

Removable Online Knapsack and Advice

Hans-Joachim Böckenhauer  

Department of Computer Science, ETH Zürich, Switzerland

Fabian Frei  

CISPA Helmholtz Center for Information Security, Saarbrücken, Germany

Peter Rossmanith  

Department of Computer Science, RWTH Aachen, Germany

Abstract

In the *proportional knapsack* problem, we are given a knapsack of some capacity and a set of variably sized items. The goal is to pack a selection of these items that fills the knapsack as much as possible. The *online* version of this problem reveals the items and their sizes not all at once but one by one. For each item, the algorithm has to decide immediately whether to pack it or not. We consider a natural variant of this online knapsack problem, which has been coined *removable knapsack*. It differs from the classical variant by allowing the removal of any packed item from the knapsack. Repacking is impossible, however: Once an item is removed, it is gone for good.

We analyze the *advice complexity* of this problem. It measures how many *advice bits* an omniscient oracle needs to provide for an online algorithm to reach any given *competitive ratio*, which is – understood in its strict sense – just the algorithm’s approximation factor. The online knapsack problem is known for its peculiar advice behavior involving three jumps in competitiveness. We show that the advice complexity of the version with removability is quite different but just as interesting: The competitiveness starts from the golden ratio when no advice is given. It then drops down to $1 + \varepsilon$ for a constant amount of advice already, which requires logarithmic advice in the classical version. Removability comes as no relief to the perfectionist, however: Optimality still requires linear advice as before. These results are particularly noteworthy from a structural viewpoint for the exceptionally slow transition from near-optimality to optimality.

Our most important and demanding result shows that the *general knapsack* problem, which allows an item’s value to differ from its size, exhibits a similar behavior for removability, but with an even more pronounced jump from an unbounded competitive ratio to near-optimality within just constantly many advice bits. This is a unique behavior among the problems considered in the literature so far.

An advice analysis is interesting in its own right, as it allows us to measure the information content of a problem and leads to structural insights. But it also provides insurmountable lower bounds, applicable to any kind of additional information about the instances, including predictions provided by machine-learning algorithms and artificial intelligence. Unexpectedly, advice algorithms are useful in various real-life situations, too. For example, they provide smart strategies for cooperation in winner-take-all competitions, where several participants pool together to implement different strategies and share the obtained prize. Further illustrating the versatility of our advice-complexity bounds, our results automatically improve some of the best known lower bounds on the competitive ratio for removable knapsack with randomization. The presented advice algorithms also automatically yield deterministic algorithms for established deterministic models such as knapsack with a resource buffer and various problems with more than one knapsack. In their seminal paper introducing removability to the knapsack problem, Iwama and Taketomi have indeed proposed a multiple knapsack problem for which we can establish a one-to-one correspondence with the advice model; this paper therefore even provides a comprehensive analysis for this up until now neglected problem.

2012 ACM Subject Classification Theory of computation → Online algorithms

Keywords and phrases Removable Online Knapsack, Multiple Knapsack, Advice Analysis, Advice Applications, Machine Learning and AI

Digital Object Identifier 10.4230/LIPIcs.STACS.2024.18

Related Version *Full Version:* <https://arxiv.org/abs/2005.01867>

Funding *Fabian Frei:* Work done in part while at ETH Zürich.



© Hans-Joachim Böckenhauer, Fabian Frei, and Peter Rossmanith;
licensed under Creative Commons License CC-BY 4.0

41st International Symposium on Theoretical Aspects of Computer Science (STACS 2024).

Editors: Olaf Beyersdorff, Mamadou Moustapha Kanté, Orna Kupferman, and Daniel Lokshantov;

Article No. 18; pp. 18:1–18:17



Leibniz International Proceedings in Informatics

LIPICs Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany



1 Introduction

In this first section, we briefly summarize what online algorithms and advice are, then informally present the problem whose advice complexity we will be analyzing, and finally describe several applications of such advice complexity results.

1.1 Online Algorithms and Advice Complexity

Online algorithms receive their input piece by piece and have to determine parts of the solution before knowing the entire instance. This often leaves them unable to compete with offline algorithms, which know the entire input in advance, in a meaningful way. In the *advice* model, we assume an omniscient oracle that provides the online algorithm with some information on how to solve the upcoming instance best. If the oracle can communicate to the algorithm an unlimited amount of such advice, it will of course be able to lead the algorithm to an optimal solution for every instance. The *advice complexity* measures the minimum amount of information necessary for the online algorithm to achieve any given approximation ratio, which is commonly called strict *competitive ratio* or *competitiveness* in this context. Advice complexity is a well-established tool to gauge the information content of an online problem [3, 9, 15]. For a detailed and careful introduction to the theory, we refer to the textbook by Komm [20]. Another classical textbook on online problems is written by Borodin and Yaniv [5]. The trade-off between a low number of transmitted advice bits on the one hand and achieving a good competitive ratio on the other hand has been examined for a wealth of problems – see the survey by Boyar et al. [6] – but one stands out for its peculiar behavior: the knapsack problem.

1.2 Knapsack and Removability

A knapsack instance presents the online algorithm with a sequence of items of different sizes. Upon the arrival of each item, the algorithm has to decide whether to pack it into a knapsack or discard it. The goal is to fill the knapsack as much as possible without ever exceeding the knapsack's given capacity. This problem is sometimes also referred to as the *proportional* or *simple* knapsack problem, as opposed to the *general* knapsack problem, in which every item has not only a size but also a value.¹ In the generalized version, the goal is to maximize the total value of all packed items. With no further specification given, we are always referring to the proportional case.

A variant of the knapsack problem has been proposed by Iwama and Taketomi [17] under the name of *removable* knapsack. In this model, we can discard an item not only when it is first presented to us; we may also remove a packed item from the knapsack at any point. This is possible only once for each item, however; once removed, an item cannot be repacked. As for the classical problem without removability, the capacity of the knapsack may not be exceeded at any point in time. Recently, Rossmanith has introduced a similar relaxed online setting for graph problems where decisions are taken only when constraints make it inevitable [23].

¹ It is also quite common for the proportional and general knapsack problem to be called unweighted and weighted, respectively. The notion *weight* is ambiguous, however, as some authors [4] use it for what is called size here, while others [18] use it for what is called the value here or profit elsewhere. For the sake of clarity, we are well advised to avoid the term weight altogether.

This model is arguably just as natural a way to translate the knapsack problem into the online setting as the more well-examined variant without removability. In many cases, it will not be hard to discard items at regular intervals, only the chance of obtaining specific objects is subject to special circumstances. For a practical example, consider a storage room in which you can store all kinds of objects that you come across over time. In the beginning you can just keep collecting everything, but by doing so you inevitably run out of space before too long. Then you will have to start disposing of some of your possessions to make room for new, potentially more interesting acquisitions. Your goal is to end up with a selection of just the most meaningful and useful items that you could have. This paper analyzes the advice complexity of both the proportional and general removable knapsack problem. It is telling you how much information about upcoming opportunities you need to ensure an outcome that is either optimal or off by at most a given factor.

1.3 Advice Applications

Besides inherently interesting insights into the information complexity of the knapsack problem, our advice algorithms also offer more concrete applications. Any algorithm reading a bounded number of advice bits can be implemented by running a bounded number of deterministic algorithms in parallel and selecting the best result. An advice analysis thus tells us, for example, how to optimally organize a betting pool in a winner-take-all scenario. Our main result in particular provides a smart selection of strategies to be assigned to a mere constant number of actors such that one is guaranteed to be as close to optimality as we desire, no matter how difficult the instances of the general knapsack problem with removability may become.

A further advantage of analyzing the advice complexity of a problem is that the resulting bounds are very versatile. The lower bounds are particularly strong. They show that a certain competitiveness cannot be achieved with a given amount of additional information, regardless of the form this advice may take. The oracle is indeed able to convey to the algorithm all kinds of structural information about the adversarial instance; for example, in the case of our knapsack problem, whether items smaller than a given threshold should or must not be ignored, whether replacing packed items by later ones will ever be beneficial, whether the values span more than a certain range, whether an optimal solution fills the knapsack completely, whether there are multiple optimal solutions, and so on. Lower bounds on the competitiveness of advice algorithms imply lower bounds for randomized algorithms, and our results indeed improve upon the best bounds known for randomization; Theorem 12 even completely closes the remaining gap in the analysis of barely random algorithms for the general knapsack problem.

There are also interesting implications for deterministic algorithms. Consider the multiple knapsack problem in which every item is either rejected or packed into one of $k > 1$ knapsacks; the goal is for the algorithm to have in the end one knapsack that is as full as possible. This problem has been analyzed with removability by Iwama and Taketomi in the proportional case. In the conclusion of their paper [17], they pose it as an open problem to analyze this model if we are allowed to copy items and pack them into arbitrarily many of the available knapsacks. It turns out that deterministic algorithms for this problem with different k s are equivalent to advice algorithms: An advice algorithm restricted to $\log k$ advice bits can read up to k different advice strings. Even if the algorithm reads the entire advice string right at the beginning, before taking any decision, it will thus implement one of k deterministic strategies. Having k knapsacks and being able to pack each item into several of them at the same time means that we can just simulate each possible strategy in one of the knapsacks

and see which one leads to the best result in the end. Conversely, the oracle in our model already knows the optimal choice and can communicate to an advice algorithm which of the knapsacks it should be simulating. Knowing the advice complexity of our problem with only one knapsack therefore automatically yields a comprehensive competitive analysis for this deterministic problem for any $k > 1$. All of this remains true for the general knapsack problem; thus our results provide a comprehensive picture for the proposed model in both the proportional and non-proportional case. We remark that algorithms for the resource buffer model are generally not applicable here. Algorithm 2 by Han et al. [13, Thm. 9], for example, keeps regrouping items in every step and thus crucially relies on having a single large buffer instead of multiple standard-sized knapsacks without an option to shuffle items between them.

2 Preliminaries

Throughout this paper, $\log$ denotes the binary logarithm. We formally define the removable knapsack problem as follows.

► **Definition 1 (Removable Knapsack Problem).** REMKNAP is an online maximization problem. An instance I is a sequence $(s_1, v_1), \dots, (s_n, v_n)$ of n items, each of which is a pair of some real positive size s_i and value v_i . Where useful, we denote size and value of an item i functionally by $s(i) = s_i$ and $v(i) = v_i$. The domain of this function naturally extends to arbitrary subsets $T \subseteq \{1, \dots, n\}$ of items by defining $s(T) = \sum_{i \in T} s(i)$ and $v(T) = \sum_{i \in T} v(i)$. The knapsack has a maximum size capacity, which we normalize to be 1.

The instance is presented item by item to an online algorithm $\mathcal{A}$ that has to maintain a packing, a set of packed items. We call the total size of the currently packed items the current filling of the knapsack. The algorithm starts out with an empty knapsack, represented by the empty set $T_0 = \emptyset$. When presented with item i , the algorithm may first remove any of the items T_{i-1} packed so far; then it may pack the new item if this does not exceed the knapsack capacity. In other words, the algorithm selects a subset $T_i \subseteq T_{i-1} \cup \{i\}$ with $s(T_i) \leq 1$ in step i . The algorithm learns the size of item i only once it is presented and only learns the total number n of items after selecting T_n . The final packing computed by $\mathcal{A}$ is denoted by $T = T_n$. The gain that we aim to maximize is the total value $v(T)$ of the final packing.

The proportional variant PROPREMKNAP additionally satisfies $s_i = v_i$ for each item i .

► **Definition 2 (Competitive Ratio).** Let an online maximization problem with instance set $\mathcal{I}$ be given and let $\mathcal{A}$ be an online algorithm solving it. For any instance $I \in \mathcal{I}$, denote by $\text{alg}(I)$ the gain that $\mathcal{A}$ achieves on I and by $\text{opt}(I)$ the gain of an optimal solution to I computed offline. The competitive performance of $\mathcal{A}$ on an instance $I \in \mathcal{I}$ is $\text{opt}(I)/\text{alg}(I)$. For any $\rho \in \mathbf{R}$, the algorithm $\mathcal{A}$ is called strictly ρ -competitive if it performs ρ -competitively across all instances, that is, if $\forall I \in \mathcal{I}: \text{opt}(I)/\text{alg}(I) \leq \rho$. The infimal competitiveness $\inf\{\rho \in \mathbf{R} \mid \mathcal{A} \text{ is strictly } \rho\text{-competitive}\}$ is called strict competitive ratio of $\mathcal{A}$. We can weaken the defining inequality above so that it only needs to hold asymptotically in the sense of $\exists \alpha \in \mathbf{R}_+: \forall I \in \mathcal{I}: \text{opt}(I) \leq \rho \cdot \text{alg}(I) + \alpha$. If this condition is met, we call $\mathcal{A}$ nonstrictly ρ -competitive.

Note that strict ρ -competitiveness implies nonstrict ρ -competitiveness but not vice versa, making it harder to prove lower bounds for nonstrict competitiveness. For the knapsack problem, however, it makes sense to always analyze competitiveness in the strict sense: On the one hand, we obtain a nonstrict lower bound from a strict one by scaling up the knapsack capacity and all item sizes in a hard instance set such that the smallest item is strictly larger than α .

If, on the other hand, scaling is impossible due to the problem being defined with a fixed knapsack capacity of 1, for example, then choosing $\alpha = 1$ shows any online algorithm is 1-competitive in the nonstrict sense.

3 Related Work

Knapsack is one of the 21 NP-complete decision problem in Karp's famous list [19]. An algorithm based on dynamic programming solves both the proportional and the general version in pseudo-polynomial time; see Bellman [1, Section 1.4] for the general technique and Dantzig [8, p. 275] for a concrete description of its application to the knapsack problem. The pseudo-polynomial time algorithm can be adapted to the optimization version, yielding a fully polynomial-time approximation scheme [16]. In the following two subsections, we list the known results on the advice complexity of the proportional knapsack problem, first for the classical version and then for the variant allowing the removal of packed items.

3.1 Knapsack without Removability

Marchetti-Spaccamela and Vercellis were the first to consider the classical online version of the knapsack problem in 1995. They called it the $\{0, 1\}$ *knapsack problem* to distinguish it from the *fractional* knapsack problem, which allows for packing items partially. They proved that both versions have an unbounded competitive ratio if items are allowed to have sizes different from their values [21, Thm. 2.1]. We denote the classical problem with neither fractional items nor removability by KNAP and its proportional variant by PROPKNAP.

The concept of advice emerged much later. When it did, KNAP quickly became one of the prime examples of a problem with an interesting advice complexity.

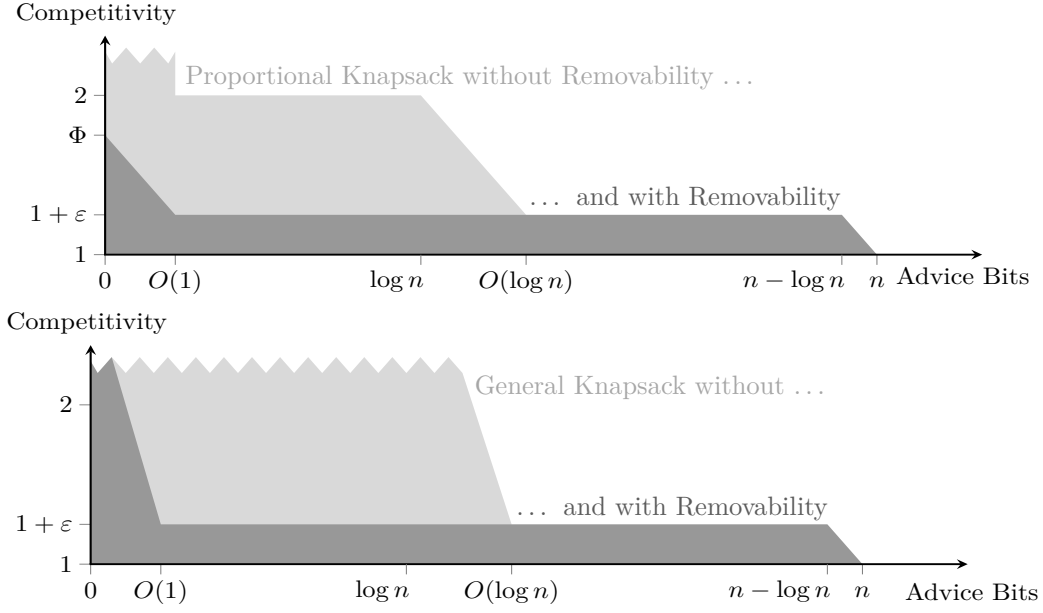
First, just a single advice bit brings with it a jump from non-competitiveness to a 2-competitive algorithm [4, Thm. 4]. More advice bits do not help however, as long as the number stays below $\lfloor \log(n - 1) \rfloor$ [4, Thm. 5]. Once this threshold is surpassed, logarithmic advice allows for a competitive ratio that is arbitrarily close to 1 [4, Thm. 6]. Achieving optimality, finally, requires at least $n - 1$ advice bits [4, Thm. 3].

The situation for the general variant is simpler: Any algorithm reading less than $\log n$ advice bits has an unbounded competitive ratio, but $\mathcal{O}(\log n)$ advice suffices for a near-optimal solution [4, Thms. 11 and 12]. A schematic plot of the advice complexity behaviors just described can be found in Figure 1 in light gray.

3.2 Online Knapsack Variants

Iwama and Taketomi [17] proposed the online knapsack model with removability as it is examined in the present paper. They proved that the competitive ratio for the proportional variant of this problem, which we denote by PROPREMKNAP, is exactly the golden ratio. Iwama and Zhang later considered the problem with resource augmentation, that is, for online algorithms that may use a larger knapsack than the offline algorithm [18].

Later still, Han et al. [12] proved an upper bound of $5/3$ on the competitive ratio for a variant of PROPREMKNAP where the value v of an item is not necessarily proportional to its size s but not arbitrary either; instead, the value is given by a convex function $v = f(s)$ known to the algorithm. They also proved the golden ratio to be optimal if f has some further technical properties. Han et al. [10] considered online knapsack with removal costs, a variant of PROPREMKNAP where items can be removed, but not for free.



■ **Figure 1** A schematic plot of the advice complexity behavior of the classical online knapsack problem in light gray and the relaxed variant with removability in dark gray. For the proportional version without removability there are two large plateaus; removability collapses to a single vast expanse. For the general version, in which an item's value may differ from its size, there is only one but a more extreme jump directly from an unbounded competitive ratio to near optimality; with removability, this jump is occurring earlier and even steeper.

Noga and Sarbua [22] considered a knapsack variant, where it is possible to split each arriving item in two parts of not necessarily equal size, and combine this with resource augmentation. Han and Makino [14] considered another partially fractional variant of PROPREMKNAP where each item can be split a constant number of times at any time. Most importantly in our context, Han et al. [11] examined randomized algorithms for PROPREMKNAP, proving an upper bound of $10/7$ and a lower bound of $5/4$ on the expected competitiveness. Cygan et al. [7] extended the study of randomization for PROPREMKNAP to a variant with multiple knapsacks. Recently, Böckenhauer et al. [2] have introduced a new model for the online proportional knapsack problem in which items can be stored outside of the knapsack until the instance ends after paying a reservation fee that is a fixed fraction α of the item's value.

4 Results for Proportional Removable Knapsack

In Section 4.1, we consider how much – or rather, how little – removability helps when trying to obtain an optimal solution. In Section 4.2, we prove bounds on what is possible with a single advice bit. Finally, we prove in Section 4.3 that a constant amount of advice is sufficient to achieve a competitive ratio of $1 + \epsilon$, for an arbitrary $\epsilon > 0$, and a constant depending on ϵ . See Figure 1 for a rough representation of these results in dark gray.

4.1 Achieving Optimality

We begin by briefly considering PROPKNAP, the classical proportional knapsack problem without removability. Solving it optimally is trivial with n advice bits: The algorithm reads one bit per item, telling it whether to accept or reject. Theorem 3 proves this to be tight by lifting the best known lower bound from $n - 1$ advice bits [4, Thm. 3] to n advice bits. Proofs immediately follow the theorems or are deferred to the full version of the paper.

► **Theorem 3.** *Any algorithm for PROPKNAP reading less than n advice bits is suboptimal.*

Having determined PROPKNAP's advice complexity for optimality, we now do the same for PROPREMKNAP, the variant with removability. It turns out that the option to remove items hardly helps at all in achieving optimality. We begin the upper bound, which is simple but instructive as to what is possible with removability.

► **Theorem 4.** *There is an optimal algorithm for PROPREMKNAP reading $n - 1$ advice bits.*

Proof. Consider an algorithm that packs the first item without reading any advice bits. For each subsequent item, it reads one advice bit, telling it whether the new item is part of a fixed optimal solution. If so, then the new item is packed; otherwise, it is rejected. The first item, which has been packed without advice, is kept in the knapsack as long as there is enough room for it. If the first item is part of the fixed optimal solution, then it will always fit in beside the other items being packed; otherwise, it will be discarded at some point. Thus the algorithm is able to reproduce the fixed optimal solution exactly. ◀

► **Theorem 5.** *Solving PROPREMKNAP optimally requires more than $n - \log n$ advice bits.*

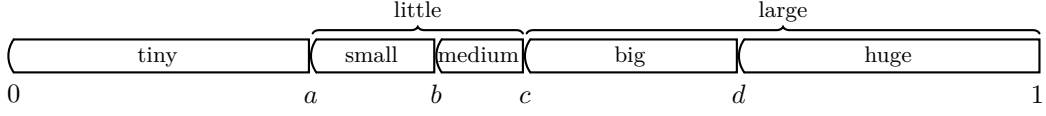
4.2 A Single Advice Bit

The previous section covered the upper end of the advice spectrum, showing that, asymptotically, reading one advice bit for each item in the instance is necessary and sufficient for ensuring an optimal solution. We now turn to the other extreme and ask what can be done with the least nonzero amount of advice, one single bit for the entire instance.

First, we describe a very simple $3/2$ -competitive advice algorithm where a single advice bit indicates whether there is an optimal solution containing more than one item from the interval $[1/3, 2/3]$: If the answer is yes, the algorithm maintains the smallest item in this interval until a second item fits in, while ignoring all items outside of the interval. As soon as a second item fits, it is packed and all remaining items are rejected. If the answer is no, the algorithm maintains in the knapsack the largest item of size at least $1/3$ seen so far while packing all items smaller than $1/3$ as long as they fit. If the knapsack capacity is never exceeded, the solution is optimal. If the knapsack capacity is exceeded at some point, then there are items that are all smaller than $1/3$ but have a total size of more than $1/3$. Discard these items one by one, in arbitrary order, until we are within the capacity of the knapsack again. The remaining gap is at most $1/3$.

Han et al. [11, Thm. 6] have presented a randomized algorithm that relies on a partition of the items into six size classes. It is rather involved and hard to analyze, yet yields an expected competitive ratio of $10/7 \approx 1.428571$. Because it uses only a single random bit, it provides an upper bound for our case of one advice bit as well. In Theorem 6, we undercut this bound with a more manageable $\sqrt{2}$ -competitive algorithm that needs only five classes. We then complement this with a lower bound of $(1 + \sqrt{17})/4 = 4/(\sqrt{17} - 1) \approx 1.2808$.

► **Theorem 6.** *There is a $\sqrt{2}$ -competitive algorithm for PROPREMKNAP reading only one advice bit.*



■ **Figure 2** The partition of the interval $(0, 1]$ of possible sizes into the five subintervals used in the proof of Theorem 6 – namely $(0, a]$, $(a, b]$, $(b, c]$, $(c, d]$, and $(d, 1]$ – plus the corresponding class names. The values are $a = 1 - 1/\sqrt{2} \approx 0.293$, and $b = \sqrt{2} - 1 \approx 0.414$, and $c = 1/2$, and $d = 1/\sqrt{2} \approx 0.707$.

Proof. We split the interval $(0, 1]$ of possible sizes into subintervals at four points $a < b < c < d$. We will call the items with sizes in one of these five intervals *tiny*, *small*, *medium*, *big*, and *huge*, respectively. Formally, we partition the items into the five classes

$$P_{\text{tiny}} = \{i \mid 0 < s(i) \leq a\}, \quad P_{\text{small}} = \{i \mid a < s(i) \leq b\}, \quad P_{\text{medium}} = \{i \mid b < s(i) \leq c\}, \\ P_{\text{big}} = \{i \mid c < s(i) \leq d\}, \quad P_{\text{huge}} = \{i \mid d < s(i) \leq 1\},$$

where $a = 1 - 1/\sqrt{2} \approx 0.29289$, $b = \sqrt{2} - 1 \approx 0.41421$, $c = 1/2$, and $d = 1/\sqrt{2} \approx 0.70711$.

We will call the small and medium items the *little* ones collectively and refer to the big and huge items as the *large* ones. Accordingly, we let $P_{\text{little}} = P_{\text{small}} \cup P_{\text{medium}}$ and $P_{\text{large}} = P_{\text{big}} \cup P_{\text{huge}}$. See Figure 2 for an illustration of the subintervals and class names.

The oracle uses the one available advice bit to tell the algorithm which of the two strategies described below to apply. For the decision, the oracle picks an arbitrary optimal solution S to the given instance. If S contains a large item, the first strategy will be chosen, with one exception: If the instance contains no huge item but a little and a big item that fit into the knapsack together, then the first strategy is chosen only if a minimal big item appears in the instance before a minimal small item. In all other cases, the second strategy is implemented.

Strategy One If at any point a huge item appears, the algorithm packs it and keeps it until the end, discarding everything else. Otherwise, the algorithm operates with two slots, a primary and a secondary one. In the primary slot, it maintains the minimal big item and in the secondary slot it maintains the minimal little item. The primary slot takes precedence; that is, in case of a conflict where a new minimal item for one slot is presented that does not fit with the minimal item in the other slot, we discard the little item.

While maintaining the slot contents, tiny items are always packed greedily. If at any point a presented tiny item does not fit, the current contents of the knapsack are frozen and kept as they are until the instance has ended. The same happens after a step in which only tiny items have been discarded.

Strategy Two This strategy manages not only two but three slots, all of which maintain minimal items of some class. In order of precedence, the primary slot maintains two medium items, the secondary slot up to three small items, and the tertiary one big one. As an exception, if at any point a big item appears that can be packed alongside a currently packed small item by discarding everything else, then this is done and these two items are kept till the end. The tiny items are handled as before: They are packed greedily and if either a presented tiny item does not fit or only tiny items have been discarded in one step, then the current knapsack configuration is kept up to the very end.

We now need to carefully work through a case distinction according to the conditions listed in Table 1 and show that the algorithm's competitiveness is indeed bounded from above by $\max\{1/d, d/c, 1/2b, 1/(a+b), 1/(1-a), b/a\} = \sqrt{2}$. For the details, we refer to the full version of the paper. ◀

■ **Table 1** The mutually exclusive cases considered in Theorem 6.

Case	Strategy	Competitiveness	Case Conditions			
A	One	$1/d$	$ P_{\text{huge}} > 0$			
B	One/Two	d/c	$ P_{\text{huge}} = 0$	$ S \cap P_{\text{big}} > 0$	$ P_{\text{medium}} \leq 1$	
C	Two	$1/2b$	$ P_{\text{huge}} = 0$	$ S \cap P_{\text{big}} \geq 0$	$ P_{\text{medium}} > 1$	
D	Two	$1/(a+b)$	$ P_{\text{huge}} = 0$	$ S \cap P_{\text{big}} = 0$	$ P_{\text{medium}} = 1$	$ P_{\text{small}} > 0$
E	Two	b/a	$ P_{\text{huge}} = 0$	$ S \cap P_{\text{big}} = 0$	$ P_{\text{medium}} = 0$	$ P_{\text{small}} > 0$
F	Two	$1/(1-a)$	$ P_{\text{huge}} = 0$	$ S \cap P_{\text{big}} = 0$	$ P_{\text{medium}} \leq 1$	$ P_{\text{small}} = 0$

► **Theorem 7.** *No algorithm for PROP-REMKNAP reading only a single advice bit can have a better competitive ratio than $(1 + \sqrt{17})/4$.*

Note again that an advice bit is at least as powerful as a random bit, hence Theorem 7 also improves the best known lower bound of $5/4$ for one random bit due to Han et al. [11, Thm. 8].

4.3 Near Optimality with Constant Advice

Having seen how much advice is necessary for optimality and what the effect of a single advice bit can be, we now address the entire range in between. For this, we prove the following generalization of Theorem 7.

► **Theorem 8.** *Let an arbitrary integer $k > 1$ be given. No algorithm for PROP-REMKNAP reading at most $\log k$ advice bits can achieve a better competitive ratio than $4/(3 - 2k + \sqrt{4k(k+1) - 7})$.*

We remark that Theorem 8 and its analogue for REMKNAP instead of PROP-REMKNAP, Theorem 13, improve upon the best known lower bounds implied by Han et al.’s results on the resource buffer model [13, Thms. 17 and 6]. In this model, the online algorithm may use a knapsack of some increased capacity $R > 1$, but only until the instance ends, at which point it has to choose from the reserved items a selection that fits a knapsack of capacity one. A resource buffer of some natural size R allows us to simulate any algorithm using up to $\log R$ advice bits: We think of the resource buffer as split into R knapsacks of capacity 1, allowing us to accommodate the items stored by the advice algorithm for every possible advice string simultaneously.

Clearly, the lower bound of Theorem 8 tends to 1 for increasingly large but still constant advice. With our most surprising result for the proportional knapsack problem, Theorem 10, we will prove that the true competitive ratio displays the same general behavior as the lower bound of Theorem 8: For any given $\varepsilon > 0$, we can guarantee a competitive ratio of $1 + \varepsilon$ with a constant number of advice bits. It is of course also possible to derive more specific upper bounds for very few advice bits such as the following one.

► **Theorem 9.** *There is a $4/3$ -competitive algorithm for PROP-REMKNAP reading two advice bits.*

We now turn to our main result for the proportional knapsack problem, which complements Theorem 8 with an upper bound. Theorem 14 will generalize this result to the general version where an item’s size may differ from its value, albeit with a far more complicated proof. To

make it as easily understandable as possible, we first present here the proof for the simple variant, which introduces the idea of slots that are reserved for items with certain properties. This will serve as a useful foundation for the proof of the general variant, which is also making use of such a slot system, although as merely one besides many more components.

► **Theorem 10.** *For any $\varepsilon > 0$, there is a strictly $(1 + \varepsilon)$ -competitive algorithm for PROPKNAP reading a constant number of advice bits.*

Proof. We describe such an algorithm called PROPPACK; see the full version of the paper for a pseudo-code implementation. We begin by describing the advice communicated to PROPPACK with a constant number of bits, then explain how the algorithm operates on this advice, prove that it is correct and terminates, and finally analyze its competitive ratio.

Notions and Notation. Without loss of generality, we assume that all items have size at most 1 and that $\varepsilon \leq 1/2$. We define the constant $K = \lceil \log_{1-\varepsilon/2} \varepsilon/2 \rceil$.

Let an instance with n items be given. Denote the items in the order of their appearance in the instance by $1, 2, \dots, n$ and denote the size of item i by $s(i)$. We divide the n items into *small* and *big* ones, with $\delta = (1 - \varepsilon/2)^K$ serving as the dividing line: $C_{\text{small}} = \{i \mid s(i) \leq \delta\}$ and $C_{\text{big}} = \{i \mid \delta < s(i)\}$. We further partition the big items into the subclasses $C_k = \{i \mid (1 - \varepsilon/2)^k < s(i) \leq (1 - \varepsilon/2)^{k-1}\}$ for $k \in \{1, \dots, K\}$. To alleviate the notation, we will often refer to C_k as class k and to C_{small} as class 0. We also use this convention when writing $C(i)$ to indicate the class to which item i belongs: We have $C(i) \in \{0, \dots, K\}$, with $C(i) = 0$ meaning that $i \in C_{\text{small}}$ and $C(i) = k \neq 0$ meaning that $i \in C_k$.

The oracle chooses an arbitrary but fixed optimal solution $S \subseteq \{1, \dots, n\}$. We denote the partition classes that are naturally induced by this solution by $S_{\text{small}} = S \cap C_{\text{small}}$, $S_{\text{big}} = S \cap C_{\text{big}}$, and $S_k = S \cap C_k$ for $k \in \{1, \dots, K\}$. Let $m = |S_{\text{big}}|$ be the number of big items in the optimal solution and denote them by $i_1 < \dots < i_m$ in order of appearance.

Constant Advice. The oracle communicates to the algorithm a tuple $(b_1, \dots, b_m)$ with the classes of the big items in the chosen optimal solution in order of appearance; that is, we have $b_j = C(i_j)$ for each $j \in \{1, \dots, m\}$. We remark that this tuple needs to be encoded in a self-delimiting way. A constant number of bits suffices for this because b_j is bounded by the constant K for every $j \in \{1, \dots, m\}$ and m is bounded by the constant $1/\delta$. The latter bound is an immediate consequence of the fact that $s(S_{\text{big}}) \leq 1$ and that any big item has a size larger than δ .

Algorithm Description. The algorithm PROPPACK proceeds in m phases as follows. In every phase, the algorithm opens a new virtual *slot* within the knapsack that can store exactly one item at a time; multiple items in succession are allowed, however. The slot opened in phase i will accommodate items belonging to class b_i exclusively; we say that items from this class *match* slot i . Slots are never closed, thus there are exactly m of them in the end. Small items are generally packed in a greedy manner and discarded one by one whenever necessary to pack a big item.

In the first phase, the algorithm rejects all big items until one of class b_1 appears. As soon as this is the case, said item is packed into the first slot, ending the first phase.

In the second phase, the algorithm opens the second slot to pack a matching item, that is, one of class b_2 . It waits for the first item from this class that fits into the knapsack alongside the item in the first slot. As soon as such an item appears, it is packed and the

phase ends. In the meantime, whenever an item of class b_1 appears during the second round, the algorithm substitutes it for the one stored in the first slot if and only if this reduces the size of the stored item.

In general, phase i begins with the opening of slot i , which is reserved for items of class b_i . The phase continues until an item appears that both matches the newly opened slot and fits in beside the items currently stored in the previously opened and filled slots without exceeding the capacity. Then this item is packed into the new slot, which ends the phase. During the entire phase, the algorithm maintains in all filled slots the smallest matching items seen so far: Whenever the algorithm is presented with a big item that either belongs to a class other than b_i or does not fit in alongside the items in the previously opened slots, then the new item replaces a largest item in the matching open slots, unless the new item itself is even larger.

The entire time, even after the last phase has terminated, small items are packed greedily and discarded one by one whenever this is necessary to make room for a big item according to the description above. Moreover, we may assume that, whenever a new item has been packed into the knapsack, the algorithm sorts the items in the matching open slots in increasing order. This sorting is not necessary for the algorithm to fulfill its duty, but it facilitates the proof by induction below.

Termination of All Phases. We need to show that PROPPACK does in fact finish all m phases; that is, all m slots will be filled with a matching item without ever exceeding the knapsack capacity. Consider the big items of the optimal solution, which we denote by $u_1 < \dots < u_m$ in their order of appearance. To ensure the termination of all phases, we prove by induction over $i \leq m$ that, after processing item u_i , the first i slots store items with a total size of $s(u_1) + \dots + s(u_i)$ or less.

We may start from $i = 0$ as the trivial, if degenerate, base case. For the induction step, assume the hypothesis for $i < m$ and observe that no item in a slot is ever replaced by a larger one. Therefore, the items in the first i slots still have a total size of at most $s(u_1) + \dots + s(u_i)$ when u_{i+1} is presented.

There are now three possibilities. If slot $i + 1$ has remained closed up to this point, it is now opened and filled with u_{i+1} , which fits in because $s(u_1) + \dots + s(u_{i+1}) \leq s(S_{\text{big}}) \leq 1$. Otherwise, slot $i + 1$ is already storing an item: If said item is larger than $s(u_{i+1})$, then u_{i+1} replaces either this item or one that is at least as large. During the subsequent sorting, u_{i+1} is then moved to slot $i + 1$ or one of the slots from 1 to i , which may force some items from slots 1 through i into higher slots but never beyond slot $i + 1$. The third possibility is that slot $i + 1$ contains an item of size at most $s(u_{i+1})$ already. We immediately obtain the induction claim for $i + 1$ in all three cases.

Competitive Analysis. We still denote by S the optimal solution that served as the basis for the given advice, by T the final output of the online algorithm, and the respective partition classes by S_{small} , S_{big} , S_k and T_{small} , T_{big} , and T_k .

Since PROPPACK opens one slot for each big item in the optimal solution T and fills it with an item from the same subclass, as proved above, we have $|S_k| = |T_k|$ for every $k \in \{1, \dots, K\}$. Moreover, the sizes within a subclass C_k vary by a factor of at most $1 - \varepsilon/2$; this means that we can bound both $s(S_{\text{big}})$ and $s(T_{\text{big}})$ from below by $L = \sum_{k=1}^K |S_k|(1 - \varepsilon/2)^k$ and from above by $L/(1 - \varepsilon/2)$. We conclude $s(T_{\text{big}}) \geq s(S_{\text{big}}) \cdot (1 - \varepsilon/2)$.

Furthermore, since small items are packed greedily and only discarded one by one whenever necessary to make room for the big items, we will not lose much from their side either. If the presented small items have a total size of at most $1 - L/(1 - \varepsilon/2)$, none is ever discarded.

In this case, we have $s(T_{\text{small}}) \geq s(S_{\text{small}})$ and thus immediately $s(T) \geq s(S) \cdot (1 - \varepsilon/2)$. If small items are discarded, however, the worst case is the following type of instance: It starts with only small items of the largest possible size δ , some of which are then discarded to accommodate big items with sizes right at the upper limit for the classes indicated by the advice, leaving a gap of almost δ , follows up with slightly smaller big items that are in the optimal solution and would not have lead to any discarded small items, and finally presents big items at the lower end of the size span, replacing all previously packed big items.

Even in this worst case, the algorithm remains $(1 - \varepsilon/2)$ -competitive on the big items and detracting the largest possible loss of δ on the small items yields $s(T) \geq s(S) \cdot (1 - \varepsilon/2) - \delta$. By the definition of δ and K and due to the simple fact that $s(S)$ is at most 1, we have $\delta = (1 - \varepsilon/2)^K \leq \varepsilon/2 \leq s(S) \cdot \varepsilon/2$. This implies $s(T)/s(S) \geq 1 - \varepsilon$, as desired. ◀

5 Results for General Removable Knapsack

First, we note that all lower bounds for the proportional removable knapsack problem carry over to the general removable knapsack problem, in particular Theorem 5.

Iwama and Zhang [18] have shown that the competitive ratio of REMKNAP is unbounded without advice. This can be seen using an interactive instance that starts with an item $(1, 1)$ and then presents items $(\varepsilon^2, \varepsilon)$ repeatedly, up to $1/\varepsilon^2$ times, until one is packed, at which point the instance ends.

The following two theorems show REMKNAP's competitiveness for one advice bit to be exactly 2.

► **Theorem 11.** *There is a 2-competitive algorithm for REMKNAP reading only a single advice bit.*

► **Theorem 12.** *No algorithm for REMKNAP reading only a single advice bit can have a competitive ratio better than 2.*

The existence of a 2-competitive algorithm already follows from a result by Han et al. [11, Thm. 9], who proved the statement even for a single random bit instead of an advice bit. We can prove Theorem 11 by describing a concrete advice algorithm. Advice being more powerful than randomness, Theorem 12 also closes the remaining gap for barely random algorithms by lifting the previously best known lower bound by Han et al. [11, Thm. 12] from $1 + 1/e \approx 1.367$ to 2.

The analysis of the hard instance presented in the proof of Theorem 12 could be adapted to the case of more than one advice bit. Two advice bits would mean that the oracle can provide to the algorithm one out of four advice strings instead of the two advice strings possible with one bit. We may also consider an intermediate advice algorithm limited to three advice strings, which corresponds to $\log 3$ advice bits. Such an algorithm cannot achieve a competitive ratio better than $2/\Phi = 4/(1 + \sqrt{5}) \approx 1.2361$ since it is forced to use one of the three advice strings to keep the most valuable item x_0 , one to pack the items $y_1, \dots, y_k$ yield-greedily, and one to keep the x_j with the value $\Phi \approx 1.618$, resulting in a competitive ratio of $2/\Phi = 2\Phi/(1 + \Phi)$.

This approach deteriorates too quickly, however. Adapting Theorem 8 to the case of REMKNAP is the better choice; this results in Theorem 13, which already yields a better bound in the case of three advice strings, that is, for $k = 3$.

► **Theorem 13.** *Let an arbitrary integer $k > 1$ be given. No algorithm for REMKNAP reading at most $\log k$ advice bits can achieve a better competitive ratio than $1/2 + \sqrt{1/4 + 1/k}$.*

We point out again that Theorem 13 slightly improves over the lower bounds known from the resource buffer model by Han et al. [13, Thm. 6], as illustrated after the proof of Theorem 13 in the full version of the paper.

We now move on to the core result of this paper, proving that a constant amount of advice bits is sufficient to reach a near-optimal competitive ratio not only for the proportional but even for the general removable knapsack problem. This will complete the picture of the global advice behavior of the online knapsack problem with removability outlined in Figure 1.

We first point out that algorithm PROPPACK generally does not work on instances where the value of an item can vary independently of its size, as seen by the following counterexample: Assume that $1/2$ lies in the interior of some size class and choose an $\varepsilon > 0$ such that $1/2 + \varepsilon$ and $1/2 - \varepsilon$ are still in the same class. Present two items $(1/2 + \varepsilon, 2)$ and $(1/2, 1)$. If the algorithm picks the first one, $(1/2, 2)$ is presented as the last item; otherwise, the instance ends with the item $(1/2 - \varepsilon, 1)$. In both cases, the algorithm achieves a total value of 2, whereas the optimum is 3. The advice does not help us to distinguish the two cases, it only tells us that the optimal solution contains two items from the class. Clearly, we have to adapt the algorithm to take the value of the items into account somehow. A major obstacle is that the online algorithm has no bound on the values of appearing items, thus the algorithm has no way of reconstructing the constantly many value classes used by the oracle just from the parameter ε .

Moreover, the proof of Theorem 10 cannot be adapted for the general case in any simple way. Using only classes based on size, it is impossible for the algorithm to know, when maintaining an item in a slot, how to balance minimizing the size against maximizing the value. On the one hand, if the size is not minimized, then the excess size may prevent other slots from being filled. On the other hand, not maximizing the values, the algorithm may incur an arbitrarily high loss because the potential values of items cannot be bounded. This is also the reason why simple value classes do not work either. The algorithm does not know the maximal value occurring in the instance until it has ended and can therefore not use it as a reference point, in contrast to the size classes that can be chosen relative to the known knapsack capacity.

A first step toward solving these issues is the definition of dynamic value classes that are anchored to both the value of the first item appearing in the instance and to the optimal solution value. The latter is of course also unknown to the algorithm until the instance ends. However, we are able to define our classes with some additional properties that enable our algorithm to compute at any point useful provisional bounds on the optimal solution value. These bounds will either turn out to be valid or the algorithm is able to notice that they are off just in time to adjust and take a fresh start before having lost too much due to bad decisions. The adversary may foil the algorithm over and over, forcing it to abandon its plans and adjust the bounds arbitrarily often.

To properly deal with these repeated resets, we develop a *level system*. One major challenge is to square the level system with some sort of *slot system* as used by the algorithm for the proportional case. We manage to do this by introducing the concept of a *virtual algorithm*, which has the special, even though only imagined, capability of keeping one item in a *splitting slot* and use arbitrary fractions of the item stored in it. We then describe an actual algorithm that tries to equal the idealized performance of the virtual algorithm without making use of the splitting slot. While it cannot quite achieve this, it will fare well enough in the end. Having one algorithm emulating another, we are going to prove the claimed competitiveness in two stages, first for the virtual version and then for the actual algorithm.

This split analysis presents several further challenges, for example, a desynchronization of the current phase and the number of slots filled by the algorithm, which coincided in the proportional case. The necessary adaptations entail a number of further challenges, for example a judicious handling of the *paltry* items, which are worth almost nothing individually, yet may be too numerous to neglect. In fact, the algorithm will need to partition the items not only by their value but simultaneously by their size as well. Overcoming these and a few other obstacles, we are able to prove in the full version of the paper our final theorem.

► **Theorem 14.** *For any $\varepsilon > 0$, there is a strictly $(1 + \varepsilon)$ -competitive algorithm for REMKNAP reading a constant number of advice bits.*

Proof. Proving this theorem requires far more effort than what was necessary for its proportional counterpart Theorem 10, where an item's value is always identical to its size. Having explained already why any straightforward adaption of the substantially simpler approach for the proportional variant is impossible, we now provide a high-level outline of the proof.

High-Level Outline. As announced, we first provide a high-level outline of the workings of the advice algorithm.

The oracle chooses an arbitrary optimal solution S and, based on its value $v(S)$, partitions the items of the instance into *precious* and *paltry* ones; the paltry ones are those worth less than $\varepsilon_{\text{paltry}} \cdot v(S)$ for a suitable $\varepsilon_{\text{paltry}} > 0$. The precious items are further split into finitely many classes such that the item values within any class are at most some factor $1 - \varepsilon_{\text{spread}}$ apart from each other.

The advice will encode exactly the classes of the precious items from the optimal solution S in their order of appearance; the goal is to ensure that the algorithm packs just as many precious items from each class as the optimal solution does, thus achieving a competitive factor of $1/(1 - \varepsilon_{\text{spread}})$ on these items. Assuming that the algorithm knows the exact value ranges of each class, it can achieve this using a system of *slots*, which will be filled with precious items in two stages: first just *virtually* – assuming the algorithm were able to do certain things that are in fact impossible – and then also *actually* at some point. Each virtual filling of a slot starts a new *phase* of the algorithm.

However, the algorithm knows only the target value but nothing about the size of the items belonging into each slot; it is necessary to prove that despite this, the algorithm is able to fill the slots in the right order without blocking important items yet to come.

And there is another problem: It is even impossible for the oracle to communicate to the algorithm the exact value range of each class with a constant amount of advice since there is no bound on the potential values occurring in an instance. Instead, the value ranges will be described in relation to the value of the first item of the instance, and merely modulo some constant factor. The algorithm will then operate under the assumption that the first item is a precious one, in which case all of the above will work out. Since the algorithm cannot know for sure which items are precious, it divides them into *presumably precious* and *provenly paltry* ones according to some computations based on the advice and the instance seen so far. If the algorithm's assumption is mistaken, it is able to recognize this just in time by continually comparing the best solution realizable with the already presented items – whether they have been accepted or not – to a rather intricate estimate for the optimal solution value $v(S)$. Once the algorithm discovers its mistake, it resets with a revised set of assumptions on the value ranges; we say that the algorithm *levels up*. This is done such that the algorithm can go through arbitrarily many levels, resetting and taking a fresh start as often as necessary without incurring more than a negligible value loss.

The algorithm also needs to take care of the paltry items, which might constitute a considerable part of the optimal solution if there are sufficiently many of them. The algorithm is packing the paltry items in a somewhat inhibited greedy manner that optimizes the value-to-size ratio, which we also call *yield*. The volume taken up by the paltry items will be restricted sufficiently to guarantee that the precious items can always fill their slots, but not as severely as to lose too much value on the paltry items. The right volume restrictions in each phase of the algorithm are communicated via a constant amount of advice as well. Again, this cannot happen directly, since the right volume range might be infinitesimally small. Instead, the volume is controlled indirectly, via bounds not on some size but instead on the value that is provided by the paltry items packed before filling the current slot with a precious item.

Communicating the necessary volumes in this way is possible only up to some precision $\varepsilon_{\text{round}}$, and an overestimation could mean the loss of a crucial precious item. If we always round down, then this cannot happen, but the algorithm might reject some paltry item it should have kept for a selection of maximum yield. This is negligible if it happens only once, but we cannot tolerate taking such a loss in every phase, with every new volume bound. The solution is to analyze the situation using a special *splitting slot*, which can accommodate one paltry item at the time outside of the knapsack. We imagine the splitting slot lending us from the item stored in it any desired fraction at any time – just always the same fraction of the value and size. We refer to the item currently stored in the splitting slot as the *split item*. We will consider an algorithm that maintains a split item of highest yield after the remaining paltry items kept in the knapsack. This imaginary algorithm is an antecedent to our real advice algorithm, which builds on it but cannot actually split any items of course; we refer to the purely hypothetical precursor as the *virtual version* of our *actual algorithm*.

The actual algorithm will mimic the virtual version as closely as possible and deliver a result that is only marginally worse. Whenever the virtual version splits an item, the actual algorithm needs to decide whether to discard this item or store it completely. The challenge is to take the right decisions to allow for all slots to be filled in time and also avoid an undue accumulation of losses by passing on too many split items.

The advice helps the actual algorithm by indicating for every phase whether an item stored in the splitting slot by the virtual version is to be packed or discarded. This is done in such a way that in the end, the actual algorithm will have relinquished only the value of a single paltry item, namely the one kept in the splitting slot when the instance ends.

Packing entire items from the splitting slot comes with problems on its own; these paltry items might block for the actual algorithm some precious items that are packed by the virtual version. This problem is addressed by further advice to the algorithm on how to prioritize the packing of precious versus paltry items in each phase. This advice, telling the algorithm when to *actualize* a virtual packing of an item, is based on the solution that the virtual version would eventually produce if it existed. The virtual version does not depend on the actual algorithm and has no need for the part of the advice on actualization, which avoids any circular reasoning.

There are a few more technical issues to be dealt with, for example the special case that the precious items contribute only marginally to the value of the optimal solution. This undermines the estimates for the optimal solution value, is thus flagged by a dedicated advice bit b_{small} , and handled by switching from the elaborate value-based limits that dampen the general yield-greedy strategy to a simpler size-sensitive strategy. ◀

References

- 1 Richard Bellman. *Dynamic Programming*. Princeton University Press, Princeton, NJ, USA, 1957.
- 2 Hans-Joachim Böckenhauer, Elisabet Burjons, Juraj Hromkovič, Henri Lotze, and Peter Rossmanith. Online simple knapsack with reservation costs. In Markus Bläser and Benjamin Monmege, editors, *38th International Symposium on Theoretical Aspects of Computer Science, STACS 2021, March 16–19, 2021, Saarbrücken, Germany (Virtual Conference)*, volume 187 of *LIPICs*, pages 16:1–16:18. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2021.
- 3 Hans-Joachim Böckenhauer, Dennis Komm, Rastislav Kráľovič, Richard Kráľovič, and Tobias Mömke. On the advice complexity of online problems. In *ISAAC 2009*, number 5878 in LNCS, pages 331–340, 2009.
- 4 Hans-Joachim Böckenhauer, Dennis Komm, Richard Kráľovič, and Peter Rossmanith. The online knapsack problem: Advice and randomization. *Theoretical Computer Science*, 527:61–72, 2014.
- 5 Allan Borodin and Ran El-Yaniv. *Online computation and competitive analysis*. Cambridge University Press, 1998.
- 6 Joan Boyar, Lene M. Favrholdt, Christian Kudahl, Kim S. Larsen, and Jesper W. Mikkelsen. Online algorithms with advice: A survey. *ACM Comput. Surv.*, 50(2):19:1–19:34, 2017.
- 7 Marek Cygan, Łukasz Jeż, and Jiří Sgall. Online knapsack revisited. *Theory Comput. Syst.*, 58(1):153–190, 2016.
- 8 George B. Dantzig. Discrete-variable extremum problems. *Operations Research*, 5(2):266–277, 1957.
- 9 Yuval Emek, Pierre Fraigniaud, Amos Korman, and Adi Rosén. Online computation with advice. *Theoretical Computer Science*, 412(24):2642–2656, 2011.
- 10 Xin Han, Yasushi Kawase, and Kazuhisa Makino. Online unweighted knapsack problem with removal cost. *Algorithmica*, 70(1):76–91, 2014.
- 11 Xin Han, Yasushi Kawase, and Kazuhisa Makino. Randomized algorithms for online knapsack problems. *Theoretical Computer Science*, 562:395–405, 2015.
- 12 Xin Han, Yasushi Kawase, Kazuhisa Makino, and He Guo. Online removable knapsack problem under convex function. *Theoretical Computer Science*, 540:62–69, 2014.
- 13 Xin Han, Yasushi Kawase, Kazuhisa Makino, and Haruki Yokomaku. Online knapsack problems with a resource buffer. In Pinyan Lu and Guochuan Zhang, editors, *30th International Symposium on Algorithms and Computation, ISAAC 2019, December 8–11, 2019, Shanghai University of Finance and Economics, Shanghai, China*, volume 149 of *LIPICs*, pages 28:1–28:14. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2019.
- 14 Xin Han and Kazuhisa Makino. Online removable knapsack with limited cuts. *Theoretical Computer Science*, 411(44-46):3956–3964, 2010.
- 15 Juraj Hromkovič, Rastislav Kráľovič, and Richard Kráľovič. Information complexity of online problems. In *MFCS 2010*, number 6281 in LNCS, pages 24–36. Springer, 2010.
- 16 Oscar H. Ibarra and Chul E. Kim. Fast approximation algorithms for the knapsack and sum of subset problems. *Journal of the ACM*, 22(4), 1975.
- 17 Kazuo Iwama and Shiro Taketomi. Removable online knapsack problems. In *ICALP 2002*, number 2380 in LNCS, pages 293–305. Springer, 2002.
- 18 Kazuo Iwama and Guochuan Zhang. Online knapsack with resource augmentation. *Information Processing Letters*, 110(22):1016–1020, 2010.
- 19 Richard M. Karp. Reducibility among combinatorial problems. In *Complexity of computer computations*, pages 85–103. Plenum, 1972.
- 20 Dennis Komm. *An Introduction to Online Computation – Determinism, Randomization, Advice*. Texts in Theoretical Computer Science. An EATCS Series. Springer, 2016.
- 21 Alberto Marchetti-Spaccamela and Carlo Vercellis. Stochastic on-line knapsack problems. *Mathematical Programming*, 68:73–104, 1995.

- 22 John Noga and Veerawan Sarbua. An online partially fractional knapsack problem. In *8th International Symposium on Parallel Architectures, Algorithms, and Networks, ISPAN 2005, December 7-9, 2005, Las Vegas, Nevada, USA*, pages 108–112. IEEE Computer Society, 2005.
- 23 Peter Rossmanith. On the advice complexity of online edge- and node-deletion problems. In *Adventures Between Lower Bounds and Higher Altitudes – Essays Dedicated to Juraj Hromkovič on the Occasion of His 60th Birthday*, number 11011 in LNCS, pages 449–462. Springer, 2018.