Fully Dynamic Algorithms for Knapsack Problems with Polylogarithmic Update Time

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

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

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

Bertrand Simon
IN2P3 Computing Center, CNRS, Villeurbanne, France

Andreas Wiese
Department of Industrial Engineering, University of Chile, Santiago, 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

Digital Object Identifier 10.4230/LIPIcs.FSTTCS.2021.18

Related Version *Full Version*: `https://arxiv.org/abs/2007.08415`

Funding This work is partly funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) – Project Number 146371743 – TRR 89 Invasive Computing.

Acknowledgements We thank Martin Böhm, Peter Kling, and Jens Schlöter for the fruitful discussions.

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 $j \in [n] := \{1, 2, \ldots, n\}$ has a size $s_j \in \mathbb{N}$ and a value $v_j \in \mathbb{N}$. The Knapsack problem asks for a subset of items, $P \subseteq [n]$, with 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 $P_1, P_2, \ldots, P_m \subseteq [n]$ 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 [43] and initiated a long sequence of follow-up work, which is still active [17, 52]. Multiple Knapsack is substantially harder and does not admit an FPTAS, unless P = NP, even with two identical knapsacks [19]. However, approximation schemes with running times of the form $n^{f(\varepsilon)}$ (PTASs) are known [19, 54] as well as improvements to only $f(\varepsilon)n^{\mathcal{O}(1)}$ (EPTASs) [48, 50]. 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., [13, 23, 59]. 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 [15, 26, 38] 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 [40, 42], the minimum spanning tree [42], and vertex cover [9, 11]. 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, 8, 10, 34]. Also, recently, algorithms have been developed for maintaining maximal independent sets, e.g., [4, 18, 64], and approximate maximum independent sets in special graph classes [12, 20, 39].

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 [45]. 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 $(1-\varepsilon)$-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 $\left(\frac{1}{\varepsilon}\log(nv_{\max})\right)^{\mathcal{O}(1)} + \mathcal{O}\left(\frac{1}{\varepsilon}\log n \log \overline{v}\right)$ 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 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 [50] 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 $\left(\frac{1}{\varepsilon}\log n\right)^{\mathcal{O}(1)}$ 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 [43] running times have been improved steadily [17,30,31,52,55,58,67] with $\mathcal{O}(n\log\frac{1}{\varepsilon}+(\frac{1}{\varepsilon})^{9/4})$ by Jin [52] being the currently fastest. Recent work on conditional lower bounds [22,57] 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 [17,65].

A PTAS for Multiple Knapsack was first presented by Chekuri and Khanna [19] and EPTASs due to Jansen [48,50] are also known. The fastest of these algorithms [50] 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 [19] invokes a guessing step with $n^{f(1/\varepsilon)}$ many options which are too many for a polylogarithmic update time. The EPTASs in [48,50] 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 [61] 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 [47], the removable online knapsack problem [21, 35–37, 46], and with advice [14]. Other models with uncertainty in the item set or the knapsack capacity include the *stochastic* knapsack problem [7, 25, 60] and *robust* knapsack problems [16, 27, 62, 70]. Related to our setting are also online models with a softened irrevocability requirement, e.g., online optimization with *recourse* [29, 33, 44, 63] or *migration* [51, 68, 69] 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 the full version of this paper [28], 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. 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. 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 $(1+\varepsilon)^\ell \leq v_j < (1+\varepsilon)^{\ell+1}$. Upon arrival of some item j, we calculate the index ℓ_j such that $j \in V_{\ell_j}$ and store the tuple (j, v_j, s_j, ℓ_j) representing j in the auxiliary data structures of the respective algorithm. In the 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 $\mathcal{O}\big(\frac{\log v_{\max}}{\varepsilon}\big)$ many value classes. (ii) For optimal solutions $\textsc{Opt}$ and $\textsc{Opt}'$ for the original and rounded instance, $v(\textsc{Opt}') \geq (1-\varepsilon) \cdot v(\textsc{Opt})$.*

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 $\mathcal{O}\big(\frac{\log^4(nv_{\max})}{\varepsilon^9}\big) + \mathcal{O}\big(\frac{1}{\varepsilon}\log n \log \overline{v}\big)$. 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 $\textsc{Opt}$ into high- and low-value items, respectively. The high-value items are the $\frac{1}{\varepsilon}$ most valuable items of $\textsc{Opt}$, and the low-value items are the remaining items of $\textsc{Opt}$. We compute a small set of candidate items $H_{\frac{1}{\varepsilon}}$ that intuitively contains all relevant high-value items in $\textsc{Opt}$. Also, we guess a placeholder item for the low-value items, that is large enough to accomodate 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 $\textsc{Opt}_{\frac{1}{\varepsilon}}$ a set of $\frac{1}{\varepsilon}$ most valuable items of $\textsc{Opt}$. 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 $\textsc{Opt}_{\frac{1}{\varepsilon}}$ and let $n_{\min} := |\textsc{Opt}_{\frac{1}{\varepsilon}} \cap V_{\ell_{\min}}| \leq \frac{1}{\varepsilon}$. Furthermore, denote by $\mathcal{V}_L$ the value of the items in $\textsc{Opt} \setminus \textsc{Opt}_{\frac{1}{\varepsilon}}$, 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 $\ell_{\max}$, $\ell_{\min}$, 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 $V_{\ell_{\min}+1}, \dots, V_{\ell_{\max}}$, 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 [52]) 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 $|\textsc{Opt}_{\frac{1}{\varepsilon}}| = \frac{1}{\varepsilon}$, this excludes items from $\textsc{Opt}$ with a total value of at most $\frac{1}{\varepsilon} \cdot \varepsilon^2 (1+\varepsilon)^{\ell_{\max}} \leq \varepsilon \cdot \textsc{Opt}$. Furthermore, due to guessing $\mathcal{V}_L$ up to a power of $(1+\varepsilon)$, we get $v_B = \mathcal{V}_L \geq \frac{1}{1+\varepsilon} \cdot v(\textsc{Opt} \setminus \textsc{Opt}_{\frac{1}{\varepsilon}})$. Finally, in Step 2, at most one item was cut fractionally. It is charged to the $\frac{1}{\varepsilon}$ items of $\textsc{Opt}_{\frac{1}{\varepsilon}}$, 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 $H_{\frac{1}{\varepsilon}}$ 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 the full version of this paper [28]. 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 [50] instead of an FPTAS (see [28]).

▶ **Theorem 4.** *For $\varepsilon > 0$, there is a dynamic algorithm for Multiple Knapsack that achieves an approximation factor of $(1-\varepsilon)$ with update time $2^{f(1/\varepsilon)}\big(\frac{m}{\varepsilon}\log(nv_{\max})\big)^{\mathcal{O}(1)} + \mathcal{O}\big(\frac{1}{\varepsilon}\log\overline{v}\log n\big)$, with f quasi-linear. Item queries are answered in time $\mathcal{O}\big(\log\frac{m^2}{\varepsilon^6}\big)$, 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 $|\textsc{Opt} \cap J'| \leq n'$ for all optimal solutions $\textsc{Opt}$, there is an algorithm with running time $\mathcal{O}\big(\frac{\log^5 n'}{\varepsilon^5}\big)$ that rounds the items in J' to item types $\mathcal{T}$ with $|\mathcal{T}| \leq \mathcal{O}\big(\frac{\log^2 n'}{\varepsilon^4}\big)$ and ensures $v(\textsc{Opt}_{\mathcal{T}}) \geq \frac{(1-\varepsilon)(1-2\varepsilon)}{(1+\varepsilon)^2} v(\textsc{Opt})$. Here, $\textsc{Opt}_{\mathcal{T}}$ is 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 $\ell_{\max}$ which is defined to be the guess for the highest value ℓ with $V'_\ell \cap \textsc{Opt} \neq \emptyset$ and let $\bar{\ell} := \ell_{\max} - \lceil \log_{1+\varepsilon}(n'/\varepsilon) \rceil$.

1) For each ℓ with $\bar{\ell} \leq \ell \leq \ell_{\max}$ 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 $\lceil \varepsilon n_\ell \rceil$ or $\lfloor \varepsilon n_\ell \rfloor$ 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 $1 \leq k \leq \frac{1}{\varepsilon}$, 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 $\textsc{Opt}'$, 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 $v(\textsc{Opt}_{\mathcal{T}}) \geq \frac{(1-\varepsilon)(1-2\varepsilon)}{(1+\varepsilon)^2} v(\textsc{Opt})$.

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 $n_\ell = |\textsc{Opt} \cap V'_\ell|$, 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 the full version of this paper [28].

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 $m \leq n$ as otherwise, the problem is trivial. We focus on instances where m is large, i.e., $m \geq \frac{16}{\varepsilon^7} \log^2 n$. If $m \leq \frac{16}{\varepsilon^7} \log^2 n$, 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 [48,50]; 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 $\textsc{Opt}$, define $\textsc{Opt}_B := \textsc{Opt} \cap J_B$ and $\textsc{Opt}_S := \textsc{Opt} \cap J_S$.

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_j, the value v_j, and the value class ℓ_j 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 $\ell_{\max}$, which we define to be the largest index ℓ with $V_\ell \cap \textsc{Opt}_B \neq \emptyset$. 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 $v(\textsc{Opt}_S)$. 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 $\ell_{\max}$ 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.

$$\begin{aligned} \max \quad & \sum_{c\in\mathcal{C}} y_c v_c & & & \\ \text{subject to} \quad & \sum_{c\in\mathcal{C}} y_c s_c & \leq & \lfloor(1-3\varepsilon)m\rfloor S - s_S & \\ & \sum_{c\in\mathcal{C}} y_c & \leq & \lfloor(1-3\varepsilon)m\rfloor & \\ & \sum_{c\in\mathcal{C}} y_c n_{tc} & \leq & n_t & \text{for all } t \in \mathcal{T} \\ & y_c & \in & \mathbb{Z}_{\geq 0} & \text{for all } c \in \mathcal{C} \end{aligned} \tag{P}$$

   By the first inequality, the configurations fit into $\lfloor(1-3\varepsilon)m\rfloor$ 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 $j^\star$ be the densest small item not in P. Pack $j^\star$ 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 [32] 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 [28] 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 [28] 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* $\mathcal{O}\left(\log n + \max\left\{\log\frac{\log n}{\varepsilon}, \frac{1}{\varepsilon}\right\}\right)$*, solution value queries can be answered in time* $\mathcal{O}(1)$*, and queries of the entire solution* P *can be answered in time* $\mathcal{O}\left(|P|\frac{\log^4 n}{\varepsilon^4}\log\frac{\log n}{\varepsilon}\right)$.

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

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.

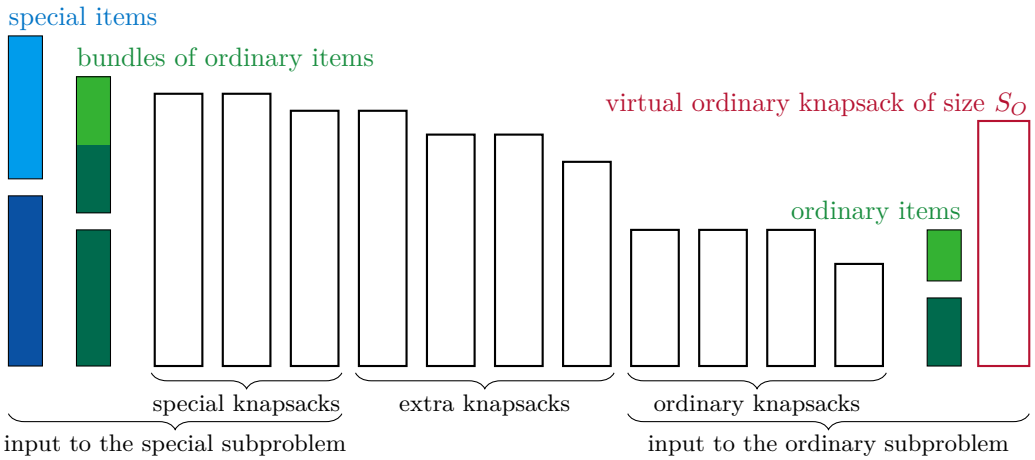

Figure 1 Input of the special and the ordinary subproblems: Based on the current guess for the extra knapsacks, the knapsacks are partitioned into three groups (special, extra, and ordinary). When an item fits into at least one ordinary knapsack, it is ordinary and special otherwise. The total size of 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.

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

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 [48]; 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; see Figure 1. 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 $L = \big(\frac{\log n}{\varepsilon}\big)^{\Theta(1/\varepsilon)}$. We assume that $m > \big(\frac{1}{\varepsilon}\big)^{4/\varepsilon} \cdot L$, 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 $\varepsilon \cdot \textsc{Opt}$. 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 $(1+\varepsilon)$. 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 $\mathcal{O}\left(\frac{\log^2 n}{\varepsilon^4}\right)$ 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 $\mathcal{O}\left(\frac{\log^2 n}{\varepsilon^4}\right)$ 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 items in extra knapsacks are stored explicitly, they can be accessed in constant time. See [28] 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.

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

We give a proof in the full version [28]. We remark 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.

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 [24] 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 $f(\varepsilon)n^{\mathcal{O}(1)}$ [56].

A recent line of research exploits fast techniques for solving convolution problems to speed up knapsack algorithms (exact and approximate); see, e.g., [3, 17, 52, 55, 66]. In fact, it has been shown that Knapsack is computationally equivalent to the $(\min, +)$-convolution problem [22]. 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 [53] and [41, 49] combined with rather straightforward data structures can be lifted to a fully dynamic algorithm framework for the respective problems.

References

1 Amir Abboud, Raghavendra Addanki, Fabrizio Grandoni, Debmalya Panigrahi, and Barna Saha. Dynamic set cover: improved algorithms and lower bounds. In *STOC*, pages 114–125. ACM, 2019.

2 Amir Abboud and Virginia Vassilevska Williams. Popular conjectures imply strong lower bounds for dynamic problems. In *FOCS*, pages 434–443. IEEE Computer Society, 2014.

3 Kyriakos Axiotis and Christos 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 Soheil Behnezhad, Mahsa Derakhshan, MohammadTaghi Hajiaghayi, Cliff Stein, and Madhu 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 Richard Bellman. *Dynamic Programming*. Princeton University Press, Princeton, NJ, USA, 1957.

6 Anton Beloglazov and Rajkumar Buyya. Energy efficient allocation of virtual machines in cloud data centers. In *CCGRID*, pages 577–578. IEEE Computer Society, 2010.

7 Anand Bhalgat, Ashish Goel, and Sanjeev Khanna. Improved approximation results for stochastic knapsack problems. In *SODA*, pages 1647–1665. SIAM, 2011.

8 Sayan Bhattacharya, Monika Henzinger, and Giuseppe 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.

9 Sayan Bhattacharya, Monika Henzinger, and Danupon 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.

10 Sayan Bhattacharya, Monika Henzinger, and Danupon Nanongkai. A new deterministic algorithm for dynamic set cover. In *FOCS*, pages 406–423. IEEE Computer Society, 2019.

11 Sayan Bhattacharya and Janardhan 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.

12 Sujoy Bhore, Jean Cardinal, John Iacono, and Grigorios Koumoutsos. Dynamic geometric independent set. *arXiv preprint*, 2020. `arXiv:2007.08643`.

13 Norman Bobroff, Andrzej Kochut, and Kirk A. Beaty. Dynamic placement of virtual machines for managing SLA violations. In *Integrated Network Management*, pages 119–128. IEEE, 2007.

14 Hans-Joachim Böckenhauer, Dennis Komm, Richard Královic, and Peter Rossmanith. The online knapsack problem: Advice and randomization. *Theor. Comput. Sci.*, 527:61–72, 2014.

15 Nicolas Boria and Vangelis Th. Paschos. A survey on combinatorial optimization in dynamic environments. *RAIRO - Operations Research*, 45(3):241–294, 2011.

16 Christina Büsing, Arie M. C. A. Koster, and Manuel Kutschka. Recoverable robust knapsacks: the discrete scenario case. *Optim. Lett.*, 5(3):379–392, 2011.

17 Timothy 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.

18 Shiri Chechik and Tianyi Zhang. Fully dynamic maximal independent set in expected poly-log update time. In *FOCS*, pages 370–381. IEEE, 2019.

19 Chandra Chekuri and Sanjeev Khanna. A polynomial time approximation scheme for the multiple knapsack problem. *SIAM J. Comput.*, 35(3):713–728, 2005.

20 Spencer Compton, Slobodan Mitrović, and Ronitt Rubinfeld. New partitioning techniques and faster algorithms for approximate interval scheduling. *arXiv preprint*, 2020. `arXiv:2012.15002`.

21 Marek Cygan, Łukasz Jeż, and Jiří Sgall. Online knapsack revisited. *Theory Comput. Syst.*, 58(1):153–190, 2016.

22 Marek Cygan, Marcin Mucha, Karol Wegrzycki, and Michal Wlodarczyk. On problems equivalent to (min, +)-convolution. *ACM Trans. Algorithms*, 15(1):14:1–14:25, 2019.

23 Khuzaima Daudjee, Shahin Kamali, and Alejandro López-Ortiz. On the online fault-tolerant server consolidation problem. In *SPAA*, pages 12–21. ACM, 2014.

24 Wenceslas Fernandez de la Vega and George S. Lueker. Bin packing can be solved within 1+epsilon in linear time. *Combinatorica*, 1(4):349–355, 1981.

25 Brian C. Dean, Michel X. Goemans, and Jan Vondrák. Approximating the stochastic knapsack problem: The benefit of adaptivity. *Math. Oper. Res.*, 33(4):945–964, 2008.

26 Camil Demetrescu, David Eppstein, Zvi Galil, and Giuseppe F. Italiano. *Dynamic Graph Algorithms*, page 9. Chapman & Hall/CRC, 2 edition, 2010.

27 Yann Disser, Max Klimm, Nicole Megow, and Sebastian Stiller. Packing a knapsack of unknown capacity. *SIAM J. Discret. Math.*, 31(3):1477–1497, 2017.

28 Franziska Eberle, Nicole Megow, Lukas Nölke, Bertrand Simon, and Andreas Wiese. Fully dynamic algorithms for knapsack problems with polylogarithmic update time. *CoRR*, abs/2007.08415, 2020. `arXiv:2007.08415`.

29 Björn Feldkord, Matthias Feldotto, Anupam Gupta, Guru Guruganesh, Amit Kumar, Sören Riechers, and David 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 George Gens and Eugene Levner. Computational complexity of approximation algorithms for combinatorial problems. In *MFCS*, volume 74 of *Lecture Notes in Computer Science*, pages 292–300. Springer, 1979.

31 George Gens and Eugene Levner. Fast approximation algorithms for knapsack type problems. In *Optimization Techniques*, pages 185–194. Springer, 1980.

32 Martin Grötschel, László Lovász, and Alexander Schrijver. The ellipsoid method and its consequences in combinatorial optimization. *Combinatorica*, 1(2):169–197, 1981.

33 Albert Gu, Anupam Gupta, and Amit Kumar. The power of deferral: Maintaining a constant-competitive steiner tree online. *SIAM J. Comput.*, 45(1):1–28, 2016.

34 Anupam Gupta, Ravishankar Krishnaswamy, Amit Kumar, and Debmalya Panigrahi. Online and dynamic algorithms for set cover. In *STOC*, pages 537–550. ACM, 2017.

35 Xin Han, Yasushi Kawase, and Kazuhisa Makino. Randomized algorithms for removable online knapsack problems. In *FAW-AAIM*, volume 7924 of *Lecture Notes in Computer Science*, pages 60–71. Springer, 2013.

36 Xin Han, Yasushi Kawase, Kazuhisa Makino, and He Guo. Online removable knapsack problem under convex function. *Theor. Comput. Sci.*, 540:62–69, 2014.

37 Xin Han and Kazuhisa Makino. Online removable knapsack with limited cuts. *Theor. Comput. Sci.*, 411(44-46):3956–3964, 2010.

38 Monika 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.

39 Monika Henzinger, Stefan Neumann, and Andreas Wiese. Dynamic Approximate Maximum Independent Set of Intervals, Hypercubes and Hyperrectangles. In *SoCG*, volume 164 of *LIPIcs*, pages 51:1–51:14. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2020. doi:10.4230/LIPIcs.SoCG.2020.51.

40 Monika Rauch Henzinger and Valerie King. Randomized fully dynamic graph algorithms with polylogarithmic time per operation. *J. ACM*, 46(4):502–516, 1999.

41 Dorit S. Hochbaum and David B. Shmoys. Using dual approximation algorithms for scheduling problems theoretical and practical results. *J. ACM*, 34(1):144–162, 1987.

42 Jacob Holm, Kristian de Lichtenberg, and Mikkel Thorup. Poly-logarithmic deterministic fully-dynamic algorithms for connectivity, minimum spanning tree, 2-edge, and biconnectivity. *J. ACM*, 48(4):723–760, 2001.

43 Oscar H. Ibarra and Chul E. Kim. Fast approximation algorithms for the knapsack and sum of subset problems. *J. ACM*, 22(4):463–468, 1975.

44 Makoto Imase and Bernard M. Waxman. Dynamic steiner tree problem. *SIAM J. Discret. Math.*, 4(3):369–384, 1991.

45 Zoran Ivkovic and Errol L. Lloyd. Fully dynamic algorithms for bin packing: Being (mostly) myopic helps. *SIAM J. Comput.*, 28(2):574–611, 1998.

46 Kazuo Iwama and Shiro Taketomi. Removable online knapsack problems. In *ICALP*, volume 2380 of *Lecture Notes in Computer Science*, pages 293–305. Springer, 2002.

47 Kazuo Iwama and Guochuan Zhang. Online knapsack with resource augmentation. *Inf. Process. Lett.*, 110(22):1016–1020, 2010.

48 Klaus Jansen. Parameterized approximation scheme for the multiple knapsack problem. *SIAM J. Comput.*, 39(4):1392–1412, 2009.

49 Klaus 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.

50 Klaus 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.

51 Klaus Jansen and Kim-Manuel Klein. A robust AFPTAS for online bin packing with polynomial migration. *SIAM J. Discret. Math.*, 33(4):2062–2091, 2019.

52 Ce 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.

53 Narendra Karmarkar and Richard M. Karp. An efficient approximation scheme for the one-dimensional bin-packing problem. In *FOCS*, pages 312–320. IEEE Computer Society, 1982.

54 Hans 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.

55 Hans Kellerer and Ulrich Pferschy. Improved dynamic programming in connection with an FPTAS for the knapsack problem. *J. Comb. Optim.*, 8(1):5–11, 2004.

56 Ariel Kulik and Hadas Shachnai. There is no EPTAS for two-dimensional knapsack. *Inf. Process. Lett.*, 110(16):707–710, 2010.

57 Marvin Künnemann, Ramamohan Paturi, and Stefan 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.

58 Eugene L. Lawler. Fast approximation algorithms for knapsack problems. *Math. Oper. Res.*, 4(4):339–356, 1979.

59 Yusen Li, Xueyan Tang, and Wentong Cai. On dynamic bin packing for resource allocation in the cloud. In *SPAA*, pages 2–11. ACM, 2014.

60 Will Ma. Improvements and generalizations of stochastic knapsack and markovian bandits approximation algorithms. *Math. Oper. Res.*, 43(3):789–812, 2018.

61 Alberto Marchetti-Spaccamela and Carlo Vercellis. Stochastic on-line knapsack problems. *Math. Program.*, 68:73–104, 1995.

62 Nicole Megow and Julián Mestre. Instance-sensitive robustness guarantees for sequencing with unknown packing and covering constraints. In *ITCS*, pages 495–504. ACM, 2013.

63 Nicole Megow, Martin Skutella, José Verschae, and Andreas Wiese. The power of recourse for online MST and TSP. *SIAM J. Comput.*, 45(3):859–880, 2016.

64 Morteza Monemizadeh. Dynamic maximal independent set. *arXiv preprint*, 2019. `arXiv:1906.09595`.

65 Marcin Mucha, Karol Wegrzycki, and Michal Wlodarczyk. A subquadratic approximation scheme for partition. In *SODA*, pages 70–88. SIAM, 2019.

66 Adam Polak, Lars Rohwedder, and Karol 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.

67 Donguk Rhee. Faster fully polynomial approximation schemes for knapsack problems. Master's thesis, Massachusetts Institute of Technology, 2015.

68 Peter Sanders, Naveen Sivadasan, and Martin Skutella. Online scheduling with bounded migration. *Math. Oper. Res.*, 34(2):481–498, 2009.

69 Martin Skutella and José Verschae. Robust polynomial-time approximation schemes for parallel machine scheduling with job arrivals and departures. *Math. Oper. Res.*, 41(3):991–1021, 2016.

70 Gang Yu. On the max-min 0-1 knapsack problem with robust optimization applications. *Oper. Res.*, 44(2):407–415, 1996.