Fully Dynamic Algorithms for Knapsack Problems with Polylogarithmic Update Time

Franziska Eberle ⊠

Faculty of Mathematics and Computer Science, University of Bremen, Germany

Nicole Megow \square

Faculty of Mathematics and Computer Science, University of Bremen, Germany

Lukas Nölke ⊠

Faculty of Mathematics and Computer Science, University of Bremen, Germany

IN2P3 Computing Center, CNRS, Villeurbanne, France

Andreas Wiese \square

Universidad de Chile, Chile

- Abstract

Knapsack problems are among the most fundamental problems in optimization. In the MULTIPLE KNAPSACK problem, we are given multiple knapsacks with different capacities and items with values and sizes. The task is to find a subset of items of maximum total value that can be packed into the knapsacks without exceeding the capacities. We investigate this problem and special cases thereof in the context of *dynamic algorithms* and design data structures that efficiently maintain near-optimal knapsack solutions for dynamically changing input. More precisely, we handle the arrival and departure of individual items or knapsacks during the execution of the algorithm with worst-case update time polylogarithmic in the number of items. As the optimal and any approximate solution may change drastically, we maintain implicit solutions and support polylogarithmic time query operations that can return the computed solution value and the packing of any given item.

While dynamic algorithms are well-studied in the context of graph problems, there is hardly any work on packing problems (and generally much less on non-graph problems). Motivated by the theoretical interest in knapsack problems and their practical relevance, our work bridges this gap.

2012 ACM Subject Classification Theory of computation → Packing and covering problems

Keywords and phrases Fully dynamic algorithms, knapsack problem, approximation schemes

Related Version Accepted for Publication at FSTTCS 2021.

Acknowledgements We thank Martin Böhm, Peter Kling, and Jens Schlöter for the fruitful discussions. This work is partly funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) — Project Number 146371743 — TRR 89 Invasive Computing.

1 Introduction

Knapsack problems are among the most fundamental optimization problems. In their most basic form, we are given a knapsack capacity $S \in \mathbb{N}$ and a set of n items, where each item $j \in [n] := \{1, 2, \dots, n\}$ has a size $s_j \in \mathbb{N}$ and a value $v_j \in \mathbb{N}$. The Knapsack problem asks for a subset of items, $P \subseteq [n]$, with maximal total value $v(P) := \sum_{j \in P} v_j$ and with a total size $s(P) := \sum_{j \in P} s_j$ that does not exceed the knapsack capacity S. In the more general Multiple Knapsack problem, we are given m knapsacks with capacities S_i for $i \in [m]$. Here, the task is to select m disjoint subsets $P_1, P_2, \dots, P_m \subseteq [n]$ such that subset P_i satisfies the capacity constraint $s(P_i) \leq S_i$ and the total value of all subsets $\sum_{i \in [m]} v(P_i)$ is maximized.

MULTIPLE KNAPSACK is strongly NP-hard, even for identical knapsack capacities, as it is a special case of bin packing. KNAPSACK, on the other hand, is only weakly NP-hard and admits pseudo-polynomial time algorithms, the first one being already published in the 1950s [5].

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

The first approximation scheme for KNAPSACK was an FPTAS by Ibarra and Kim [44] and initiated a long sequence of follow-up work, which is still active [18,53]. MULTIPLE KNAPSACK is substantially harder and does not admit an FPTAS, unless P = NP, even with two identical knapsacks [20]. However, approximation schemes with running times of the form $n^{f(\varepsilon)}$ (PTASs) are known [20, 55] as well as improvements to only $f(\varepsilon)n^{\mathcal{O}(1)}$ (EPTASs) [49,51]. All these algorithms are *static* in the sense that the full instance is given to an algorithm and is then solved.

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

A practical application is the dynamic estimation of the profit for scheduling jobs in computing clusters in which virtual machines can be moved among physical machines [6]. This allows the service provider to adapt the provided capacity, i.e., the currently running servers, to the current demand, see, e.g., [14, 24, 60]. An efficient framework for MULTIPLE KNAPSACK can be viewed as a first-stage decision tool: In real-time, it determines whether the customer in question should be allowed into the system based on the cost of possibly powering and using additional servers. As the service provider has to decide immediately which request she wants to accept, she needs to obtain the information fast, i.e., sublinear in the number of requests already in the system.

Generally, dynamic algorithms constitute a vibrant research field in the context of graph problems. We refer to surveys [16, 27, 39] for an overview on dynamic graph algorithms. Interestingly, only for a small number of graph problems there are dynamic algorithms known with polylogarithmic update time, among them connectivity problems [41, 43], the minimum spanning tree [43], and vertex cover [10,12]. Recently, this was complemented by conditional lower bounds that are typically linear in the number of nodes or edges; see, e.g., [2]. Over the last few years, the generalization of dynamic vertex cover to dynamic set cover gained interest leading to near-optimal approximation algorithms with polylogarithmic update times [1,9,11,35]. Also, recently, algorithms have been developed for maintaining maximal independent sets, e.g., [4, 19, 65], and approximate maximum independent sets in special graph classes [13, 21, 40].

For packing problems, there are hardly any dynamic algorithms with small update time

known. A notable exception is a result for bin packing that maintains a $\frac{5}{4}$ -approximative solution with $\mathcal{O}(\log n)$ update time [46]. This lack of efficient dynamic algorithms is in stark contrast to the aforementioned intensive research on computationally efficient algorithms for packing problems. Our work bridges this gap initiating the design of data structures and algorithms that efficiently maintain near-optimal solutions.

Our Contribution In this paper, we present dynamic algorithms for maintaining approximate solutions for three problems of increasing complexity: KNAPSACK, MULTIPLE KNAPSACK with identical knapsack sizes, and general Multiple Knapsack. Our algorithms are fully dynamic which means that in an update operation they can handle the arrival or departure of an item and of a knapsack. Further, we consider the *implicit solution* or *query* model, in which an algorithm is not required to store the solution explicitly in memory such that the solution can be read in linear time at any given point of the execution. Instead, the algorithm may maintain the solution implicitly with the guarantee that a query about the packing can be answered in polylogarithmic time.

We give worst-case guarantees for update and query times that are polylogarithmic in n, the number of items currently in the input, and bounded by a function of $\varepsilon > 0$, the desired approximation accuracy. For some special cases, we can even ensure a polynomial dependency on $\frac{1}{\varepsilon}$. In others, we justify the exponential dependency with corresponding lower bounds. Denote by v_{max} the currently largest item value and by \overline{v} an upper bound on v_{max} that is known in advance.

- 1. For Multiple Knapsack, we design a dynamic algorithm maintaining a (1ε) approximate solution with update time $2^{f(1/\varepsilon)} \left(\frac{1}{\varepsilon} \log n \log \overline{v}\right)^{\mathcal{O}(1/\varepsilon)} (\log S_{\max})^{\mathcal{O}(1)}$, where f is quasi-linear, and query time $\left(\frac{1}{\varepsilon}\log n\right)^{\mathcal{O}(1)}$.

 2. The exponential dependency on $\frac{1}{\varepsilon}$ in the update time for MULTIPLE KNAPSACK is indeed
- necessary, even for two identical knapsacks. We show that there is no $(1-\varepsilon)$ -approximate dynamic algorithm with update time $\left(\frac{1}{\varepsilon}\log n\right)^{\mathcal{O}(1)}$, unless P=NP.

 3. For KNAPSACK, we give a dynamic $(1-\varepsilon)$ -approximation algorithm with update time
- $\left(\frac{1}{\varepsilon}\log\left(nv_{\max}\right)\right)^{\mathcal{O}(1)} + \mathcal{O}\left(\frac{1}{\varepsilon}\log n\log \overline{v}\right)$ and constant query times.
- 4. For MULTIPLE KNAPSACK with identical knapsacks with capacity S each, we improve the update time to $\left(\frac{1}{\varepsilon}\log n\log v_{\max}\log S\right)^{\mathcal{O}(1)}$ if $m\geq \frac{16}{\varepsilon^7}\log^{-2}n$ with query time $\left(\frac{1}{\varepsilon}\log n\right)^{\mathcal{O}(1)}$.

In each update step, we compute only implicit solutions and provide query operations for the solution value, the knapsack of a queried item, and the complete solution. These queries are consistent between two update steps and run efficiently, i.e., run in time polynomial in log n and log \overline{v} and linear in the output size. We remark that it is not possible to maintain a solution with a non-trivial approximation guarantee explicitly with only polylogarithmic update time (even amortized) since it might be necessary to change $\Omega(n)$ items per iteration, e.g., if a very large and very profitable item is inserted and removed in each iteration.

We remark that our result yields a static algorithm with a near-linear running time in n.

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

4 Fully Dynamic Algorithms for Knapsack Problems with Polylogarithmic Update Time

extracted efficiently from suitable data structures in which we store the input items and knapsacks. Even more, we show that we can maintain these data structures in polylogarithmic time per update.

On a high level, we reduce the overall problem to two subproblems solved independently. In the first one, we are given only few knapsacks, $m = \left(\frac{1}{\varepsilon}\log n\right)^{\mathcal{O}(1)}$ many, which are the largest knapsacks in the original input. Here, we observe that if we select the $\frac{m}{\varepsilon}$ most valuable items in the optimal solution correctly, we can afford to fill the remaining space in the knapsacks greedily, i.e., highest density (value divided by size) first, and charge the resulting loss to the valuable items. We cannot guess these most valuable items explicitly, but we show that we can select a small set of candidates for these items and guess a few placeholder items for the remaining ones. This yields an instance with only $\left(\frac{1}{\varepsilon}\log n\right)^{\mathcal{O}(1)}$ items on which we run a known EPTAS for MULTIPLE KNAPSACK [51] yielding a running time of $\left(\frac{1}{\varepsilon}\log n\right)^{\mathcal{O}(1)}$. For the special case of a single knapsack, we show that we can invoke an FPTAS instead, which improves the running time.

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

Further Related Work Since the first approximation scheme for KNAPSACK [44] running times have been improved steadily [18,31,32,53,56,59,69] with $\mathcal{O}(n\log\frac{1}{\varepsilon}+(\frac{1}{\varepsilon})^{9/4})$ by Jin [53] being the currently fastest. Recent work on conditional lower bounds [23,58] implies that KNAPSACK does not admit an FPTAS with running time of $\mathcal{O}((n+\frac{1}{\varepsilon})^{2-\delta})$, for any $\delta > 0$, unless (min, +)-convolution has a subquadratic algorithm [18,66].

A PTAS for MULTIPLE KNAPSACK was first presented by Chekuri and Khanna [20] and EPTAS s due to Jansen [49,51] are also known. The fastest of these algorithms [51] has a running time of $2^{\mathcal{O}(\log^4(1/\varepsilon)/\varepsilon)} + n^{\mathcal{O}(1)}$. The mentioned algorithms are all static and assume full knowledge about the instance for which a complete solution has to be found. In particular, their solutions might change completely when a single item is added to the input which makes a full recomputation necessary. The algorithm in [20] invokes a guessing step with $n^{f(1/\varepsilon)}$ many options which are too many for a polylogarithmic update time. The EPTASs in [49,51] use a configuration linear program of size $\Omega(n)$ which is also prohibitively

large for such an update time.

The dynamic arrival and removal of items exhibits some similarity to knapsack models with incomplete information. For example, in the *online* knapsack problem [62] items arrive online one by one. When an item arrives, an algorithm must irrevocably accept or reject it before the next item arrives. Various problem variants have been studied, e.g., with resource augmentation [48], the removable online knapsack problem [22,36–38,47], and with advice [15]. Other models with uncertainty in the item set or the knapsack capacity include the *stochastic* knapsack problem [8, 26, 61] and *robust* knapsack problems [17, 28, 63, 72]. Related to our setting are also online models with a softened irrevocability requirement, e.g., online optimization with *recourse* [29, 34, 45, 64] or *migration* [52, 70, 71] allows to adapt previously taken decisions in a limited way. We are not aware of work on knapsack problems in these settings and, again, the goal is to bound the amount of change needed to maintain good online solutions regardless of the computational effort.

2 Roadmap and Preliminaries

First, in this section, we formalize the operations that our data structures support, describe auxiliary data structures that we need, and define how we round the item values. Then, in Section 3, we describe algorithms for one knapsack and for a polylogarithmic number of knapsacks. In Section 4, we present an algorithm for (many) identical knapsacks and an algorithm under resource augmentation (in the form of a polylogarithmic number of additional knapsacks) in the setting of (many) knapsacks with possibly different capacities. Finally, we present in Section 5 an algorithm for the general case that uses the previously mentioned algorithms as subroutines. Additionally, in Appendix G, we show that our update time cannot be improved to $(\log n/\varepsilon)^{\mathcal{O}(1)}$, unless P=NP.

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

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

A new solution can be output, entirely or in parts, using the following query operations.

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

Importantly, queries are consistent in-between two update operations. However, their answers are not independent of each other but depend on the queries as well as their order.

For simplicity, we assume that elementary operations (e.g., additions) can be handled in constant time. Additionally, we assume without loss of generality that $\frac{1}{\varepsilon} \in \mathbb{N}$. We also assume that at the very beginning we start with no items and no knapsacks, and initialize all needed auxiliary data structures accordingly. If one wants to start with a specific set of items and/or knapsacks, one can insert them with our insertion routines, using polylogarithmic time per insertion.

Auxiliary Data Structures We employ auxiliary data structures in which we store (subsets of) input items and input knapsacks, sorted according to some specific values, e.g., size or capacity. We need to be able to quickly access elements, compute the largest prefix of elements such that the sum according to some property, e.g., the total size, is below a given threshold, and compute in such a prefix the sum according to some element property, e.g., the total value. Note that these prefixes are w.r.t. the fixed ordering of the elements, while the element property for the threshold or computing the sum might be different. To this end, we employ as an auxiliary data structure a variation of balanced search trees that store elements according to some given ordering. For computing the mentioned prefix sums, we store in each internal node v the sums of the elements in the subtree rooted at v according to each property, e.g., size, value, or capacity. When we need to compute some largest prefix, we simply output the index of its last element.

▶ Lemma 1. There is a data structure maintaining a sorting of n' elements w.r.t. to some key value such that (i) insertion, deletion, or search by key value of an element takes $\mathcal{O}(\log n')$ time, and (ii) prefixes and prefix sums w.r.t. to any element property can be computed in time $\mathcal{O}(\log n')$.

Rounding Values A crucial ingredient of our algorithms is the partitioning of items into only few value classes V_{ℓ} , where for each ℓ the class V_{ℓ} consists of each input item j with $(1+\varepsilon)^\ell \le v_j < (1+\varepsilon)^{\ell+1}$. Upon arrival of some item j, we calculate the index ℓ_j such that $j \in V_{\ell_i}$ and store the tuple (j, v_j, s_j, ℓ_j) representing j in the auxiliary data structures of the respective algorithm. In the following, we pretend for each ℓ that each item in V_{ℓ} has value $(1+\varepsilon)^{\ell}$, which loses only a factor of $\frac{1}{1+\varepsilon}$ in the total profit of any solution.

▶ **Lemma 2.** (i) There are at most $\mathcal{O}(\frac{\log v_{\text{max}}}{\varepsilon})$ many value classes. (ii) For optimal solutions OPT and OPT' for the original and rounded instance, $v(OPT') \ge (1 - \varepsilon) \cdot v(OPT)$.

3 A Single Knapsack

In this section, we first present a dynamic algorithm for the case of one single knapsack, summarized in the following theorem. Afterwards, we will argue how to extend our techniques to the setting of a polylogarithmic number of knapsacks.

▶ Theorem 3. For $\varepsilon > 0$, there is a fully dynamic algorithm for KNAPSACK that maintains $(1-\varepsilon)$ -approximate solutions with update time $\mathcal{O}(\frac{\log^4(nv_{\max})}{\varepsilon^9}) + \mathcal{O}(\frac{1}{\varepsilon}\log n\log \overline{v})$. Furthermore, queries of single items and the solution value can be answered in time $\mathcal{O}(1)$.

We partition the items in the optimal solution OPT into high- and low-value items, respectively. The high-value items are the $\frac{1}{\varepsilon}$ most valuable items of OPT, and the low-value items are the remaining items of OPT. We compute a small set of candidate items $H_{\frac{1}{2}}$ that intuitively contains all relevant high-value items in OPT. Also, we guess a placeholder item for the low-value items, that is large enough to accommodate low-value items of enough profit fractionally. We can assume that in an optimal fractional solution (of low-value items) at most one item is selected non-integrally. Hence, we can drop this item and charge it to the $\frac{1}{\varepsilon}$ high-value items. This results in a knapsack instance with only $\mathcal{O}\left(\frac{1}{\varepsilon^3}\right)$ items which we solve with an FPTAS.

Formally, denote by $OPT_{\frac{1}{2}}$ a set of $\frac{1}{\epsilon}$ most valuable items of OPT. We break ties by picking smaller items. Denote by $V_{\ell_{\max}}$ and $V_{\ell_{\min}}$ the highest resp. lowest value class of an element in $OPT_{\frac{1}{\varepsilon}}$ and let $n_{\min} \coloneqq |OPT_{\frac{1}{\varepsilon}} \cap V_{\ell_{\min}}| \le \frac{1}{\varepsilon}$. Furthermore, denote by \mathcal{V}_L the value of the items in $\text{OPT} \setminus \text{OPT}_{\frac{1}{\varepsilon}}$, rounded down to the next power of $(1+\varepsilon)$. To efficiently implement our algorithm, we maintain several data structures, using Lemma 1. We store items of each non-empty value class V_{ℓ} (at most $\log_{1+\varepsilon}v_{\max}$) in a data structure ordered non-decreasingly by size. Second, for each possible value class V_{ℓ} (at most $\log_{1+\varepsilon}\overline{v}$), we maintain a data structure that contains each input item j with $j \in V_{\ell'}$ for some $\ell' \leq \ell$, ordered non-increasingly by density $\frac{v_j}{s_j}$. In particular, we maintain such a data structure even if V_{ℓ} itself is empty (since the data structure might still contain items from classes $V_{\ell'}$ with $\ell' < \ell$). This leads to the additive term in the update time of $\mathcal{O}(\log n \log_{1+\varepsilon}\overline{v})$. We use additional auxiliary data structures to store our solution and support queries.

Algorithm The algorithm computes an implicit solution as follows.

- 1) Compute a set $H_{\frac{1}{\varepsilon}}$ of high-value candidates: Guess the values ℓ_{\max} , ℓ_{\min} , and n_{\min} . If $(1+\varepsilon)^{\ell_{\min}} \geq \varepsilon^2 \cdot (1+\varepsilon)^{\ell_{\max}}$, define $H_{\frac{1}{\varepsilon}}$ to be the set containing the $\frac{1}{\varepsilon}$ smallest items of each of the value classes $V_{\ell_{\min}+1}, \ldots, V_{\ell_{\max}}$, plus the n_{\min} smallest items from $V_{\ell_{\min}}$. Otherwise, set $H_{\frac{1}{\varepsilon}}$ to be the union of the $\frac{1}{\varepsilon}$ smallest items of each of the value classes with values in $[\varepsilon^2 \cdot (1+\varepsilon)^{\ell_{\max}}, (1+\varepsilon)^{\ell_{\max}}]$.
- 2) Create a placeholder item B: Guess \mathcal{V}_L and consider items with value at most $(1+\varepsilon)^{\ell_{\min}}$ sorted by density. Remove the n_{\min} smallest items of $V_{\ell_{\min}}$ until the next iteration. For the remaining items, compute the minimal size of fractional items necessary to reach a value \mathcal{V}_L . We do this via prefix sum computations on the data structure that contains all items in $V_{\ell'}$ for each $\ell' \leq \ell_{\min}$, ordered non-increasingly by density. Then B is given by $v_B = \mathcal{V}_L$ and with s_B equal to the size of those low-value items.
- 3) Use an FPTAS: On the instance I, consisting of $H_{\frac{1}{\varepsilon}}$ and the placeholder item B, run an FPTAS parameterized by ε (we use the one by Jin [53]) to obtain a packing P.
- 4) **Implicit solution:** Among all guesses, keep the solution P with the highest value. Pack items from $H_{\frac{1}{\varepsilon}}$ as in P and, if $B \in P$, also pack the low-value items completely contained in B (note that at most one item is packed fractionally in B). While used candidate items from $H_{\frac{1}{\varepsilon}}$ can be stored explicitly, low-value items are given only implicitly by saving the correct guesses and computing membership in B on a query.

Analysis We show that the above algorithm attains an approximation ratio of $(1 - \varepsilon)$. A factor of $(1 - \varepsilon)$ is lost due to the approximation ratio of the FPTAS. An additional factor of $(1 - \varepsilon)$ is lost in each of the following steps. To obtain a candidate set $H_{\frac{1}{\varepsilon}}$ of constant cardinality, we restrict the item values to $[\varepsilon^2 \cdot (1 + \varepsilon)^{\ell_{\max}}, (1 + \varepsilon)^{\ell_{\max}}]$. Since $|\operatorname{OPT}_{\frac{1}{\varepsilon}}| = \frac{1}{\varepsilon}$, this excludes items from OPT with a total value of at most $\frac{1}{\varepsilon} \cdot \varepsilon^2 (1 + \varepsilon)^{\ell_{\max}} \le \varepsilon \cdot \operatorname{OPT}$. Furthermore, due to guessing \mathcal{V}_L up to a power of $(1 + \varepsilon)$, we get $v_B = \mathcal{V}_L \ge \frac{1}{1 + \varepsilon} \cdot v(\operatorname{OPT} \setminus \operatorname{OPT}_{\frac{1}{\varepsilon}})$. Finally, in Step 2, at most one item was cut fractionally. It is charged to the $\frac{1}{\varepsilon}$ items of $\operatorname{OPT}_{\frac{1}{\varepsilon}}$, using that each of them has a larger value.

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

Queries We show how to efficiently handle the different types of queries.

■ Single Item Query: If the queried item is contained in $H_{\frac{1}{\varepsilon}}$, its packing was saved explicitly. Otherwise, if B is packed, we save the last, i.e., least dense, item contained entirely in B. By comparing with this item, membership in B can be decided in constant time on a query.

- **Solution Value Query:** While the algorithm works with rounded values, we use the data structures of Lemma 1 to retrieve the actual item values. We store the actual solution value in the update step by adding the actual values of the packed items from H_{\perp} and determining the actual value of items in B with a prefix computation. On query, we return the stored value.
- **Query Entire Solution:** Output the stored packing of candidates. If B was packed, iterate over items in B in the respective density-sorted data structure and output them.

Polylogarithmically many knapsacks One can show that the queries can be performed in the claimed running times which completes the proof of Theorem 3, see Appendix A. We can extend the above technique to the setting of m knapsacks, at the expense of increasing the update time and query time by a factor $m^{\mathcal{O}(1)}$, and using an EPTAS for MULTIPLE KNAPSACK [51] instead of an FPTAS (see Appendix B).

▶ Theorem 4. For $\varepsilon > 0$, there is a dynamic algorithm for MULTIPLE KNAPSACK that achieves an approximation factor of $(1-\varepsilon)$ with update time $2^{f(1/\varepsilon)} \left(\frac{m}{\varepsilon} \log (nv_{\max})\right)^{\mathcal{O}(1)} +$ $\mathcal{O}(\frac{1}{\varepsilon}\log \overline{v}\log n)$, with f quasi-linear. Item queries are answered in time $\mathcal{O}(\log \frac{m^2}{\varepsilon^6})$, solution value queries in time $\mathcal{O}(1)$, and queries of one knapsack or the entire solution in time linear in the output.

4 Identical Knapsacks

In this section, we present our algorithm for an arbitrary (large) number of identical knapsacks. Also, we describe an extension to the case where the knapsacks have different sizes and we can use some additional knapsacks as resource augmentation.

4.1 Oblivious Linear Grouping

We start with our oblivious linear grouping routine that we use in order to round the item sizes, aiming at only few different types of items. We say that two items j, j' are of the same type if $\{j,j'\}\subseteq V_{\ell}$ for some ℓ and if $s_j=s_{j'}$. We round the items implicitly, i.e., we compute thresholds $\{\bar{s}_1, ..., \bar{s}_k\}$ and we round up the size s_j of each item j to the next larger value in this set.

▶ Lemma 5. Given a set J' with $|OPT \cap J'| \leq n'$ for all optimal solutions OPT, there is an algorithm with running time $\mathcal{O}(\frac{\log^5 n'}{\varepsilon^5})$ that rounds the items in J' to item types \mathcal{T} with $|\mathcal{T}| \leq \mathcal{O}(\frac{\log^2 n'}{\varepsilon^4})$ and ensures $v(OPT_{\mathcal{T}}) \geq \frac{(1-\varepsilon)(1-2\varepsilon)}{(1+\varepsilon)^2}v(OPT)$. Here, $OPT_{\mathcal{T}}$ is the optimal solution attainable by packing item types \mathcal{T} instead of the items in J' and using $J \setminus J'$ as is.

Algorithm In the following, we use the notation X' for a set X to refer to $X \cap J'$ while X'' refers to $X \setminus J'$. Recall that item values of items in J are rounded to powers of $1 + \varepsilon$ to create the value classes V_{ℓ} where each item $j \in V_{\ell}$ has value $(1+\varepsilon)^{\ell}$. We guess ℓ_{max} which is defined to be the guess for the highest value ℓ with $V'_{\ell} \cap \text{OPT} \neq \emptyset$ and let $\bar{\ell} := \ell_{\text{max}} - \lceil \log_{1+\varepsilon}(n'/\varepsilon) \rceil$.

1) For each ℓ with $\bar{\ell} \leq \ell \leq \ell_{\text{max}}$ and each $n_{\ell} = (1 + \varepsilon)^k$ with $0 \leq k \leq \log_{1+\varepsilon} n'$ do: Consider the n_ℓ smallest elements of V'_ℓ (sorted by increasing size) and determine the $\frac{1}{\varepsilon}$ many (almost) equal-sized groups $G_1(n_\ell), \ldots, G_{1/\varepsilon}(n_\ell)$ of $[\varepsilon n_\ell]$ or $[\varepsilon n_\ell]$ elements. If $\varepsilon n_\ell \notin \mathbb{N}$, ensure that $|G_k(n_\ell)| \leq |G_{k'}(n_\ell)| \leq |G_k(n_\ell)| + 1$ for $k \leq k'$. If $\frac{1}{\varepsilon}$ is not a natural power of $(1+\varepsilon)$, create $G_1(\frac{1}{\varepsilon}), \ldots, G_{1/\varepsilon}(\frac{1}{\varepsilon})$ where $G_k(\frac{1}{\varepsilon})$ is the kth smallest item in V'_{ℓ} . Let $G_1(n_\ell), \ldots, G_{1/\varepsilon}(n_\ell)$ be the corresponding groups sorted increasingly by the size

of the items. Let $j_k(n_\ell) = \max\{j : j \in G_k(n_\ell)\}$ be the last index belonging to group $G_k(n_\ell)$. After having determined $j_k(n_\ell)$ for each possible value n_ℓ (including $\frac{1}{\varepsilon}$) and for each $1 \le k \le \frac{1}{\varepsilon}$, the size of each item j is rounded up to the size of the next larger item j' such that there exists k and ℓ satisfying $j' = j_k(n_\ell)$.

2) Discard each item j with $j \in V'_{\ell}$ for $\ell < \bar{\ell}$.

Analysis Despite the new approach to apply linear grouping simultaneously to many possible values of n_{ℓ} , the analysis builds on standard techniques. The loss in the objective function due to rounding item values is bounded by a factor of $\frac{1}{1+\varepsilon}$ by Lemma 2. As $\bar{\ell}$ is chosen such that n' items of value at most $(1+\varepsilon)^{\bar{\ell}}$ contribute less than an ε -fraction of OPT', the loss in the objective function by discarding items in value classes V'_{ℓ} with $\ell < \bar{\ell}$ is bounded by a factor $(1-\varepsilon)$. By taking only $(1+\varepsilon)^{\lfloor \log 1+\varepsilon n_{\ell} \rfloor}$ items of V'_{ℓ} instead of n_{ℓ} , we lose at most a factor $\frac{1}{1+\varepsilon}$. The groups created by oblivious linear grouping are an actual refinement of the groups created by classical linear grouping. Thus, we pack our items similarly: not packing the group with the largest items (at the loss of a factor of $(1-2\varepsilon)$) allows us to "move" all rounded items of group $G_k(n_{\ell})$ to the positions of the (not rounded) items in group $G_{k+1}(n_{\ell})$. Combining, we obtain $v(\text{OPT}_{\mathcal{T}}) \geq \frac{(1-\varepsilon)(1-2\varepsilon)}{(1+\varepsilon)^2} v(\text{OPT})$.

group $G_{k+1}(n_{\ell})$. Combining, we obtain $v(\text{OPT}_{\mathcal{T}}) \geq \frac{(1-\varepsilon)(1-2\varepsilon)}{(1+\varepsilon)^2}v(\text{OPT})$. Since \mathcal{T} contains at most $\frac{1}{\varepsilon}\left(\left\lceil\frac{\log n'/\varepsilon}{\log (1+\varepsilon)}\right\rceil+1\right)$ different value classes, and as it suffices to use $\left\lceil\frac{\log n'}{\log (1+\varepsilon)}\right\rceil+1$ many different values for $n_{\ell}=|\text{OPT}\cap V_{\ell}'|$, we have $|\mathcal{T}|\leq \mathcal{O}(\frac{\log^2 n'}{\varepsilon^4})$. Using the access times given in Lemma 1 bounds the running time. For details, see Appendix C.

4.2 A Dynamic Algorithm for Many Identical Knapsacks

We give a dynamic algorithm with approximation ratio $(1-\varepsilon)$ for Multiple Knapsack, assuming that all knapsacks have the same size S. We assume $m \le n$ as otherwise, the problem is trivial. We focus on instances where m is large, i.e., $m \ge \frac{16}{\varepsilon^7} \log^2 n$. If $m \le \frac{16}{\varepsilon^7} \log^2 n$, we use the algorithm due to Theorem 4. In the following, we prove Theorem 6.

▶ Theorem 6. If $m \geq \frac{16}{\varepsilon^7}\log^2 n$, there is a dynamic algorithm for MULTIPLE KNAPSACK with identical knapsacks with approximation factor $(1-\varepsilon)$ and update time $\left(\frac{\log U}{\varepsilon}\right)^{\mathcal{O}(1)}$, where $U = \max\{Sm, nv_{\max}\}$. Queries for single items and the solution value can be answered in time $\mathcal{O}\left(\frac{\log n}{\varepsilon}\right)^{\mathcal{O}(1)}$ and $\mathcal{O}(1)$, respectively. The solution P can be returned in time $|P|\left(\frac{\log n}{\varepsilon}\right)^{\mathcal{O}(1)}$.

Our strategy is the following: we partition the input items into large and small items, which are defined w.r.t. the size S of each knapsack. To the large items, we apply oblivious linear grouping, obtaining a polylogarithmic number of item types. We guess the total size of the small items in the optimal solution. Then, we formulate the problem as a configuration linear program (LP) which has a variable for each feasible configuration for a knapsack. A configuration describes how many large items of each type are packed in a knapsack. Also, we ensure that there will be enough space for the small items left. This is similar in spirit to the LPs used in [49,51]; however, we use variables only for the configurations of the big items and we have only a polylogarithmic number of item types, which yields a smaller LP which we can solve faster. We round the obtained fractional solution, using that $m > \frac{16}{\varepsilon^7} \log^2 n$ and that basic feasible solutions to the LP are sparse.

Definitions and Data Structures We partition the items into two sets, J_B , the *big* items, and J_S , the *small* items, with sizes $s_j \geq \varepsilon S$ and $s_j < \varepsilon S$, respectively. For an optimal solution OPT, define OPT_B := OPT $\cap J_B$ and OPT_S := OPT $\cap J_S$.

We maintain three types of auxiliary data structures from Lemma 1: we maintain one such data structure in which we store all items in the order of their arrivals and store the size s_i , the value v_i , and the value class ℓ_i of each item j. For each value class V_{ℓ} , we maintain a data structure which contains all big items of V_{ℓ} , ordered non-decreasingly by size. Finally, for the small items (of all value classes together), we maintain a data structure in which they are sorted non-increasingly by density. Upon arrival of a new item j, we insert j into each corresponding data structure.

Algorithm

- 1) Linear grouping of big items: Guess ℓ_{max} , which we define to be the largest index ℓ with $V_{\ell} \cap \text{OPT}_B \neq \emptyset$. Via oblivious linear grouping with $J' = J_B$ and $n' = \min\{\frac{m}{\varepsilon}, n_B\}$ we obtain \mathcal{T} ; for each item type t, denote by n_t the number of items of this type (the multiplicity of t).
- 2) Configurations: Let \mathcal{C} denote the set of all configurations, i.e., of all multisets of item types whose total size is at most S. For each $c \in \mathcal{C}$, denote by v_c and s_c the total value and size of the item types in c.
- 3) Small items: We guess v_S which we define to be the largest power of $1+\varepsilon$ that is at most $v(OPT_S)$. Let P be the maximal prefix of small items (sorted by non-increasing density) with $v(P) < v_S$. Set $s_S := s(P)$.
- 4) Configuration ILP: We compute an extreme point solution of the LP relaxation of the following configuration ILP with variables y_c for $c \in \mathcal{C}$ for the current guesses ℓ_{max} and v_S (implying s_S). Here, y_c counts how often a certain configuration c is used and n_{tc} denotes the number of items of type t in configuration c.

max
$$\sum_{c \in \mathcal{C}} y_c v_c$$
subject to
$$\sum_{c \in \mathcal{C}} y_c s_c \leq \lfloor (1 - 3\varepsilon)m \rfloor S - s_S$$

$$\sum_{c \in \mathcal{C}} y_c \leq \lfloor (1 - 3\varepsilon)m \rfloor \qquad (P)$$

$$\sum_{c \in \mathcal{C}} y_c n_{tc} \leq n_t \qquad \text{for all } t \in \mathcal{T}$$

$$y_c \in \mathbb{Z}_{\geq 0} \qquad \text{for all } c \in \mathcal{C}$$

By the first inequality, the configurations fit into $|(1-3\varepsilon)m|$ knapsacks while reserving sufficient space for the small items. The second constraint limits the total number of configurations that are packed. The third inequality ensures that only available items are used.

- 5) Obtaining an integral solution: We round up each variable of the obtained fractional solution, yielding an integral solution \bar{y} . As $m \geq \frac{16}{\varepsilon^7} \log^2 n$ and extreme point solutions have only $|\mathcal{T}| + 2$ non-zero variables, one can show that \bar{y} still satisfies the relaxed constraints $\sum_{c \in \mathcal{C}} \bar{y}_c s_c \leq \lfloor (1 - 2\varepsilon)m \rfloor S - s_S$ and $\sum_{c \in \mathcal{C}} \bar{y}_c \leq \lfloor (1 - 2\varepsilon)m \rfloor$. In case that a constraint $\sum_{c\in\mathcal{C}} \bar{y}_c n_{tc} \leq n_t$ is violated for some type t, we intuitively drop items of type t from some knapsacks until the constraint is satisfied. Let P_B denote the resulting packing.
- 6) Packing small items: Consider the maximal prefix P of small items with $v(P) < v_S$ and let j^* be the densest small item not in P. Pack j^* into one of the knapsacks kept empty by P_B . Then, fractionally fill up the $\lfloor (1-2\varepsilon)m \rfloor$ knapsacks used by P_B and place any "cut" item into the $\lceil \varepsilon m \rceil$ additional knapsacks that are still empty.

Analysis The loss in the objective function value due to linear grouping of big items is bounded by $\frac{(1-\varepsilon)(1-2\varepsilon)}{(1+\varepsilon)^2}$ by Lemma 5. Restricting a solution to its $\lfloor (1-3\varepsilon)m \rfloor$ most valuable knapsacks and guessing the value of small items in these knapsacks only up to a factor of $(1+\varepsilon)$ as done by (P) costs at most a factor of $\frac{1-4\varepsilon}{1+\varepsilon}$ in the objective function value.

For solving the LP-relaxation of the configuration ILP (P), we apply the Ellipsoid method [33] on its dual, using an FPTAS for KNAPSACK as a separation oracle. For this, we need to handle some technical complications due to the first two constraints of (P), which yield additional variables in the dual, and due to the fact that we can solve the separation problem only up to a factor of $(1+\varepsilon)$ (see Appendix D for details). Via Gaussian elimination, we transform the obtained fractional solution into a basic feasible solution with the same objective function value. As argued above, since any basic feasible solution has at most $|\mathcal{T}| + 2$ non-zero variables, our integral solution \bar{y} uses at most $\lfloor (1-2\varepsilon)m \rfloor$ knapsacks and it has at least the profit of the fractional solution. Given the packing of big items, we pack the small items in a FIRST FIT manner as described in the algorithm.

To bound the running time of our algorithm, we use Lemma 5, show that the relaxation of the configuration ILP can be solved in time $\left(\frac{\log U}{\varepsilon}\right)^{\mathcal{O}(1)}$ with the Ellipsoid method, and use the fact that the algorithm needs at most $\mathcal{O}\left(\frac{\log (nv_{\max})\log v_{\max}}{\varepsilon^2}\right)$ many guesses, see Appendix D for details.

Queries In contrast to the previous section, for transforming an implicit solution into an explicit packing, the query operation has to compute the knapsack where a queried item j is packed. We do not explicitly store the packing of any item, but instead we define and update pointers for small items and for each item type, that indicate the knapsacks where the corresponding items are packed. To stay consistent with the precise packing of a particular item between two update operations, we additionally cache query answers.

- Single Item Query: For small items, only the prefix of densest items is part of our solution. For big items of a certain type, only the smallest items are packed by the implicit solution. In both cases, we use the corresponding pointer to determine the knapsack.
- Solution Value Query: As the algorithm works with rounded values, we use prefix computations on the small items and on any value class of big items to calculate and store the current solution value. Given a query, we return the stored solution value.
- Query Entire Solution: We use prefix computations on the small items as well as on the value classes of the big items to determine the packed items. Then, we use the Single Item Query to determine their respective knapsacks.
- ▶ Lemma 7. The solution determined by the query algorithms is feasible and achieves the claimed total value. The query times of our algorithm are as follows: Single item queries can be answered in time $\mathcal{O}(\log n + \max\left\{\log\frac{\log n}{\varepsilon}, \frac{1}{\varepsilon}\right\})$, solution value queries can be answered in time $\mathcal{O}(1)$, and queries of the entire solution P can be answered in time $\mathcal{O}(|P|\frac{\log^4 n}{\varepsilon^4}\log\frac{\log n}{\varepsilon})$.

We extend our techniques above to an algorithm for knapsacks of arbitrary sizes, assuming that we have $\left(\frac{\log n}{\varepsilon}\right)^{\Theta(1/\varepsilon)}$ additional knapsacks (of capacity at least as large as the largest original knapsack) as resource augmentation available. The intuition is that these additional knapsacks are sufficient to compensate errors when rounding the LP-relaxation of (P). However, additional care is needed since whether an item is big or small now depends on the knapsack.

▶ Theorem 8. For $\varepsilon > 0$, there is a dynamic algorithm for MULTIPLE KNAPSACK that, given $\left(\frac{\log n}{\varepsilon}\right)^{\Theta(1/\varepsilon)}$ additional knapsacks as resource augmentation, achieves an approximation

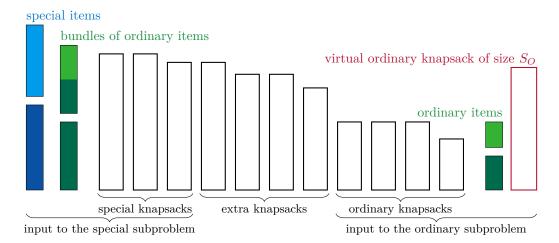


Figure 1 Input of the special and the ordinary subproblems: Based on the current guess for the extra knapsacks, the knapsacks are partitioned into three groups (special, extra, and ordinary). When an item fits into at least one ordinary knapsack, it is ordinary and special otherwise. The total size of ordinary items placed by OPT in special knapsacks gives the size of the virtual ordinary knapsack. The ordinary items packed into this virtual knapsack are further assigned to bundles of equal size, which are then part of the input to the special subproblem.

factor of $(1+\varepsilon)$ with update time $(\frac{1}{\varepsilon}\log n)^{\mathcal{O}(1/\varepsilon)}(\log m\log S_{\max}\log v_{\max})^{\mathcal{O}(1)}$. Item queries are answered in time $\mathcal{O}(\log m + \frac{\log n}{\varepsilon^2})$, and the solution P is output in time $\mathcal{O}(|P| \frac{\log^3 n}{\varepsilon^4} (\log m + \frac{\log n}{\varepsilon^2}))$.

5 Solving Multiple Knapsack

Having laid the groundwork with the previous two sections, we finally show how to maintain solutions for arbitrary instances of the MULTIPLE KNAPSACK problem, and give the main result of this paper, summarized in the following theorem. Note that we assume $n \geq m$ as otherwise only the n largest knapsacks are used.

▶ Theorem 9. For $\varepsilon > 0$, there is a dynamic, $(1 - \varepsilon)$ -approximate algorithm for MULTIPLE KNAPSACK with update time $2^{f(1/\varepsilon)} \left(\frac{1}{\varepsilon}\log n\log v_{\max}\right)^{\mathcal{O}(1/\varepsilon)} (\log S_{\max})^{\mathcal{O}(1)} + \mathcal{O}\left(\frac{1}{\varepsilon}\log \overline{v}\log n\right)$, where f is quasi-linear. Item queries are served in time $\mathcal{O}\left(\frac{\log n}{\varepsilon^2}\right)$ and the solution P can be output in time $\mathcal{O}\left(\frac{\log^4 n}{\varepsilon^6}|P|\right)$.

We obtain this result by partitioning the knapsacks into three sets, special, extra and ordinary knapsacks, and solving the respective subproblems. This has similarities to the approach in [49]; however, there it was sufficient to have only two groups of knapsacks. On a high level, the special knapsacks are the $(\log n)^{\mathcal{O}(1/\varepsilon)}$ largest input knapsacks and, intuitively, we apply the algorithm due to Theorem 4 to them (for a suitably defined set of input items). The extra knapsacks are $(\log n)^{\mathcal{O}(1/\varepsilon)}$ knapsacks that are smaller than the special knapsacks, but larger than the ordinary knapsacks. We ensure that there is a (global) $(1-\varepsilon)$ -approximate solution in which they are all empty. We apply the algorithm due to Theorem 8 to the ordinary and extra knapsacks, where the extra knapsacks form the additional knapsacks used as resource augmentation.

Definitions and Data Structures Let $L = \left(\frac{\log n}{\varepsilon}\right)^{\Theta(1/\varepsilon)}$. We assume that $m > \left(\frac{1}{\varepsilon}\right)^{4/\varepsilon} \cdot L$, since otherwise we simply apply Theorem 8. Consider $\frac{1}{\varepsilon}$ groups of knapsacks with sizes $\frac{L}{\varepsilon^{3i}}$, for $i=0,1,\ldots,\frac{1}{\varepsilon}-1$, such that the first group, i.e., i=0, consists of the L largest knapsacks, the second, i.e., i=1, of the $\frac{L}{\varepsilon^3}$ next largest, and so on. In OPT, one of these contains items with total value at most ε · OPT. Let $k \in \{0,1,\ldots,\frac{1}{\varepsilon}-1\}$ be the index of such a group and let $L_S := \sum_{i=0}^{k-1} \frac{L}{\varepsilon^{3i}}$. We define the L_S largest input knapsacks to be the *special* knapsacks. The *extra* knapsacks are the $\frac{L}{\varepsilon^{3k}} > \frac{L_S}{\varepsilon^2} + L$ next largest, and the *ordinary* knapsacks the remaining ones.

Call an item ordinary if it fits into the largest ordinary knapsack and special otherwise. Denote by J_O and J_S the set of ordinary and special items, respectively, and by S_O the total size of ordinary items that OPT places in special knapsacks, rounded down to the next power of $(1 + \varepsilon)$; see Figure 1. Since we use the algorithms from Theorems 4 and 8 as subroutines, we require the maintenance of the corresponding data structures.

Algorithm

- 1) Oblivious linear grouping: Compute $\mathcal{O}(\frac{\log^2 n}{\varepsilon^4})$ item types as described in Section 4.1. Guess k and determine whether items of a certain type are ordinary or special.
- 2) **High-value ordinary items:** Place each of the $\frac{L_S}{\varepsilon^2}$ most valuable ordinary items in an empty extra knapsack. On a tie choose the larger item. Denote this set of items by J_E .
- 3) Virtual ordinary knapsack: Guess S_O and add a virtual knapsack with capacity S_O to the ordinary subproblem. In the LP used in the proof of Theorem 8, treat every ordinary item as small item in this knapsack and do not use configurations.
- 4) Solve ordinary instance: Remove temporarily the set J_E from the data structures of the ordinary subproblem. Solve the subproblem with the virtual knapsack as in Theorem 8 and use extra knapsacks for resource augmentation. When rounding up variables, fill the $\mathcal{O}(\frac{\log^2 n}{\epsilon^4})$ rounded items from the virtual knapsack into extra knapsacks.
- 5) Create bundles Consider the items that remain in the virtual ordinary knapsack after rounding. Sort them by type (first value, then size) and cut them to form $\frac{L_S}{\varepsilon}$ bundles B_O of equal size. For each bundle, remember how many items of each type are placed entirely inside it. Place cut items into extra knapsacks. Consider each $B \in B_O$ as an item of size and value equal to the fractional size respectively value of items placed entirely in B.
- 6) Solve special instance: Temporarily insert the bundles in B_O into the data structures used in the special subproblem. Solve this subproblem with the algorithm due to Theorem 4.
- 7) Implicit solution: Among all guesses, keep the solution P_F with the highest value. Store items in J_E and their placement explicitly. Revert the removal of J_E from the ordinary data structures after the next update. For the remaining items, the solutions are given as in the respective subproblem, with the exception of items packed in the virtual ordinary knapsack. The solution of these items is stored implicitly by deciding membership in a bundle on a query.

Queries We essentially use the same approach as in Theorems 4 and 8 for the ordinary and special subproblem, respectively. However, special care has to be taken with items in the virtual knapsack. In the ordinary subproblem, we assume that items of a certain type which are packed in the virtual knapsack are the first, i.e., smallest, of that type. We can therefore decide in constant time whether or not an item is contained in the virtual knapsack and, if this is the case, fill it into the free space in special knapsacks reserved by bundles. We do this efficiently by using a first fit algorithm on the knapsacks with reserved space. Since

14 Fully Dynamic Algorithms for Knapsack Problems with Polylogarithmic Update Time

items in extra knapsacks are stored explicitly, they can be accessed in constant time. See Appendix F for details.

Hardness of approximation It is a natural question whether the update time of our algorithms for Multiple Knapsack can be improved to $\left(\frac{1}{\varepsilon}\log n\right)^{\mathcal{O}(1)}$. We show that this is impossible, unless P=NP; see Appendix G.

▶ **Theorem 10.** Unless P = NP, there is no fully dynamic algorithm for MULTIPLE KNAP-SACK that maintains a $(1 - \varepsilon)$ -approximate solution in update time polynomial in $\log n$ and $\frac{1}{\varepsilon}$, for $m < \frac{1}{3\varepsilon}$.

6 Conclusion

Any dynamic algorithm can be turned into a non-dynamic one by having n items arrive one by one, incurring an additional linear factor in the running time. Hence, lower bounds for the running times of static approximation schemes yield lower bounds for update times of dynamic algorithms. Our running times for the problems with identical capacities are tight in the sense that the algorithms yield a static FPTAS (resp. EPTAS) matching known lower bounds.

Clearly, it would be interesting to generalize our results beyond MULTIPLE KNAPSACK. A natural generalization is d-dimensional KNAPSACK, where the items and knapsacks have a size in each of the d dimensions, and a feasible packing of a subset of items must meet the capacity constraint in each dimension. A reduction to one dimension by [25] immediately yields a dynamic $\frac{1-\varepsilon}{d}$ -approximation, but designing a dynamic framework with a better guarantee than this remains open. Note that unless W[1] = FPT, 2-dimensional knapsack does not admit a dynamic algorithm maintaining a $(1-\varepsilon)$ -approximation in worst-case update time $f(\varepsilon)n^{\mathcal{O}(1)}$ [57].

A recent line of research exploits fast techniques for solving convolution problems to speed up knapsack algorithms (exact and approximate); see, e.g., [3,18,53,56,68]. In fact, it has been shown that KNAPSACK is computationally equivalent to the (min, +)-convolution problem [23]. It seems worth exploring whether such techniques are useful in the dynamic setting. Here, it is unclear whether the re-computation of a solution in a new iteration can be done in polylogarithmic time. It is also open whether such techniques can be applied for solving MULTIPLE KNAPSACK, even in the static setting.

We hope to foster further research for other packing, scheduling and, generally, non-graph problems. For bin packing and for makespan minimization on uniformly related machines, we notice that existing PTAS techniques from [54] and [42,50] combined with rather straightforward data structures can be lifted to a fully dynamic algorithm framework for the respective problems.

References

- A. Abboud, R. Addanki, F. Grandoni, D. Panigrahi, and B. Saha. Dynamic set cover: improved algorithms and lower bounds. In *STOC*, pages 114–125. ACM, 2019.
- A. Abboud and V. V. Williams. Popular conjectures imply strong lower bounds for dynamic problems. In *FOCS*, pages 434–443. IEEE Computer Society, 2014.
- 3 K. Axiotis and C. Tzamos. Capacitated dynamic programming: Faster knapsack and graph algorithms. In *ICALP*, volume 132 of *LIPIcs*, pages 19:1–19:13. Schloss Dagstuhl Leibniz-Zentrum für Informatik, 2019.

- 4 S. Behnezhad, M. Derakhshan, M. Hajiaghayi, C. Stein, and M. Sudan. Fully dynamic maximal independent set with polylogarithmic update time. In 2019 IEEE 60th Annual Symposium on Foundations of Computer Science (FOCS), pages 382–405. IEEE, 2019.
- 5 R. Bellman. Dynamic Programming. Princeton University Press, Princeton, NJ, USA, 1957.
- 6 A. Beloglazov and R. Buyya. Energy efficient allocation of virtual machines in cloud data centers. In CCGRID, pages 577–578. IEEE Computer Society, 2010.
- 7 D. Bertsimas and J. N. Tsitsiklis. *Introduction to linear organisation*, volume 6 of *Athena scientific optimization and computation series*. Athena Scientific, 1997.
- 8 A. Bhalgat, A. Goel, and S. Khanna. Improved approximation results for stochastic knapsack problems. In SODA, pages 1647–1665. SIAM, 2011.
- 9 S. Bhattacharya, M. Henzinger, and G. F. Italiano. Design of dynamic algorithms via primal-dual method. In *ICALP* (1), volume 9134 of *Lecture Notes in Computer Science*, pages 206–218. Springer, 2015.
- S. Bhattacharya, M. Henzinger, and D. Nanongkai. Fully dynamic approximate maximum matching and minimum vertex cover in $O(\log^3 n)$ worst case update time. In SODA, pages 470–489. SIAM, 2017.
- S. Bhattacharya, M. Henzinger, and D. Nanongkai. A new deterministic algorithm for dynamic set cover. In *FOCS*, pages 406–423. IEEE Computer Society, 2019.
- S. Bhattacharya and J. Kulkarni. Deterministically maintaining a $(2 + \varepsilon)$ -approximate minimum vertex cover in $o(1/\varepsilon^2)$ amortized update time. In SODA, pages 1872–1885. SIAM, 2019.
- 13 S. Bhore, J. Cardinal, J. Iacono, and G. Koumoutsos. Dynamic geometric independent set. arXiv preprint arXiv:2007.08643, 2020.
- N. Bobroff, A. Kochut, and K. A. Beaty. Dynamic placement of virtual machines for managing SLA violations. In *Integrated Network Management*, pages 119–128. IEEE, 2007.
- 15 H. Böckenhauer, D. Komm, R. Královic, and P. Rossmanith. The online knapsack problem: Advice and randomization. *Theor. Comput. Sci.*, 527:61–72, 2014.
- N. Boria and V. T. Paschos. A survey on combinatorial optimization in dynamic environments. RAIRO - Operations Research, 45(3):241–294, 2011.
- 17 C. Büsing, A. M. C. A. Koster, and M. Kutschka. Recoverable robust knapsacks: the discrete scenario case. *Optim. Lett.*, 5(3):379–392, 2011.
- 18 T. M. Chan. Approximation schemes for 0-1 knapsack. In SOSA@SODA, volume 61 of OASICS, pages 5:1–5:12. Schloss Dagstuhl Leibniz-Zentrum für Informatik, 2018.
- 19 S. Chechik and T. Zhang. Fully dynamic maximal independent set in expected poly-log update time. In 2019 IEEE 60th Annual Symposium on Foundations of Computer Science (FOCS), pages 370–381. IEEE, 2019.
- 20 C. Chekuri and S. Khanna. A polynomial time approximation scheme for the multiple knapsack problem. SIAM J. Comput., 35(3):713–728, 2005.
- 21 S. Compton, S. Mitrović, and R. Rubinfeld. New partitioning techniques and faster algorithms for approximate interval scheduling. arXiv preprint arXiv:2012.15002, 2020.
- 22 M. Cygan, Ł. Jeż, and J. Sgall. Online knapsack revisited. Theory Comput. Syst., 58(1):153–190, 2016.
- M. Cygan, M. Mucha, K. Wegrzycki, and M. Wlodarczyk. On problems equivalent to (min, +)-convolution. *ACM Trans. Algorithms*, 15(1):14:1–14:25, 2019.
- 24 K. Daudjee, S. Kamali, and A. López-Ortiz. On the online fault-tolerant server consolidation problem. In SPAA, pages 12–21. ACM, 2014.

- W. F. de la Vega and G. S. Lueker. Bin packing can be solved within 1+epsilon in linear time. Combinatorica, 1(4):349–355, 1981.
- 26 B. C. Dean, M. X. Goemans, and J. Vondrák. Approximating the stochastic knapsack problem: The benefit of adaptivity. *Math. Oper. Res.*, 33(4):945–964, 2008.
- 27 C. Demetrescu, D. Eppstein, Z. Galil, and G. F. Italiano. *Dynamic Graph Algorithms*, page 9. Chapman & Hall/CRC, 2 edition, 2010.
- 28 Y. Disser, M. Klimm, N. Megow, and S. Stiller. Packing a knapsack of unknown capacity. SIAM J. Discret. Math., 31(3):1477–1497, 2017.
- 29 B. Feldkord, M. Feldotto, A. Gupta, G. Guruganesh, A. Kumar, S. Riechers, and D. Wajc. Fully-dynamic bin packing with little repacking. In *ICALP*, volume 107 of *LIPIcs*, pages 51:1–51:24. Schloss Dagstuhl Leibniz-Zentrum für Informatik, 2018.
- 30 M. R. Garey and D. S. Johnson. *Computers and intractability*, volume 174. freeman San Francisco, 1979.
- 31 G. Gens and E. Levner. Computational complexity of approximation algorithms for combinatorial problems. In MFCS, volume 74 of Lecture Notes in Computer Science, pages 292–300. Springer, 1979.
- 32 G. Gens and E. Levner. Fast approximation algorithms for knapsack type problems. In *Optimization Techniques*, pages 185–194. Springer, 1980.
- 33 M. Grötschel, L. Lovász, and A. Schrijver. The ellipsoid method and its consequences in combinatorial optimization. *Combinatorica*, 1(2):169–197, 1981.
- A. Gu, A. Gupta, and A. Kumar. The power of deferral: Maintaining a constant-competitive steiner tree online. SIAM J. Comput., 45(1):1–28, 2016.
- 35 A. Gupta, R. Krishnaswamy, A. Kumar, and D. Panigrahi. Online and dynamic algorithms for set cover. In *STOC*, pages 537–550. ACM, 2017.
- X. Han, Y. Kawase, and K. Makino. Randomized algorithms for removable online knapsack problems. In FAW-AAIM, volume 7924 of Lecture Notes in Computer Science, pages 60–71. Springer, 2013.
- 37 X. Han, Y. Kawase, K. Makino, and H. Guo. Online removable knapsack problem under convex function. *Theor. Comput. Sci.*, 540:62–69, 2014.
- 38 X. Han and K. Makino. Online removable knapsack with limited cuts. *Theor. Comput. Sci.*, 411(44-46):3956–3964, 2010.
- 39 M. Henzinger. The state of the art in dynamic graph algorithms. In *SOFSEM*, volume 10706 of *Lecture Notes in Computer Science*, pages 40–44. Springer, 2018.
- M. Henzinger, S. Neumann, and A. Wiese. Dynamic Approximate Maximum Independent Set of Intervals, Hypercubes and Hyperrectangles. In S. Cabello and D. Z. Chen, editors, 36th International Symposium on Computational Geometry (SoCG 2020), volume 164 of Leibniz International Proceedings in Informatics (LIPIcs), pages 51:1–51:14, Dagstuhl, Germany, 2020. Schloss Dagstuhl-Leibniz-Zentrum für Informatik.
- 41 M. R. Henzinger and V. King. Randomized fully dynamic graph algorithms with polylogarithmic time per operation. *J. ACM*, 46(4):502–516, 1999.
- 42 D. S. Hochbaum and D. B. Shmoys. Using dual approximation algorithms for scheduling problems theoretical and practical results. *J. ACM*, 34(1):144–162, 1987.
- 43 J. Holm, K. de Lichtenberg, and M. Thorup. Poly-logarithmic deterministic fully-dynamic algorithms for connectivity, minimum spanning tree, 2-edge, and biconnectivity. J. ACM, 48(4):723-760, 2001.
- 44 O. H. Ibarra and C. E. Kim. Fast approximation algorithms for the knapsack and sum of subset problems. *J. ACM*, 22(4):463–468, 1975.

- 45 M. Imase and B. M. Waxman. Dynamic steiner tree problem. SIAM J. Discret. Math., 4(3):369–384, 1991.
- 46 Z. Ivkovic and E. L. Lloyd. Fully dynamic algorithms for bin packing: Being (mostly) myopic helps. SIAM J. Comput., 28(2):574–611, 1998.
- 47 K. Iwama and S. Taketomi. Removable online knapsack problems. In *ICALP*, volume 2380 of *Lecture Notes in Computer Science*, pages 293–305. Springer, 2002.
- 48 K. Iwama and G. Zhang. Online knapsack with resource augmentation. *Inf. Process. Lett.*, 110(22):1016–1020, 2010.
- 49 K. Jansen. Parameterized approximation scheme for the multiple knapsack problem. SIAM J. Comput., 39(4):1392–1412, 2009.
- 50 K. Jansen. An EPTAS for scheduling jobs on uniform processors: Using an MILP relaxation with a constant number of integral variables. SIAM J. Discrete Math., 24(2):457–485, 2010.
- 51 K. Jansen. A fast approximation scheme for the multiple knapsack problem. In *SOFSEM*, volume 7147 of *Lecture Notes in Computer Science*, pages 313–324. Springer, 2012.
- 52 K. Jansen and K. Klein. A robust AFPTAS for online bin packing with polynomial migration. SIAM J. Discret. Math., 33(4):2062–2091, 2019.
- 53 C. Jin. An improved FPTAS for 0-1 knapsack. In *ICALP*, volume 132 of *LIPIcs*, pages 76:1–76:14. Schloss Dagstuhl Leibniz-Zentrum für Informatik, 2019.
- 54 N. Karmarkar and R. M. Karp. An efficient approximation scheme for the one-dimensional bin-packing problem. In *FOCS*, pages 312–320. IEEE Computer Society, 1982.
- 55 H. Kellerer. A polynomial time approximation scheme for the multiple knapsack problem. In RANDOM-APPROX, volume 1671 of Lecture Notes in Computer Science, pages 51–62. Springer, 1999.
- 56 H. Kellerer and U. Pferschy. Improved dynamic programming in connection with an FPTAS for the knapsack problem. J. Comb. Optim., 8(1):5–11, 2004.
- 57 A. Kulik and H. Shachnai. There is no EPTAS for two-dimensional knapsack. *Inf. Process. Lett.*, 110(16):707–710, 2010.
- 58 M. Künnemann, R. Paturi, and S. Schneider. On the fine-grained complexity of one-dimensional dynamic programming. In ICALP, volume 80 of LIPIcs, pages 21:1–21:15. Schloss Dagstuhl -Leibniz-Zentrum für Informatik, 2017.
- 59 E. L. Lawler. Fast approximation algorithms for knapsack problems. *Math. Oper. Res.*, 4(4):339–356, 1979.
- **60** Y. Li, X. Tang, and W. Cai. On dynamic bin packing for resource allocation in the cloud. In *SPAA*, pages 2–11. ACM, 2014.
- W. Ma. Improvements and generalizations of stochastic knapsack and markovian bandits approximation algorithms. *Math. Oper. Res.*, 43(3):789–812, 2018.
- 62 A. Marchetti-Spaccamela and C. Vercellis. Stochastic on-line knapsack problems. Math. Program., 68:73–104, 1995.
- N. Megow and J. Mestre. Instance-sensitive robustness guarantees for sequencing with unknown packing and covering constraints. In ITCS, pages 495–504. ACM, 2013.
- N. Megow, M. Skutella, J. Verschae, and A. Wiese. The power of recourse for online MST and TSP. SIAM J. Comput., 45(3):859–880, 2016.
- 65 M. Monemizadeh. Dynamic maximal independent set. arXiv preprint arXiv:1906.09595, 2019.
- 66 M. Mucha, K. Wegrzycki, and M. Wlodarczyk. A subquadratic approximation scheme for partition. In SODA, pages 70–88. SIAM, 2019.

18 Fully Dynamic Algorithms for Knapsack Problems with Polylogarithmic Update Time

- 67 C. H. Papadimitriou and K. Steiglitz. Combinatorial Optimization: Algorithms and Complexity. Prentice-Hall, 1982.
- 68 A. Polak, L. Rohwedder, and K. Wegrzycki. Knapsack and subset sum with small items. In ICALP, volume 198 of LIPIcs, pages 106:1–106:19. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021.
- 69 D. Rhee. Faster fully polynomial approximation schemes for knapsack problems. Master's thesis, Massachusetts Institute of Technology, 2015.
- 70 P. Sanders, N. Sivadasan, and M. Skutella. Online scheduling with bounded migration. *Math. Oper. Res.*, 34(2):481–498, 2009.
- M. Skutella and J. Verschae. Robust polynomial-time approximation schemes for parallel machine scheduling with job arrivals and departures. *Math. Oper. Res.*, 41(3):991–1021, 2016.
- 72 G. Yu. On the max-min 0-1 knapsack problem with robust optimization applications. Oper. Res., 44(2):407-415, 1996.

Appendices

A Proofs for Single Knapsack

In this section, we give the detailed analysis of our algorithm for KNAPSACK in Section 3. We consider the iteration in which the guesses $\ell_{\text{max}}, \ell_{\text{min}}, n_{\text{min}}$ and \mathcal{V}_L are correct and show that the obtained solution has a value of at least $(1 - 4\varepsilon) \cdot v(\text{OPT})$.

Let \mathcal{P}_1 be the set of solutions respecting: (i) packed items not in $H_{\frac{1}{\varepsilon}}$ have a value of at most $(1+\varepsilon)^{\ell_{\min}}$ but are not part of the n_{\min} smallest items of the value class $V_{\ell_{\min}}$, and (ii) the total value of these items lies in $[\mathcal{V}_L, (1+\varepsilon)\mathcal{V}_L]$. Denote by OPT₁ the solution of highest value in \mathcal{P}_1 .

▶ **Lemma 11.** Consider OPT_1 defined as above. Then, $v(OPT_1) \ge (1 - \varepsilon) \cdot v(OPT)$.

Proof. Let OPT* be the packing obtained from OPT by removing all items belonging to OPT₁ whose value is strictly smaller than $\varepsilon^2 (1+\varepsilon)^{\ell_{\max}}$. Since OPT₁ consists of $\frac{1}{\varepsilon}$ many items, the total value of removed items is at most $\frac{1}{\varepsilon} \cdot \varepsilon^2 (1+\varepsilon)^{\ell_{\max}} \leq \varepsilon \cdot \text{OPT}$. We show that OPT* $\in \mathcal{P}_1$.

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

▶ **Lemma 12.** Let OPT_2 be the optimal solution of the instance I on which the FPTAS is run at Step 3. Then, $v(OPT_2) \ge (1 - \varepsilon) \cdot v(OPT_1)$.

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

▶ **Lemma 13.** For the solution P_F of the algorithm, we have $v(P_F) \ge (1 - 4\varepsilon) \cdot v(OPT)$.

Proof. The solution P_{FPTAS} returned by the FPTAS in Step 3 has a value of at least $(1 - \varepsilon) \cdot v(\text{OPT}_2)$. The solution P_F is obtained from P_{FPTAS} by replacing the placeholder with the corresponding low-value items, except possibly the fractional item j. Since there are $\frac{1}{\varepsilon}$ items in OPT that are of higher value than j, namely the ones in OPT $\frac{1}{2}$, this implies

$$v(P_F) > v(P_{\text{FPTAS}}) - \varepsilon \cdot v(\text{OPT}).$$

Using Lemmas 11 and 12, we obtain:

$$v(P_F) \ge v(P_{\text{FPTAS}}) - \varepsilon \cdot v(\text{OPT})$$

$$\ge (1 - \varepsilon)^2 \cdot v(\text{OPT}_1) - \varepsilon \cdot v(\text{OPT})$$

$$\ge (1 - \varepsilon)^3 \cdot v(\text{OPT}) - \varepsilon \cdot v(\text{OPT})$$

$$> (1 - 4\varepsilon) \cdot v(\text{OPT}).$$

◀

▶ Lemma 14. The algorithm has update time $\mathcal{O}(\frac{1}{\varepsilon^9} \cdot \log n \cdot \log (n \cdot v_{\text{max}}) \cdot \log^2 v_{\text{max}} + \frac{1}{\varepsilon} \log \overline{v} \log n)$.

Proof. In the first step, guessing ℓ_{max} and ℓ_{min} , and therefore enumerating over all possible values, leads to $\mathcal{O}(\frac{1}{\varepsilon^2} \cdot \log^2 v_{\text{max}})$ many iterations. Guessing n_{min} adds an additional factor of $\frac{1}{\varepsilon}$.

In the second step, again guessing \mathcal{V}_L adds a factor to the running time, specifically $\mathcal{O}(\frac{1}{\varepsilon}\log{(n \cdot v_{\text{max}})})$. Temporarily removing the $n_{\text{min}} \leq \frac{1}{\varepsilon}$ elements from the data structure costs a total of $\mathcal{O}(\frac{1}{\varepsilon}\log{n})$, as does adding back removed items from a previous iteration. Computing the size of B can be done by querying the prefix of value just above \mathcal{V}_L in time $\mathcal{O}(\log{n})$, see Section 2.

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

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

▶ Lemma 15. The query times of our algorithm are as follows. (i) Single item queries are answered in time $\mathcal{O}(1)$. (ii) Solution value queries are answered in time $\mathcal{O}(1)$. (iii) Queries of the entire solution P are answered in time $\mathcal{O}(|P|)$.

B Few Different Knapsacks

It is not very difficult to extend the approach from Section 3 to the case of multiple but few knapsacks. While theoretically applicable for any number of knapsacks, the running time is reasonable when $m = (\frac{1}{\varepsilon} \log n)^{\mathcal{O}_{\varepsilon}(1)}$. The main difference to Section 3 comes from the fact that in order to reserve space for low-value items, a single placeholder is no longer sufficient. Instead, we utilize several smaller placeholders. Since guessing the size of low-value items for every knapsack would lead to a running time exponential in m, we instead employ a sufficiently large number of placeholder items, namely $\frac{m}{\varepsilon}$ many.

This leads to additional changes as there are more fractionally cut items, i.e., one per placeholder. To be able to charge them as before in Lemma 13, we now consider the $\frac{m}{\varepsilon^2}$ most profitable items in OPT. This in turn leads to a larger candidate set of size $\frac{m}{\varepsilon^2}$. Furthermore, since we consider multiple knapsacks, we need to utilize an EPTAS instead of an FPTAS. Besides these changes, the algorithm remains unchanged.

▶ Theorem 4. For $\varepsilon > 0$, there is a dynamic algorithm for MULTIPLE KNAPSACK that achieves an approximation factor of $(1 - \varepsilon)$ with update time $2^{f(1/\varepsilon)} \left(\frac{m}{\varepsilon} \log (nv_{\max})\right)^{\mathcal{O}(1)} + \mathcal{O}\left(\frac{1}{\varepsilon} \log \overline{v} \log n\right)$, with f quasi-linear. Item queries are answered in time $\mathcal{O}(\log \frac{m^2}{\varepsilon^6})$, solution value queries in time $\mathcal{O}(1)$, and queries of one knapsack or the entire solution in time linear in the output.

Definitions and Data Structures

Let OPT be the set of items used in an optimal solution and $\text{OPT}_{\frac{m}{\varepsilon^2}}$ the set containing the $\frac{m}{\varepsilon^2}$ most valuable items of OPT; in both cases, break all ties by picking smaller-size items. Further, denote by $V_{\ell_{\text{max}}}$ and $V_{\ell_{\text{min}}}$ the highest and lowest value (class) of an element in $\text{OPT}_{\frac{m}{\varepsilon^2}}$ respectively and by n_{min} the number of elements of $\text{OPT}_{\frac{m}{\varepsilon^2}}$ with value $(1+\varepsilon)^{\ell_{\text{min}}}$.

Let \mathcal{V}_L be the total value of the items in $OPT \setminus OPT_{\frac{m}{\varepsilon^2}}$, rounded down to a power of $(1 + \varepsilon)$. The data structures used are identical to those of Section 3.

Algorithm

- 1) Compute high-value candidates $H_{\frac{m}{\varepsilon^2}}$: Guess the three values ℓ_{\max} , ℓ_{\min} and n_{\min} . If $(1+\varepsilon)^{\ell_{\min}} \cdot m \geq \varepsilon^3 \cdot (1+\varepsilon)^{\ell_{\max}}$, then define $H_{\frac{m}{\varepsilon^2}}$ to be the set that contains the $\frac{m}{\varepsilon^2}$ smallest items of each of the value classes $V_{\ell_{\min}+1}, \ldots, V_{\ell_{\max}}$, plus the n_{\min} smallest items from $V_{\ell_{\min}}$.
 - Otherwise, we set $H_{\frac{m}{\varepsilon^2}}$ to be the union of the $\frac{m}{\varepsilon^2}$ smallest items of each of the value classes with values in $\left[\frac{\varepsilon^3}{m}\cdot(1+\varepsilon)^{\ell_{\max}},(1+\varepsilon)^{\ell_{\max}}\right]$.
- 2) Create bundles of low-value items as placeholders: Guess the value \mathcal{V}_L and consider the data structure containing all the items of value at most $(1+\varepsilon)^{\ell_{\min}}$ sorted by decreasing density. Remove from it (temporarily) the n_{\min} smallest items of value $(1+\varepsilon)^{\ell_{\min}}$. Insert them back into the data structure right before the next iteration. From the remaining items, compute the amount of fractional items necessary to reach a value of \mathcal{V}_L . That is, sum the sizes of the densest items until their total value equals \mathcal{V}_L and, if necessary, cut the last item fractionally. In the same manner, cut this range of items again fractionally to obtain bundles $B_1, B_2, \ldots, B_{\frac{m}{\varepsilon}}$ of equal value $\frac{\varepsilon}{m} \cdot \mathcal{V}_L$.
- 3) Use an EPTAS: Consider the instance I consisting of the items in $H_{\frac{m}{\varepsilon^2}}$ and the placeholder bundles $B_1, B_2, \ldots, B_{\frac{m}{\varepsilon}}$. Run the EPTAS designed by Jansen [49,51], parameterized by ε , to obtain a packing P for this instance.
- 4) **Implicit Solution:** Among all guesses, keep the feasible solution P with the highest value. Then, for any knapsack, place into the knapsack items from $H_{\frac{m}{\varepsilon^2}}$ as in P and, if B_k is placed in P on this knapsack, also place the low-value items that constitute B_k , except possibly items cut fractionally. While used candidates can be stored explicitly, low-value items are given only implicitly by saving the correct guesses and recomputing B_k on a query.

Analysis

The analysis is almost identical to that of Section 3 with only slight changes to accommodate the alterations described above. For completeness, we give the full proofs. We consider the iteration in which all guesses $(\ell_{\text{max}}, \ell_{\text{min}}, n_{\text{min}}, \mathcal{V}_L)$ are correct, and show that the obtained solution has a value of at least $(1 - 6\varepsilon) \cdot v(\text{OPT})$. To this end, we consider intermediate results to analyze the impact of each step.

Let \mathcal{P}_1 be the set of solutions respecting: (i) items not in $H_{\frac{m}{\varepsilon^2}}$ have a value of at most $(1+\varepsilon)^{\ell_{\min}}$ but are not part of the n_{\min} smallest items of the value class $V_{\ell_{\min}}$, and (ii) the total value of these items lies in $[\mathcal{V}_L, (1+\varepsilon)\mathcal{V}_L]$. Denote by OPT₁ the solution of highest value in \mathcal{P}_1 .

▶ **Lemma 16.** Consider OPT_1 defined as above. Then, $v(OPT_1) \ge (1 - \varepsilon) \cdot v(OPT)$.

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

Consider an item j in $OPT_{\frac{m}{\varepsilon^2}}$ of value $v_j \geq \frac{\varepsilon^3}{m} \cdot (1+\varepsilon)^{\ell_{\max}}$. If $v_j = (1+\varepsilon)^{\ell_{\min}}$, then $j \in H_{\frac{m}{\varepsilon^2}}$ by definition of n_{\min} and $OPT_{\frac{m}{\varepsilon^2}}$; specifically, due to the tie-breaking rules. Assume

now that $v_j > (1+\varepsilon)^{\ell_{\min}}$ and $j \notin H_{\frac{m}{\varepsilon^2}}$. Recall that $H_{\frac{m}{\varepsilon^2}}$ contains the $\frac{m}{\varepsilon^2}$ smallest items of value v_j , and $|\operatorname{OPT}_{\frac{m}{\varepsilon^2}}| = \frac{m}{\varepsilon^2}$. Thus, there exists an item of value v_j , smaller than j, which belongs to $H_{\frac{m}{\varepsilon^2}}$ but not to $\operatorname{OPT}_{\frac{m}{\varepsilon^2}}$. Exchanging j for this item contradicts the definition of Opt. Therefore, $j \in H_{\frac{m}{\varepsilon^2}}$ and Condition (i) is satisfied. Condition (ii) follows directly from the definition of \mathcal{V}_L , and therefore $\operatorname{OPT}^* \in \mathcal{P}_1$, concluding the proof.

▶ **Lemma 17.** Let OPT_2 be the optimal solution of the instance I on which the EPTAS is run in Step 3. Then, $v(OPT_2) \ge (1 - 2\varepsilon) \cdot v(OPT_1)$.

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

Denote by J_L the set of items packed by OPT_1 that are not in $H_{\frac{m}{\epsilon^2}}$, i.e., the low-value items. Since the bundles consists of the densest low-value items, it must be the case, that $\sum_{k=1}^m s(B_k) \leq s(J_L)$. Therefore, OPT_1^* is a feasible solution for I and the statement follows.

By definition of the bundles, we have $\sum_{k=1}^{m} v(B_k) = \mathcal{V}_L \geq \frac{1}{1+\varepsilon}v(J_L) \geq (1-\varepsilon)v(J_L)$. Further, since there are $\frac{m}{\varepsilon}$ bundles of equal value and at most m of them are cut fractionally and discarded, we conclude that $v(\text{OPT}_1^*) \geq (1-2\varepsilon) \cdot v(\text{OPT}_1)$.

▶ **Lemma 18.** For the solution P_F of the algorithm, we have $v(P_F) \ge (1 - 6\varepsilon) \cdot v(OPT)$.

Proof. The solution P_{EPTAS} returned by the EPTAS in Step 3 has a value of at least $(1 - \varepsilon) \cdot v(\text{OPT}_2)$. The solution P_F is obtained from P_{EPTAS} by replacing the placeholder bundles with the corresponding low-value items with the exception of fractionally cut ones, of which there are at most $\frac{m}{\varepsilon}$ many. Since there are $\frac{m}{\varepsilon^2}$ items in OPT that are of higher value than these items, namely the ones in OPT $\frac{m}{\varepsilon^2}$, this implies

$$v(P_F) \ge v(P_{\text{EPTAS}}) - \varepsilon \cdot v(\text{OPT}).$$

Using Lemmas 16 and 17, we obtain:

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

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

▶ Lemma 19. The algorithm has update time $2^{\mathcal{O}(\frac{1}{\varepsilon} \cdot \log^4(\frac{1}{\varepsilon}))} \cdot (\frac{m}{\varepsilon} \log (nv_{\max}))^{\mathcal{O}(1)} + \mathcal{O}(\frac{1}{\varepsilon} \log \overline{v} \log n)$.

Proof. In the first step, guessing ℓ_{max} and ℓ_{min} leads to $\mathcal{O}(\frac{1}{\varepsilon^2}\log^2 v_{\text{max}})$ many iterations. Guessing n_{min} adds an additional factor of $\frac{m}{\varepsilon^2}$. In the second step, guessing \mathcal{V}_L leads to $\mathcal{O}(\frac{1}{\varepsilon}\log\ (nv_{\text{max}}))$ many additional iterations, so the factor due to guessing is $\mathcal{O}(\frac{m}{\varepsilon^5}\log^2 v_{\text{max}}\log\ (nv_{\text{max}}))$

Temporarily removing the $n_{\min} \leq \frac{m}{\varepsilon^2}$ elements from the data structure costs a total of $\mathcal{O}(\frac{m}{\varepsilon^2}\log n)$, as does adding back removed items from a previous iteration. Computing

the size of the bundles can be done by querying the prefixes of value just above \mathcal{V}_L , so in time $\mathcal{O}(\log n)$. Computing the cut items of the bundles takes time $\frac{m}{\varepsilon}\log n$.

The set $H_{\frac{m}{\varepsilon^2}}$ spans value classes ranging from values of $(1+\varepsilon)^{\ell_{\max}}$ to a value at least $\frac{\varepsilon^3}{m} \cdot (1+\varepsilon)^{\ell_{\max}}$. As the value classes correspond to powers of $(1+\varepsilon)$, this means we consider at most $\log_{1+\varepsilon} \frac{m}{\varepsilon^3}$ many. Since each of them contains at most $\frac{m}{\varepsilon^2}$ items, $H_{\frac{m}{\varepsilon^2}}$ contains $\mathcal{O}(\frac{m^2}{\varepsilon^6})$ items in total. Thus, in the third step, the EPTAS, used on $\mathcal{O}(\frac{m^2}{\varepsilon^6})$ many items, runs in time $2^{\mathcal{O}(\frac{1}{\varepsilon}\cdot\log^4(\frac{1}{\varepsilon}))} + (\frac{m}{\varepsilon})^{\mathcal{O}(1)}$. Together, this gives the first term in the desired update time.

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

Queries

We show how to efficiently handle the different types of queries and state their running time.

- Single Item Query: If the queried item is contained in $H_{\frac{m}{\varepsilon^2}}$, its packing was saved explicitly. For low-value items, we save the first and last element entirely inside a bundle and on query of an item decide its membership in a bundle by comparing its density with those pivot elements.
- Solution Value Query: While the algorithm works with rounded values, we may set up the data structure of Section 2 to additionally store the actual values of items and enable prefix computation on the actual values. We can compute and store the actual solution value after an update by summing the actual values of packed candidates and determining the actual value of items in B using prefix computations while subtracting the values of discarded fractional bundles and items. On query, we return the stored solution value.
- Single Knapsack Query: Output the saved packing of all candidates packed in the knapsack. Then, in the respective density sorted data structure, iterate over items in bundles that were packed in the queried knapsack and output them. As above, this is possible since the first and last item of a bundle were saved during the update step.
- Query Entire Solution: Output saved packing of all candidates and iterate over items in packed bundles in the respective density sorted data structure as above.
- ▶ Lemma 20. The query times of our algorithm are as follows.
 - (i) Single item queries are answered in time $\mathcal{O}(\log \frac{m^2}{\epsilon^6})$.
- (ii) Solution value queries are answered in time $\mathcal{O}(1)$.
- (iii) Queries of a single knapsack P_K are answered in time $\mathcal{O}(|P_K|)$.
- (iv) Queries of the entire solution P are answered in time $\mathcal{O}(|P|)$.
- **Proof.** (i): Since the packing of candidates is stored explicitly, each of the packed candidates can be output in time $\mathcal{O}(1)$. The part of the solution corresponding to low-value items is stored implicitly, by saving the correct guesses and the first and last items of each bundle. The latter are stored in a tree sorted by density first and item index second, as in the data structure that was used to compute the bundles. Also save a pointer to and from the respective adjoining bundles of these items. This preparation is done during an update. When a low-value item is queried, use these pivot items to determine whether it is contained in packed bundles and if so in which it lies. This takes time $\mathcal{O}(\log \frac{m^2}{\varepsilon^6})$.
- (ii): The computations for this query are done during an update of the instance, with the update clearly dominating the running time. Thus, on a query, the answer can be given in constant time.

(iii): As in (i), the packing of candidates in P_K can be output in time $\mathcal{O}(1)$. For low-value items, we create, during an update, pointers from bundles to the first, i.e., densest, item contained in them. On a query, we then simply consider each bundle in P_K and iterate over the density sorted data structure used to find and output all items of the bundle.

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

C Proofs for Oblivious Linear Grouping (Section 4.1)

In this section, we give the technical details of the analysis of the oblivious linear grouping approach developed in Section 4.1. We start by analysing the approximation ratio, i.e., by formally proving Lemma 21. Recall that OPT is the optimal solution and OPT_T is the optimal solution attainable by packing item types \mathcal{T} instead of items in J' and using $J \setminus J'$ without any changes.

▶ Lemma 21. Let OPT and OPT_T be as defined above. Then, $v(OPT_{\mathcal{T}}) \geq \frac{(1-\varepsilon)(1-2\varepsilon)}{(1+\varepsilon)^2}v(OPT)$.

The loss in the objective function due to rounding item values to natural powers of $(1+\varepsilon)$ is bounded by a factor of $(1-\varepsilon)$ by Lemma 2. As already pointed out, the analysis of the approximation ratio consists of three steps. In Lemma 22 we show that the loss in the objective function value when restricting the items in J' to the value classes with $\bar{\ell} \leq \ell \leq \ell_{\text{max}}$ is bounded by a factor of $(1-\varepsilon)$. If an optimal solution contains n_{ℓ} items of V'_{ℓ} , it is feasible to pack the n_{ℓ} smallest such items. Then, Lemma 23 shows that we do not need n_{ℓ} exactly but it suffices to guess n_{ℓ} up to a factor of $(1+\varepsilon)$. Finally, in Lemma 24, we argue that using the introduced oblivious linear grouping approach costs at most a factor $(1-2\varepsilon)$. In Lemma 25, we show that the number of item types within one value class is reduced to $\mathcal{O}(\frac{\log n'}{\varepsilon^2})$. In Lemma 26, we bound the running time of the algorithm.

Let \mathcal{P}_1 be the set of solutions that (i) may use all items in J'' and (ii) uses items in J' only of the value classes V_ℓ with $\bar{\ell} \leq \ell \leq \ell_{\max}$. Let OPT_1 be an optimal solution in \mathcal{P}_1 . The following lemma bounds the value of OPT_1 in terms of OPT .

▶ **Lemma 22.** Let OPT_1 be defined as above. Then, $v(OPT_1) \ge (1 - \varepsilon)v(OPT)$.

Proof. Given ℓ_{\max} , it follows that $v(\text{OPT}) \geq (1+\varepsilon)^{\ell_{\max}}$. As n' is an upper bound on the cardinality of OPT', the items in the value classes with $\ell < \bar{\ell}$ contribute at most n' - 1 items to OPT' while the value of one item is bounded by $(1+\varepsilon)^{\bar{\ell}}$. Thus, the total value of items in $V_0, \ldots, V_{\bar{\ell}}$ contributing to OPT' is bounded by

$$n'(1+\varepsilon)^{\bar{\ell}} = n'(1+\varepsilon)^{\ell_{\max} - \left\lceil \frac{\log n'/\varepsilon}{\log (1+\varepsilon)} \right\rceil} \le \varepsilon (1+\varepsilon)^{\ell_{\max}} \le \varepsilon v(\mathrm{OPT}).$$

Let J_1 be the items in OPT' restricted to the value classes with $\bar{\ell} \leq \ell \leq \ell_{\text{max}}$. Clearly, J_1 and OPT" can be feasibly packed. Hence,

$$v(\operatorname{OPT}_1) \ge v(J_1) + v(\operatorname{OPT}'') \ge v(\operatorname{OPT}') - \varepsilon v(\operatorname{OPT}) + v(\operatorname{OPT}'') \ge (1 - \varepsilon)v(\operatorname{OPT}).$$

From now on, we only consider packings in \mathcal{P}_1 , i.e., we restrict to the value classes V_ℓ with $\bar{\ell} \leq \ell \leq \ell_{\max}$ for the items in J'. Let V_ℓ be a value class contributing to OPT_1' . As explained above, knowing $n_\ell = |V_\ell \cap \mathrm{OPT}_1'|$ would be sufficient to determine the items of V'_ℓ contributing to OPT_1 , i.e., to determine $V'_\ell \cap \mathrm{OPT}_1$. In the following lemma we show that we can additionally assume that $n_\ell = (1+\varepsilon)^{k_\ell}$ for some $k_\ell \in \mathbb{N}_0$. To this end, let \mathcal{P}_2 contain all the packings in \mathcal{P}_1 where the number of big items of each value class V_ℓ is a natural power of $(1+\varepsilon)$. Let OPT_2 be an optimal packing in \mathcal{P}_2 .

▶ Lemma 23. $v(OPT_2) \ge \frac{1}{(1+\varepsilon)}v(OPT_1)$.

Proof. Consider OPT_1 , the optimal packing in \mathcal{P}_1 . We set $OPT_1' := OPT_1 \cap J'$ and $OPT_1'' := OPT_1 \setminus J' = OPT_1 \cap J''$. We construct a feasible packing in \mathcal{P}_2 that achieves the desired value of $\frac{1}{(1+\varepsilon)}v(OPT_1)$.

Let J_2 be the subset of OPT_1' where each value class V_ℓ' is restricted to the smallest $(1 + \varepsilon)^{\lfloor \log_{1+\varepsilon} n_\ell \rfloor}$ items in V_ℓ' if $V_\ell \cap \mathrm{OPT}_1' \neq \emptyset$.

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

$$v(V_{\ell} \cap J_2) \ge (1+\varepsilon)^{\lfloor \log_{1+\varepsilon} n_{\ell} \rfloor} (1+\varepsilon)^{\ell} \ge \frac{1}{1+\varepsilon} (1+\varepsilon)^{\log_{1+\varepsilon} n_{\ell}} (1+\varepsilon)^{\ell} = \frac{1}{1+\varepsilon} v(V_{\ell} \cap \operatorname{OPT}'_1).$$

Clearly, $J_2 \cup \text{OPT}_1''$ is a feasible packing in \mathcal{P}_2 . Since $v(\text{OPT}_1') = \sum_{\ell=\bar{\ell}}^{\ell_{\text{max}}} v(V_{\ell} \cap \text{OPT}_1')$,

$$v(\mathrm{OPT}_2) \ge v(J_2) + v(\mathrm{OPT}_1'') \ge \frac{1}{1+\varepsilon}v(\mathrm{OPT}_1') + v(\mathrm{OPT}_1'') \ge \frac{1}{1+\varepsilon}v(\mathrm{OPT}_1).$$

From now on, we only consider packings in \mathcal{P}_2 . This means, we restrict the items in J' to value classes V'_{ℓ} with $\bar{\ell} \leq \ell \leq \ell_{\max}$ and assume that $n_{\ell} = (1+\varepsilon)^{k_{\ell}}$ for $n_{\ell} \in \mathbb{N}_0$ or $n_{\ell} = 0$. Even with n_{ℓ} being of the form $(1+\varepsilon)^{k_{\ell}}$, guessing the exponent for each value class V'_{ℓ} independently is intractable in time polynomial in $\log n$ and $\frac{1}{\varepsilon}$. To resolve this, the oblivious linear grouping creates groups that take into account all possible guesses of n_{ℓ} . This rounding is done for each value class individually and results in item types \mathcal{T}_{ℓ} for the set V'_{ℓ} . Let $\mathcal{P}_{\mathcal{T}}$ be the set of all feasible packings of items in \mathcal{T}_{ℓ} for $\bar{\ell} \leq \ell \leq \ell_{\max}$ and any subset of items in J''. That is, instead of the original items in J' the packings in $\mathcal{P}_{\mathcal{T}}$ pack the corresponding item types. Note that packings in $\mathcal{P}_{\mathcal{T}}$ are not forced to pack natural powers of $(1+\varepsilon)$ many items per value class. Let $\mathrm{OPT}_{\mathcal{T}}$ be the optimal solution in $\mathcal{P}_{\mathcal{T}}$. The next lemma shows that $v(\mathrm{OPT}_{\mathcal{T}})$ is at most a factor $(1-2\varepsilon)$ less than $v(\mathrm{OPT}_2)$, the optimal solution in \mathcal{P}_2 .

▶ Lemma 24. $v(OPT_T) \ge (1 - 2\varepsilon)v(OPT_2)$.

Proof. We construct a feasible packing J_3 in $\mathcal{P}_{\mathcal{T}}$ based on the optimal packing OPT_2 . Let $\mathrm{OPT}_2' := J' \cap \mathrm{OPT}_2$ and $\mathrm{OPT}_2'' := J'' \cap \mathrm{OPT}$. We let $J_3'' := \mathrm{OPT}_2''$ be the items of J'' in our new packing J_3 . These items will be packed exactly where they are packed in OPT_2 . For items in J', we consider each value class $V_\ell \cap \mathrm{OPT}_2'$ individually and carefully construct the set $J_{3,l}$, the items of V_ℓ' contributing to J_3 . Then, we show that the items in $J_{\ell,3}$ can be packed into the knapsacks where the items in $V_\ell \cap \mathrm{OPT}_2'$ are placed while ensuring that $v(J_{\ell,3}) \geq (1-2\varepsilon)v(V_\ell \cap \mathrm{OPT}_2')$.

If $V_{\ell} \cap \text{OPT}'_2 = \emptyset$, we set $J_{\ell,3} = \emptyset$. Then, both requirements are trivially satisfied.

If $|V'_{\ell} \cap \text{OPT}'_2| \leq \frac{1}{\varepsilon}$. Then, we set $J_{\ell,3} := V'_{\ell} \cap \text{OPT}'_2$. Clearly, $v(J_{\ell,3}) \geq (1-2\varepsilon)v(V_{\ell} \cap \text{OPT}'_2)$. For packing $J_{\ell,3}$, we observe that \mathcal{T}_{ℓ} actually contains the smallest $\frac{1}{\varepsilon}$ items as item types. Hence, their sizes are not affected by the rounding procedure and whenever OPT_2 packs one of these items, we can pack the same item into the same knapsack.

Let ℓ be a value class with $n_{\ell} := |V'_{\ell} \cap \operatorname{OPT}'_2| > \frac{1}{\varepsilon}$. Let $G_1(n_{\ell}), \ldots, G_{1/\varepsilon}(n_{\ell})$ be the corresponding $\frac{1}{\varepsilon}$ groups of $\lfloor \varepsilon n_{\ell} \rfloor$ or $\lceil \varepsilon n_{\ell} \rceil$ many items created by the (traditional) linear grouping for n_{ℓ} . We set $J_{\ell,3} = G_1(n_{\ell}) \cup \ldots \cup G_{1/\varepsilon-1}(n_{\ell})$. As $v(G_{1/\varepsilon}(n_{\ell})) = \lceil \varepsilon n_{\ell} \rceil (1+\varepsilon)^{\ell} \le 2\varepsilon n_{\ell}(1+\varepsilon)^{\ell} = 2\varepsilon v_{\ell,2}$, we have $v(J_{\ell,3}) \ge (1-2\varepsilon)v(V'_{\ell} \cap \operatorname{OPT}'_2)$. For packing these items, we observe that the item types created by our algorithm are a refinement of $G_1(n_{\ell}), \ldots, G_{1/\varepsilon}(n_{\ell})$. As the oblivious linear grouping ensures $|G_{1/\varepsilon}(n_{\ell})| \ge |G_{1/\varepsilon-1}(n_{\ell})| \ge \ldots \ge |G_1(n_{\ell})|$ and that

the item sizes are increasing in the group index, we can pack the items of group $G_k(n_\ell)$ where OPT₂ packs the items of group $G_{k+1}(n_\ell)$ for $1 \le k < \frac{1}{\varepsilon}$. We conclude

$$v(\mathrm{OPT}_{\mathcal{T}}) \ge v(J_3) + v(\mathrm{OPT}_2'') \ge (1 - 2\varepsilon)v(\mathrm{OPT}_2') + v(\mathrm{OPT}_2'') \ge (1 - 2\varepsilon)v(\mathrm{OPT}_2).$$

As \mathcal{T} contains at most $\frac{1}{\varepsilon} \left(\left\lceil \frac{\log n'/\varepsilon}{\log (1+\varepsilon)} \right\rceil + 1 \right)$ many different value classes and using $\left\lceil \frac{\log n'}{\log (1+\varepsilon)} \right\rceil + 1$ many different values for $n_{\ell} = |\operatorname{OPT} \cap V'_{\ell}|$ suffices as explained above, the next lemma follows.

▶ **Lemma 25.** The algorithm reduces the number of item types to $\mathcal{O}(\frac{\log^2 n'}{\varepsilon^4})$.

Next, we formally prove the bound on the running time, i.e., the following lemma.

▶ Lemma 26. For a given guess ℓ_{\max} , the set \mathcal{T} can be determined in time $\mathcal{O}(\frac{\log^4 n'}{\epsilon^4})$.

Proof. Remember that n' is an upper bound on the number of items in J' in any feasible solution. Observe that the boundaries of the linear grouping created by the algorithm per value class are actually independent of the value class and only refer to some kth item in class V_{ℓ} . Hence, the algorithm first computes the different indices needed in this round. We denote the set of these indices by $I' = \{j_1, \ldots\}$ sorted in an increasing manner. There are at most $\lfloor \log_{1+\varepsilon} n' \rfloor$ many possibilities for n_{ℓ} . Thus, the algorithm needs to compute at most $\frac{1}{\varepsilon}(\log_{1+\varepsilon} n' + 1)$ many different indices. This means that these indices can be computed and stored in time $\mathcal{O}(\frac{\log n'}{\varepsilon^2})$ while each index is bounded by n.

Given the guess ℓ_{\max} and $\bar{\ell}$, fix a value class V_{ℓ} with $\bar{\ell} \leq l \leq \ell_{\max}$. We want to bound the time the algorithm needs to transform the big items in V_{ℓ} into the modified item set T_{ℓ} . We will ensure that the dynamic algorithms in the following sections maintain a balanced binary search for each value class V_{ℓ} that stores the items in J' sorted by increasing size. Hence, the sizes of the items corresponding to J' can be accessed in time $\mathcal{O}(\frac{\log^3 n'}{\varepsilon^2})$. These sizes correspond to the item size s_t for an item type $t \in \mathcal{T}_{\ell}$. Given an item type $t \in \mathcal{T}_{\ell}$, $n_t = j_t - j_{t-1}$, which can again be pre-computed independently of the value class. Thus, \mathcal{T}_{ℓ} can be computed in time $\mathcal{O}(\frac{\log^3 n'}{\varepsilon^2})$.

As there are $\mathcal{O}(\frac{\log n'}{\varepsilon^2})$ many value classes that need to be considered for a given guess ℓ_{\max} , calculating the set $\mathcal{T}^{(\ell_{\max})}$ needs $\mathcal{O}(\frac{\log^4 n'}{\varepsilon^4})$ many computation steps.

Proof of Lemma 5. Lemmas 2 and 21 bound the approximation ratio, Lemma 25 bounds the number of item types, and Lemma 26 bounds the running time of oblivious linear grouping.

D Proofs for Identical Knapsacks (Section 4.2)

In this section, we give the technical details of Section 4.2. The first step is to analyze the loss in the objective function value due to the linear grouping. Set $J' = J_B$ and $n' = \min\{\frac{m}{\varepsilon}, n\}$. Moreover, let $OPT_{\mathcal{T}}$ be the optimal packing when using the corresponding item types \mathcal{T} obtained from applying oblivious linear rounding instead of the items in J_B . Then, the next corollary immediately follows from Lemma 5.

▶ Corollary 27. Let OPT and OPT_T be defined as above. Then, $v(OPT_T) \ge \frac{(1-\varepsilon)(1-2\varepsilon)}{(1+\varepsilon)^2}v(OPT)$.

In Figure 2 we give a feasible solution to the configuration ILP. In the next lemma, we show that there is a guess v_S with the corresponding size s_S such that $v_{\text{ILP}}^* + v_S + v_{j^*}$ for the optimal solution value v_{ILP}^* of (P) is a good guess for the optimal solution $v(\text{OPT}_{\mathcal{T}})$. Here, j^* is the densest small item not contained in P while P is the maximal prefix of small items with $v(P) < v_S$. The high-level idea of the proof is to restrict an optimal solution $\text{OPT}_{\mathcal{T}}$ to the $\lfloor (1-3\varepsilon)m \rfloor$ most valuable knapsacks and show that s_S underestimates the size of small items in these $\lfloor (1-3\varepsilon)m \rfloor$ knapsacks. Interpreting these knapsacks as configurations gives a feasible solution for the configuration ILP.

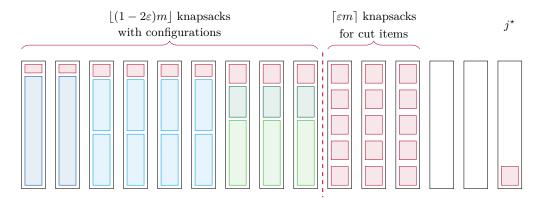


Figure 2 A possible solution: Blue and green rectangles represent the packed big item types. Red rectangles on the left side represent the space left empty by the configurations and on the right represent the slots for cut small items reserved in $\lceil \varepsilon m \rceil$ knapsacks while the densest small item, j^* , not in P gets its own knapsack.

▶ Lemma 28. There is a guess v_S with $v_{ILP} + v_S \ge \frac{1-4\varepsilon}{1+\varepsilon}v(OPT_T)$. Moreover, $v(P_S) + v_{j^*} \ge v_S$.

Proof. Let $\mathrm{OPT}_{B,\mathcal{T}} := \mathrm{OPT}_{\mathcal{T}} \cap J_B$ and $\mathrm{OPT}_{S,\mathcal{T}} := \mathrm{OPT}_{\mathcal{T}} \cap J_S$. We construct a candidate set J_{ILP} of items that are feasible for (P) and obtain a value of at least $(1-4\varepsilon)v(\mathrm{OPT}_{B,\mathcal{T}})$. To this end, take an optimal packing $\mathrm{OPT}_{\mathcal{T}}$ and consider the $\lfloor (1-3\varepsilon)m \rfloor$ most valuable knapsacks in this packing. Let $J_{B,\mathcal{T}}$ and $J_{S,\mathcal{T}}$ consist of the big and small items, respectively, in these knapsacks. Since $m \geq \frac{16}{\varepsilon^7}\log^2 n \geq \frac{1}{\varepsilon}$, we have $\lfloor (1-3\varepsilon)m \rfloor \geq (1-4\varepsilon)m$. Hence,

$$v(J_{B,\mathcal{T}}) + v(J_{S,\mathcal{T}}) \ge (1 - 4\varepsilon)v(\text{Opt}_{\mathcal{T}}).$$

Create the variable values y_c corresponding to the number of times configuration c is used by the items in $J_{B,\mathcal{T}}$. We observe that $J_{B,\mathcal{T}} \cup J_{S,\mathcal{T}}$ can be feasibly packed into $\lfloor (1-3\varepsilon)m \rfloor$ knapsacks. Therefore, $\sum_{c \in \mathcal{C}} y_c \leq \lfloor (1-3\varepsilon)m \rfloor$, and $\sum_{c \in \mathcal{C}} y_c s_c + s(J_{S,\mathcal{T}}) \leq \lfloor (1-3\varepsilon)m \rfloor S$.

Since we guess the value of the small items in the dynamic algorithm up to a factor of $(1+\varepsilon)$, there is one guess v_S satisfying $v_S \leq v(J_{S,\mathcal{T}}) < (1+\varepsilon)v_S$. Let P_S be the maximal prefix of small items with $v(P_S) < v_S$ and let j^* be the densest small item not in P_S . Hence, $v(P_S) + v_{j^*} \geq v_S \geq \frac{1}{1+\varepsilon}v(J_{S,\mathcal{T}})$. As P_S contains the densest small items, this implies $s_S := s(P_S) \leq s(J_{S,\mathcal{T}})$. Thus,

$$\sum_{c \in \mathcal{C}} y_c s_c \le \lfloor (1 - 3\varepsilon) m \rfloor S - s(J_{S,\mathcal{T}}) \le \lfloor (1 - 3\varepsilon) m \rfloor S - s_S.$$

Therefore, the just created y_c are feasible for the ILP with the guess v_s , and

$$v_{\text{ILP}} + v_S \ge v(J_{B,\mathcal{T}}) + \frac{1}{1+\varepsilon}v(J_{S,\mathcal{T}}) \ge \frac{1}{1+\varepsilon}(v(J_{B,\mathcal{T}}) + v(J_{S,\mathcal{T}})) \ge \frac{1-4\varepsilon}{1+\varepsilon}v(\text{Opt}_{\mathcal{T}}),$$

which concludes the proof.

Solving the configuration LP

In this part, we provide the full proof of our approach to solving the LP relaxation of the configuration ILP when m satisfies $\frac{16}{\varepsilon^7}\log^2 n \leq m$. More specifically, we prove the following lemma.

▶ Lemma 29. Let $U = \max\{Sm, nv_{\max}\}$. There is an algorithm that finds a feasible solution for the LP relaxation of (P) with value at least $\frac{1-\varepsilon}{1+\varepsilon}v_{LP}$ with running time $\left(\frac{\log U}{\varepsilon}\right)^{\mathcal{O}(1)}$.

In the following, we abuse notation and also refer to the LP relaxation of (P) by (P):

max
$$\sum_{c \in \mathcal{C}} y_c v_c$$
subject to
$$\sum_{c \in \mathcal{C}} y_c s_c \leq \lfloor (1 - 3\varepsilon)m \rfloor S - s_S$$

$$\sum_{c \in \mathcal{C}} y_c \leq \lfloor (1 - 3\varepsilon)m \rfloor \qquad (P)$$

$$\sum_{c \in \mathcal{C}} y_c n_{tc} \leq n_t \qquad \text{for all } t \in \mathcal{T}^{(l_{\text{max}})}$$

$$y_c \in \geq 0 \qquad \text{for all } c \in \mathcal{C}$$

Let γ and β be the dual variables of the capacity constraint and the number of knapsacks constraint, respectively. We set $\mathcal{T} := \mathcal{T}^{(\ell_{\text{max}})}$ for simplicity. Let α_t for $t \in \mathcal{T}$ be the dual variables of the constraint ensuring that only n_t items of type t are packed. Then, the dual is given by the following linear program.

$$\begin{aligned} & \min & & \lfloor (1-3\varepsilon)m \rfloor \beta + (\lfloor (1-3\varepsilon)m \rfloor S - s_S) \gamma + \sum_{t \in \mathcal{T}} n_t \alpha_t \\ & \text{subject to} & & \beta + s_c \gamma + \sum_{t \in T} \alpha_t n_{tc} \\ & & \geq & v_c & \text{for all } c \in \mathcal{C} \\ & & \alpha_t \\ & & \beta, \gamma & \geq & 0. \end{aligned}$$

As discussed above, for applying the Ellipsoid method we need to solve the separation problem efficiently. The separation problem decides if the current solution $(\alpha^*, \beta^*, \gamma^*)$ is feasible or finds a violated constraint. As verifying the first constraint of (D) corresponds to solving a KNAPSACK problem, we do not expect to optimally solve the separation problem in time polynomial in log n and $\frac{1}{\varepsilon}$. Instead, we apply a dynamic program (DP) for the single knapsack problem after restricting the item set further and rounding the item values as follows

Let $\bar{v}_t := v_t - \alpha_t^* - \gamma^* s_t$ for $t \in \mathcal{T}$. If there exists an item type with $\bar{v}_t > \beta^*$, we return the configuration using only this item. Otherwise, we define $\tilde{v}_t := \left\lfloor \frac{\bar{v}_t}{\varepsilon^4 \beta^*} \right\rfloor \cdot \varepsilon^4 \beta^*$. By running the dynamic program for the KNAPSACK problem on the item set \mathcal{T} with multiplicities $\min\{\frac{1}{\varepsilon}, n_t\}$ and values \tilde{v}_t , we obtain a solution x^* where x_t^* indicates how often item type t is packed. If $\sum_{t \in \mathcal{T}} x_t^* \tilde{v}_t > \beta^*$, we return the configuration defined by x^* as separating hyperplane. Otherwise, we return DECLARED FEASIBLE for the current solution.

The next lemma shows that this algorithm approximately solves the separation problem by either correctly declaring infeasibility or by finding a solution that is almost feasible for (D). The slight infeasibility for the dual problem translates to a slight decrease in the optimal objective function value of the primal problem. In the proof we use that x^* is optimal for the rounded values \tilde{v}_t to show that $(\alpha^*, \beta^*, \gamma^*)$ is almost feasible if $\sum_{t \in \mathcal{T}} x_t^* \tilde{v}_t \leq \beta^*$. Noticing that $\bar{v}_t \geq \tilde{v}_t$ then concludes the proof.

▶ **Lemma 30.** Given $(\alpha^*, \beta^*, \gamma^*)$, there is an algorithm with running time $\mathcal{O}\left(\frac{\log^2 n}{\varepsilon^{10}}\right)$ which either finds a configuration $c \in \mathcal{C}$ such that $\beta^* + s_c \gamma^* + \sum_{t \in \mathcal{T}} \alpha_t^* n_{tc} < v_c$ or guarantees that $\beta^* + s_c \gamma^* + \sum_{t \in \mathcal{T}} \alpha_t^* n_{tc} \ge (1 - \varepsilon) v_c$ holds for all $c \in \mathcal{C}$.

Proof. Fix a configuration c and recall that $s_c = \sum_{t \in \mathcal{T}} n_{tc} s_t$ and $v_c = \sum_{t \in \mathcal{T}} n_{tc} v_t$. Then, checking $\beta^* + s_c \gamma^* + \sum_{t \in \mathcal{T}} \alpha_t^* n_{tc} \ge v_c$ for all configurations $c \in \mathcal{C}$ is equivalent to showing $\max_{c \in \mathcal{C}} \sum_{t \in \mathcal{T}} (v_t - \alpha_t^* - \gamma^* s_t) n_{tc} \leq \beta^*$. This problem translates to solving the following ILP and comparing its objective function value to β^* .

$$cx_{c \in \mathcal{C}} \sum_{t \in \mathcal{T}} (v_t - \alpha_t^* - \gamma^* s_t) n_{tc} \leq \beta^*$$
. This problem translates to solving the following ILP d comparing its objective function value to β^* .

 $cx_{c \in \mathcal{C}} \sum_{t \in \mathcal{T}} (v_t - \alpha_t^* - \gamma^* s_t) n_{tc} \leq \beta^*$.

 $cx_{c \in \mathcal{C}} \sum_{t \in \mathcal{T}} (v_t - \alpha_t^* - \gamma^* s_t) n_{tc} \leq \beta^*$.

 $cx_{c \in \mathcal{C}} \sum_{t \in \mathcal{T}} s_t n_t \leq \beta^*$.

 $cx_{c \in \mathcal{C}} \sum_{t \in \mathcal{T}} s_t n_t \leq \beta^*$.

(S)

 $cx_{c \in \mathcal{C}} \sum_{t \in \mathcal{T}} s_t n_t \leq \beta^*$.

 $cx_{c \in \mathcal{C}} \sum_{t \in \mathcal{T}} s_t n_t \leq \beta^*$.

(S)

 $cx_{c \in \mathcal{C}} \sum_{t \in \mathcal{T}} s_t n_t \leq \beta^*$.

(S)

 $cx_{c \in \mathcal{C}} \sum_{t \in \mathcal{T}} s_t n_t \leq \beta^*$.

(S)

(S)

is ILP is itself a (single) KNAPSACK problem. Hence, the solution x^* found by the contribution is indeed for sible for (S) .

This ILP is itself a (single) KNAPSACK problem. Hence, the solution x^* found by the algorithm is indeed feasible for (S).

We start by bounding the running time of the algorithm. Recall that, for each $t \in \mathcal{T}$, $\bar{v}_t := v_t - \alpha_t^* - \gamma^* s_t$ and $\tilde{v}_t := \left| \frac{\bar{v}_t}{\varepsilon^4 \beta^*} \right| \cdot \varepsilon^4 \beta^*$. Observe that \mathcal{T} only contains big items. Hence, it suffices to consider $\min\{n_t, \frac{1}{\varepsilon}\}$ items per value class in the DP. It can be checked in time $\mathcal{O}(\frac{\log^2 n}{\varepsilon^4})$, if $\bar{v}_t \leq \beta^*$ is violated for one $t \in \mathcal{T}$. Otherwise, $\tilde{v}_t \leq \bar{v}_t$ and $\bar{v}_t - \tilde{v}_t \leq \varepsilon^4 \beta^*$ hold. Thus, the running time of the DP is bounded by $\mathcal{O}\left(\frac{|\mathcal{T}|^2}{\varepsilon^6}\right) = \mathcal{O}\left(\frac{\log^2 n}{\varepsilon^{10}}\right)$ [44]. It remains to show that the solution x^* either defines a configuration with $\beta^* + s_c \gamma^* + s_c \gamma^*$

 $\sum_{t \in \mathcal{T}} \alpha_t^* n_{tc} < v_c \text{ or ensures that } \beta^* + s_c \gamma^* + \sum_{t \in \mathcal{T}} \alpha_t^* n_{tc} \ge (1 - \varepsilon) v_c \text{ holds for all } c \in \mathcal{C}.$ If $\sum_{t \in \mathcal{T}} x_t^* \tilde{v}_t > \beta^*, \text{ it holds } \sum_{t \in \mathcal{T}} x_t^* \tilde{v}_t \ge \sum_{t \in \mathcal{T}} x_t^* \tilde{v}_t > \beta^* \text{ and, thus, } x^* \text{ defines a separating}$

Suppose now that $\sum_{t \in \mathcal{T}} x_t^* \tilde{v}_t \leq \beta^*$. We assume for the sake of contradiction that there is a configuration c', defined by packing x_t items of type t, such that $\sum_{t \in \mathcal{T}} x_t ((1-\varepsilon)v_t - \alpha_t^* - (1-\varepsilon)v_t) = 0$ $\gamma^* s_t$) > β^* . As \mathcal{T} contains only big item types, we have that $\sum_{t \in \mathcal{T}} x_t \leq \frac{1}{\varepsilon}$. This implies that there exists at least one item type t' in \mathcal{T} with $x_{t'} \geq 1$ and $(1 - \varepsilon)v_{t'} - \alpha_{t'}^* - \gamma^* s_{t'} \geq \varepsilon \beta^*$.

$$\bar{v}_t = v_t - \alpha_t^* - \gamma^* s_t \ge (1 - \varepsilon)v_t - \alpha_t^* - \gamma^* s_t$$

holds for all item types $t \in \mathcal{T}$. This implies for t' that $\bar{v}_{t'} \geq \varepsilon \beta^*$. Hence,

$$\sum_{t \in \mathcal{T}} x_t \bar{v}_t \ge \varepsilon x_{t'} \bar{v}_{t'} + \sum_{t \in \mathcal{T}} x_t \left((1 - \varepsilon) v_t - \alpha_t^* - \gamma^* s_t \right) > \varepsilon v_{t'} + \beta^* \ge (1 + \varepsilon^2) \beta^*.$$

By definition of \tilde{v} , we have $\bar{v}_t - \tilde{v}_T \leq \varepsilon^4 \beta^*$ and $\sum_{t \in \mathcal{T}} x_t (\bar{v}_t - \tilde{v}_t) \leq \varepsilon^3 \beta^*$. This implies

$$\sum_{t \in \mathcal{T}} x_t \tilde{v}_t = \sum_{t \in \mathcal{T}} x_t \tilde{v}_t + \sum_{t \in \mathcal{T}} x_t (\bar{v}_t - \tilde{v}_t) > (1 + \varepsilon^2) \beta^* - \varepsilon^3 \beta^* \ge \beta^*,$$

where the last inequality follows from $\varepsilon \leq 1$. By construction of the DP, x^* is the optimal solution for the values \tilde{v} and achieves a total value less than or equal to β^* . Hence,

$$\beta^* \ge \sum_{t \in \mathcal{T}} x_t^* \tilde{v}_t \ge \sum_{t \in \mathcal{T}} x_t \tilde{v}_t > \beta^*;$$

a contradiction.

We now present the proof of Lemma 29.

Proof of Lemma 29. As discussed above, the high-level idea is to solve (D), the dual of (P), with the Ellipsoid Method and to consider only the variables corresponding to constraints added by the Ellipsoid Method for solving (P).

As (S) is part of the separation problem for (D), there is no efficient way to exactly solve the separation problem, unless P = NP. Lemma 30 provides us a way to approximately solve the separation problem. As an approximately feasible solution for (D) cannot be directly used to determine the important variables in (P), we add an upper bound r on the objective function as a constraint to (D) and search for the largest r such that the Ellipsoid Method returns infeasible. This implies that r is an $upper\ bound$ on the objective function of (D) which in turn guarantees a $lower\ bound$ on the objective function value of (P) by weak duality.

Of course, testing all possible values for r is intractable and we restrict the possible choices for r. Observe that $v_{\rm LP} \in [v_{\rm max}, nv_{\rm max}]$ where $v_{\rm LP}$ is the optimal value of (P). Thus, for $k \in \mathbb{N}$ with $\lceil \log_{1+\varepsilon} v_{\rm max} \rceil \le k \le \lceil \log_{1+\varepsilon} (nv_{\rm max}) \rceil$, we use $r = (1+\varepsilon)^k$ as upper bound on the objective function. That is, we test if (D) extended by the objective function constraint

$$\lfloor (1 - 3\varepsilon)m \rfloor \beta + (\lfloor (1 - 3\varepsilon)m \rfloor S - s_S)\gamma + \sum_{t \in \mathcal{T}} n_t \alpha_t \le r$$

is declared feasible by the Ellipsoid Method with the approximate separation oracle for (S). We refer to the feasibility problem by (D_r) .

For a given solution $(\alpha^*, \beta^*, \gamma^*)$ of (D_r) , the separation problem asks for one of the two: either the affirmation that the point is feasible or a separating hyperplane that separates the point from any feasible solution. The non-negativity of α_t^*, β^* , and γ^* an be checked in time $\mathcal{O}(|\mathcal{T}|) = \mathcal{O}(\frac{\log^2 n}{\varepsilon^4})$. In case of a negative answer, the corresponding non-negativity constraint is a feasible separating hyperplane. Similarly, in time $\mathcal{O}(|\mathcal{T}|)$, we can check whether the objective function constraint $\lfloor (1-3\varepsilon)m\rfloor\beta + (\lfloor (1-3\varepsilon)m\rfloor S - s_S)\gamma + \sum_{t\in\mathcal{T}} n_t\alpha_t \leq r$ is violated and add it as a new inequality if necessary. In case the non-negativity and objective function constraints are not violated, the separation problem is given by the knapsack problem in (S). The algorithm in Lemma 30 either outputs a configuration that yields a valid separating hyperplane or declares $(\alpha^*, \beta^*, \gamma^*)$ feasible, i.e., $\beta^* + s_c \gamma^* + \sum_{t\in\mathcal{T}} \alpha_t^* n_{tc} \geq (1-\varepsilon)v_c$ for all $c\in\mathcal{C}$. This implies that $(\alpha^*, \beta^*, \gamma^*)$ is feasible for the following LP. Note that we changed the right side of the constraints when compared to (D).

$$\min \quad \lfloor (1 - 3\varepsilon)m \rfloor \beta + (\lfloor (1 - 3\varepsilon)m \rfloor S - s_S)\gamma + \sum_{t \in \mathcal{T}} n_t \alpha_t$$
s.t.
$$\beta + s_c \gamma + \sum_{t \in \mathcal{T}} \alpha_t n_{tc} \qquad \geq (1 - \varepsilon)v_c \quad \text{for all } c \in \mathcal{C}$$

$$\alpha_t \qquad \geq 0 \qquad \text{for all } t \in \mathcal{T}$$

$$\beta, \gamma \qquad \geq 0 \qquad \qquad (D^{(1 - \varepsilon)})$$

Let r^* be minimal such that (D_{r^*}) is declared feasible. Let $v_D^{(1-\varepsilon)}$ denote the optimal solution value of $(D^{(1-\varepsilon)})$. As $(\alpha^*, \beta^*, \gamma^*)$ is feasible with objective value at most r^* , we

have $v_D^{(1-\varepsilon)} \leq r^*$. Let $v^{(1-\varepsilon)}$ denote the optimal solution value of its dual, i.e., of the following LP.

$$\max \sum_{c \in \mathcal{C}} y_c (1 - \varepsilon) v_c$$
subject to
$$\sum_{c \in \mathcal{C}} y_c s_c \leq \lfloor (1 - 3\varepsilon) m \rfloor S - s_S$$

$$\sum_{c \in \mathcal{C}} y_c \leq \lfloor (1 - 3\varepsilon) m \rfloor \qquad (P^{(1 - \varepsilon)})$$

$$\sum_{c \in \mathcal{C}} y_c n_{tc} \leq n_t \qquad \text{for all } t \in \mathcal{T}$$

$$y_c \geq 0 \qquad \text{for all } c \in \mathcal{C}$$

Then, y = 0 is feasible for $(P^{(1-\varepsilon)})$, and by weak duality, we have

$$v^{(1-\varepsilon)} \le v_D^{(1-\varepsilon)} \le r^*.$$

Note that (P) and (P^(1-\varepsilon)) have the same feasible region and their objective functions only differ by the factor $(1-\varepsilon)$. This implies that

$$v_{\rm LP} = \frac{v^{(1-\varepsilon)}}{1-\varepsilon} \le \frac{r^*}{1-\varepsilon}.\tag{1}$$

Because of this relation between v_{LP} and r^* it suffices to find a feasible solution for (P) with objective function value close to r^* in order to prove the lemma.

To this end, let C_r be the configurations that correspond to the inequalities added by the Ellipsoid Method while solving (D_r) for $r = \frac{r^*}{1+\varepsilon}$. Consider the problems (P) and (D) restricted to the variables y_c , for $c \in C_r$, and to the constraints corresponding to $c \in C_r$, respectively, and denote these restricted LPs by (P') and (D'). Let v' and v'_D be their respective optimal values.

It holds that $v'_D > r$ as the Ellipsoid Method also returns infeasibility for (D') when run on (D') extended by the objective function constraint for r. As y=0 is feasible for (P') and $\alpha=0$, $\beta=\max_{c\in\mathcal{C}_r}v_c$, and $\gamma=0$ are feasible for (D'), their objective function values coincide by strong duality, i.e., $v'=v'_D>r$. If we have an optimal solution to (P'), then this solution is also feasible for (P) and achieves an objective function value

$$v' > \frac{r^*}{1+\varepsilon} \ge \frac{1-\varepsilon}{1+\varepsilon} v_{\rm LP},$$

where we used Equation (1) for the last inequality.

It remains to show that the Ellipsoid Method can be applied to the setting presented here and that the running time of the just described algorithm is indeed bounded by a polynomial in $\log n$, $\frac{1}{\varepsilon}$, and $\log U$. Recall that U is an upper bound on the absolute values of the denominators and numerators appearing in (D), i.e., on Sm and nv_{max} . Observe that by Lemma 30, the separation oracle runs in time $\mathcal{O}(\frac{\log^4 n}{\varepsilon^{14}})$. The number of iterations of the Ellipsoid Method will be bounded by a polynomial in $\log U$ and $\tilde{n} \in \mathcal{O}(\frac{\log^2 n}{\varepsilon^4})$. Here, \tilde{n} is an upper bound on the number of variables in the problem (D_r) (and hence also in (D^(1-\varepsilon))).

The feasible region of (D_r) is a subset of the feasible region of $(D^{(1-\varepsilon)})$, even when the objective function constraint is added to the latter LP. The Ellipsoid Method usually is applied to full-dimensional, bounded polytopes that guarantee two bounds: If the polytope is non-empty, then its volume is at least v > 0. The polytope is contained in a ball of volume

at most V. As shown in the book by Bertsimas and Tsitsiklis [7], these assumptions can always be ensured and the parameters v and V can be chosen as polynomial functions of \tilde{n} and U. Since we cannot check feasibility of (D_r) directly, we choose the parameters v and V as described in [7, Chapter 8] for the problem $(D^{(1-\varepsilon)})$ extended by the objective function constraint for r. After $N = \mathcal{O}(\tilde{n}\log\frac{V}{v})$ iterations, the modified Ellipsoid Method either finds a feasible solution to $(D^{(1-\varepsilon)})$ with objective function value at most r or correctly declares (D_r) infeasible. In [7, Chapter 8] it is shown that the number of iterations N satisfies $N = \mathcal{O}(\tilde{n}^4\log(\tilde{n}U))$ and that the overall running time is polynomially bounded in \tilde{n} and $\log U$.

Hence, (P'), the problem (P) restricted to variables corresponding to constraints added by the Ellipsoid Method, has at most N variables and, thus, a polynomial time algorithm for linear programs can be applied to (P') to obtain an optimal solution in time $\left(\frac{\log U}{\varepsilon}\right)^{\mathcal{O}(1)}$.

Integrally Packing Fractional Solutions

One of the main ingredients to the dynamic algorithms in this section is a configuration ILP. As solving general ILPs is NP-hard, in a first step, we relax the integrality constraints and accept fractional solutions before rounding the obtained solution to an integral one. The first lemma of this section describes how to obtain an integral solution with slightly more knapsacks given a fractional solution to a certain class of packing ILPs. Even after rounding, the configuration ILPs only take care of integrally packing big items, i.e., items with $s_j \geq \varepsilon S_i$. Therefore, the second lemma focuses on packing small items integrally given an integral packing of big items that reserves enough space for packing these items fractionally using resource augmentation.

Formally, we consider a packing problem of items into a given set K of knapsacks with capacities S_i . These knapsacks are grouped to obtain the set $\mathcal G$ where group $g \in \mathcal G$ contains m_g knapsacks and has total capacity S_g . The objective is to maximize the total value without violating any capacity constraint. Each item j has a certain type t, i.e., value $v_j = v_t$ and size $s_j = s_t$, and in total there are n_t items of type t. Items can either be packed as single items or as part of configurations. A configuration c, that packs $n_{c,t}$ items of type t, has total value $v_c = \sum_t n_{c,t} v_t$ and size $s_c = \sum_t n_{c,t} s_t$. The set E represents the items and the configurations that we are allowed to pack for maximizing the total value. Without loss of generality, we assume that for each element $e \in E$ there exists at least one knapsack e where this element fits, i.e, e is e in the configuration of th

Let $0 \le \delta \le 1$ and $s \ge 0$. Later we will choose $\delta = 1 - \Theta(\varepsilon)$ since intuitively an $\Theta(\varepsilon)$ -fraction of the knapsacks remains unused. Consider the packing ILP for the above described problem with variables $z_{e,g}$, where $e \in E$ and $g \in \mathcal{G}$. The ILP may additionally contain constraints of the form

$$\sum_{e \in E, g \in \mathcal{G}'} s_e z_{e,g} \leq \delta \sum_{g \in \mathcal{G}'} S_g - s \text{ and } \sum_{e \in E', g \in \mathcal{G}'} z_{e,g} \leq \delta \sum_{g \in \mathcal{G}'} m_g \,,$$

i.e., the elements assigned to a subset of knapsack types \mathcal{G}' do not violate the total capacity of a δ -fraction of the knapsacks in \mathcal{G}' while reserving a space of size s and a particular subset E' of these elements uses at most a δ -fraction of the available knapsacks.

Let v(z) be the value attained by a certain solution z and let n(z) be the number of non-zero variables of z. The following lemma shows that there is an integral solution of value at least v(z) using at most n(z) extra knapsacks. The high-level idea of the proof is to round down each non-zero variable $z_{e,g}$ and pack the corresponding elements as described by $z_{e,g}$.

For achieving enough value, we additionally place one extra element e into the knapsacks given by resource augmentation for each variable $z_{e,g}$ that was subjected to rounding.

More precisely, for each element e and each knapsack group g, we define $\bar{z}'_{e,g} = \lfloor z_{e,g} \rfloor$ and $\bar{z}''_{e,g} = \lceil z_{e,g} - \bar{z}'_{e,g} \rceil$. Note that $\bar{z}' + \bar{z}''$ may require more items of a certain type than are available. Hence, for each item type t that is now packed more than n_t times, we reduce the number of items of type t in $\bar{z}' + \bar{z}''$ by either adapting the chosen configurations if t is packed in a configuration or by decreasing the variables of type $z_{t,g}$ if items of type t are packed as single items in knapsacks of group g. Let z' and z'' denote the solutions obtained by this transformation. For some elements e, the packing described by $z'_{e,g} + z''_{e,g}$ may now use more or less elements than $z_{e,g}$ due to the just described reduction of items.

▶ **Lemma 31.** Any fractional solution z to the packing ILP described above can be rounded to an integral solution with value at least v(z) using at most n(z) additional knapsacks of capacity $\max_{i \in K} S_i$.

Proof. Consider a particular item type t. If $\bar{z}' + \bar{z}''$ packs at most n_t items of this type, then the value achieved by z for this particular item type is upper bounded by the value achieved by z' + z''. If an item type was subjected to the modification, then z' + z'' packs exactly n_t items of this type while z packs at most n_t items. This implies that $v(z' + z'') \geq v(z)$.

It remains to show how to pack $\bar{z}' + \bar{z}''$ (and, thus, z' + z'') into the knapsacks given by K and potentially n(z) additional knapsack. Clearly, \bar{z}' can be packed exactly as z was packed. If $z_{e,g} = 0$ for $e \in E$ and $g \in \mathcal{G}$, then $\bar{z}'_{e,g} = 0$. Hence, the number of non-zero entries in \bar{z}'' is bounded by n(z). Consider one element $e \in E$ and a knapsack group g with $\bar{z}''_{e,g} = 1$ and let i be a knapsack where e fits. Pack e into i.

Since reducing the number of packed items of a certain type only decreases the size of the corresponding configuration or the number of individually packed elements, the solution z' + z'' can be packed exactly as described for $\bar{z}' + \bar{z}''$. Therefore, we need at most n(z) extra knapsacks to pack z'', which concludes the proof.

Having found a feasible solution with the Ellipsoid Method, we use Gaussian elimination to obtain a basic feasible solution with no worse objective function value. We note that this procedure has a running time bounded by $(N|\mathcal{T}|)^{\mathcal{O}(1)}$, where N is the number of non-zero variables in the solution found by the Ellipsoid Method. Since basic feasible solutions have at most $|\mathcal{T}| + 2$ non-vanishing variables, the assumptions $\frac{16}{\varepsilon^7} \log^2 n \leq m$ and m < n imply $\frac{16}{\varepsilon^7} \log^2 m \leq m$. This in turn guarantees $|\mathcal{T}| + 2 \leq \lfloor \varepsilon m \rfloor$. Hence, rounding the solution as described above uses at most $\lfloor (1 - 2\varepsilon)m \rfloor$ knapsacks and achieves a value of at least v_{LP} .

▶ Corollary 32. If $\frac{16}{\varepsilon^7} \log^2 n \leq m$, any feasible solution of the LP relaxation of (P) with at most N non-zero variables can be rounded to an integral solution using at most $\lfloor (1-2\varepsilon)m \rfloor$ knapsacks with total value at least v_{LP} in time $(N|\mathcal{T}|)^{\mathcal{O}(1)}$.

Given an integral packing of big items, we explain how to pack small items, i.e., items with $s_j < \varepsilon S$, using resource augmentation. More precisely, let K be a set of knapsacks and let $J_S' \subseteq J$ be a subset of items that are small with respect to every knapsack in K. Let $J' \subset J$ be a set of items admitting an integral packing into m = |K| knapsacks that preserves a space of at least $s(J_S')$ in these m knapsacks. We develop a procedure to extend this packing to an integral packing of all items $J' \cup J_S'$ in $\lceil (1+\varepsilon)m \rceil$ knapsacks where the $\lceil \varepsilon m \rceil$ additional knapsacks can be chosen to have the smallest capacity of knapsacks in K.

We use a packing approach similar to NEXT FIT for the problem BIN PACKING. That is, consider an arbitrary order of the small items and an arbitrary order of the knapsacks

filled with big items. We open the first knapsack in this order for small items. If the next small item i still fits into the open knapsack, we place it there and decrease the remaining capacity accordingly. If it does not fit anymore, we pack this item into the next empty slot of an additional knapsacks (possibly opening a new one), close the current original knapsack, and open the next one for packing small items. We call such an item cut.

▶ **Lemma 33.** The procedure described above feasibly packs all items $J' \cup J'_S$ in $\lceil (1+\varepsilon)m \rceil$ knapsacks where the $\lceil \varepsilon m \rceil$ additional knapsacks can be chosen to have the smallest capacity of knapsacks in K.

Proof. We start by showing that all small items are packed after the last original knapsack is closed. Toward a contradiction, suppose that there is a small item j left after all original knapsacks were closed while packing small items. As a knapsack is only closed if the current small item does not fit anymore, this implies that the volume of all small items that are packed so far have a total volume at least as large as the total remaining capacity of knapsacks in K after packing J'. Since j is left unpacked after all original knapsacks have been closed, the total volume of all items in $J' \cup J'_S$ is strictly larger than the total capacity of the original knapsacks in K. This contradicts the assumption imposed on J'_B and on J'_S . Hence, all items in J'_S are packed. Therefore, the packing created by the procedure is integral and feasible.

It remains to bound the number of additional knapsacks. Observe that each item that we packed into a knapsack given by resource augmentation while an original knapsack was still available, implied the closing of the current knapsack and the opening of a new one. Hence, for each original knapsack at most one small item was placed into the additional knapsacks. Thus, at most m small items are packed into the additional knapsacks. Since by definition of small items at least $\frac{1}{\varepsilon}$ items fit into one additional knapsack, we only need $\lceil \varepsilon m \rceil$ extra knapsacks for such items.

Answering Queries

Note that, throughout the course of the dynamic algorithm, we only implicitly store solutions. In the remainder of this section, we explain how to answer the queries stated in the main part and bound the running times of the corresponding algorithms. We refer to the time frame between two updates as a round and introduce a counter τ that is increased after each update and denotes the current round. Since answers to queries have to stay consistent in a round, we cache existing query answers by additionally storing a round t(j) and a knapsack k(j) for each item in the data structure for items where t(j) stores the last round in which item j has been queried and k(j) points to the knapsack of j in round t(j). Storing t(j) is necessary since resetting the cached query answers after each update takes too much running time. If j was not selected in t(j), we store and return this with k(j) = 0.

Let \bar{y}_c , for $c \in \mathcal{C}$, be the packing for the big items in terms of the variables of the configuration ILP. During the Ellipsoid Method and the rounding of the fractional solution to an integral solution, the set $\overline{\mathcal{C}} := \{c \in \mathcal{C} : \overline{y}_c \geq 1\}$ was constructed. We assume that this set is ordered in some way and stored in a list. In the following we use the position of $c \in \overline{\mathcal{C}}$ in that list as the index of c. For assigning \bar{y}_c distinct knapsacks to $c \in \overline{C}$, we use the ordering of the configurations and map the knapsacks $\sum_{c'=1}^{c-1} \bar{y}_{c'} + 1, \ldots, \sum_{c'=1}^{c} \bar{y}_{c'}$ to c.

For small items, we store all items in a balanced binary search tree sorted by non-increasing density. For simplicity, let $P_S = \{1, \dots, j^* - 1\}$ be the set of items (sorted by non-increasing density) that translate the guess v_S into the size s_S of small items in the current solution. Item j^* is packed into its own knapsack. Any item $j \leq j^* - 1$ is either packed regularly into the empty space of a knapsack with a configuration or it is packed into a knapsack

designated for packing cut small items. Therefore, we maintain two pointers: κ^r points to the next knapsack where a small item is supposed to go if it is packed regularly and κ^c points to the knapsack where the next cut small item is packed. We initialize these values with $\kappa^r = 1$ and $\kappa^c = \lfloor (1-2\varepsilon)m \rfloor + 1$. To determine if an item is packed regularly or as cut item, we store in ρ^r the remaining capacity of κ^r initialized with $\kappa^r = S - s_1$ where s_1 is the size of the first configuration in $\overline{\mathcal{C}}$. We store in ρ^c the remaining slots of small items in knapsack κ^c and initialize this with $\rho^c = \frac{1}{\varepsilon}$.

For each type t of big items, we maintain a pointer κ_t to the knapsack where the next queried item of type t is supposed to be packed. Moreover, the counter η_t stores how many slots κ_t still has available for items of type t. These two values are initialized with the first knapsack that packs items of type t and $\eta_t = n_{c,t}$ where c is the configuration of κ_t . If no items of type t are packed, we set $\kappa_t = 0$. Let \bar{n}_t denote the number of items of type t belonging to solution \bar{y} . We will only pack the first, i.e., smallest, \bar{n}_t items of type t. Figure 3 depicts the pointers and counters after some items already have been queried.

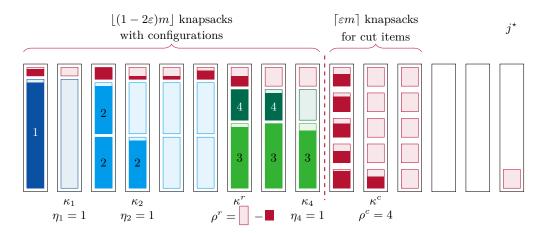


Figure 3 Pointers and counters used for answering queries: Lightly colored rectangles represent slots to be filled with items. Big (blue and green) items are packed one item per slot. Item type 3 does not have any slots left. Small (red) items are packed either until the slot is filled (left side) or one item per slot (right side). The not yet queried, small item j^* gets its own knapsack.

Consider a queried small item j. If $t(j) = \tau$, we return k(j). Otherwise, set $t(j) = \tau$ and determine whether j is currently part of the solution. If j does not belong to the densest j^* items, we return k(j) = 0. Otherwise, we determine where j is packed. If $j = j^*$, we return k(j) = m. Else, we figure out whether j is packed into the knapsack κ^r or into κ^c . If $\rho^r \geq s_j$, we simply update ρ^r to $\rho^r - s_j$ and return $k(j) = \kappa^r$. Otherwise, we decrease ρ^c by one and pack j as cut item in κ^c . If $\rho^c = 0$ holds after the update, we increase κ^c by one and set $\rho^c = \frac{1}{\varepsilon}$. Further, we need to close κ^r and update κ^r and ρ^r accordingly. To this end, we increase κ^r by one and determine ρ^r , the remaining capacity in knapsack κ^r . Then, we return k(j).

Consider a queried big item j. If $t(j) = \tau$, we return k(j). Otherwise, we set $t(j) = \tau$ and compute whether item j is packed by the current solution. Let V_{ℓ} be the value class of j. If $\ell \notin \{\bar{\ell}, \dots, \ell_{\max}\}$, we return k(j) = 0. Otherwise, we retrieve the type t of item j. Given t, we determine if j belongs to the first \bar{n}_t items of type t. If this is not the case, we return k(j) = 0. If this is the case, then we set $k(j) = \kappa_t$ instead and we decrease η_t by one. If this remains non-zero, we return $k(j) = \kappa_t$. Otherwise, we find the next knapsack that packs items of type t and update κ_t and η_t accordingly before returning k(j).

Answering Item Queries.

- 1) Check cache. Let τ be the current round and let j be the queried item. If $t(j) = \tau$, return k(j).
- 2) Answer queries for non-cached items. Set $t(j) = \tau$. If $s_j \leq \varepsilon S$, item j is small. Otherwise, j is big.

Small items. If $j > j^* + 1$, return NOT SELECTED and set k(j) = 0. If $j = j^* + 1$, return k(j) = m.

Otherwise, determine if j is packed regularly or as cut item: If $s_j \leq \rho^r$, return $k(j) = \kappa^r$ and update ρ accordingly. Otherwise, return $k(j) = \kappa^c$. Decrease ρ^c by one and if ρ^c now holds, increase κ^c by one and set $\rho^c = \frac{1}{\varepsilon}$. Increase κ^r by one and update ρ^r accordingly to reflect the empty space in κ^r .

Big items. Determine the value class V_{ℓ} of j. If $\ell < \ell_{\min}$, return NOT SELECTED. Otherwise, determine the item type t of j by retracing the steps of the oblivious linear grouping.

If j is not among the first \bar{n}_t items of type t, return NOT SELECTED and set k(j) = 0. Otherwise, return $k(j) = \kappa_t$ and decrease η_t by one. If $\eta_t = 0$, increase κ_t to the next knapsack for type t and update η_t accordingly. If no such knapsack exists, set $\kappa_t = 0$.

For being able to return the solution value in constant query time, we actually compute the solution value once after each update operation and store it. More precisely, the value achieved by the small items, v_S can be computed with a prefix computation of the first j^* items in the density-sorted tree for small items. For computing the value of big items, we consider each value class V_ℓ with $\ell \in \{\bar{\ell}, \dots, \ell_{\text{max}}\}$ individually. Per value class and per item type, we use prefix computation to determine the value v_t of the first \bar{n}_t items of type t. Lemma 37 guarantees that the running time is indeed upper bounded by the update time and, thus, does not change the order of magnitude.

Answering the Solution Value Query.

- 1) Value of small items. Calculate $v_S = \sum_{j=1}^{j^*} v_j$ with prefix computation.
- 2) Value of big items. For each item type t, calculate $v_{B,t}$ the value of the first \bar{n}_t items[3)] of type t using prefix computation.
- 1. Value. Return $v_S + \sum_{t \in \mathcal{T}} v_{B,t}$.

When queried the complete solution, we return a list of packed items together with their respective knapsacks. To this end, we start by querying the j^* densest small items using the algorithm for item queries. For big items, we query the first \bar{n}_t items of each item type $t \in \mathcal{T}$.

Answering the Solution Query

- 1) **Small items.** Query each item $j = 1, ..., j^* + 1$ and return the solution.
- 2) Big items. For each type $t \in \mathcal{T}$, query the first \bar{n}_t items and return the solution.

We prove the parts of the following lemmas individually.

▶ Lemma 7. The solution determined by the query algorithms is feasible and achieves the claimed total value. The query times of our algorithm are as follows: Single item queries can be answered in time $\mathcal{O}(\log n + \max\left\{\log\frac{\log n}{\varepsilon}, \frac{1}{\varepsilon}\right\})$, solution value queries can be answered in time $\mathcal{O}(1)$, and queries of the entire solution P can be answered in time $\mathcal{O}(|P|\frac{\log^4 n}{\varepsilon^4}\log\frac{\log n}{\varepsilon})$.

▶ Lemma 34. The solution determined by the query algorithms is feasible and achieves the claimed total value.

Proof. By construction of t(j) and k(j), the answers to queries happening between two consecutive updates are consistent.

For small items, observe that $1, \ldots, j^*$ are the densest small items in the current instance. By Lemma 33, the packing obtained by our algorithms is feasible for these items.

For big items, we observe that their actual size is at most the size of their item types. Hence, packing an item of type t where the implicit solution packs an item of type t is feasible. The algorithms correctly pack the first \bar{n}_t items of type t. A knapsack with configuration $c \in \overline{\mathcal{C}}$ correctly obtains $n_{c,t}$ items of type t. Moreover, each configuration $c \in \overline{\mathcal{C}}$ gets assigned \overline{y}_c knapsacks. Hence, the algorithm packs exactly the number of big items as dictated by the implicit solution \bar{y} .

▶ **Lemma 35.** The data structures for big items can be generated in time $\mathcal{O}(\frac{\log^4 n}{\varepsilon^9})$. Queries for big items can be answered in time $\mathcal{O}(\log n + \log \frac{\log n}{n})$.

Proof. We assume that $\overline{\mathcal{C}}$ is already stored in some list. We start by formally mapping knapsacks to configurations. To this end, we create a list $\alpha = (\alpha_c)_{c \in \overline{C}}$, where $\alpha_c = \sum_{c'=1}^{c-1} \bar{y}_{c'}$ is the first knapsack with configuration $c \in \overline{\mathcal{C}}$. Using $\alpha_c = \alpha_{c-1} + \bar{y}_{c-1}$, we can compute these values in constant time. Hence, by iterating once through $\overline{\mathcal{C}}$, list α can be generated in $\mathcal{O}(|\mathcal{C}|)$.

We start by recomputing the indices needed for the oblivious linear grouping approach. For each value class V_{ℓ} with $\ell \in {\ell, \dots, \ell_{\text{max}}}$, we access the items corresponding to the boundaries of the item types \mathcal{T}_{ℓ} in order to obtain the item types \mathcal{T}_{ℓ} . By construction, these types are already ordered by non-decreasing size s_t . By Lemma 26, these item types can be computed in time $\mathcal{O}(\frac{\log^4 n}{\epsilon^4})$ and stored in one list \mathcal{T}_ℓ per value class V_ℓ .

For maintaining and updating the pointer κ_t , we generate a list \mathcal{C}_t of all configurations $c \in$ $\overline{\mathcal{C}}$ with $n_{c,t} \geq 1$. By iterating through each $c \in \overline{\mathcal{C}}$, we can add c to the list of t if $n_{c,t} \geq 1$. We additionally store $n_{c,t}$ and α_c in the list C_t . While iterating through the configurations, we additionally compute $\bar{n}_t = \sum_{c \in \bar{\mathcal{C}}} \bar{y}_c n_{c,t}$ and store \bar{n}_t in the same list as the item types \mathcal{T}_{ℓ} . Note that, since the list of \overline{C} is ordered by index, the created lists C_t are also sorted by index. For each item type, we point κ_t to the first knapsack of the first added configuration c and set $\eta_t = n_{c,t}$. If the list of an item type remains empty, we set $\kappa_t = 0$. Since each configuration contains at most $\frac{1}{\varepsilon}$ item types, the lists \mathcal{C}_t can be generated in time $\mathcal{O}(\frac{|\mathcal{C}||\mathcal{T}|}{\varepsilon})$.

Now consider a queried big item j. In time $\mathcal{O}(\log n)$, we can decide whether j has already been queried in the current round. If not, let V_{ℓ} be the value class of j, which was computed upon arrival of j. If $\ell \notin \{\ell, \ldots, \ell_{\text{max}}\}$, then j does not belong to the current solution and no data structures need to be updated. Otherwise, the type of j is determined by accessing the item types \mathcal{T}_{ℓ} in time $\mathcal{O}(\log \frac{\log n}{\varepsilon})$. Once t is determined, \bar{n}_t can be added to the left boundary of type t in order to determine if j is packed or not. If j belongs to the current solution, pointer κ_t dictates the answer to the query.

In order to update κ_t and η_t , we extract c, the configuration of knapsack κ_t in time $\mathcal{O}(\log |\mathcal{C}|)$ by binary search over the list α . If $\kappa_t + 1 < \alpha_{c+1}$, then κ_t is increased by one and η_t set to $n_{c,t}$ in constant time. If not, then the next configuration c' containing t can be found with binary search over the list \mathcal{C}_t in time $\mathcal{O}(\log |\overline{\mathcal{C}}|)$. If no such configuration is found, we set $\kappa_t = 0$. Otherwise, we set $\kappa_t = \alpha_{c'}$ and $\eta_t = n_{c',t}$. Overall, queries for big items can be answered in time $\mathcal{O}\left(\max\left\{\log|\overline{\mathcal{C}}|,\log\frac{\log n}{\varepsilon}\right\}\right)$. Observing that $|\overline{\mathcal{C}}| \in \mathcal{O}(|\mathcal{T}|) = \mathcal{O}\left(\frac{\log^2 n}{\varepsilon^4}\right)$ completes the proof.

▶ Lemma 36. Given the data structures for big items, the data structures for small items can be generated in time $\mathcal{O}(\log \frac{\log n}{\varepsilon})$. The running time for answering queries for small items is $\mathcal{O}(\log n + \max \{\log \frac{\log n}{\varepsilon}, \frac{1}{\varepsilon}\})$.

Proof. We initialize $\kappa^r = 1$ and $\rho^r = S - s_1$ where s_1 is the total size of the configuration assigned to the first knapsack. For packing cut items, we use the pointer κ^c to the current knapsack for cut items while ρ^c stores the remaining slots of small items. We initialize these values with $\kappa^c = \lfloor (1 - 2\varepsilon)m \rfloor + 1$ and $\rho^c = \frac{1}{\varepsilon}$. These initializations can be computed in time $\mathcal{O}(\log |\overline{\mathcal{C}}|)$ (for extracting s_1).

Now consider a queried small item j. In time $\mathcal{O}(\log n)$ we can decide whether j has already been queried in the current round. In constant time, we can decide whether $j > j^*$. If $j > j^*$, the answer is NOT SELECTED. If $j = j^*$, we return m. If $j < j^*$, the algorithm only needs to decide if j is packed into κ^r or κ^c , which can be done in constant time. Finally, κ^r and κ^c as well as ρ^r and ρ^c need to be updated. While κ^c , κ^r , and ρ^c can be updated in constant time, we need to compute the configuration c and remaining capacity $S - s_c$ of knapsack κ^r when the pointer is increased. By using binary search over the list α , the configuration can be determined in time $\mathcal{O}(\log |\overline{\mathcal{C}}|)$. Once the configuration is known, ρ^r can be calculated in time $\mathcal{O}(\frac{1}{\varepsilon})$. Overall, queries for small items can be answered in time $\mathcal{O}(\log n + \max \{\log |\overline{\mathcal{C}}|, \frac{1}{\varepsilon}\})$.

Using that $|\overline{\mathcal{C}}| \in \mathcal{O}(|\mathcal{T}|) = \mathcal{O}(\frac{\log^2 n}{\varepsilon^4})$ concludes the proof.

▶ **Lemma 37.** The total solution value can be computed in $\mathcal{O}(\frac{\log^3 n}{\varepsilon^4})$. A query for the solution value can be answered in time $\mathcal{O}(1)$.

Proof. The true value \tilde{v}_S achieved by the small items can be determined by computing the prefix of the first j^* items in the density-sorted tree for small items in time $\mathcal{O}(\log n)$ by Lemma 1.

For computing the value of a big item, we consider each value class V_{ℓ} with $\ell \in \{\bar{\ell}, \dots, \ell_{\max}\}$ individually. There are at most $\mathcal{O}(\frac{\log n}{\varepsilon^2})$ many value classes by Lemma 22. For one value class, in time $\mathcal{O}(\frac{\log n}{\varepsilon^2})$, iterate through the item types t. For each item type, we can access the total value of the first \bar{n}_t items in time $\mathcal{O}(\log n)$ by Lemma 1.

As these running times are subsumed by the running time of the update operation, we actually compute the solution value once after each update operation and store the value allowing for constant running time to answer the query.

▶ **Lemma 38.** A query for the complete solution can be answered in time $\mathcal{O}(|P| \frac{\log^4 n}{\varepsilon^4} \log \frac{\log n}{\varepsilon})$, where P is the set of items in our solution.

Proof. The small items belonging to P can be accessed in time $\mathcal{O}(j^*\log n)$ by Lemma 1. By Lemma 36, their knapsacks can be determined in time $\mathcal{O}(\log n + \max\{\log \frac{\log n}{\epsilon}, \frac{1}{\epsilon}\})$.

For big items, we consider again at most $\mathcal{O}(\frac{\log n}{\varepsilon^2})$ many value classes individually. In time $\mathcal{O}(\frac{\log n}{\varepsilon^2})$, we access the boundaries of the corresponding item types. In time $\mathcal{O}(\bar{n}_t \log n)$, we can access the \bar{n}_t items of type t belonging to our solutions by Lemma 1. Lemma 35 ensures that their knapsacks can be determined in time $\mathcal{O}(\log n + \log \frac{\log n}{\varepsilon})$.

In total, this bounds the running time by $\mathcal{O}(|P| \frac{\log^4 n}{\varepsilon^4} \log \frac{\log n}{\varepsilon})$.

E Knapsacks with Resource Augmentation

In this section, we consider instances for MULTIPLE KNAPSACK with many knapsacks and arbitrary capacities. We show how to efficiently maintain a $(1 + \varepsilon)$ -approximation when

given, as resource augmentation, L additional knapsacks that have the same capacity as a largest knapsack in the input instance, where $L \in \left(\frac{\log n}{\varepsilon}\right)^{\mathcal{O}(1/\varepsilon)}$. While we may pack items into the additional knapsacks, an optimal solution is not allowed to use them. The algorithm will again solve the LP relaxation of a configuration ILP and round the obtained solution to an integral packing. However, in contrast to the problem for identical knapsacks, not every configuration fits into every knapsack and we therefore cannot just reserve a fraction of knapsacks in order to pack the rounded configurations since the knapsack capacities might not suffice. For this reason, we employ resource augmentation in the case of arbitrary knapsack capacities.

Again, we assume that item values are rounded to powers of $(1+\varepsilon)$ which results in value classes V_{ℓ} of items with value $v_i = (1+\varepsilon)^{\ell}$. We prove the following theorem.

▶ Theorem 8. For $\varepsilon > 0$, there is a dynamic algorithm for MULTIPLE KNAPSACK that, given $\left(\frac{\log n}{\varepsilon}\right)^{\Theta(1/\varepsilon)}$ additional knapsacks as resource augmentation, achieves an approximation factor of $(1+\varepsilon)$ with update time $\left(\frac{1}{\varepsilon}\log n\right)^{\mathcal{O}(1/\varepsilon)}(\log m\log S_{\max}\log v_{\max})^{\mathcal{O}(1)}$. Item queries are answered in time $\mathcal{O}(\log m + \frac{\log n}{\varepsilon^2})$, and the solution P is output in time $\mathcal{O}(|P| \frac{\log^3 n}{\varepsilon^4}(\log m + \frac{\log n}{\varepsilon^2}))$.

E.1 Algorithm

Data structures

In this section, we maintain three different types of data structures. For storing every item j together with its size s_j , its value v_j , and the index of its value class ℓ_j , we maintain one balanced binary search tree where the items are sorted by non-decreasing time of arrival. For each value class V_{ℓ} , we maintain one balanced binary tree for sorting the items with $\ell_j = \ell$ in order of non-decreasing size. We store the knapsacks sorted in non-increasing capacity in one balanced binary tree.

Algorithm

The algorithm we develop in this section is quite similar to the dynamic algorithm for MULTIPLE KNAPSACK with identical capacities. First, we use oblivious linear grouping for the current set of items to obtain item types. However, in contrast to identical knapsacks, one particular item may be big with respect to one knapsack, small with respect to another, and may not even fit in a third knapsack. Thus, we use the item types to partition the knapsacks into groups to simulate knapsacks with identical capacities; see Figure 4. Within one group, we give an explicit packing of the big items into slightly less knapsacks than belonging to the group by solving a configuration ILP. For packing small items, we would like to use a guess of the size of small items per groups and later use again NEXT FIT to pack them integrally. However, since items classify as big in one knapsack group and as small in another group, instead of guessing the size of small items per knapsack group, we incorporate their packing into the configuration ILP by reserving sufficient space for the small items in each group. More precisely, we assign items as big items via configurations or as small items by number to the various groups. The remainder of the algorithm is straight-forward: we relax the integrality constraint to find a fractional solution and use the tools developed in Lemmas 31 and 33 to obtain an integral packing. Figure 5 shows a possible solution including some knapsacks given by resource augmentation.

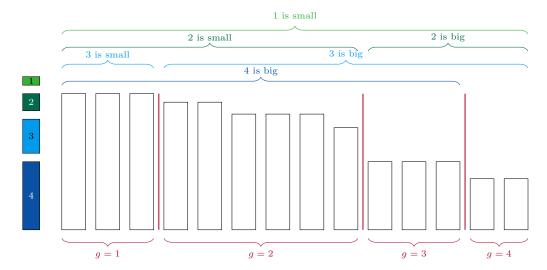


Figure 4 Item types and resulting knapsack groups

- 1) Linear grouping of big items: Guess ℓ_{max} , the index of the highest value class that belongs to OPT and use oblivious linear grouping with J' = J and n' = n to obtain \mathcal{T} , the set of item types t with their multiplicities n_t .
- 2) **Knapsack Grouping:** Consider the knapsacks sorted increasingly by their capacity and determine for each item size for which knapsacks a corresponding item would be big or small. This yields a set \mathcal{G} of $O(\frac{\log^2 n}{\varepsilon^4})$ many knapsack groups. Denote by \mathcal{F}_g the set of all item types that are small with respect to group g, and by S_g the total capacity of all knapsacks in group g. Let m_g be the number of knapsacks in group g and let $\mathcal{G}^{(1/\varepsilon)}$ be the groups in \mathcal{G} with $m_g \geq \frac{1}{\varepsilon}$. For each $g \in \mathcal{G}^{(1/\varepsilon)}$, define $S_{g,\varepsilon}$ as the total capacity of the smallest εm_g many knapsacks in g. Similar to the ILP for identical knapsacks, the ILP reserves some knapsacks to pack small 'cut' items. We distinguish between $\mathcal{G}^{(1/\varepsilon)}$ and $\mathcal{G} \setminus \mathcal{G}^{(1/\varepsilon)}$ to restrict only large enough groups g, i.e, $g \in \mathcal{G}^{(1/\varepsilon)}$, to the $(1-\varepsilon)m_g$ most valuable knapsacks of g.
- 3) Configurations: For each group $g \in \mathcal{G}$, create all possible configurations consisting of at most $\frac{1}{\varepsilon}$ items which are big with respect to knapsacks in g. This amounts to $O((\frac{\log^2 n}{\varepsilon^4})^{1/\varepsilon})$ configurations per group. Order the configurations decreasingly by size and denote the set of such configurations by $\mathcal{C}_g = \{c_{g,1}, c_{g,2} \dots c_{g,k_g}\}$. Let $m_{g,\ell}$ be the total number of knapsacks in group g in which we could possibly place configuration $c_{g,\ell}$. Further, denote by $n_{c,t}$ the number of items of type t in configuration c, and by s_c and v_c the size and value of c respectively.
- 4) Configuration ILP: Solve the following configuration ILP with variables y_c and $z_{g,t}$. Here, y_c counts how often a certain configuration c is used, and $z_{g,t}$ counts how many items of type t are packed in knapsacks of group g if type t is small with respect to g. Note that by the above definition of C_g , we may have duplicates of the same configuration

for several groups.

max
$$\sum_{g \in \mathcal{G}} \sum_{c \in \mathcal{C}_g} y_c v_c + \sum_{g \in \mathcal{G}} \sum_{t \in \mathcal{F}_g} z_{g,t} v_t$$
s.t.
$$\sum_{h=1}^{\ell} y_{c_{g,h}} \leq m_{g,\ell} \quad \text{for all } g \in \mathcal{G}, \ell \in [k_g]$$

$$\sum_{c \in \mathcal{C}_g} y_c \leq (1-\varepsilon) m_g \quad \text{for all } g \in \mathcal{G}^{(1/\varepsilon)}$$

$$\sum_{c \in \mathcal{C}_g} y_c s_{c_{g,h}} + \sum_{t \in \mathcal{F}_g} z_{g,t} s_t \leq S_g \quad \text{for all } g \in \mathcal{G} \setminus \mathcal{G}^{(1/\varepsilon)}$$

$$\sum_{c \in \mathcal{C}_g} y_c s_{c_{g,h}} + \sum_{t \in \mathcal{F}_g} z_{g,t} s_t \leq S_g - S_{g,\varepsilon} \quad \text{for all } g \in \mathcal{G}^{(1/\varepsilon)}$$

$$\sum_{g \in \mathcal{G}} \sum_{c \in \mathcal{C}_g} y_c n_{c,t} + \sum_{g \in \mathcal{G}: t \in \mathcal{F}_g} z_{g,t} \leq n_t \quad \text{for all } t \in \mathcal{T}$$

$$y_c \qquad \qquad \in \mathbb{Z}_{\geq 0} \quad \text{for all } t \in \mathcal{T}$$

$$y_c \qquad \qquad \in \mathbb{Z}_{\geq 0} \quad \text{for all } t \in \mathcal{T}, g \in \mathcal{G}$$

$$z_{g,t} \qquad \qquad \in \mathbb{Z}_{\geq 0} \quad \text{for all } t \in \mathcal{T}, g \in \mathcal{G} : t \notin \mathcal{F}_g$$
The first inequality ensures that the configurations chosen by the LP actually fit into the

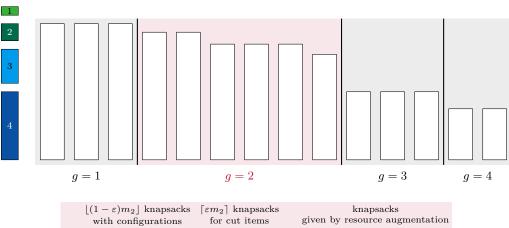
The first inequality ensures that the configurations chosen by the ILP actually fit into the knapsacks of the respective group while the second inequality ensures that an ε -fraction of knapsacks in $\mathcal{G}_{1/\varepsilon}$ remains empty for packing small 'cut' items. The third and fourth inequality guarantee that the total volume of large and small items together fits within the designated total capacity of each group. Finally, the fifth inequality makes sure that only available items are used by the ILP.

- 1. Obtaining an integral solution: After relaxing the above ILP and allowing fractional solutions, we are able to solve it efficiently. Let OPTLP be an optimal (fractional) solution to (P) with objective function value $v_{\rm LP}$. With Lemma 31 we obtain an integral solution that uses the additional knapsacks given by the resource augmentation with value at least $v_{\rm LP}$. Let P_F denote this final solution.
- 2. Packing small items: Observe that small item types $t \in \mathcal{F}_q$ are only packed fractionally by P_F . Lemma 33 provides us with a way to pack the small items integrally.

Queries

Since we do not maintain an explicit packing of any item, we define and update pointers for each item type that dictate the knapsacks where the corresponding items are packed. We note that special pointers are also used for packing items into the additional knapsacks given by resource augmentation. To stay consistent between two update operations, we cache query answers for the current round in the data structure that store items. We give the details in the next section.

- **Single Item Query:** For a queried item, we retrieve its item type and check if it belongs to the smallest items of this type that our implicit solution packs. In this case, we use the pointer for this item type to determine its knapsack.
- Solution Value Query: After having found the current solution, we use prefix computation for every value class for the corresponding item types to calculate and store the actual solution value. Then, we return this value on query.
- **Entire Solution Query:** With prefix computation on each value class, we determine the packed items. Then, the single item query is used to determine their knapsack.



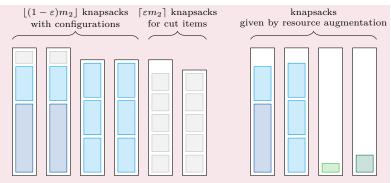


Figure 5 Possible solution of the algorithm: Group 2 accommodates the knapsacks for cut small items within the original knapsacks. Group 1, 3, and 4 use resource augmentation.

E.2 Analysis

We start again by showing that the loss in the objective function value due to the linear grouping of items is bounded by a factor of at most $\frac{(1-\varepsilon)(1-2\varepsilon)}{(1+\varepsilon)^2}$ with respect to v(OPT). To this end, let OPT be an optimal solution to the current, non-modified instance and let J be the set of items with values already rounded to powers of $(1+\varepsilon)$. Setting J'=J, we apply Lemma 5 to obtain the following corollary. Here, $\text{OPT}_{\mathcal{T}}$ is a optimal solution for the instance induced by the item types \mathcal{T} with multiplicities n_t .

▶ Corollary 39. There exists an index
$$\ell_{\max}$$
 such that $v(OPT_T) \ge \frac{(1-\varepsilon)(1-2\varepsilon)}{(1+\varepsilon)^2} v(OPT)$.

We have thus justified the restriction to item types \mathcal{T} instead of packing the actual items. In the next two lemmas, we show that (P) is a linear programming formulation of the MULTIPLE KNAPSACK problem described by the set \mathcal{T} of item types and their multiplicities and that we can obtain a feasible integral packing (using resource augmentation) if we have a fractional solution (without resource augmentation) to (P). Let v_{LP} be the optimal objective function value of the LP relaxation of (P).

Similar to the proof of Lemma 28, we restrict an optimal solution $\text{OPT}_{\mathcal{T}}$ to the $\lfloor (1-\varepsilon)m_g \rfloor$ most valuable knapsacks of a group g if $m_g \geq \frac{1}{\varepsilon}$ and otherwise we do not restrict the part of the solution corresponding to a group g with $m_g < \frac{1}{\varepsilon}$.

▶ **Lemma 40.** It holds that $v_{LP} \ge (1 - 2\varepsilon)v(O_{PT_T})$.

Proof. We show the statement by explicitly stating a solution (y, z) that is feasible for (P) and achieves an objective function value of at least $(1 - 2\varepsilon)v(\text{OPT}_{\mathcal{T}})$.

Consider a feasible optimal packing $OPT_{\mathcal{T}}$ for item types. The construction of (y, z) considers each group $g \in \mathcal{G}$ separately. We fix a group $g \notin \mathcal{G}^{(1/\varepsilon)}$. Let y_c count how often a configuration $c \in \mathcal{C}_g$ is used in $OPT_{\mathcal{T}}$ and let $z_{g,t}$ denote how often an item that is small with respect to g is packed by $OPT_{\mathcal{T}}$ in group g. By construction, the first and the third constraint of (P) are satisfied. The part of the solution (y, z) corresponding to group g achieves the same value as $OPT_{\mathcal{T}}$ restricted to this group.

If $g \in \mathcal{G}^{(1/\varepsilon)}$, i.e., if there are at least $\frac{1}{\varepsilon}$ knapsacks in group g, consider the $\lfloor (1-\varepsilon)m_g \rfloor$ most valuable knapsacks in group g with respect to $\mathrm{OPT}_{\mathcal{T}}$. Define y_c to count how often $\mathrm{OPT}_{\mathcal{T}}$ uses configuration $c \in \mathcal{C}_c$ in this reduced knapsack set and let $z_{g,t}$ denote how often $\mathrm{OPT}_{\mathcal{T}}$ uses item type $t \in \mathcal{F}_g$ in these knapsacks. Clearly, this solution satisfies the first constraint of (P). By construction, $\sum_{c \in \mathcal{C}_g} y_c \leq \lfloor (1-\varepsilon)m_g \rfloor$ and, hence, the second constraint of the ILP is also satisfied. Clearly, the $\lfloor (1-\varepsilon)m_g \rfloor$ most valuable knapsacks can be packed into the $\lfloor (1-\varepsilon)m_g \rfloor$ largest knapsacks in g, which implies the feasibility for the fourth constraint of the ILP. Observe that $\lfloor (1-\varepsilon)m_g \rfloor \geq (1-\varepsilon)m_g - 1 \geq (1-2\varepsilon)m_g$. Thus, the value of the corresponding packing is at least a $(1-2\varepsilon)$ -fraction of the value that $\mathrm{OPT}_{\mathcal{T}}$ obtains with group g.

As (y, z) uses no more items of a certain item type than $OPT_{\mathcal{T}}$ does, the last constraint of the ILP is also satisfied. Hence, (y, z) is feasible and

$$v_{\text{LP}} \ge \sum_{g \in \mathcal{G}} \left(\sum_{c \in \mathcal{C}_g} y_c v_c + \sum_{t \in \mathcal{F}_g} z_{g,t} v_t \right) \ge (1 - 2\varepsilon) v(\text{Opt}_{\mathcal{T}}),$$

with which we conclude the proof.

The next corollary shows how to round any fractional solution of (P) to an integral solution (possibly) using additional knapsacks given by resource augmentation. It follows immediately from Lemma 31 if we bound the number of variables in (P). To this end, we observe that $|\mathcal{G}|$ and $|\mathcal{T}|$ are in $\mathcal{O}(\frac{\log^2 n}{\varepsilon^4})$, and $|\mathcal{C}_g| \in (\frac{\log n}{\varepsilon})^{\mathcal{O}(1/\varepsilon)}$ for every group $g \in \mathcal{G}$. Let L' denote the exact number of variables and let $L = L' + |\mathcal{G}|$. Thus, $L \in (\frac{\log n}{\varepsilon})^{\mathcal{O}(1/\varepsilon)}$.

▶ Corollary 41. Any feasible solution (y, z) of the LP relaxation of (P) with objective value v can be rounded to an integral solution with value at least v using at most L extra knapsacks.

In the next lemma, we bound the value obtained by our algorithm in terms of v(OPT), for an optimal solution OPT. Let P_F be the solution returned by our algorithm.

▶ Lemma 42.
$$v(P_F) \ge \frac{(1-2\varepsilon)^2(1-\varepsilon)}{(1+\varepsilon)^2}v(OPT)$$
.

Proof. Fix an optimal solution OPT. Observe that our algorithm outputs the solution P_F with the maximum value over all guesses of ℓ_{\max} , the index of the highest value class in OPT. Hence, we find a guess ℓ_{\max} and a corresponding solution P that satisfies $v(P) \geq \frac{(1-2\varepsilon)^2(1-\varepsilon)}{(1+\varepsilon)^2}v(\text{OPT})$.

Let $\ell_{\max} = \max\{\ell : V_{\ell} \cap \operatorname{OPT} \neq \emptyset\}$. Then, ℓ_{\max} is considered in some round of the algorithm. Let v_{ILP} be the optimal solution value of the configuration ILP (P) and let v_{LP} be the solution value of its LP relaxation. Corollary 41 provides a way to round the corresponding LP solution (y,z) to an integral solution (\bar{y},\bar{z}) using at most L extra knapsacks with objective function value at least $v_{\operatorname{LP}} \geq v_{\operatorname{ILP}}$. The construction of (\bar{y},\bar{z}) guarantees that only small items in the original knapsacks might be packed fractionally.

Consider one particular group g. Lemma 33 shows how to pack the small items assigned by (\bar{z}_g) to group g into $\lceil (1+\varepsilon)m_g \rceil$ knapsacks. If $m_g < \frac{1}{\varepsilon}$, we use one extra knapsack per group to pack the cut items. If $m_g \geq \frac{1}{\varepsilon}$, then $g \in \mathcal{G}^{(1/\varepsilon)}$ which implies that the configuration

ILP (and its relaxation) already reserved $\lceil \varepsilon m_g \rceil$ knapsacks of this group for packing small items. Hence, the just obtained packing P is feasible. By Corollary 39 and Lemma 40,

$$v(P_F) \ge v(P) \ge \frac{(1-2\varepsilon)^2(1-\varepsilon)}{(1+\varepsilon)^2}v(\text{OPT}),$$

which gives the desired bound on the approximation ratio.

Now, we bound the running time of our algorithm.

▶ Lemma 43. In time $\left(\frac{1}{\varepsilon}\log n\right)^{\mathcal{O}(1/\varepsilon)} (\log m\log S_{\max}\log v_{\max})^{\mathcal{O}(1)}$, the dynamic algorithm executes one update operation.

Proof. By assumption, upon arrival, the value of each item is rounded to natural powers of $(1+\varepsilon)$. The algorithm starts with guessing ℓ_{\max} , the largest index of a value class to be considered in the current iteration. There are $\log v_{\max}$ many guesses possible, where v_{\max} is the highest value appearing in the current instance.

By Lemma 26, the oblivious linear grouping of all items has at most $\mathcal{O}(\frac{\log^4 n}{\varepsilon^4})$ iterations. Let the knapsacks be sorted by increasing capacity and stored in a binary balanced search tree as defined in Lemma 1. Then, the index of the smallest knapsack i with $S_i \geq S$ or the largest knapsack with $S_i \leq S$ can be determined in time $\mathcal{O}(\log m)$, where S is a given number. Thus, the knapsack groups depending on the item types can be determined in time $\mathcal{O}(\log m \frac{\log^2 n}{\varepsilon^4})$ as the number of item types is bounded by $\mathcal{O}(\frac{\log^2 n}{\varepsilon^4})$. The number of big items per knapsack is bounded by $\frac{1}{\varepsilon}$ and, hence, the number of configurations is bounded by $\mathcal{O}(\frac{\log^2 n}{\varepsilon^4}(\frac{\log^2 n}{\varepsilon^4})^{1/\varepsilon})$.

Let N be the number of variables in the configuration ILP. We have $N \in \left(\frac{\log n}{\varepsilon}\right)^{\mathcal{O}(1/\varepsilon)}$. Hence, there is a polynomial function $g(N, \log S_{\max}, \log v_{\max})$ that bounds the running time of finding an optimal solution to the LP relaxation of the configuration ILP [7,67]. Clearly, the computational complexity of setting up and rounding the fractional solution is dominated by solving the LP. Thus, $\left(\frac{1}{\varepsilon}\log n\right)^{\mathcal{O}(1/\varepsilon)}(\log m\log S_{\max}\log v_{\max})^{\mathcal{O}(1)}$ bounds the running time.

In similar time, we can store y and z, the obtained solutions to the configuration LP. Let \bar{y} and \bar{z} be the variables obtained by (possibly) rounding down y and z and let \tilde{y} and \tilde{z} be the variables corresponding to the resource augmentation as in Lemma 31. The time needed to obtain these variables is dominated by solving the LP relaxation of the configuration ILP.

Answering Queries

Since we only store implicit solutions, it remains to show how to answer the corresponding queries. In order to determine the relevant parameters of a particular item, we assume that all items are stored in one balanced binary search tree that allows us to access one item in time $\mathcal{O}(\log n)$ by Lemma 1. We additionally assume that this balanced binary search tree also stores the value class of an item. We use again the round parameter t(j) and the corresponding knapsack k(j) to cache given answers in order to stay consistent between two updates. If j was NOT SELECTED in round t(j), we represent this by k(j) = 0. We assume that these two parameters are stored in the same binary search tree that also stores the items and, thus, can be accessed in time $\mathcal{O}(\log n)$.

We now design an algorithm for non-cached items. The high-level idea is similar to the algorithm developed in Section 4 for identical knapsacks. As the knapsacks have different capacities in this section, the relative size of an item depends on the particular knapsack group:

Counter/Pointer	Meaning
$\overline{\overline{\mathcal{C}}_g}$	Configurations that are used by group g
$\alpha_{c,g}$	First knapsack with configuration c in group g
$R_{c,g}^{(y)}$	Knapsack in $\mathbb{R}^{(y)}$ used for group g and configuration c
$R_{g,t}^{(z)} \\ R_g^{(\varepsilon)}$	Knapsack in $R^{(z)}$ used for group g and type t
$R_g^{(arepsilon)}$	Knapsack in $R^{(\varepsilon)}$ used for group g with $m_g < \frac{1}{\varepsilon}$
\mathcal{G}_t	Knapsack groups where items of type t are packed
$\mathcal{C}_{g,t}$	List of configurations $c \in \overline{\mathcal{C}}_g$ with $n_{c,t} \geq 1$
γ_t	Current knapsack group where items of type t are packed
κ_t	Current knapsack for packing items of a big type t
η_t^S	Remaining number of slots for items of type t in γ_t
η_t^B	Remaining number of slots for items of type t in κ_t
κ_g^r	Current knapsack in g for packing small items regularly
κ_g^c	Current knapsack in g (or in $R^{(\varepsilon)}$) for packing cut small items
ρ_g^r	Remaining capacity in κ_g^r for packing small items
ρ_g^c	Remaining number of slots for small items in κ_g^c

Table 1 Counters and pointers used during querying items

An item can be big with respect to one knapsack and small with respect to another. Thus, the distinction between small and big items does not hold for all knapsacks simultaneously anymore and needs to be handled carefully. More precisely, upon query of an item j of type t, we start by determining the group γ_t in which the next item of type t is packed. The pointers and counters we use correspond mostly to the ones in Section 4 except that we additionally have a dependency on the particular group g for each parameter. Additionally, we use $R_q^{(z)}$, $R_q^{(y)}$ and $R_q^{(z)}$ to refer to knapsacks given by resource augmentation for group g.

If t is small with respect to γ_t , then j is packed by NEXT FIT either as regular or as cut item. We use the two pointers κ_g^r for packing small items regularly in group g and κ_g^c for packing cut items. If there are at most $\frac{1}{\varepsilon}-1$ knapsacks in group g, then κ_g^c points to the knapsack $R_g^{(\varepsilon)}$ given by resource augmentation. Otherwise, the configuration ILP left the smallest $\lceil \varepsilon m_g \rceil$ knapsacks in group g empty for packing cut small items. Further, we use $R_{g,t}^{(z)}$ to refer to the knapsack given by resource augmentation that is used for packing one item of type t if the variable $z_{g,t}$ was subjected to rounding. Since we may only pack as many items of type t in group t0 as indicated by the implicit solution, the counter t1 determines how many items of type t1 can still be packed in group t2 if t3 is small with respect to t3.

If t is big with respect to γ_t , then j is packed in the next slot for items of type t determined by the configuration ILP. To this end, we use again the counter κ_t to determine the knapsack where the next item of type t is packed and the counter η_t^B to determine how many items of type t can still be packed in knapsack κ_t if t is big with respect to γ_t . The knapsack $R_{c,g}^{(y)}$, for $c \in \mathcal{C}_g$, refers to the knapsack given by resource augmentation used when the variable $y_{c,g}$ was subjected to rounding.

Table 1 summarizes the parameters and counters used to answer queries, and in Figure 6, we give an example of the current packing after some items have been queried. Next, we define the data structures for answering queries before we formally explain how to answer queries.

Data structures We assume that the knapsacks are sorted by non-increasing capacity and stored in one binary search tree together with S_i , the capacity of the knapsacks. The

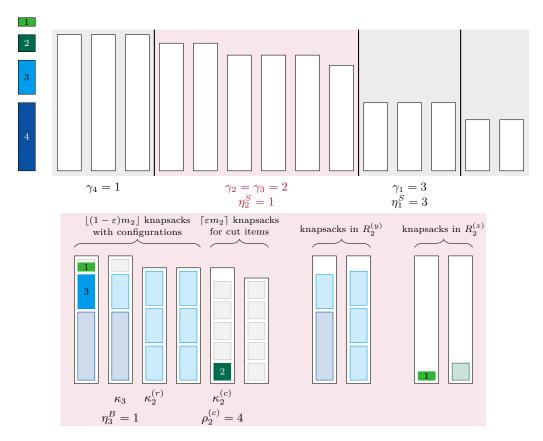


Figure 6 Counters and pointers for answering queries: Gray rectangles inside knapsacks represent small items. The next item of type 2 (dark green) is placed in the knapsack given by resource augmentation $R_2^{(z)}$ since $\eta_2^S = \tilde{z}_{2,2}$. Items of type 1 (light green) already filled all their slots in group 2 and are now placed in group 3.

knapsacks given by resource augmentation are stored in three different lists, $R^{(y)}$, $R^{(z)}$, and $R^{(\varepsilon)}$, needed due to rounding y or z or because $m_q < \frac{1}{\varepsilon}$, respectively. The knapsack groups are stored in the list \mathcal{G} sorted by non-increasing knapsack capacity. For each group g, we additionally store the number m_g of knapsacks belonging to g.

Let $\bar{y}, \tilde{y}, \bar{z}$, and \tilde{z} be the implicit solution of the algorithm. Here $\bar{*}$ refers to packing configurations or items into the original knapsacks while $\tilde{*}$ refers to the knapsacks given by resource augmentation. Let $\overline{\mathcal{C}}_q$ be the set of configurations c with $\overline{y}_{c,q} + \widetilde{y}_{c,q} \geq 1$ ordered in non-increasing size s_c and stored in one list per group. In the following, we use the position of a configuration $c \in \overline{\mathcal{C}}_g$ in that list as the index of c. For mapping the configurations to knapsacks, we assign the knapsacks $\sum_{g'=1}^{g-1} m_{g'} + \sum_{c'=1}^{c-1} \bar{y}_{c',g} + 1, \dots, \sum_{g'=1}^{g-1} m_{g'} + \sum_{c'=1}^{c} \bar{y}_{c',g}$ to configuration c. For the knapsacks in the resource augmentation, we set $R_{c,g}^{(y)} =$ $\sum_{g'=1}^{g-1} \sum_{c' \in \overline{\mathcal{C}}_{g'}} \tilde{y}_{c',g'} + \sum_{c' \leq c} \tilde{y}_{c',g} \text{ for each group } g \text{ and each configuration } c \in \overline{\mathcal{C}}_g.$

For each item type t, let \bar{n}_t denote the number of items of type t in the solution. We maintain a pointer γ_t to the group where the next queried item of type t is supposed to go. We initialize γ_t with the first group that packs items of type t. Since the number of items of type t assigned to group g as small items is determined by $\bar{z}_{g,t} + \tilde{z}_{g,t}$, we additionally use the counter η_t^S , initialized with $\bar{z}_{\gamma_t,t} + \tilde{z}_{\gamma_t,t}$, to reflect how many slots group γ_t still has for items of type t. For accessing the knapsacks $R^{(z)}$ given by resource augmentation, we set $R_{g,t}^{(z)} = \sum_{g'=1}^{g-1} \sum_{t' \in \mathcal{T}} \tilde{z}_{g',t'} + \sum_{t'=1}^{t} \tilde{z}_{g,t'}$ for each group g and item type t. Note that $z_{g,t} = 0$ holds if t is big with respect to g.

When packing small items in group g, we use group pointers κ_g^r and κ_g^c to refer to the knapsack for packing items regularly or for packing cut items. The pointer κ_g^r is initialized with $\kappa_g^r = \sum_{g'=1}^{g-1} m_{g'} + 1$. Further, we use ρ_g^r to store the remaining capacity for small items in κ_g^r and initialize it with $\rho_g^r = S_{\kappa_g^r} - s_1$, where s_1 is the size of the first configuration in group g. If $m_g \geq \frac{1}{\varepsilon}$, we set $\kappa_g^c = \sum_{g'=1}^{g-1} m_{g'} + \lfloor (1-\varepsilon)m_g \rfloor + 1$, while $m_g < \frac{1}{\varepsilon}$ implies that κ_g^c points to the knapsack $R_g^{(\varepsilon)}$ given by resource augmentation. The counter ρ_g^c stores again the remaining slots for cut small items in group g and is initialized with $\frac{1}{\varepsilon}$.

If t is big with respect to γ_t , we use the pointer κ_t to direct us to the particular knapsack where the next item of type t goes, while η_t^B stores how many slots κ_t still has available for items of type t. Initially, κ_t points to the first knapsack with a configuration that contains t in the first group where t is packed as big item. If c is the corresponding configuration, we set $\eta_t^B = n_{c,t}$. Because of resource augmentation, κ_t may point to a knapsack in $R^{(y)}$, the additional knapsacks for rounding y.

Answering Item Queries.

- 1) Check cache. Let τ be the current round and let j be the queried item. If $t(j) = \tau$, return k(j).
- 2) Answer queries for non-cached items. Set $t(j) = \tau$ and determine t, the type of j. Let γ be the group of t. If $\gamma = 0$, return NOT SELECTED. Decide if j is small or big with respect to the group γ .

Small items. If $\eta_t^S = z_{\gamma,t}''$, determine if j goes to the resource augmentation $R^{(z)}$: If $z_{\gamma,t}'' = 1$, set k(j) to the knapsack in $R^{(z)}$ reserved for $z_{\gamma,t}''$ and increase γ_t to the next group for type t. If no such group exists, set $\gamma_t = 0$. Otherwise, update η_t and possibly κ_t accordingly.

If $z_{\gamma,t}''=0$, increase γ_t to the next group for type t and go to Step 2. If no such group exists, set $\gamma_t=0$, k(j)=0, and return NOT SELECTED.

Otherwise, determine if j is packed regularly or as a cut item. If $s_t \leq \rho_{\gamma}^r$, return $k(j) = \kappa_{\gamma}^r$ and decrease ρ_{γ}^r accordingly. Otherwise, return $k(j) = \kappa_{\gamma}^c$ and decrease ρ_{γ}^c by one. If now $\rho_{\gamma}^c = 0$, increase κ_{γ}^c by one and set $\rho_{\gamma}^c = \frac{1}{\varepsilon}$.

Big items. If $\gamma_t^B = 0$, return NOT SELECTED and set k(j) = 0. Otherwise, return $k(j) = \kappa_t$ and decrease η_t by one. If this implies $\eta_t = 0$, let c be the configuration of κ_t .

If $\kappa_t \in R^{(y)}$, let c' be the next configuration for type t in group γ and update κ_t and η_t^B accordingly. If no such configuration exists, increase γ_t to the next group for type t and update κ_t and η_t^B accordingly. If no such group exists, set $\gamma_t = 0$.

If κ_t belongs to the original knapsacks and is the last knapsack assigned to configuration c, check if there is resource augmentation for configuration c. In this case, point κ_t to the knapsack reserved for rounding $y''_{c,\gamma}$. Otherwise, let c' be the next configuration for type t in group γ and update κ_t and η^B_t accordingly. If no such configuration exists, increase γ_t to the next group for type t and update κ_t and η^B_t accordingly. If no such group exists, set $\gamma_t = 0$.

Otherwise, increase κ_t by one and update η_t^B accordingly.

For calculating the value of the current solution, we need to calculate the total value of the first \bar{n}_t items. We do this by iterating through the value classes once and per value class, we iterate once through the list \mathcal{T}_{ℓ} of item types for value class V_{ℓ} to access the number \bar{n}_t . Then, we use prefix computation twice in order to access the total value of the first \bar{n}_t items

of type t. Again, we do this computation once after each update operation. Lemma 47 bounds the running time of these calculations and shows that incorporating these does not change the order of magnitude of the running time given in Lemma 43.

Answering the Solution Value Query.

- 1) Value per item type. For each item type t, calculate v_t , the total value of the first \bar{n}_t items with prefix computation.
- 2) Value. Return $\sum_{t \in \mathcal{T}} v_t$.

For returning the complete solution, we iterate once through the value classes and for each value class, we iterate through the list \mathcal{T}_{ℓ} to access the number \bar{n}_{t} . Then, we use prefix computation based on the indices of the items for accessing the first \bar{n}_t items of type t. Then, we access and query each item individually.

Answering the Solution Query.

- 1) For each item type t, query the first \bar{n}_t items and return these items with their knapsacks. We prove the parts of the next lemma again separately.
- ▶ **Lemma 44.** The solution determined by the query algorithm is feasible as well as consistent and achieves the claimed total value. The query times of our algorithm are as follows.
- (i) Single item queries can be answered in time $\mathcal{O}(\log m + \frac{\log n}{\varepsilon^2})$.
- (ii) Solution value queries can be answered in time $\mathcal{O}(1)$.
- (iii) Queries of the entire solution P are answered in time $\mathcal{O}(|P| \frac{\log^3 n}{\varepsilon^4} (\log m + \frac{\log n}{\varepsilon^2}))$.
 - ▶ Lemma 45. The query algorithms return a feasible and consistent solution obtaining the total value given by the implicit solution.

Proof. By construction of k(j) and t(j), the solution returned by the query algorithms is consistent between updates.

Observe that \bar{y} and \bar{z} is a feasible solution to the configuration ILP (P). Hence, showing that the algorithm does not assign more than $\bar{y}_{c,q}$ times configuration c and not more than $\bar{z}_{q,t}$ items of type t to group g is sufficient for having a feasible packing of the corresponding elements into the $\lfloor (1-\varepsilon)m_g \rfloor$ largest knapsacks of group g if $m_g \geq \frac{1}{\varepsilon}$ or into the m_g knapsacks of group g if $m_q < \frac{1}{\varepsilon}$. When defining L, we made sure that the items and configurations specified by \tilde{y} and \tilde{z} fit into the knapsacks given by resource augmentation.

If the item type t is small with respect to the group g, then at most $\bar{z}_{g,t}$ items of type t are packed in group g. Thus, Lemma 33 ensures that all small items assigned to group gfit in the knapsacks for regular and the cut items. Moreover, the treatment of $\eta_t^S = \tilde{z}_{q,t}$ guarantees that the value obtained by small items packed in q and its additional knapsacks is as in the implicit solution.

If t is big with respect to group g, then the constructions of κ_t and η_t^B ensure that exactly $\sum_{c \in \overline{C}_g} (\bar{y}_c + \tilde{y}_c) n_{c,t}$ items of type t are packed in group g and in $R_g^{(y)}$. Hence, the total value achieved is as given by the implicit solution.

▶ **Lemma 46.** The data structures can be generated in $\mathcal{O}\left(\frac{\log^4 n}{\varepsilon^8}\left(\log m + \frac{\log^{2/\varepsilon} n}{\varepsilon^{4/\varepsilon}}\right)\right)$ many iterations. Queries for a particular item can be answered in $\mathcal{O}(\log m + \frac{\log n}{\epsilon^2})$ many steps.

Proof. We start by retracing the steps of the oblivious linear grouping in order to obtain the set \mathcal{T} of item types. We store the types \mathcal{T}_{ℓ} of one value class in one list, sorted by non-decreasing size. By Lemma 26, the set \mathcal{T} can be determined in time $\mathcal{O}(\frac{\log^4 n}{\varepsilon^4})$.

We first argue about the generation of the data structures and the initialization of the various pointers and counters. We start by generating a list $(\alpha_{c,g})_{c\in\overline{\mathcal{C}}_g}$ for each group g where $\alpha_{c,g}$ stores the first (original) knapsack of configuration $c\in\overline{\mathcal{C}}_g$, i.e.,

$$\alpha_{c,g} = \alpha_{c-1,g} + \bar{y}_{c-1,g} + 1,$$

where $\alpha_{0,g} = \sum_{g'=1}^{g-1} m_{g'}$ and $y_{0,g} = 0$. Next, we set

$$R_{g,t}^{(z)} = \sum_{g'=1}^{g-1} \sum_{t' \in \mathcal{T}} \tilde{z}_{g',t'} + \sum_{t'=1}^{t} \tilde{z}_{g,t'}$$

and

$$R_{c,g}^{(y)} = \sum_{g'=1}^{g-1} \sum_{c' \in \overline{\mathcal{C}}_{g'}} \tilde{y}_{c',g'} + \sum_{c' \in \mathcal{C}_g, c' \leq c} \tilde{y}_{c',g} ,$$

where $R_{g,t}^{(z)}$ corresponds to the resource augmentation needed because of rounding $z_{g,t}$ and $R_{c,g}^{(y)}$ corresponds to the resource augmentation for rounding $y_{c,g}$. These lists can be generated by iterating through the list $\overline{\mathcal{C}}_g$ for each group g in time $\mathcal{O}(\sum_{g \in \mathcal{G}} |\mathcal{C}_g|) = \mathcal{O}(\frac{\log^2 n}{\varepsilon^4} \frac{\log^{2/\varepsilon} n}{\varepsilon^{4/\varepsilon}})$.

For maintaining and updating the pointer γ_t , we generate the list \mathcal{G}_t that contains all groups g where items of type t are packed in the implicit solution. By iterating through the groups once more and checking $\sum_{c \in \overline{\mathcal{C}}_g} (\bar{y}_{c,g} + \tilde{y}_{c,g}) n_{c,t} \geq 1$ or $\bar{z}_{g,t} + \tilde{z}_{g,t} \geq 1$, we can add the corresponding groups g to \mathcal{G}_t . Then, γ_t points to the head of the list. While iterating through the groups, we also calculate $\bar{n}_t = \sum_{g \in \mathcal{G}} \left(\sum_{c \in \mathcal{C}'_g} (\bar{y}_{c,g} + \tilde{y}_{c,g}) + \bar{z}_{g,t} + \tilde{z}_{g,t} \right)$ and store the corresponding value together with the item type. The lists \mathcal{G}_t can be generated in $\mathcal{O}(|\mathcal{T}|\sum_{g \in \mathcal{G}} |\mathcal{C}_g|) = \mathcal{O}\left(\frac{\log^4 n}{\varepsilon^8} \frac{\log^{2/\varepsilon} n}{\varepsilon^{4/\varepsilon}}\right)$ many iterations.

For maintaining and updating the pointer κ_t , we create the list $\mathcal{C}_{g,t}$ storing all configurations $c \in \overline{\mathcal{C}}_g$ with $n_{c,t} \geq 1$. While iterating through the groups and creating \mathcal{G}_t , we also add c together with $n_{c,t}$ to the list $\mathcal{C}_{g,t}$ if $n_{c,t} \geq 1$. Initially, κ_t points to the head of $\mathcal{C}_{g,t}$, where g is the first group that packs t as big item. If c is the corresponding configuration, we start with $\eta_t^B = n_{c,t}$. The time needed for this is bounded by $\mathcal{O}(|\mathcal{T}|\sum_{g\in\mathcal{G}}|\mathcal{C}_g|) = \mathcal{O}\left(\frac{\log^4 n}{\varepsilon^8}\frac{\log^{2/\varepsilon} n}{\varepsilon^{4/\varepsilon}}\right)$.

The pointer κ_g^r is initialized with $\kappa_g^r = \sum_{g'=1}^{g-1} m_{g'} + 1$. By using binary search on the list $\overline{\mathcal{C}}_g$, we get s_1 , the total size of configuration 1 assigned to κ_g^r , and binary search over the knapsacks allows us to obtain $S_{\kappa_g^r}$, the capacity of knapsack κ_g^r . Thus, $\rho_g^r = S_{\kappa_g^r} - s_1$ can be initialized in time $\mathcal{O}(\sum_{g \in \mathcal{G}} (\log (|\overline{\mathcal{C}}_g| + \log m)) = \mathcal{O}\left(\frac{\log^2 n}{\varepsilon^5} (\log \frac{\log n}{\varepsilon} + \log m)\right)$.

If $m_g \geq \frac{1}{\varepsilon}$, we set $\kappa_g^c = \sum_{g'=1}^{g-1} m_{g'} + \lfloor (1-\varepsilon)m_g \rfloor + 1$, while $m_g < \frac{1}{\varepsilon}$ implies that κ_g^c points to the knapsack $R^{(\varepsilon)} = |\{g': g' \leq g, m_{g'} < \frac{1}{\varepsilon}\}|$ given by resource augmentation. The time needed for initializing κ_g^c is $\mathcal{O}(|\mathcal{G}|)$. In order to determine the position of the next cut item, we also maintain ρ^c , initialized with $\rho_g^c = \frac{1}{\varepsilon}$, that counts how many slots are still left in knapsack κ_g^c .

Now consider the query for an item j. We can decide in time $\mathcal{O}(\log n)$ if j has already been queried in the current round. Upon arrival of j, we calculated the index ℓ of its value class. If $\ell \in \{\bar{\ell}, \dots, \ell_{\max}\}$, then the item types \mathcal{T}_{ℓ} together with their first and last item can be determined in time $\mathcal{O}(\frac{\log n}{\varepsilon^2})$ by retracing the steps of the linear grouping,. By binary search, the item type of j can be determined in time $\mathcal{O}(\log \frac{\log n}{\varepsilon})$. Once the item type is

known, we check if j belongs to the first \bar{n}_t items of this type. If not, then NOT SELECTED is returned. Otherwise, the pointer γ_t answers the question in which group item j is packed.

If j is small and $\eta_t^S > \tilde{z}_{t,\gamma_t}$, the knapsack k(j) can be determined in constant time by nested case distinction and having the correct pointer (either $\kappa_{\gamma_t}^r$ or $\kappa_{\gamma_t}^c$) dictate the answer. In order to bound the update time of the data structures, note that packing j as regular item only implies the updates of ρ_{γ_t} and of η_t^S , which take constant time. Hence, it remains to consider the case where j is packed as a cut item. The capacity of the new knapsack $\kappa_{\gamma_t}^r$ can be determined in $\mathcal{O}(\log m)$ by binary search over the knapsack list while the configuration c of the new knapsack $\kappa_{\gamma_t}^r$ and its total size are determined by binary search over the list α_{γ_t} in time $\mathcal{O}\left(\log |\overline{C}_{\gamma_t}|\right) = \mathcal{O}\left(\frac{1}{\varepsilon}\log\frac{\log n}{\varepsilon}\right)$. Then, $\rho_{\gamma_t}^r = S_{\kappa_{\gamma_t}^r} - s_c$ can be computed with constantly many operations. If $\rho_{\gamma_t}^c = 0$ after packing j in $\kappa_{\gamma_t}^c$, we increase the knapsack pointer by one and update $\rho_{\gamma_t}^c = \frac{1}{\varepsilon}$. In case $\eta_t^S = \tilde{z}_{\gamma_t,t} = 1$, item j is packed in the knapsack $R_{\gamma_t,t}^{(z)}$ which can be decided in constant time. Otherwise the group pointer γ_t is increased and either η_t^S is updated according to the new group or κ_t and η_t^B are used. Updating γ_t can be done by binary search over the list \mathcal{G}_t in time $\mathcal{O}(\log |\mathcal{G}|)$. The pointer γ_t is updated at most once before determining k(j). Hence, the case distinction on the relative size of t is invoked at most twice.

If j is big, the pointer κ_{γ_t} dictates the answer which can be returned in time $\mathcal{O}(1)$. For bounding the running time of the possible update operations, observe that η_t is updated in constant time with values bounded by n. If $\eta_t^B = 0$ after the update, the knapsack pointer κ_t needs to be updated as well. The most time consuming update operations are finding a new configuration c' and possibly even a new group g'. Finding configuration $c' \in \overline{\mathcal{C}}_{\gamma_t,t}$ can be done by binary search in time $\mathcal{O}\left(\log |\overline{\mathcal{C}}_{\gamma_t}|\right) = \mathcal{O}\left(\frac{1}{\varepsilon}\log\frac{\log n}{\varepsilon}\right)$. To update κ_t and η_t^B , we extract κ_t from the list α_{γ_t} and $n_{c',t}$ from the list $\overline{\mathcal{C}}_{\gamma_t,t}$ in time $\mathcal{O}\left(\log |\overline{\mathcal{C}}_{\gamma_t}|\right) = \mathcal{O}\left(\frac{1}{\varepsilon}\log\frac{\log n}{\varepsilon}\right)$ by binary search. If the algorithm needs to update γ_t as well, this can be done by binary search on the list \mathcal{G}_t in time $\mathcal{O}(\log |\mathcal{G}_t|) = \mathcal{O}\left(\log\frac{\log (n)}{\varepsilon}\right)$.

In both cases, the running time of answering the query and possibly updating data structures is bounded by the running time of the linear grouping step and by the routine to access one particular knapsack, i.e., by $\mathcal{O}(\log m + \frac{\log n}{\varepsilon^2})$.

▶ Lemma 47. The solution value can be calculated in time $\mathcal{O}(\frac{\log^3 n}{\varepsilon^4})$.

Proof. For obtaining the value of the current solution, we calculate the total value of the first \bar{n}_t items. We do this by iterating through the value classes once and per value class, we iterate once through the list \mathcal{T}_ℓ to access the number \bar{n}_t . Then, we use prefix computation twice in order to access the total value of the first \bar{n}_t items of type t. Lemma 1 bounds this time by $\mathcal{O}(\log n)$. By Lemma 25, the number of item types is bounded by $\mathcal{O}(\frac{\log^3 n}{\varepsilon^4})$. Combining these two values bounds the total running time by $\mathcal{O}(\frac{\log^3 n}{\varepsilon^4})$. As this time is clearly dominated by obtaining the implicit solution in the first place, we calculate and store the solution value when computing the implicit solution value and thus are able to return it in constant time.

▶ Lemma 48. In time $\mathcal{O}(|P| \frac{\log^3 n}{\varepsilon^4} (\log m + \frac{\log n}{\varepsilon^2}))$ a query for the complete solution P can be answered.

Proof. For returning the complete solution, we determine the packed items and query each packed item individually. Lemma 46 bounds their query times by $\mathcal{O}(\log m + \frac{\log n}{\varepsilon^2})$ while

Lemma 1 bounds the running time for accessing item j. Lemma 25 bounds the number of item types by $\mathcal{O}(\frac{\log^2 n}{\varepsilon^4})$. In total, the running time is bounded by $\mathcal{O}(|P|\frac{\log^3 n}{\varepsilon^4}(\log m + \frac{\log n}{\varepsilon^2}))$, where P is the current solution.

E.2.1 Proof of main result

Proof of Theorem 8. Lemma 42 gives the bound on the approximation ratio of our algorithm and Lemma 43 bounds the running time of an update operation. Further, Lemma 46 gives the running time for query operations.

F Multiple Knapsack

Analysis.

We consider the iteration in which all the guesses, ℓ_{max} , k and S_O are correct. Let \mathcal{P}_1 be the set of solutions on the ordinary knapsacks (without the additional virtual knapsack) and the special knapsacks such that the total size of ordinary items placed in special knapsacks lies in the range $[S_O, (1+\varepsilon)S_O]$. Denote by OPT_1 a solution of highest value in \mathcal{P}_1 . Altering OPT by deleting the extra knapsacks gives a solution in \mathcal{P}_1 of value at least $(1-\varepsilon) \cdot v(\text{OPT})$. This holds since for correct guesses the extra knapsacks by definition contribute at most an ε -fraction to OPT. Further, the correctness of the guessed S_O implies that the altered OPT is indeed a packing in \mathcal{P}_1 .

- ▶ **Observation 49.** For OPT_1 defined as above, we have $v(OPT_1) \ge (1 \varepsilon) \cdot v(OPT)$.
- ▶ Lemma 50. Consider an optimal solution OPT_O to the ordinary subproblem, i.e., exclude items in J_E but include the virtual knapsack. Then $v(OPT_O) \ge v(OPT_{1,O}) 2\varepsilon \cdot v(OPT)$, where we use the shorthand $OPT_{1,O} := (OPT_1 \cap J_O) \setminus J_E$.
- **Proof.** Consider the ordinary items in OPT_1 that are not in J_E . Leave items on ordinary knapsacks in their current position and place ordinary items on special knapsacks into the virtual ordinary knapsack. The latter is possible with the exception of possibly an ε -fraction of the items (with respect to size) due to S_O being rounded down. Deleting the least dense items until the remainder fits into the virtual knapsack causes a loss of at most an ε -fraction of the value of OPT_1 plus an additional ordinary item j_O . This item j_O contributes at most an ε -fraction to OPT as its value is not larger than that of the least valuable element in J_E which has a value of less than $\varepsilon v(OPT)$.
- ▶ **Lemma 51.** Let P_F be the final solution the algorithm computes. Then $v(P_F) \ge (1 7\varepsilon)v(OPT)$.
- **Proof.** Consider P_O , the solution of the ordinary subproblem returned by the algorithm of Appendix E (including virtual knapsack and resource augmentation). We know that $v(P_O) \ge (1 \varepsilon) \cdot v(\text{OPT}_O) \ge v(\text{OPT}_{1,O}) 3\varepsilon \cdot v(\text{OPT})$ by Theorem 8 and Lemma 50.

Let $\mathrm{OPT}_S \coloneqq \mathrm{OPT}_1 \cap J_S$, and $P_2 \coloneqq P_O \cup \mathrm{OPT}_S \cup J_E$. Then, $\mathrm{OPT}_1 = \mathrm{OPT}_{1,O} \cup (\mathrm{OPT}_1 \cap J_E) \cup (\mathrm{OPT}_1 \cap J_S)$ implies

$$v(\text{OPT}_1) = v(\text{OPT}_{1,O}) + v(\text{OPT}_1 \cap J_E) + v(\text{OPT}_1 \cap J_S)$$

$$\leq v(P_O) + 3\varepsilon v(\text{OPT}) + v(J_E) + v(\text{OPT}_S)$$

$$\leq v(P_2) + 3\varepsilon v(\text{OPT}).$$

With Observation 49 we then obtain $v(P_2) \ge v(\text{OPT}) - 4\varepsilon v(\text{OPT})$.

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

Therefore, $v(P_3) \geq v(P_2) - 2\varepsilon v(\text{OPT})$. Moreover, the portion of P_3 on special knapsacks is a valid solution for the request sent to the special subproblem. Therefore, using Theorem 4, the overall solution P_F satisfies $v(P_F) \geq (1 - 7\varepsilon)v(\text{OPT})$.

▶ **Lemma 52.** The algorithm has update time $\left(\frac{1}{\varepsilon}\log\left(nv_{\max}\right)\right)^{f(1/\varepsilon)} + \mathcal{O}(\frac{1}{\varepsilon}\log\overline{v}\log n)$, where f is a quasi-linear function.

Proof. Guessing k adds a factor of $\frac{1}{\varepsilon}$ to the update time. Placing the $\frac{L_S}{\varepsilon^2}$ most valuable ordinary items on extra knapsacks and removing them from data structures takes time $\mathcal{O}(\frac{L_S}{\varepsilon^2}\log n)$ which is within the time bound. The same holds for the updates of the ordinary and special data structures and for solving the subproblems with the algorithms of Appendices B and E.

Cutting the items placed in the virtual ordinary knapsack info $\frac{L_S}{\varepsilon}$ equal-sized bundles can be archived efficiently as follows. Compute the total size of these items, using the number of items used for each of the $\mathcal{O}(\frac{\log^2 n}{\varepsilon^4})$ item types and deduce the size of a bundle. Sort the item types, e.g., by value then size, and then iteratively pack items of the same type by computing how many items of this type fit in the next non-empty bundle. This takes time $\mathcal{O}(\frac{\log^2 n}{\varepsilon^4} \cdot \frac{L_S}{\varepsilon})$ which is sufficient.

Additionally, the maintenance of data structures is dominated in running time by that of the subproblems. These takes time $\mathcal{O}(\frac{1}{\varepsilon}\log \overline{v}\log n)$ and cause the additive factor.

▶ Lemma 53. The query times of our algorithm are as follows. (i) Single item queries are answered in time $\mathcal{O}(\frac{\log n}{\varepsilon^2})$. (ii) Solution value queries are answered in time $\mathcal{O}(1)$. (iii) Queries of the entire solution P are answered in time $\mathcal{O}(\frac{\log^4 n}{\varepsilon^6}|P|)$.

G Hardness of Approximation

The following theorems provide a justification why our algorithms for Multiple Knapsack have different running times depending on the number of knapsacks. As Chekuri and Khanna [20] observed, Multiple Knapsack with m=2 does not admit an FPTAS unless P=NP.

▶ Theorem 54 (Proposition 2.1 in [20]). If MULTIPLE KNAPSACK with two identical knapsacks has an FPTAS, then PARTITION can be solved in polynomial time. Hence there is no FPTAS for MULTIPLE KNAPSACK even with m = 2, unless P = NP.

In the fully dynamic setting, this implies that there is no dynamic algorithm with running time polynomial in log n and $\frac{1}{\varepsilon}$ unless P = NP. We are able to extend this result to the case where $m \leq \frac{1}{3\varepsilon}$. The statement focuses on dynamic algorithms, our main interest here, but the proof does not use the dynamic nature, only the final complexity.

▶ Theorem 55. Unless P = NP, there is no fully dynamic algorithm for MULTIPLE KNAP-SACK that maintains a $(1 - \varepsilon)$ -approximate solution in update time polynomial in $\log n$ and $\frac{1}{\varepsilon}$, for $m < \frac{1}{3\varepsilon}$.

Proof. Consider the strongly NP-hard problem 3-Partition [30] where there are 3m items with sizes $a_j \in \mathbb{N}$ such that $\sum_{j=1}^{3m} a_j = mA$. We use the restricted-input variant where sizes belong to (A/2, A/4) so that only subsets of size 3 may sum to A. The task is to decide whether there exists a partition $\bigcup_{i=1}^m J_i = [3m]$ such that $|J_i| = 3$ and $\sum_{j \in J_i} a_j = A$ for $1 \le i \le m$.

Consider the following instance for Dynamic Multiple Knapsack: There are m knapsacks with S=A and 3m many items. Each item corresponds to a 3-Partition item with $s_j=a_j$ and $v_j=1$ for $1\leq j\leq 3m$. Observe that the 3-Partition instance is a Yes-instance if and only if the optimal solution to the Knapsack problem contains 3m items. Indeed, in such a Knapsack instance, each knapsack must have 3 items.

If Dynamic Multiple Knapsack admits a dynamic algorithm with approximation guarantee at least $(1-\varepsilon)$ and running time polynomial in $\frac{1}{\varepsilon}$ and $\log n_0$ where $m<\frac{1}{3\varepsilon}$, such an algorithm is able to optimally solve the Knapsack instance reduced from 3-Partition. Thus, such an algorithm decides 3-Partition in polynomial time which is not possible, unless P=NP.

Note that this result can be extended to a larger number of knapsacks by adding an appropriate number of sufficiently small knapsacks, i.e., polynomially many in n.