Semi Online Problems

Von der Fakultät für Informatik der RWTH Aachen University
zur Erlangung des akademischen Grades

eines Doktors der Naturwissenschaften

genehmigte Dissertation

vorgelegt von

Matthias Gehnen, M. Sc.

Berichter:

Univ.-Prof. Dr. rer. nat. Peter Rossmanith

Univ.-Prof. Yossi Azar, Ph.D.

Tag der mündlichen Prüfung: 26.03.2026

Contents

Acknowledgements

I believe that no PhD project can be completed successfully without support and an inspiring environment. This research is no exception, and therefore I owe my thanks to many people around me.

First, I would like to thank my research group:

To Elisabet, who was the main driver of my research in the first year of my PhD, with ideas and encouragement. It took me some more years (and teaching) to realize how valuable colleagues are who help you to not get too distracted.

To Henri, who was in the unfortunate situation of having to endure my constant approaches to start chatting while sharing an office. I am grateful for not only the common research and teaching interests, but also for the high-level debates about, e.g., the goals of education.

To Daniel, in particular for being there in recent years. I found it to be incredibly valuable to have a colleague for everything from discussing the daily business to traveling to conferences.

To Peter, for various things: First and foremost, I am grateful for his continuous trust in me, starting with the offer to start my career as a researcher in his group, but also later by giving me the free rein to conduct research I am interested in, or by allowing me to gain my own experience in teaching and supervising. At the same time, I knew that I always could count on an open door and an open ear to talk about everything.

To Birgit, as without her, I would just have been drowned in administrative matters. I am glad that I could still count on her support even after she left our group.

As someone who has worked in a rather small team, I am particularly thankful for the colleagues outside of my group at RWTH:

To the whole team from the chair of algorithms and complexity, namely to Dennis, Janosch, Svenja, Christoph, Tim, Martin, Lars, Tom, Tim, Jakob, Komal, Sukanya, Rilind, Finn, Erika, Walter, Robin, and Gerhard. Every attempt to count the visits to local restaurants on Fridays or the number of ice creams we shared throughout the year would have been bound to fail.

To Daniel, Stefan, Ira, Martin, Louis, and Nils, as well as dozens of student assistants, with whom I had the pleasure of organizing and teaching the basic lectures in data structures and algorithms or in automaton theory in recent years. Even though we should not understate the workload those courses require, it was still fun and also inspiring to teach them together.

To the many people across the university departments, with whom I had the chance to talk or learn about algorithms and discrete mathematics in various ways: While it is impossible to name everyone, I would particularly like to thank the groups of Christina, Yubao, Arie, Britta, and Eberhard; with their courses

they motivated me to do research in this area myself some years ago, and they are still inspiring me today.

Just as my colleagues were an important driver of the research in this thesis, my students did so as well: The comments, questions, and work during all the seminars, lab courses, or lectures are more helpful to me than most students probably believe. In particular, I would like to thank Julian, Mats, Mira, Aman, Chris, David, Magnus, Jannis, Kübra, Jakob, Christian, Anna, Pascal, Julius, Emilie, Andreas, Luca, and Shiyu for giving me the chance to learn from them as their supervisor for a Bachelor's or Master's thesis. It was a great pleasure.

To my colleagues and collaborators from different institutions: Every time I met Hajo, Fabian, Juraj, Ralf, Dennis, Rastislav, Richard, Émile, Moritz, or Philip, I left with more inspiration and ideas than I could possibly use in my lifetime.

To Yossi, Martin, and Ulrik, who volunteered to be in my committee: Thank you for the time and effort you invested for my defense! In particular I would like to thank Yossi, as I feel really honored that someone, whose work had a significant impact on my thesis, volunteered to be my second examiner without any compensation.

But most importantly, I feel like I never would have been able to finish this project without the amazing support I received from my friends and family. I would easily double the pages of this thesis if I attempted to put into words what this support means to me. Just the pages dedicated to Christina, Gerrit, Marion, and Stephanie would be equivalent to at least a chapter each.

Thank you.

Abstract

Assume you have an arbitrary problem. Likely, you will then try to find a good solution for it. If you are fully aware of the situation, the future, and your options with their consequences, making a good decision is usually possible. However, often you do not have all the necessary information but need to decide anyway.

Luckily, this does not imply you need to decide completely in the dark. You also do not have to stick to the decisions made if new information appears.

In this thesis, we will discuss two approaches that help to deal with those problems, as they are modeled best between classical offline and online problems:

Often, a decision maker is allowed to delay some decisions. In real life, often such options exist, even though it may not for free: If you are considering whether to book something, purchasing some insurance that allows for cancellation might be possible. In finance, buying options allows you to decide at a later point, but again, this does not come for free. In this thesis, various problems are studied within this *Reservation Setting*:

For two variants of the Secretary Problem, the Simple Knapsack with Removability, and the Vertex Cover, we provide matching upper and lower bounds for the whole range of potential reservation costs from zero to expensive. For the General Knapsack and general vertex deletion problems, such as Feedback Vertex Set, we provide asymptotically tight bounds.

Assuming that a setting is either in the complete darkness of an online or with full knowledge of an offline problem is probably even more unrealistic: Usually, you might not have all the information available when deciding, but perhaps a rough overview based on experience or machine learning. Even if the necessary data is available, often it is not exact due to e.g., rounding. In those cases, you will likely have this rough overview of the information available from the beginning. More details will likely be available when decisions need to be made, as is known from online problems. Also in this setting, various problems are studied and called a problem *with Estimates*:

For the Simple Knapsack and Simple Knapsack with Removability, as well as for a restricted version of Bin Packing and Graph Exploration, we provide tight bounds for all accuracy factors, and present bounds for Bin Packing and Graph Exploration.

Additionally, we present bounds and an algorithm for the Online Feedback Vertex Set. As it is a natural online problem with non-trivial structure, it is surprising that its study is just initiated within this thesis.

Zusammenfassung

Stell dir vor, du hast ein beliebiges Problem, für das du eine gute Lösung suchst. Wenn du mit der Situation vertraut bist, weißt, was die Zukunft bringt, welche deine Optionen und deren Konsequenzen sind, dann ist es sicherlich möglich, eine gute Entscheidung zu treffen. In der Realität sind all diese Informationen jedoch nicht immer vorhanden; eine Entscheidung muss dennoch getroffen werden.

Glücklicherweise muss man seine Entscheidungen allerdings auch nicht komplett im Dunkeln treffen. Außerdem gibt es häufig die Gelegenheit, Entscheidungen zu revidieren, falls neue Informationen verfügbar sind.

In dieser Arbeit werden zwei Ansätze untersucht, die diese Situationen modellieren, und damit zwischen klassischen *Offline*- und *Online*-Problemen liegen:

Oft kann eine Entscheidung verzögert werden, allerdings häufig nicht umsonst: Beim Buchen gibt es oft die Gelegenheit, statt einer festen Buchung auch eine teurere mit Stornierungsoption zu tätigen. Im Finanzwesen erlauben einem Optionen, zu einem späteren Punkt zu entscheiden, allerdings ist auch dies nicht umsonst. Wir modellieren verschiedene Probleme in diesem *Reservierungssetting*:

Für zwei Varianten des Sekretärinnenproblems, des Simple Knapsack mit Entfernbarkeit sowie des Vertex Cover präsentieren wir optimale Lösungen für alle Reservierungskosten. Für das General Knapsack und allgemeine Knotenlöschungsprobleme wie dem Feedback Vertex Set zeigen wir asymptotische geschlossene Schranken.

Noch unrealistischer ist es, anzunehmen, dass eine Entscheidung entweder komplett im Dunkeln (wie bei einem Online-Problem) oder mit allen relevanten Informationen getroffen wird (wie bei einem Offline-Problem): Für gewöhnlich sind nicht alle relevanten Daten verfügbar, man hat aber zumindest eine grobe Idee durch die Erfahrung oder Anwendungen aus dem maschinellen Lernen. Hier kann man annehmen, dass die Größenordnung der Instanz am Anfang bekannt ist, aber die Details erst während der Laufzeit wie bei einem Online-Problem aufgedeckt werden. Auch in diesem *Schätzungssetting* studieren wir verschiedene Probleme:

Für das Simple Knapsack mit und ohne Reservierung und für eingeschränkte Varianten des Bin Packing sowie des Graph Exploration erzielen wir optimale Ergebnisse für jede Güte der Schätzung. Für das Bin Packing sowie das Graph Exploration präsentieren wir obere und untere Schranken.

Dazu wird auch eine Studie zum Online Feedback Vertex Set vorgestellt. Überraschenderweise ist diese Arbeit die erste, die dieses natürliche Problem mit interessanter Struktur betrachtet.

Chapter 1

Introduction

Whenever decisions need to be made, one usually benefits from available knowledge about the problem. As this is the case for many real-world problems that need to be handled, it is also true for all the optimization problems which strategies were made within the field of algorithms. While it might be plausible to assume that all the information is available when attacking e.g., a made-up riddle from a book, it is not natural for most settings. This can be due to various reasons.

Sometimes information exists, but is not available:

Should I buy a stock? Surely it would be helpful to know about the current numbers or plans, but those might not be public yet. What questions will there be in my next exam? It would be helpful when preparing for it, but usually the examiner will not share them even if he has already finished the exam. How many moving boxes do I need when moving? You could measure all your belongings to then compute the necessary number, but hardly anyone takes this effort.

Often, the necessary data is also hidden in the future:

Will there be rain in a particular week in August? Then I could plan my vacation accordingly. Should I buy something in today's sale? Or will I regret it, as a better offer will be available soon? Should I work on some heavy proofs today, or will I get interrupted regularly? What is worth trying to memorize (in my head or in a cache of a CPU)? What will I need again soon, what not?

Such settings can be analyzed within the framework of *Online Algorithms*. Various questions are commonly asked for each problem:

How far will I be off due to the lack of relevant data? How much data is missing to make a better decision, and what do I need to know? If someone tells me how I should decide, do I trust him, or is some trap unavoidable in case of wrong hints?

In this thesis, we will mainly discuss two, arguably realistic, questions:

Should I decide now, or is it worth waiting until more information is available, even though this does not come for free? How much will a rough idea about the data help me in making better decisions, depending on how good the estimate is?

1.1 Online Computation

The formal introduction of online algorithms is credited to Sleator and Tarjan in 1984 [87], who also motivated the focus on worst-case analysis as performed in the introduced concept of competitive ratio. But, as one can expect due to the natural motivation and the many use cases of online problems, several results in the field of online computation are notably older. For example, some of the bin packing results stated later are from the 1970s.

In this thesis, we will use common notation for online problems, as used in Komm [71] or Borodin and El-Yaniv [30].

Definition 1.1.1 (Online Problem)**.** An online problem Π is an optimization problem that consists of an instance presented as a sequence $I = (x_1, x_2, \dots x_n)$, for which an algorithm must compute an output sequence $O = (y_1, y_2, \dots, y_n)$. The output y_i can only depend on the input $x_1, \dots, x_i$ and on $y_1, \dots, y_{i-1}$, and must be a feasible solution at the end.

As online algorithms are a special case of optimization problems, they come in two flavors: minimization and maximization problems. When analyzing how well an online algorithm performs on some instance, we compare the quality of the online algorithm's solution to an optimal solution on the same input.

Definition 1.1.2 (Competitive Ratio)**.** Let $ALG(I)$ be the solution of an online algorithm ALG and $OPT(I)$ the optimal solution on instance I.

For maximization problems we call ALG a *c-competitive* algorithm, when $ALG(I) \leq c \cdot OPT(I) + \alpha$ with $c \geq 1$ holds for all instances I.

For minimization problems we call ALG a *c-competitive* algorithm, when $OPT(I) \leq c \cdot ALG(I) + \alpha$ with $c \geq 1$ holds for all instances I.

When $\alpha = 0$, we say that ALG is *strictly c-competitive*.

We also say that the online problem Π itself is c-competitive, when the best possible online algorithm is c-competitive. If there is no constant c for which an algorithm (or a problem) is c-competitive, we also say that the algorithm (or problem) is *non-competitive*.

It is also worth noting that although this is the commonly used definition for competitiveness, in some cases, it is not a sensible wording: For several online problems, algorithms are known that provide an interesting, non-constant competitive ratio. Examples include logarithmic bounds for e.g., Graph Exploration, or in Online Coloring variants.

For deterministic algorithms, a lower bound can be visualized as a game. Here, two players compete: On the one side, there is the so-called *adversary*, whose main task is to construct the next piece of the instance sequence. On the other side, the

algorithm must decide on what to do with the last part of the instance. While the algorithm aims for achieving the lowest possible competitive ratio, the adversary tries to construct the sequence in a way that hinders an algorithm from achieving low competitive ratio. Whenever the adversary has a strategy to construct an instance on which the algorithm cannot achieve a better competitive ratio than c, no matter how he decides, the underlying problem must be at least c-competitive. An easy example for this is the construction of the instance for the knapsack problem as presented in example 1.2.1.

1.2 Online Problems

Formal definitions, like the one of online problems from the previous chapter, can best be understood when illustrated with examples. Therefore, this section will introduce some well-known online problems in their classical variant. Additionally, several variants of these problems are mentioned, which have been studied in the literature. For each problem mentioned in this chapter, a variant in the reservation- or estimate setting is studied in the next chapters.

While most of this section describes well-known problems and results, the Online Feedback Vertex Set Problem in its unmodified setting is studied for the first time within this thesis.

1.2.1 Online Knapsack

One of the most studied online models in online computation is the online knapsack problem, together with dozens of its variants. In the classical form, it can be defined as follows:

Definition 1.2.1 (The Online Knapsack Problem)**.** An instance of the Online Knapsack Problem consists of a series of *items* $I = (x_1, \ldots, x_n)$ that arrives sequentially, with each *item* x_i consisting of a weight $w(x_i) \in [0, 1]$ and a value $v(x_i) > 0$. At each step $i \in \{1, \ldots, n\}$, the item size and value of the request sequence are revealed. An algorithm then has the option to either pack the item in an initially empty knapsack K of size 1 if it fits, or to reject the item:

- **Pack** If $\sum_{x_k \in K} w(x_k) + w(x_i) \leq 1$, set $K := K \cup x_i$.
- **Reject** Do nothing.

The Online Knapsack problem is then for an algorithm `ALG` to minimize the competitive ratio between the value $\sum_{x_k \in K} v(x_k)$ of his packing, compared to the optimal solution in the same instance.

When restricting to items for which the weight equals the value $w(x_i) = v(x_i)$, the problem is also known as the Online Simple Knapsack or Online Proportional Knapsack problem. To point out the difference to the simple variant, the Online Knapsack problem is also known as the Online General Knapsack problem.

One reason and motivation to study the online knapsack problem in various variants is the existence of arguably pathological instances that show its uncompetitiveness. The following example was introduced by Marchetti-Spaccamela and Vercellis in 1995, and even works for the restricted online simple knapsack problem [77].

Example 1.2.1. Let $\varepsilon > 0$. The adversary first presents a single item with size and value of ε. Now an algorithm needs to decide whether to pack this item or not:

- If the algorithm decides to pack the item, an adversary can present the next item with value and size of 1. The algorithm cannot pack it, as the knapsack would be overfull then. Thus, the optimal solution can achieve a packing of 1, while the algorithm only packed a value ε. Therefore, in this case, the competitive ratio is $\frac{1}{\varepsilon}$.
- If the algorithm decides to reject the item, the adversary may decide to present no further items. Here, the optimal solution has a value of ε, while the algorithm only achieves a value of 0, resulting in an unbounded competitive ratio.

As ε can be chosen arbitrarily small, an algorithm has no chance to achieve any constant competitive ratio. Thus, even the online simple knapsack problem is non-competitive.

Since then, many variants were studied. Those were also designed to avoid failing on those unrealistic examples.

If randomization is allowed for an algorithm, then there exists a simple 2-competitive algorithm for the simple knapsack using only one randomization bit [29], while the general problem still does not allow a bounded competitive ratio [96].

Another option is to relax the irrevocability of an algorithm's packing decisions. The setting, when an algorithm is allowed to discard already packed items, is also known as knapsack with removability:

Definition 1.2.2 (The Online Knapsack with Removability Problem)**.** An instance of the Online Knapsack with Removability problem consists of a series of *items* $I = (x_1, \ldots, x_n)$ that arrives sequentially, with each *item* x_i consisting of a weight $w(x_i) \in [0, 1]$ and a value $v(x_i) > 0$. At each step $i \in \{1, \ldots, n\}$,

the item size and value of the request sequence are revealed. An algorithm then has the option to discard a subset of already packed items

- **Discard** Set $K := K \setminus S$ for some subset $S \subseteq K$.

Afterwards, it can pack the new item in an initially empty knapsack K of size 1 if it fits, or reject the item:

- **Pack** If $\sum_{x_k \in K} w(x_k) + w(x_i) \leq 1$, set $K := K \cup x_i$.
- **Reject** Do nothing.

The Online Knapsack with Removability problem is then for an algorithm `ALG` to minimize the competitive ratio between the value $\sum_{x_k \in K} v(x_k)$ of its packing, compared to the optimal solution in the same instance.

This classical modification is known to be Φ-competitive for the simple knapsack [65].

Building on this variant, some models also allow repacking a limited amount of removed items [28]. Also, variants when removing items are connected to costs are studied [61], even if only for the simple knapsack.

Some variants also relax the packing restriction for an online algorithm, for example, by allowing the algorithm to exceed the knapsack size [66, 62] or by allowing it to cut items into parts [63].

Additionally, various attempts were made to relax the blindness of an algorithm. Those approaches include the advice and the prediction setting, which are discussed in the next part of this thesis.

A recent survey by Böckenhauer et al. provides an overview about the Online Knapsack problem together with its variants [27].

1.2.2 Online Bin Packing

One of the first problems that was studied in its online version is the Bin Packing problem. Similar to the Online Simple Knapsack problem, items of various sizes arrive sequentially, which then can be packed into bins. Other than the Knapsack problem, where just one bin is available and a subset of items must be selected to be packed into the one bin, for the Online Bin Packing problem, multiple bins exist. Here, all items must be packed into a bin, and the goal is to use as few bins as possible to do so.

Definition 1.2.3 (The Online Bin Packing Problem)**.** In the Online Bin Packing problem the algorithm receives receives a list of *items* $L = a_1, ..., a_n$ with sizes $c(a_i) \in [0, 1]$ for all $i \in \{1, ..., n\}$ sequentially. The algorithm needs to

decide in which bin the item will be packed before the next item is revealed, or the end of the instance is announced. The resulting partition of the items (also called *packing*) into *bins* must be in a way that the sum of the item sizes from each bin does not exceed 1. The algorithm's goal is to find a packing using as few bins as possible.

Thus, other than the Online Knapsack problem, the Online Bin Packing problem is a classical example of a minimization problem.

It is well known that already very simple strategies can provide good competitive ratios for Online Bin Packing, some might already be sufficient in some applications: The strategy Next Fit, where each item is just placed in the last used bin (if it fits there) or otherwise placed in a new bin, is already known to be 2-competitive [67]. If each new item is placed in the bin that is fullest among those in which the new item fits (and if it fits in none, it will be placed in a new bin), it is known as Best Fit and improves the competitive ratio to 1.7 [68]. The same ratio can also be achieved when each new item is placed in the first bin it fits into, following a strategy called First Fit[68].

This upper bound was beaten by Yao [93], who introduced an algorithm called Refined-First-Fit. This strategy attempts to pack items with a size larger than $\frac{1}{3}$ more efficiently by dividing them into subclasses based on their size and preferring to pack items of certain subclasses together. It yields a competitive ratio of $\frac{5}{3}$ and thus performs significantly better than the simple strategies discussed above.

Further improvements of the competitive ratio usually rely on the concept of Harmonic Algorithms. Here, each item will be classified by its size. Depending on its class, it is then treated differently [73]. In its simple version, first, a constant $M \geq 3$ is fixed. Then, an item is classified to be an I_k item, if its size is in in the interval $(\frac{1}{k+1}, \frac{1}{k}]$ for $k \leq M-1$. Items, which are smaller than $\frac{1}{M}$, are classified to be an I_M item. Items of each class are packed together greedily, meaning each bin only contains items from the same class. The competitive ratio of an algorithm Harmonic$_M$ decreases with increasing M, and converges towards 1.691 as M goes to infinity. The ratio of Best-Fit however (1.7) is already beaten for $M = 7$ [73].

Lee and Lee [73] combined a similar idea from Refined-First-Fit with Harmonic$_M$ to create an algorithm called Refined-Harmonic which achieves a ratio of $\frac{373}{228} \approx 1.6369$. Most approaches that improve these results use Harmonic$_M$ as their basis. The first improvements were presented by Ramanan et al. [83] who introduced the algorithms Modified-Harmonic and Modified-Harmonic-2 with competitive ratios of around 1.6156 and 1.612, respectively. Later, Seiden [86] managed to generalize multiple previously known algorithms to a strategy he called Super-Harmonic and thereby showed that all these algorithms can be analyzed similarly. He also introduced a new algorithm called Harmonic++ based

on Super-Harmonic and achieves a competitive ratio of 1.58889. The currently best-known strategy for online bin packing was proposed in [16] and yields a ratio of approximately 1.57829.

However, one cannot expect to significantly decrease the competitive ratio: Yao proved that every algorithm for the online bin packing problem has to be at least 1.5-competitive [93]. Brown and Liang [37][74] independently improved on this lower bound by adding more item sizes to the instance described above, obtaining a new lower bound of around 1.5363. This value was improved to 1.54015 by van Vliet [89] and later to 1.54037 and then 1.54278 by Balogh et al. [19, 18]. This is the currently best lower bound on the online bin packing problem.

Additional improvements are possible if adding further constraints: For example, when assuming that a solution might only consist of up to two items per bin, a ratio of $1 + \frac{1}{\sqrt{5}} \approx 1.4472$ is achievable. Thus, its competitive ratio significantly undercuts 1.5, but unfortunately, there remains little room to improve as the currently best known lower bound is 1.4286 for this particular variant (with previous bounds of $\sqrt{2}$ or 1.4276)([12, 52, 17]). Just like Refined-Harmonic, the algorithm of Babel et al. attempts to preserve a ratio between the number of bins with different item configurations [12]. However, if estimates are available, it can help to overcome the balancing problem between the bins with different configurations.

An extensive survey on upper and lower bounds for Online Bin Packing together with some of its variants can be found in [46].

1.2.3 Secretary Problem

A different flavor of online problems is studied in the setting of the Secretary problem. The main difference to the other problems discussed in this thesis is that the instance is not presented adversarially. Instead, items arrive in random order. An algorithm however, does not know the instance beforehand, thus it cannot know what else will come up in the future.

The secretary problem has been extensively researched in the areas of online algorithms and stopping theory, and can be defined as follows:

Definition 1.2.4 (The Secretary Problem)**.** In the Secretary problem, an algorithm receives the number of items in the instance n in the beginning, followed by n randomly ordered distinct numbers x_i. An algorithm can compare all items it has been presented so far, and has two options for each presented item:

- Take the current item, then the instance ends.
- Reject the item, move on to the next one (if it was not the last item)

The algorithm's goal is to maximize the probability of taking the best item of the instance.

The crucial point is that you have to decide whether to choose a number as soon as it arrives without seeing the following numbers, and that you cannot take back your decision. This classic problem was first solved by Lindley [75] and Dynkin [49]. The solution basically is as follows: Look at the first n/e numbers without selecting one. Afterwards, choose the first number that is the biggest one of all numbers seen so far. Then the probability of getting the highest number is asymptotically $1/e$, which is the best possible. In the secretary problem, the gain is either 1 or 0, depending on whether the algorithm has chosen the highest number or not.

Today, the Secretary problem is well-known, and many variants have been studied. For example, instead of hiring just one person, we might be looking for k persons. Ideally, the highest numbers will be chosen, but the gain is defined as the number of persons chosen that are among the best k ones. Not surprisingly, the competitive ratio will get better and better as k grows. This so-called k-secretary problem was introduced by Kleinberg [70] and has seen a lot of attention, too [2, 38, 10, 11, 44].

1.2.4 Graph Exploration

The Graph Exploration setting usually refers to a problem in which an agent needs to visit all vertices of a given graph. However, is only aware of the graph induced by the vertices visited so far and their neighbors.

Definition 1.2.5 (The Graph Exploration Problem)**.** Given a weighted graph $G = (V, E)$, a vertex $s \in V$ as a start-vertex and a vertex $t \in V$ as an end-vertex, the Graph Exploration problem is the problem of finding the shortest walk visiting all vertices, starting and ending at the designated vertices. The graph gets revealed sequentially. As soon as a vertex is visited for the first time, its neighborhood (the adjacent vertices together with the edge weight of the incident edges $w(e)$) will get revealed.

The study of this problem was initiated by Kalyanasundaram and Pruhs in 1994 [69], where they proved a 16-competitive ratio for planar graphs.

Since then, the Graph Exploration model attracted interest in various settings. It was shown by Miyazaki et al. [81] that there cannot be an algorithm achieving a better competitive ratio than 2 on unweighted graphs. The lower bound for the graph exploration problem on general graphs was then improved to 2.5 by Dobrev et al. [48], and then by Birx et al. [22] with a competitive ratio of 10/3 for planar graphs. The current best known lower bound is 4 given by Baligács [14].

On the other hand, no algorithm achieving a constant competitive ratio is known for general graphs. The so far best approach by Rosenkrantz et al. [85]

achieves a logarithmic competitive ratio, using a nearest neighbor approach. Another algorithm that achieves a logarithmic competitive ratio is a modification of depth-first search by Megow et al. [78]. However, constant competitive algorithms are known for restricted graph classes.

In Megow et al. [78], an algorithm with a constant competitive ratio for bounded genus graphs was presented. They also proved that this algorithm does not yield a constant competitive ratio on general graphs. Thus, we may wonder whether knowing the structure of the graph in advance might help to construct exploration algorithms with a constant competitive ratio.

Their results are generalized by Baligács et al. [15], who prove that one can achieve a constant competitive ratio on any minor-free graph class.

Another research direction of the graph exploration problem is given by the setting where, instead of one agent, several can traverse through the graph simultaneously. Here, the goal is either to visit every vertex as soon as possible or to minimize the total distance traversed by all agents. A recent overview of the different results in this direction can be found in van den Akker et al. [88].

1.2.5 Online Vertex Cover

The goal in the general Vertex Cover problem is, given a graph $G = (V, E)$, to find a minimum set of vertices $S \subseteq V$ such that $G[V \setminus S]$ contains no edges, i.e., the obstruction set is a path of length 1. One could define an online version of vertex cover, where the graph is revealed vertex by vertex, including all induced edges, and an online algorithm must immediately and irrevocably decide for each vertex whether to add it to the proposed vertex cover or not.

It is easy to see that this definition is not competitive: If the first vertex is added to the solution set, the instance stops and thus leaving a single-vertex instance with an optimal solution size of zero. On the other hand, not selecting the first vertex will lead to an instance where this vertex becomes a central vertex of a star, forcing an algorithm to delete all the leaves.

These adversarial strategies are arguably pathological and unnatural, as decisions are enforced that are not based on the properties of the very problem to be solved: We need to start constructing a vertex cover before any edge is presented. To address this issue in general, for online vertex- and edge-deletion problems, Boyar et al. [31, 32] introduced the *late accept* online model. It later was reintroduced by Chen et al. [45] as the *delayed-decision* model. This model allows an online algorithm to remain idle until a "need to act" occurs, which in our case means waiting until a graph from the obstruction set appears in the online graph. The online algorithm may then choose to delete any vertices in the current online graph. The main remaining restriction is that an online algorithm may not undo

any of these deletions. We will use this variant for defining the Online Vertex Cover problem in this thesis:

Definition 1.2.6 (The Online Vertex Cover problem)**.** Let G be an online graph induced by its vertices $V(G) = v_1, \ldots, v_n$, ordered by their occurrence in an online instance. The Online Vertex Cover problem is to select, for every i, a subset of vertices $S_i \subseteq \{v_1, \ldots, v_i\}$ with $S_1 \subseteq \ldots \subseteq S_n$ such that the induced subgraph $G[\{v_1, \ldots, v_i\} \setminus S_i]$ contains no edge. The goal is to minimize $|S_n|$.

A constant competitive ratio of 2 for the online vertex cover problem is simple to prove and given in the introduction of the paper by Chen et al. [45]. It can be achieved by selecting both vertices for each edge, which is not covered at the point of its presentation. This is also known to be best possible.

The Online Vertex Cover problem has received some attention in the past years. Demange and Paschos [47] analyzed the Online Vertex Cover problem with two variations of how the online graph is revealed: One variant is that the graph is presented vertex by vertex. Alternatively, it is presented in clusters, per induced subgraphs of the final graph. The proven competitive ratios are functions on the maximum degree of the graph.

Zhang et al. [95] studied a variant called the Online 3-Path Vertex Cover problem, where every induced path on three vertices needs to be covered. In this setting, the competitive ratio is again dominated by the maximum degree of the graph. Buchbinder and Naor [39] considered online integral and fractional covering problems formulated as linear programs where the covering constraints arrive online. As these are strong generalizations of the online vertex cover problem, they achieve only logarithmic and not constant competitive ratios.

There has been some work on improving upon the bound of 2 for some special cases of vertex cover in the model with delayed decisions (under different names). For the vertex cover problem on *bipartite* graphs where one side is offline, Wang and Wong [91] give an algorithm achieving a competitive ratio of $\frac{1}{1-1/e} \approx 1.582$. Using the same techniques, they achieve a competitive ratio of 1.901 for the full online vertex cover problem for bipartite graphs and for the online fractional vertex cover problem on general graphs.

1.2.6 Online Feedback Vertex Set

As the Feeback Vertex Set problem can be defined similarly to the Vertex Cover problem. The same is possible for their online variants: In this thesis, we will also study the Online Feedback Vertex Set problem in the same variant as the Online Vertex Cover problem, namely in its *Late Accept* or *Delayed Decision* variant.

Definition 1.2.7 (The Online Feedback Vertex Set Problem)**.** Let G be an online graph induced by its vertices $V(G) = v_1, \ldots, v_n$, ordered by their occurrence in an online instance. The Online Feedback Vertex Set problem is to select, for every i, a subset of vertices $S_i \subseteq \{v_1, \ldots, v_i\}$ with $S_1 \subseteq \ldots \subseteq S_n$ such that the induced subgraph $G[\{v_1, \ldots, v_i\} \setminus S_i]$ is cycle-free. The goal is to minimize $|S_n|$.

This definition is almost identical to the definition of the Online Vertex Cover problem 1.2.6, and differs only by replacing "has no edge" with "is cycle-free".

Interestingly, and to the best of our knowledge, the Online Feedback Vertex Set problem has not been studied prior to this thesis. This is surprising, as the problem definition is very natural, the Feedback Vertex Set problem a very well-famous problem in its offline variant with various known results for especially approximation algorithms. We will see, that it also yields non-trivial results in its online version.

One notable algorithm is the one in the paper of Bar-Yehuda et al. [20], yielding an approximation ratio of $4 - 2/n$ on an undirected, unweighted graph. We adapt their notation and our online algorithm with a competitive ratio of 5 is based on their aforementioned approximation algorithm. The currently best known approximation ratio of 2 by an (offline) polynomial-time algorithm was given by e.g., Becker and Geiger [21]. An adaptation of those algorithms are likely not possible in an online setting, as they very heavily relay on an ordering of the vertices by their degree.

Therefore, in this thesis, we present initial results and first bounds for the Online Feedback Vertex Set problem, also hoping to initiate further research on this problem.

Lower Bound

Theorem 1.2.1 (Lower Bound)**.** *Given an $\varepsilon > 0$, there is no algorithm for* Online Feedback Vertex Set *achieving a competitive ratio of $4 - \varepsilon$.*

Proof. The adversarial strategy depicted in Figure 1.1 provides the lower bound.

First, a cycle is presented, which forces any algorithm to delete a single vertex. The adversary now repeats the following scheme n times: From the cycle that was presented in the previous step, select two different vertices. Then connect these vertices by a new path, thus forming a new cycle (the large, "layered" cycles in the figure).

Every time such a semicycle is added, any algorithm has to delete a vertex from it. This is sketched in Figure 1.1 by the black cycles with red dots for exemplary deletions of some algorithm. We assume w.l.o.g. that an algorithm chooses one

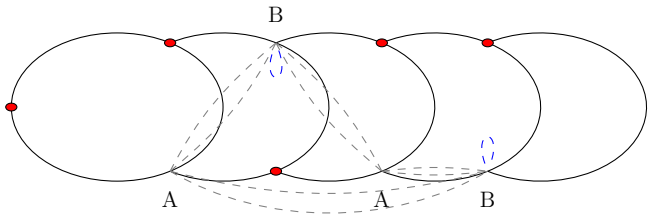


Figure 1.1: Sketch of the lower bound graph, revealed from left to right. The marked vertices are deleted by an algorithm. Then dashed edges (gray) and finally the self-loops (blue) force more deletions. The competitive ratio tends to 4 with increasing instance size.

of the vertices of degree 3 between two cycles for deletion; we call these vertices of degree 3 branchpoints. If an algorithm deletes vertices of degree 2 instead, the adversary can force the algorithm to delete even more vertices.

After following this scheme, we split the remaining branchpoints (one for each new cycle) into two sets A and B alternatingly. We now take each (a, b)-pair with $a \in A$ and $b \in B$ and connect the two vertices with two paths, forming a new cycle between each pair of branching vertices. In order to remove all cycles with the lowest amount of deletions, any algorithm has to delete either all vertices from the set A or all vertices from the set B. Once such a set is chosen, say A, the adversary adds "self-loops", i.e., new cycles each only connected to a vertex of B, forcing the deletion of all (branchpoint) vertices of B.

The optimal solution consists only of the vertices of B and possibly, and depending on whether n is even and on the choice of A or B, another vertex from the first and the last cycle.

Thus, any online algorithm has to delete at least $2n - 1$ vertices, while an optimal algorithm only deletes at most $n/2+1$ vertices. The resulting competitive ratio is hence worse than $4 - \varepsilon$ for large enough values of n. □

A 5-competitive Algorithm

Next, we present Algorithm 1, which guarantees a competitive ratio of 5. It is a modified version of the 4-approximative algorithm for Feeback Vertex Set problem by Bar-Yehuda et al. [20]. Algorithm 1 maintains a maximal so-called $2-3$-subgraph H in the presented graph G and selects a feedback vertex set F for G within H. It is important to note that H is not necessarily an *induced* subgraph of G and also does not have to be connected.

Definition 1.2.8 (**$2-3$**-subgraph)**.** A $2-3$-*subgraph* of a graph G is a subgraph H of G where every vertex has degree exactly 2 or 3 in H.

Given a $2-3$-subgraph H, a vertex is called a *branchpoint of* H if it has exactly degree 3 in H. We say a vertex v is a *linkpoint in* H if it has exactly degree 2 in

H and there is a cycle in G whose only intersection with H is v. A cycle is called *isolated in* H if every vertex of this cycle (in G) is contained in H and has exactly degree 2 in H.

Algorithm 1 adds every branchpoint, linkpoint, and one vertex per isolated cycle without linkpoints in H to the solution set F.

Note that it is possible that vertices, that were added to F as linkpoints or due to isolated cycles, can still become branchpoints with degree 3, if H is expanded in later steps. For the sake of the analysis, we consider them as branchpoints. The extension of H is not necessarily unique, as often there are several ways of extending. The algorithm just chooses an arbitrary way.

Algorithm 1 Online SubG-2-3

$H \leftarrow \emptyset$, $F \leftarrow \emptyset$
for every new edge in G **do**
  if H is not a maximal $2-3$-subgraph **then**
    Extend H to a maximal $2-3$-subgraph, converting linkpoints to branchpoints
    whenever possible.
  $F \leftarrow F \cup$ All new branchpoints
  $F \leftarrow F \cup$ All new linkpoints
  $F \leftarrow F \cup$ A set of one vertex per new isolated cycle without linkpoints in H
return F

Lemma 1.2.1 (Correctness of Online SubG-2-3)**.** *Algorithm 1 returns a feedback vertex set for the given input graph G.*

Proof. By contradiction, assume there is a cycle C in G without a vertex in F.

If C contains no point of H, then the complete cycle C can be added to the subgraph H as an isolated cycle, thus H is not maximal. This contradicts the procedure of Algorithm 1.

Therefore, we can assume that there is at least one vertex of C in H. If the cycle contains no branchpoints, three situations are possible: First, C is an isolated cycle, where all vertices are in H and one vertex is added to F. Second, C intersects with H at just a single vertex, which will be a linkpoint and part of F as well. The third case is that C intersects H on two or more vertices and none of them are branchpoints (thus of degree 2). Then H would have been extended from one of those points along C towards another one, making the two vertices branchpoints.

Thus, every cycle in G has to contain at least one vertex of F, which is a feedback vertex set at the end.

□

Proving that Algorithm 1 is 5-competitive is trickier. First, note that an optimal feedback vertex set does not change if we consider a *reduction graph* G' instead of a graph G.

Definition 1.2.9 (Reduction Graph)**.** A *reduction graph* G' of a graph G is obtained by deleting vertices of degree 1 and their incident edges, and by deleting vertices of degree 2 without a self-loop and connecting the two neighbors directly.

The following lemma bounds the size of a reduced graph by its maximum degree and the size of any feedback vertex set. This will be used in the analysis of Algorithm 1. The lemma is due to Voss [90] and is used in the proof of 4-competitiveness by Bar-Yehuda et al. [20].

Lemma 1.2.2. *Let G be a graph where no vertex has degree less than 2. Then for every feedback vertex set F, that contains all vertices of degree 2,*

$$|V(G)| \leq (\Delta(G) + 1)|F| - 2$$

holds, where $\Delta(G)$ is the maximum degree of G. In particular, if every vertex has a degree of at most 3, $|V(G)| \leq 4|F| - 2$ holds.

Theorem 1.2.2. *Algorithm 1 achieves a strict competitive ratio of $5 - 2/|V(G)|$ for the* Online Feedback Vertex Set *problem.*

Proof. At the end of the instance, after running Algorithm 1, call the set of branchpoints $B \subseteq F$, the set of linkpoints $L \subseteq F$, and the set of vertices added due to isolated cycles I. Again, note that vertices can become branchpoints in later steps, even if they were added to F as a linkpoint or for an isolated cycle. Let μ be the size of an optimal feedback vertex set for the graph G.

The vertices in I are part of pairwise independent cycles. This follows from the fact that every isolated cycle was added completely to H. Thus, new isolated cycles cannot contain a vertex of one already added to H. Therefore, $|I| \leq \mu$, since an optimal solution must contain at least one vertex for each of the pairwise independent cycles.

Moreover, the set of cycles that caused adding vertices as linkpoints to F are also pairwise independent, if no linkpoint was relabeled to a branchpoint later:

For a contradiction, assume that two cycles overlap at a vertex v and and intersect with H at the linkpoints ℓ_1 and ℓ_2. Now H could easily be extended by a path from ℓ_1 through v to ℓ_2, if none of those vertices are added to H before. This would make ℓ_1 and ℓ_2 branchpoints, contradicting the assumption.

The algorithm would also convert either ℓ_1 or ℓ_2 to a branchpoint, if — at one point — it was possible to extend H by connecting one of the linkpoints along the mentioned path to one vertex in H of degree 2 (with respect to H).

The only remaining case is when the first vertex of H along this path was a branchpoint immediately at the time it was added to H. This makes it impossible to connect it with one of the linkpoints.

Note that w.l.o.g. ℓ_1 must have been considered as a linkpoint at this time, since — by definition— no cycle would have caused any vertex to become a linkpoint if some vertices of the cycle were already considered as branchpoints. In particular, x cannot have been added immediately to H when it was presented.

Since the adversary presents the instance vertex-wise, there must be a vertex s such that there was no possibility to add x to H before s was presented, but such that x immediately could become a branchpoint when s was added to G.

Therefore, there must be at least two independent paths in $G \setminus H$ from x to s. Therefore, in this situation, it would also be possible to extend H by adding the two independent paths from x to s towards H, and also one path from x to ℓ_1 along the cycle. Since the algorithm prioritizes extensions to H which convert linkpoints to branchpoints over others that do not, it has to add the path from x to ℓ_1 first. Therefore, ℓ_1 becomes a branchpoint. This finally contradicts our assumption. Note that priority is unambiguous, since there are no paths in $G \setminus H$ from x to some other linkpoints: Then ℓ_1 and the other linkpoint would already be connected within H, thus making them branchpoints.

Thus we have $|L| \leq \mu$. This also shows that there cannot be a branchpoint inside a cycle that was used to delete a linkpoint, without also converting the linkpoint to a branchpoint.

It follows that $|L|+|I| \leq 2\mu$. If $|B| \leq 2|L|$, then we have $|F| = |I|+|L|+|B| \leq 3|L| + |I| \leq 4\mu$, which proves the statement.

In every other case, assume $|B| > 2|L|$. We now consider a reduction graph H' of the graph $H \setminus L$, and delete every component consisting of only a single vertex. Every vertex in the resulting graph has degree 3. In the graph H we can have up to $2|L|$ more branchpoints than in the resulting graph here.

By Lemma 1.2.2, $|B| - 2|L|$ is less than $4\mu(H') - 2$, where $\mu(H')$ is the optimal size of a feedback vertex set for the graph H'.

The size of an optimal feedback vertex set of the original graph G must be at least $\mu(H')+|L|$ since every linkpoint in G is part of a cycle that is not intersecting H'.

The inequality chain $|F| = |I| + |L| + |B| = |I| + 3|L| + |B| - 2|L| \leq |I| + 4|L| + 4\mu(H') - 2 \leq |I| + 4\mu(G) - 2 \leq 5\mu(G) - 2$ concludes the proof. □

One could think that the reason Algorithm 1 does not match the competitive ratio of 4 is that the algorithm deletes vertices even in cases where it is not necessary. However, this is not the case.

Lemma 1.2.3. *The competitive ratio of Algorithm 1 is at least* $5 - 2/|V(G)|$*, even if vertices are only deleted whenever necessary.*

Proof. The following adversarial strategy, illustrated in Figure 1.2, provides the desired lower bound.

First, the adversary presents n independent cycles $C_1, ..., C_n$. Algorithm 1 deletes at least one vertex per cycle.

Second, the adversary connects each pair of neighboring cycles C_i and C_{i+1} for $i \leq n-1$ with two independent paths. This adds $4n-4$ branchpoints. Everything that is presented is part of the $2-3$-subgraph, marked in blue in Figure 1.2.

Now, a vertex c (the top vertex in Figure 1.2) is added and connected to every branchpoint with two edges. This new vertex will not be part of the $2-3$-subgraph. Therefore, even a modified Algorithm 1 that only deletes branchpoints whenever necessary has to delete every branchpoint. Here, Algorithm 1 deletes $n + 4n - 4$ vertices, whereas an optimal solution consists of only one vertex per independent cycle and the vertex c. □

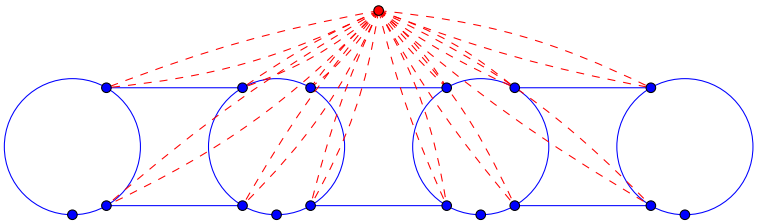


Figure 1.2: The $2-3$-subgraph is given in blue, marked vertices will be deleted by Algorithm 1. The top vertex cannot be added to the $2-3$-subgraph, so it will not be deleted.

1.3 Other Variants in Online Computation

As already pointed out, both traditional algorithmic approaches and online methods are not realistic in many real-world applications. It is often unrealistic to assume an algorithm can utilize complete information from the start. It is also unrealistic to assume there is no information at all about the missing input data.

Over the last fifteen years, significant efforts have been dedicated to bridging the gap between full and no information, driven by a motivation to develop good algorithmic solutions for real-world applications.

1.3.1 Online Algorithms with Advice

This process began with the concept of online algorithms utilizing *advice*. In this context, an algorithm without prior knowledge of the future input can ask a limited number of questions about the input, which are then answered truthfully. This approach highlights the importance of determining how much information is required to produce an output of sufficient quality. Although it is evident that supplementary information can enhance an algorithm's solution, such information may only be available with some effort. The research on online algorithms with advice thus suggests a user-oriented approach to identifying which additional information to provide in order to achieve the desired level of quality in the solution, while minimizing the amount of additional information needed. An overview of the advice model can be found in Komm [72] and in a survey by Boyar et al. [33]

For the Online Knapsack problem and the variant with removability, Böckenhauer et al. [24, 29] showed that already one bit of advice can significantly impact an algorithms perfomance, make the problem competitive.

For Online Bin Packing, A simple heuristic that yields a competitive ratio of $\frac{3}{2}$ and requires $O(\log n)$ advice bits (n being the number of items in the input list) is called Reserve-Critical and was introduced by Boyar et al. in [35]. Here, the advice is used to tell the algorithm the number of items with a size in the interval $(\frac{1}{2}, \frac{2}{3}]$. The algorithm then reserves a bin for each of these items and tries to pack them with smaller ones. A major improvement on this result was given by Angelopoulos et al. [4] who showed that it is even possible to beat $\frac{3}{2}$ with a constant number of advice bits; their algorithm DR+ yields a competitive ratio of around 1.47012. In the same paper, they also observed that it only takes as few as 16 bits for an algorithm to outperform every algorithm for classical online bin packing. Renault et al. also presented an algorithm that can construct an optimal packing using only linear advice [84]. Finally, it is known that any algorithm needs to be able to access an advice string with a length of at least $\Omega(n)$ to beat a competitive ratio of $4 - 2\sqrt{2} \approx 1.172$ [79].

For the Graph Exploration problem on unweighted graphs, a recent study

is provided in Böckenhauer et al. [25], where they show that an online algorithm can compute an optimal solution on sparse graphs if it is provided with $O(m)$ bits of correct advice, where m is the number of edges.

1.3.2 Online Algorithms with Predictions

While advice requires accurate information, many real-world applications inherently involve uncertainty. Therefore, Purohit et al. [82] introduced online algorithms with *predictions* in 2018 (see also [80]). In this setting, an algorithm must operate without knowledge of future inputs but still receives guidance from a predictor for each decision. A good algorithm should provide an optimal solution if the given suggestions are correct (consistency), yet avoid blindly following hints, as incorrect predictions could lead to poor solution quality (robustness). Ideally, solution quality should improve with better predictions (smoothness). This setting is also known as online algorithms with *untrusted* or *machine-learned advice*, where the necessary predictions result from machine learning approaches. In essence, an online algorithm with predictions advises on how to use recommendations generated by a machine learning tool to avoid poor solutions in case of incorrect recommendations. The main advantage of using such an online algorithm connected to a machine learning tool is that it allows for provable guarantees against low-quality solutions. Given the relevance of machine learning in recent years, it is not surprising that there is significant interest in this area, with numerous publications on top-level machine learning conferences. The website *Algorithms with Predictions* provides an overview of recent activity in this field [1]. Also, problems studied in this thesis were already investigated under the lens of predictions:

The Online Knapsack problem is studied by Im et al. [64] in the general variant, where model predicting the frequency of items of each size. Boyar, Favrholdt, and Larsen [34] considered the online simple knapsack problem with predictions, but working with predictions on the *average* size of the items an optimal solution would pack. In Xu and Zhang [92], again the simple knapsack problem in a learning-augmented setting is studied.

For Online Bin Packing, Angelopoulos et al. provided an algorithm which is 1.5-competitive in case of correct predictions, and achieves a competitive ratio of 6 if a prediction turn out to be incorrect [3]. Later, Angelopoulos [5] assume that the item sizes are drawn from a fixed set of integers, and the predictions are used to predict the frequency at which different sizes occur in the instance. Here, also an extensive overview of the most important results for online bin packing under advice and prediction models is given.

For the Graph Exploration problem, Eberle et al. [50] present an algorithm that achieves a constant competitive ratio if the predictions are correct, but degrades the worse the predictions turn out to be.

1.4 Bibliographical Note

Excerpts of this thesis were published already at various conferences or made available to the public on preprints.

Results related to the reservation setting from chapter 2 can also be found in the following articles:

1. Elisabet Burjons, Matthias Gehnen, Henri Lotze, Daniel Mock, Peter Rossmanith: "The Secretary Problem with Reservation Costs", COCOON 2021 [42]

2. Elisabet Burjons, Matthias Gehnen, Henri Lotze, Daniel Mock, Peter Rossmanith: "The Online Simple Knapsack Problem with Reservation and Removability", MFCS 2023 [43]

3. Elisabet Burjons, Fabian Frei, Matthias Gehnen, Henri Lotze, Daniel Mock, Peter Rossmanith: "Delaying Decisions and Reservation Costs", COCOON 2023 [40]

4. Elisabet Burjons, Matthias Gehnen: "Online General Knapsack with Reservation Costs", WAOA 2025 [41]

The results on the Online Feedback Vertex Set Problem are also contained in the article "Delaying Decisions and Reservation Costs".

For the setting of algorithms with estimates from chapter 3, results are published in the following articles

5. Matthias Gehnen, Ralf Klasing, Émile Naquin: "Graph Exploration with Edge Weight Estimates", LATIN 2026 [54]

6. Matthias Gehnen, Andreas Usdenski: "Online Bin Packing with Item Size Estimates" [59]

7. Jakub Balabán, Matthias Gehnen, Henri Lotze, Finn Seesemann, Moritz Stocker: "Online Knapsack Problems with Estimates", MFCS 2025 [13]

The article "Online Bin Packing with Item Size Estimates" contains results that have been found together with Andreas Usdenski in the scope of his master's thesis.

The article

8. Matthias Gehnen, Henri Lotze, Peter Rossmanith: "Online Simple Knapsack with Bounded Predictions", STACS 2024 [**?**]

also contributes to the understanding of the estimate setting, but will not be a main part of this thesis. An extended version of the findings of this article can be found in the PhD-Thesis "Going Offline- Delays, Reservations and Predictions in Online Computation" by Henri Lotze [76].

Additionally, the author also contributed to the following publications that are not covered in this thesis.

9. Matthias Gehnen, Eberhard Triesch: "Transitive avoidance games on boards of odd size", The Electronic Journal of Combinatorics 28(4) (2021) [58]

10. Matthias Gehnen, Luca Venier: "Tetris Is Not Competitive", FUN 2024 [60]

11. Matthias Gehnen, Julius Stannat: "Fog of War Chess", FUN 2026 [56]

12. Fabian Frei, Matthias Gehnen, Dennis Komm, Rastislav Královič, Richard Královič, Peter Rossmanith, Moritz Stocker: "Tree Coloring with Predictions", Discrete Applied Mathematics, 2026 [51]

13. Hans-Joachim Böckenhauer, Matthias Gehnen, Juraj Hromkovič, Ralf Klasing, Dennis Komm, Henri Lotze, Daniel Mock, Peter Rossmanith, Moritz Stocker: "Online Unbounded Knapsack", Theory of Computing Systems 69(1)(2025) [26]

14. Matthias Gehnen, Moritz Stocker: "Stealing From The Dragon's Hoard: Online Unbounded Knapsack With Removal", STACS 2026 [57]

15. Matthias Gehnen, Kübra Güven, Valentin Hächler, Dennis Komm, Richard Královič: "Improved Results for Knapsack with Removal", MFCS 2026 [53]

The article "Transitive avoidance games on boards of odd size" is a follow-up work on the master's thesis of Matthias Gehnen under the supervision of Eberhard Triesch. The article "Tetris Is Not Competitive" builds upon the findings from the master's thesis of Luca Venier, the article "Fog of War Chess" of the bachelor's thesis of Julius Stannat, and the article "Improved Results for Knapsack with Removal" of the bachelor's theses of of Kübra Güven and Valentin Hächler. The theses projects of Andreas Usdenski, Luca Venier, Julius Stannat, and Kübra Güven were supervised by Matthias Gehnen.

1.5 Structure of this Thesis

In this thesis, we will focus on discussing two variants of various problems.

In the introduction we started with an overview within the world of online algorithms. First, we have introduced online computation formally. After fixing the setting, the online problems that are studied within this thesis were defined. It is worth noting that the well-known *Feedback Vertex Set* is studied for the first time in its online variant within this thesis. With the basic definitions and some problems in mind, we briefly looked into some questions that are studied within the area of online computation. In the introduction, the popular variants of *Advice* and *Predictions* were described. In the next sections, we will focus on the *Reservation* variant and the *Estimates* setting, the main variants that are studied within this thesis. Here will also see why the estimate setting is a logical next step after the advice and prediction model.

In chapter 2, we will focus on the reservation model. After giving a motivation and overview of the section, we will discuss various problems in its reservation variant. As packing problems are realistic within this setting, we will consider the knapsack problem first. Here we will focus on two flavors, the Simple Knapsack with Removability as well as the General Knapsack. Afterwards, we will discuss the Secretary Problem, which turns out to be interesting as the used proof technique differs drastically from other online problems studied in this thesis. Lastly, we will discuss how graph problems such as Vertex Cover or Feedback Vertex Set can be analyzed within the reservation framework.

Afterwards, in chapter 3, we will study the variant of online algorithms with estimates. As in the previous chapter, we will start with a motivation and overview, followed by packing problems. Again, several knapsack variants are studied: this time, we will discuss both the Knapsack Problem and the Knapsack Problem with Removability in its simple variant. Afterwards, as another packing problem and perhaps one of the most studied online problems overall, we will discuss Bin Packing with estimated item sizes. Finally, we will see that the setting of estimates is also natural for online graph problems, such as the famous Graph Exploration Problem.

This thesis will be concluded with some motivation for further work in chapter 4.

Chapter 2

Algorithms with Reservations

2.1 Introduction

The aforementioned variants of online computation deal with the question of how to handle if additional information about the instance can be made available. Arguably, one can ask whether it is realistic to assume that such oracles exist at all, no matter which form it may have.

While likely one will have some advice-giver available for some problems, sometimes this is not the case. Then the inherent darkness about the future in some online problems is unavoidable.

Luckily, sometimes options exist that avoid the classical, harsh online setting:

Take, for example, one of the very basic problems in online computation, the so-called Ski Rental problem. It can be defined as follows:

Definition 2.1.1 (Ski Rental Problem)**.** An instance of the Ski Rental problem consists of a price b for buying skis, a price for renting skis for one day r, and a list of days which are either skiing days or not. The list of days is presented iteratively, and an algorithm must either rent or buy skis for each skiing day until skis are bought.

The goal of the Ski Rental Problem is to minimize the costs of an algorithm for buying and renting combined.

For this problem, the best deterministic strategy is called *Break Even*. This strategy reserves skis as long as the paid rental costs equal the buying price, at which it finally buys the skis. It is easy to see that this strategy achieves a competitive ratio of 2 [71].

In this example, one assumes that buying and renting are the only options for an algorithm. In reality, one usually has more options when making decisions. Those might involve, e.g., buying an insurance to cut losses for some realizations

of the instance, or to undo decisions previously made (it might be possible to sell used skis, some sellers might give a discount if one decides to buy previously rented gear). However, those options typically do not come for free. Usually, the more flexibility an option allows, the more expensive it gets.

Another example is bookings for hotels, rental cars, trains, or flights: when booking, one often has the chance to select options with different flexibility. Bookings that cannot be canceled typically are the cheapest, but by selecting this option, one has made a fixed decision. When reality happens such that this booking is not fitting anymore, the costs (which one usually tries to minimize) need to be paid regardless. Alternatively, options which allow cancellation or switching to different dates will be more expensive first, but allow one to decide at a later point about whether one actually needs to use the booking or not.

In this chapter, we model the option within a framework called *reservations*.

Definition 2.1.2 (Online Algorithms with *Reservations*)**.** Given an online Problem Π, the online problem Π *with reservations* is defined as Π with the extension that options of reserving are added to the options for possible output y_i. Each reservation allows for delaying a decision for one of the options y_i in Π to a later point. Additionally, costs depending on a variable $\alpha \geq 0$ must be paid for each reservation, which are then deducted (for maximization problems) or added (for minimization problems) to the value of the algorithm's solution.

The competitive ratio for an online problem with reservations is then defined as in definition 1.1.2. Note that the reservation costs only potentially influence the value of an online algorithm, while an optimal solution typically does not need to pay reservation costs until additional information is revealed.

Given some variable for the reservation costs α, an online problem with the option of reserving yields a competitive ratio at most as high as the respective online problem without the option of reserving. This is easy to see, as all online algorithms might decide not to use the option of reserving at all. This implies that all algorithms for the online problem without reservations are also valid algorithms in the setting with reservations.

For most problems, one can expect a competitive ratio of 1 for $\alpha = 0$, as all decisions might be delayed until the end of the instance (thus making the online problem essentially an offline problem). Thus, even if not explicitly stated, we will assume $\alpha > 0$ for the rest of this chapter. The competitive ratio then will increase monotonously for rising reservation costs α. For some α, the option to reserve will likely become unattractive. Beginning at this α, the competitive ratio will be identical to the problem without the option of reserving items.

2.2 Related Work and Contributions

The reservation setting was studied first by Böckenhauer et al. in 2021. Here, the behavior of the Online Simple Knapsack Problem with the option to reserve items was studied [23]. A later, more detailed study can be found in the PhD-Thesis by Henri Lotze [76].

Here, the authors present a complete study with matching upper- and lower bounds for all reservation costs which are linear dependent on the size of an item (so for an item of size (and value) x, the costs of reserving this item are given by αx for a given factor α).

As for many online knapsack problems without removability, a natural lower bound of 2 can be observed even for very low reservation costs. This is mainly due to the same reason why, e.g., an algorithm with constant many advice bits in an advice setting cannot yield a better competitive ratio than 2: If there are multiple items of size just shy above 0.5, an algorithm cannot know which of them will be the smallest. However, packing the smallest of those items might be necessary, as an adversary can present a counterpart to this item, that only fits together with the smallest item of all available items larger than 0.5. As constant advice cannot tell which item to pack here, it is also not possible to reserve a constant number of items to ensure the smallest item larger than 0.5 is reserved. When, however, reserving an unbounded number of such items, the cumulative reservation costs become unbounded. This makes the algorithm non-competitive even for an arbitrarily small factor $\alpha > 0$.

The competitive ratio of 2 is also achievable by an algorithm for every $\alpha \leq 0.25$. Afterwards, and not surprisingly, with an increasing factor for the reservation costs the competitive ratio also degrades. For a factor approaching 1, an algorithm must become uncompetitive. The behaviour can be found in figure 2.1.

Additionally, in the thesis also the variant of the Online Simple Knapsack Problem with Reservation and Reimbursement is studied, which models a variant in that the costs of reserving an item are only deducted of the algorithms solution quality when the item is not packed into the knapsack (or, equivalent, get reimbursed when an item is packed at the end). One can motivate this variant by several real-life applications, in which some costs only occur when some reserved option is not taken finally (e.g., a cancellation fee).

As in the variant without the reimbursement, an algorithm cannot be better than 2-competitive until 0.25. Afterwards, the competitive ratio increases almost linearly, with the same rate as in Han et al. [61] in the setting of removing items for some costs. Interestingly, the problem remains competitive even for arbitrarily large reservation costs ≥ 1.

In this thesis, further knapsack variants are studied, as well as new problems: First, and as arguably one of the most natural variants of the Online Simple

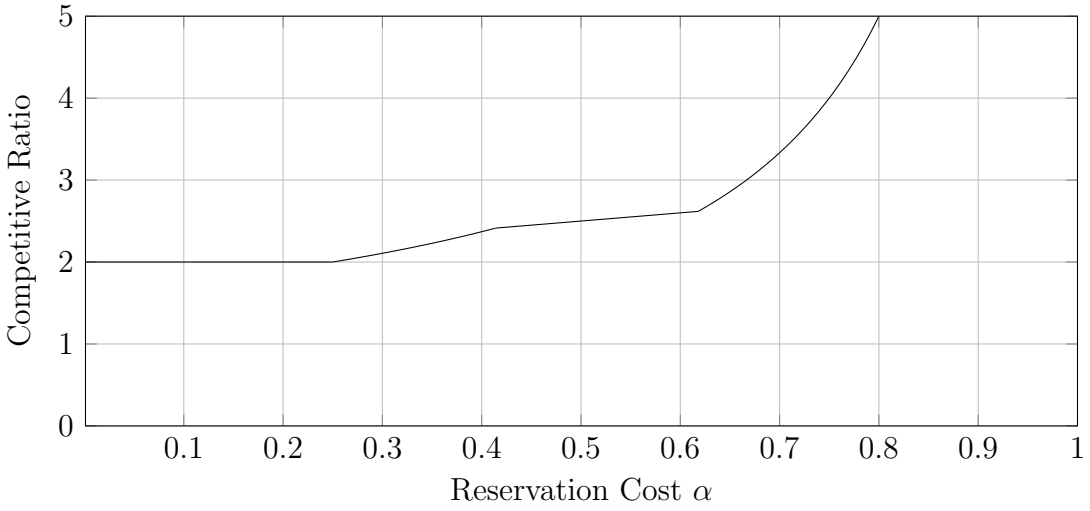


Figure 2.1: Competitive ratio of the Online Simple Knapsack Problem with Reservations, depending on the fixed value of α. This plot was first used in Böckenhauer et al. [23].

Knapsack Problem, the Online Simple Knapsack Problem with Removability will be studied with the additional option to reserve items. Here, we will see that even for arbitrarily small reservation costs, no algorithm will be able to achieve a competitive ratio of better than 1.5. The competitive ratio then rises with the increasing reservation costs immediately, before it finally reaches a ratio of Φ at $1 - \sqrt{5}/3 \approx 0.2546$. Afterwards, a ratio of Φ can be reached even without reserving items, as also shown in Iwama and Taketomi [65].

Arguably, the most natural next step when starting with a variant of the Online Simple Knapsack Problem is to ask which parts of the results translate when considering the Online General Knapsack Problem. Here, however, it is not as clear on how the reservation costs should be modeled: While for the Online Simple Knapsack Problem with Reservations, in all the three previously mentioned variants, the costs are depended based on the size/value of each item, for the Online General Knapsack Problem with Reservations those two parameters can differ. In this thesis, we study both:

- When the reservation costs depend on the item value, we will see that no algorithm can achieve a competitive ratio of better than 2 for any $\alpha > 0$, which is also optimal for small values of α. Unlike the Online Simple Knapsack Problem with Reservations, the competitive ratio of the general variant diverges at 0.5 instead of 1.

- When the reservation costs depend on the item size, it is rather easy to see that either no algorithm can be competitive (if using the definition of the

competitive ratio in the strict sense), or, when using the non-strict variant, the problem is 2-competitive even for arbitrarily large $\alpha > 0$.

The reservation setting however, is also sensible for other online optimization problems, apart from variants of the Online Knapsack Problem: In this thesis, we will also consider the Secretary Problem as one classical online problem with a different flavor due to its inherent random order variant, as well as graph problems such as the Online Vertex Cover Problem.

For the Secretary Problem with Reservations, next to accepting or rejecting a candidate on the spot, the option to delay that decision for some price is added - like asking a candidate to wait for the decision for some time. This will likely not come for free, as those candidates might otherwise apply at some other place and need some incentive not to do so. Here again, we consider two variants on how to include the reservation costs: First, the candidates can be asked to wait until the whole interviewing process has ended for some fixed price; or, in the second variant, the candidates can be asked to wait for some time and are paid for the duration of the time. While both models differ on technical aspects, it is not surprising that in both settings an algorithm can be arbitrarily close to optimality for small reservation costs, and degrades until the reservation is not sensible and the well-known 37%-rule of the classical Secretary Problem without reservations is best possible.

Finally, we will also apply the reservation setting to node-deletion problems such as Vertex Cover Problem or Feedback Vertex Set Problem, where each deletion can be postponed for some price. We will provide matching bounds for the Vertex Cover Problem with Reservations, as well as general bounds that apply to all node-deletion problems.

2.3 Knapsack with Removability

In this section, we study the Online Simple Knapsack with Removability problem in the variant where reservations are allowed. The Online Knapsack with Removability and Reservation problem can be defined in the following way:

Definition 2.3.1 (The Online Knapsack with Removability and Reservation Problem)**.** An instance of the Online Knapsack with Removability and Reservation problem consists of a series of *items* $I = (x_1, \dots, x_n)$ that arrives sequentially, with each *item* x_i consisting of a weight $w(x_i) \in [0, 1]$ and a value $v(x_i) > 0$. Additionally, a price $\alpha \geq 0$ is given. At each step $i \in \{1, \dots, n\}$, the item size and value of the request sequence are revealed. An algorithm then has the option to discard a subset of already packed items:

- **Discard** Set $K := K \setminus S$ for some subset $S \subseteq K$.

Afterwards, it can pack the new item in an initially empty knapsack K of size 1 if it fits, reserve it, or reject the item:

- **Pack** If $\sum_{x_k \in K} w(x_k) + w(x_i) \leq 1$, set $K := K \cup x_i$.
- **Reserve** Reserve the item for costs $\alpha v(x_i)$
- **Reject** Do nothing.

The Online Knapsack with Removability and Reservation problem is then for an algorithm `ALG` to minimize the competitive ratio between the value of an optimal packing with the items contained in the packing K together with the reserved items minus the paid reservation costs, compared to the optimal solution in the same instance.

As we will focus on the variant where the weight of an item equals its value, we will refer to the problem as the Online Simple Knapsack with Removability and Reservations Problem (OSKRR). In a slight abuse of notation, we will use the label x_i for an identifier of an item as well as to denote its size/value. The usage should be clear given the context at each point.

In this section, we provide tight upper and lower bounds of $\frac{3-1.5\alpha}{2-1.5\alpha}$ for $0 < \alpha \leq 1 - \sqrt{5}/3 \approx 0.2546$, which goes from 1.5 to ϕ. For larger values of α, the competitive ratio stays constant at ϕ. These bounds are depicted in Figure 2.2. This means that, even if the reservation costs are arbitrarily small the competitive ratio is worse than 1.5, and for any $\alpha > 0.2546$ the best achievable competitive ratio is the one that can be achieved without reserving any items.

This is in contrast to the behavior of the competitive ratio for the reservation model without removability, where the competitive ratio is constant at value 2 on the small range, and grows until being unbounded for large values of α.

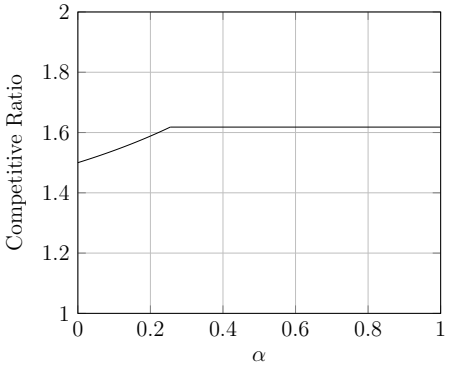


Figure 2.2: A schematic plot of the competitive ratio with respect to the reservation costs α for the online simple knapsack problem with reservation and removability.

2.3.1 Lower Bound

First note that no algorithm can be better than optimal, in particular, we have a lower bound of 1 for $\alpha = 0$.

In this section we show that no algorithm for the OSKRR can reach a competitive ratio smaller than $\min\{\frac{3-1.5\alpha}{2-1.5\alpha}, \phi\}$ for all $0 < \alpha < 1$. The ratio $\frac{3-1.5\alpha}{2-1.5\alpha}$ is equal to ϕ at $\alpha = 1 - \frac{\sqrt{5}}{3} \approx 0.2546$. For any larger values of α, we prove that it is not possible to construct an algorithm that performs better than the strategy of Iwama and Taketomi for the knapsack with removability and without reservation [65].

Theorem 2.3.1. *Given a parameter $0 < \alpha \leq 1 - \frac{\sqrt{5}}{3}$, there exists no algorithm for* OSKRR *achieving a competitive ratio better than $\frac{3-1.5\alpha}{2-1.5\alpha}$.*

Proof. We present an adversarial strategy to show that no algorithm can reach a competitive ratio of $\frac{3-1.5\alpha}{2-1.5\alpha} - \epsilon$ for any given $\epsilon > 0$. For the following analysis, we will abbreviate $\frac{2-1.5\alpha}{3-1.5\alpha}$ with k (the inverted competitive ratio, so the size an algorithm needs to achieve the competitive ratio independent on the optimal filling). Consider the following set of adversarial instances with $0 < \delta \ll \epsilon$:

The adversary starts by presenting the first item x_1 of size $1 - k + \delta$. In this situation, an arbitrary algorithm A can choose between the following options:

Case 1, Pack x_1: A packs the first item of size $1 - k + \delta$ into the knapsack. The adversary next presents the second item x_2 of size $k + \delta$ that barely does not fits together with the first item.

Case 1.1, Remove x_1 and pack x_2: A discards the item of size $1 - k + \delta$ and packs x_2. The adversary then presents an item of size $1 - x_1 = k - \delta$. This would perfectly fit together with the discarded item x_1, so OPT has a solution of size 1. Since this item does not fit together with x_2, A only holds x_2 as the larger of the two items, which results in the desired competitive ratio.

Case 1.2, Reserve x_2: A reserves the item of size $k+\delta$. The adversary presents the item x_3 of size $k+\delta^2$.
Case 1.2.1, Remove x_1 and pack x_3: Assuming A discards the first item that is still held in its knapsack to pack the item of size $k+\delta^2$. Similar to Case 1.1, the adversary then presents the counterpart to the discarded item of size $1-x_1 = k-\delta$. A cannot pack its reserved item together with the newly packed item, since $k-\delta+k+\delta^2 > 1$ for all $\alpha \in (0, 1-\frac{\sqrt{5}}{3})$. The gain of A is even smaller than in Case 1.1, due to the costs of reserving an item.
Case 1.2.2, Reserve x_3: A reserves the item of size $k+\delta^2$. The adversary presents x_4 of size $k+\delta^3$ and will continue to present items x_{j+1} of size $k+\delta^j$ as long as A reserves those items. At some point, A must stop reserving these items due to the accumulating reservation costs exceeding any possible gain. As soon as A rejects an item, or discards the item in its knapsack to pack an item, the case can be handled analogously to 1.2.1 or 1.2.3. The gain of the algorithm will be lower than in these cases due to the added reservation costs.
Case 1.2.3, Reject x_3: The algorithm does not pack the item of size $k+\delta^2$. The adversary presents an item of size $1-x_3$. So while OPT reaches 1, A can at the most pack x_1 and the item of size $1-x_3$. Together, with subtracted reservation costs, the algorithm then has a gain of $2-\frac{4-3\alpha}{3-1.5\alpha}-\alpha k$. A simple analysis shows that this yields a ratio higher than $\frac{3-1.5\alpha}{2-1.5\alpha}$.
Case 1.3, Reject x_2: A does not pack or reserve the item of size $k+\delta$. No further items are presented. The algorithm thus only holds x_1, while an optimal solution would be to pack the item x_2. The ratio of k to $1-k$ is worse than ϕ for every $\alpha < 1-\frac{\sqrt{5}}{3}$ and therefore, in particular, larger than the desired ratio.

Case 2, Reserve x_1: A reserves the first item of size $1-k+\delta$. The adversary presents the item x_2' of size $\frac{2}{3}+2\delta$.
Case 2.1 Pack x_2': The algorithm packs the item of size $\frac{2}{3}+2\delta$. The next item is x_3' of size $\frac{1}{3}+2\delta$.
Case 2.1.1 Remove x_2' and pack x_3': The algorithm discards the item of size $\frac{2}{3}+2\delta$ from its knapsack and instead packs the item of size $\frac{1}{3}+2\delta$. The adversary next presents an item of size $1-x_2'$. OPT then has a solution of size 1. A can at most reach a gain of $x_1+x_3'-\alpha x_1 = \frac{1}{3}+1-k+3\delta-\alpha(1-k) = k$.
Case 2.1.2 Reserve x_3': A reserves the item of size $\frac{1}{3}+2\delta$. The adversary continues to present items x_{j+1}' of size $\frac{1}{3}+\delta+\delta^{j-1}$, as long as A reserves those items. As soon as A rejects an item or discards the item already in its knapsack to pack an item, these cases can be handled analogously to 2.1.1 or 2.1.3. The gain of the algorithm will be lower than in these cases due to the added reservation costs.
Case 2.1.3 Reject x_3': The algorithm does not pack the item of size $\frac{1}{3}+2\delta$. The adversary then presents an item of size $1-x_3'$. Since x_3' was the smallest item of all that have been presented so far, the counterpart will not fit together with any

other item of A. Therefore OPT has a gain of 1, whereas the largest packing of A consists of the item x'_2 which is of smaller size than k.
Case 2.2 Reserve x'_2: A reserves the item of size $\frac{2}{3}+2\delta$. The adversary presents x''_2 of size $\frac{2}{3}+\delta+\delta^2$.

Case 2.2.1 Pack x''_2: A packs the item of size $\frac{2}{3}+\delta+\delta^2$ into the knapsack. The next item is x'_3 of size $\frac{1}{3}+2\delta$.
Case 2.2.1.1 Remove x''_2 and pack x'_3: The algorithm discards the item of size $\frac{2}{3}+\delta+\delta^2$ from its knapsack and instead packs the item of size $\frac{1}{3}+2\delta$. The adversary next presents an item of size $1-x''_2$. OPT then has a solution of size 1. A can at most reach a gain of $x_1+x'_3-\alpha x_1 = \frac{1}{3}+1-k+3\delta-\alpha(1-k) = k$.
Case 2.2.1.2 Reserve x'_3: A reserves the item of size $\frac{1}{3}+2\delta$. The adversary continues to present items x'_{j+1} of the size $\frac{1}{3}+\delta+\delta^{j-1}$, as long as A reserves those. As soon as A rejects an item or discards the item already in its knapsack to pack an item, these cases can be handled analogously to 2.2.1.1 or 2.2.1.3. The gain of the algorithm will be lower than in these cases due to the added reservation costs.
Case 2.2.1.3 Reject x'_3: The algorithm does not pack the item of size $\frac{1}{3}+2\delta$. The adversary then presents an item of size $1-x'_3$. Since x'_3 was the smallest item of all that have been presented so far, the counterpart will not fit together with any other item of A. Therefore OPT has a gain of 1, whereas the largest packing of A consists of the item $x'_2 < k$.
Case 2.2.2 Reserve x''_2: The algorithm reserves the item of size $\frac{2}{3}+\delta+\delta^2$. The adversary continues to present items x''_j of the size $\frac{2}{3}+\delta+\delta^j$, as long as A reserves these items. As soon as A rejects an item or discards the item already in its knapsack to pack an item, these cases can be handled analogously to 2.2.1 or 2.2.3. The gain of the algorithm will be lower than in these cases due to the added reservation costs.
Case 2.2.3 Reject x''_2: A does not pack the item of size $\frac{2}{3}+\delta+\delta^2$. The adversary then presents an item of size $1-x''_2$. The gain of OPT is then 1. The maximum gain of A is then $x_1+1-x''_2-\alpha(x_1+x'_2) = 1-k+\frac{1}{3}-\delta-\delta^2-\alpha(1-k+\frac{1}{3}-\delta-\delta^2)$ which is slightly smaller than the gain of A in Case 2.1.1 due to increased reservation costs and thus especially smaller than k.
Case 2.3 Reject x'_2: The algorithm does not pack the item of size $\frac{2}{3}+2\delta$. No further items are presented. Similarly to Case 1.3, the algorithm only holds x_1, while an optimal packing would be x'_2. Since even $x'_2 > x_2$, the same argumentation from Case 1.3 concludes this case.

Case 3, Reject x_1: No further items are presented. Therefore, the optimal solution is exactly the item $x_1 = 1-k$, while A has packed no items. The competitive ratio is then unbounded. □

It is not surprising that for increasing reservation costs, the competitive ratio

is increasing, too. In particular, it is easy to see that the competitive ratio cannot decrease if the reservation costs increase.

Corollary 2.3.1. *For $\alpha > 1 - \frac{\sqrt{5}}{3}$, no algorithm can be better than ϕ-competitive for the online simple knapsack problem with reservation and removability.*

Proof. The competitive ratio of Theorem 2.3.1 reaches ϕ at $\alpha = 1 - \frac{\sqrt{5}}{3}$. For every $\alpha > 1 - \frac{\sqrt{5}}{3}$ it is possible to present the lower bound with exactly the same items as in Theorem 2.3.1 for $\alpha = 1 - \frac{\sqrt{5}}{3}$. The only difference is the higher costs for every reservation. Thus, in every case, the gain of an optimal offline solution stays the same, while the algorithm gets at most the gain of the case $\alpha = 1 - \frac{\sqrt{5}}{3}$. □

2.3.2 Upper Bound

If $\alpha = 0$, then the OSKRR becomes an offline problem, since an algorithm can reserve every item for free. Thus, a competitive ratio of 1 is achievable. Trivially, this matches the lower bound.

However, as we will see now, as soon as the value of α is positive, the best achievable competitive ratio is worse than 1.5-competitive.

To handle the range of $0 < \alpha \leq 1 - \frac{\sqrt{5}}{3}$, we present Algorithm 2 with a competitive ratio of $\frac{3-1.5\alpha}{2-1.5\alpha}$, matching the lower bound established in the previous subsection.

Algorithm 2, analyzed in this section, can be split into three phases. All of the phases work similarly: First, if small items arrive, they are packed since they will not cause harm (we can throw them out at any time).

Then there are two kinds of interesting ranges for medium-sized items: A large-medium range, where the algorithm tries to keep the smallest items to maintain the possibility to get fitting counterparts (similar to [65]). And a small-medium range, where an item gets reserved, because otherwise the ratio between large and small items in the large-medium range, where only the smallest items are kept, gets too large by itself. After the first reservation, the algorithm continues in the second phase; after the second reservation in the third phase.

Finally, very large items, which are sufficiently large to guarantee the desired competitive ratio by themselves, are the easiest case since the algorithm can just pack these items to achieve the desired competitive ratio immediately.

The main difference between the three phases is that the ranges for the different actions are shifted — due to the ability to use the reserved items, but also due to the paid reservation costs that need to be compensated. Due to these shifts, in the third phase, there is no small-medium range, and there is no need to reserve any more items; thus, Algorithm 2 reserves at most two items.

Algorithm 2 Upper bound Algorithm for $0 < \alpha \leq 1 - \frac{\sqrt{5}}{3}$. OPT($S$) denotes the size of the optimal packing of the set S, $c := \frac{3-1.5\alpha}{2-1.5\alpha}$.

1: $K := \emptyset$; $R := \emptyset$
2: **for** $k = 1, \ldots, n$ **do**
3: **if** $|R| = 0$ **then**
4: **if** OPT$(K \cup \{x_k\}) \geq \frac{1}{c}$ **then** Pack OPT$(K \cup \{x_k\})$ **END**
5: **else if** $x_k \leq 1 - \frac{1}{c}$ **then** $K := K \cup \{x_k\}$;
6: **else if** $x_k \geq \frac{1}{c^2}$ **then** Keep only smallest $x'_k \in K \cup x_k$ with $x'_k \in [\frac{1}{c^2}, \frac{1}{c}]$;
7: **else**$(1 - \frac{1}{c} < x_k < \frac{1}{c^2})$ $R := R \cup x_k$, $x_f := x_k$;
8: **else if** $|R| = 1$ **then**
9: **if** OPT$(K \cup \{x_f, x_k\}) \geq \frac{1}{c} + \alpha x_f$ **then** Pack OPT$(K \cup \{x_f, x_k\})$ **END**
10: **else if** $1 - \frac{1}{c} - \alpha x_f < x_k \leq \frac{1}{3}$ **then** $R := R \cup x_k$, $g := k$;
11: **else if** $x_k \leq 1 - \frac{1}{c} - \alpha x_f$ **then** $K := K \cup \{x_k\}$;
12: **else** Keep only smallest $x'_k \in K \cup x_k$ with $x'_k \in [1 - x_f, \frac{1}{c} + \alpha x_f]$;
13: **else**
14: $M := \emptyset$; $L := \emptyset$
15: **if** OPT$(K \cup \{x_f, x_g, x_k\}) \geq \frac{1}{c} + \alpha(x_f + x_g)$ **then**
16: Pack OPT$(K \cup \{x_f, x_g, x_k\})$ **END**
17: **else if** $x_k \leq 1 - \frac{1}{c} + \alpha(x_f + x_g)$ **then** $K := K \cup \{x_k\}$;
18: **else if** $1 - x_g - x_f < x_k < \frac{1}{c} + \alpha(x_g + x_f) - x_f$ **then**
19: Keep x_k if $x_k + x_g < \min(x_l \in L) \wedge x_k < \min(x_m \in M)$; $M := M \cup x_k$;
20: **else if** $1 - x_g < x_k < \frac{1}{c} + \alpha(x_f + x_g)$ **then**
21: Keep x_k if $x_k + x_g < \min(x_m \in M) \wedge x_k < \min(x_l \in L)$; $L := L \cup x_k$;
22: Pack $OPT(K \cup R)$

Theorem 2.3.2. *For $0 < \alpha \leq 1 - \frac{\sqrt{5}}{3}$, there exists an algorithm solving* OSKRR *that is not worse than $\frac{3-1.5\alpha}{2-1.5\alpha}$ competitive.*

Proof. We prove that Algorithm 2 achieves the competitive ratio on the statement. For ease of notation, we define $c := \frac{3-1.5\alpha}{2-1.5\alpha}$. If Algorithm 2 is at any point able to pack a knapsack such that, even when paying the reservation costs, the desired competitive ratio is reached, it stops afterwards. This is the case in lines 4, 9, and 16.

Depending on the instance, Algorithm 2 reserves zero, one, or two items. In the following, we show that the algorithm yields the desired competitive ratio in those three cases.

Case 1: Algorithm 2 ends in line 4 or 22, before the first reservation is made.

When presented with a new item, the algorithm first checks if this item together

with a subset of its packed items already yields a packing of size $\frac{1}{c}$. In the following analysis, assume that this is not the case, since if this were the case, the algorithm would have achieved the desired competitive ratio anyway.

Items that are smaller than $1 - \frac{1}{c}$ are packed by the algorithm in line 5.

Items with sizes between $1 - \frac{1}{c}$ and $\frac{1}{c^2}$ will be reserved in line 7, if the condition in line 4 does not hold: We investigate this case as Case 2.

The first item of size between $\frac{1}{c^2}$ and $\frac{1}{c}$ is packed. If further items in this range arrive, only the smallest item of this range is packed (or kept) in the knapsack, and the other items from this range are discarded.

If an item x_k in this range, which should be kept in the knapsack according to line 6, does not fit, it must be due to the items already in the knapsack, which are each smaller than $1 - \frac{1}{c}$. Thus, in this case, it is possible to combine this item x_k with a subset of the small items in the knapsack to get a packing of size at least $\frac{1}{c}$, triggering the end of the packing in line 4.

On the other hand, if the instance ends before reserving the first item, the only possible difference between an optimal solution and a solution of the algorithm is that the algorithm only has the smallest item in the range between $\frac{1}{c^2}$ and $\frac{1}{c}$, while the optimal solution might have a different item in this range. Since the smallest item in this range is at least of size $\frac{1}{c^2}$ and the largest item is at most of size $\frac{1}{c}$, the ratio is smaller than $\frac{1/c}{1/c^2} = c$. Additional small items can be part of both the optimal solution and the solution provided by the algorithm, which only makes the ratio smaller.

Note that an optimal solution cannot contain two items in the range between $\frac{1}{c^2}$ and $\frac{1}{c}$, since the algorithm would also be able to combine the two items in this range. The algorithm always keeps the smallest possible item within the range, a second item within the range that could fit into the knapsack would trigger the condition in line 4, since two items of size at least $\frac{1}{c^2}$ are sufficient to reach $\frac{1}{c}$ ($\frac{2}{c^2} \geq \frac{1}{c}$ for $c \leq 2$).

Case 2: Algorithm ends in line 9 or 22 with one reserved item

Assume one item x_f in the range $1 - \frac{1}{c}$ to $\frac{1}{c^2}$ was reserved. The reservation costs for this item are αx_f. Thus, the algorithm needs to pack at least $\frac{1}{c} + \alpha x_f$ to guarantee the desired competitive ratio independently of the gain achieved by an optimal solution.

For items smaller than $1 - \frac{1}{c} - \alpha x_f$, the same argument for the small items in Case 1 holds: These items can be packed in a greedy manner until the threshold $\frac{1}{c} + \alpha x_f$ is reached. An item of size between $1 - \frac{1}{c} - \alpha x_f$ and $\frac{1}{3}$ will be reserved in line 10, if the algorithm cannot pack a knapsack of size $\frac{1}{c} + \alpha x_f$ at this point. This case is discussed in the next part of the proof, as Case 3.

Note that it is sufficient to combine the first reserved item with a single item of size at least $\frac{1}{3}$ to reach the desired competitive ratio, even after paying the

reservation costs, as

$$\frac{1}{3} + (1-\alpha)x_f \geq \frac{1}{3} + (1-\alpha)\Big(1 - \frac{1}{c}\Big) = \frac{1}{c},$$

in the considered range of α. In addition, any item that reaches $\frac{1}{c} + \alpha x_f$ either by its own or together with small items is sufficient as well. Thus, with every item between $\frac{1}{3}$ and $1 - x_f$, or larger than $\frac{1}{c} + \alpha x_f$, the threshold is reached already and the algorithm stops in line 9.

We are left to consider items of sizes between $1 - x_f$ and $\frac{1}{c} + \alpha x_f$, which will reach line 12 in the algorithm. With the items in this range, we follow the same strategy as for the large-medium items before the first reservation: The smallest of those items will be kept in the knapsack. This works for the same reasons as in Case 1, as explained below.

Assume the instance stops here and the algorithm was not able to reach $\frac{1}{c} + \alpha x_f$: The only difference between the solution of the algorithm and an optimal solution is that the algorithm sticks to the smallest item in the range between $1 - x_f$ and $\frac{1}{c} + \alpha x_f$, which gives us the worst possible case. Note that no item of size smaller than $1 - x_f$ can be rejected before the first reservation, since it otherwise would have been packed in combination with f. The ratio between $\frac{1}{c} + \alpha x_f$ and $1 - x_f$ is less than c, since x_f is at most $\frac{1}{c^2}$. Again, additional small items can be added to both the optimal solution and the solution provided by the algorithm, only decreasing the ratio.

Case 3: Algorithm ends in line 16 or 22 with two reservations made

Assume two items were reserved, the first item x_f of size between $1 - \frac{1}{c}$ and $\frac{1}{c^2}$, and the second item x_g of size between $1 - \frac{1}{c} - \alpha x_f$ and $\frac{1}{3}$. Therefore, the algorithm has already paid $\alpha(x_g + x_f)$ as reservation costs, and now needs a packing of size $\frac{1}{c} + \alpha(x_g + x_f)$ to guarantee the competitive ratio of c independent of any optimal solution size.

First note that an item or a subset of items is sufficient to reach $\frac{1}{c} + \alpha(x_f + x_g)$ in three cases: it either reaches the threshold by its own, reaches the threshold together with one of the reserved items, or with the two reserved items combined.

To reach the threshold with one of the reserved items, a subset of items must have a size between $\frac{1}{c} + \alpha(x_f + x_g) - x_f$ and $1 - x_f$, or between $\frac{1}{c} + \alpha(x_f + x_g) - x_g$ and $1 - x_g$ (depending on which of the reserved items will be used to combine this subset with). The subset sizes that combined with x_f are sufficient to reach the necessary threshold can be smaller than the subsets that can be combined with x_g, since $x_g < \frac{1}{3}$ and $x_f > \frac{1}{3}$. With simple analysis, it can be shown that for any α between 0 and $1 - \frac{\sqrt{5}}{3}$ and for every choice of x_g and x_f in the given ranges, $1 - x_g$ must be larger than $\frac{1}{c} + \alpha(x_f + x_g) - x_f$. Therefore, these intervals overlap, as displayed in Figure 2.3.

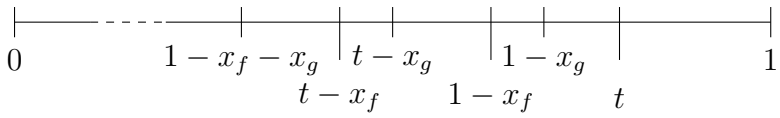


Figure 2.3: A schematic plot of the size ranges in the algorithm when two items have already been reserved. The targeted size for the items is $t = \frac{1}{c} + \alpha(x_g + x_f)$.

It follows that there are three intervals in which an item can be presented, and the algorithm might not stop immediately:

1. Items smaller than $\frac{1}{c} + \alpha(x_g + x_f) - x_g - x_f$ are too small to reach the necessary size in combination with both of the reserved items. Note that those are smaller than $1 - \frac{1}{c} - \alpha(x_g + x_f)$, thus they can be packed greedily. As soon as the knapsack is overfull due to those items, a subset of the small items can be selected to reach $\frac{1}{c}+\alpha(x_g+x_f)$. This argumentation is analogous to the small items argument in the two previous cases.

2. Items between $1 - x_g - x_f$ and $\frac{1}{c} + \alpha(x_g + x_f) - x_f$, which are too large to fit together with both reserved items, but too small to reach the threshold in combination with the largest reserved item. We call those items *mid-sized*.

3. Items that cannot be combined with any of the reserved items but also do not reach the threshold by themselves, i.e., items between $1 - x_g > \frac{2}{3}$ and $\frac{1}{c} + \alpha(x_f + x_g)$. Those items will be called *large*.

As we have already justified, the small items in 1. are not a concern. We argue now that the desired competitive ratio is also achieved when mid-sized and large items appear in the request sequence after reserving two items.

The algorithm keeps either the smallest large item or the smallest mid-sized item. This is dependent on the situation, as detailed in lines 18 to 21: If the smallest large item is smaller than the smallest mid-sized item in combination with x_g, the algorithm keeps the smallest large item. On the other hand, if the smallest mid-sized item in combination with x_g is smaller than the smallest large item, the algorithm keeps the smallest mid-sized item. Note that we can assume that every large item is larger than $\frac{2}{3}$, since otherwise it can be combined with x_g, reaching the threshold $\frac{1}{c} + \alpha(x_g + x_f)$.

Therefore, the following crucial observation holds: If the algorithm is able to pack either one large item together with x_f, x_g, or a mid-sized item, or the algorithm is able to pack three items among x_f, x_g, and mid-sized items, the algorithm will also reach the desired threshold and competitive ratio.

To see this, first note that every mid-sized item of size smaller than or equal to $\frac{1}{3}$ will be packed into the knapsack and will never be replaced by a large item.

This is the case because every large item must be larger than $\frac{2}{3}$, and an item of size smaller than $\frac{1}{3}$ in combination with x_g must be smaller than or equal to $\frac{2}{3}$.

Assume that, at some point, an arbitrary algorithm is about to pack the last item of a combination of one large and one mid-sized or reserved item, or a combination of three mid-sized or reserved items. Due to the construction and the selection of items before, we are able to do so as well:

- If the last item is a large item, the algorithm is able to pack such a combination, since the smallest mid-sized item is in the knapsack if it is smaller than the reserved x_g. Otherwise, the reserved x_g is available for a combination.
- If the last item is a mid-sized one, it fits together with the current combination of items selected by Algorithm 1 (either two mid-sized or one large item), since it is the smallest possible combination by construction.

A simple calculation then shows that every such combination of items (one large and one mid-sized/reserved item, or three mid-sized/reserved items) is of size at least $\frac{1}{c} + \alpha(x_g + x_f)$, even if chosen as small as possible.

The only thing left to prove is that even if there is no such combination, the algorithm still reaches the designated competitive ratio:

First, note that large items are only rejected or removed from the knapsack in the first two phases. Those items had at most size $\frac{1}{c} + \alpha x_f$. Items that remained in the knapsack can be considered as either small items that can be packed in a greedy manner, or as large items. If there were an item that is not in the allowed range for large or small items in this phase, it would have been used to reach the threshold for a guaranteed competitive ratio immediately, just like it would have been possible if the item had been presented in the third phase.

By using mid-sized and large items together with the reserved ones, every optimal solution can only reach $\frac{1}{c} + \alpha(x_g + x_f)$, while Algorithm 1 cannot be worse than $x_g + x_f$ on any instance. An easy calculation shows that even for the smallest possible sizes of x_g and x_f, their combination is sufficient to reach the competitive ratio of c, since an optimal solution cannot be better than $\frac{1}{c} + \alpha(x_g + x_f)$. Since Algorithm 1 can use small items in the same manner as an optimal solution, the appearance of small items cannot worsen the ratio. □

This concludes the analysis for an upper bound that matches the lower bound for values of α smaller than $\alpha \geq 1 - \sqrt{5}/3$.

For any $\alpha \geq 1 - \sqrt{5}/3$, note that the algorithm for Online Simple Knapsack with Removability (without reservation) presented by Iwama and Taketomi [65] achieves a competitive ratio of ϕ, thus matching our lower bound as well.

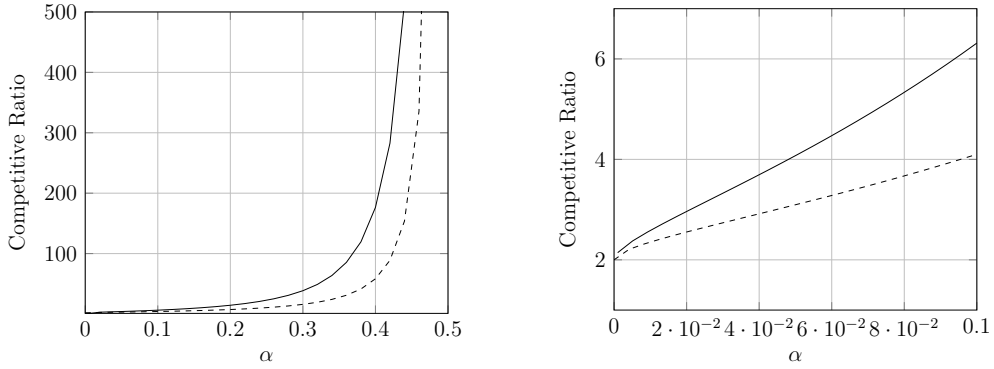


Figure 2.4: A schematic plot of the upper and lower bounds for reservation depending on item value. The dashed line represents the general lower bound proven in Theorem 2.4.1 and the continuous line is the upper bound achieved in Corollary 2.4.1 by analyzing Algorithm 3.

2.4 The General Knapsack Problem

In this section, we present upper and lower bounds for the online general knapsack with reservation costs, when the reservation costs depend both on the size and value of the items.

For the reservation costs depending on the value of the items, we get the upper and lower bounds as depicted in Figure 2.4. The upper bound is

$$\frac{2(2\alpha+\sqrt{\alpha(1+2\alpha)})(1+2\alpha+2\sqrt{\alpha(1+2\alpha)})}{(1-2\alpha)^2\sqrt{\alpha(1+2\alpha)}},$$

whereas the lower bound is $\frac{2(1+\sqrt{\alpha(2-3\alpha)}-\alpha)}{(1-2\alpha)^2}$. Here, the strict notation of competitive ratio can be used.

We can see that the (strict) competitive ratio is 2 when the reservation costs go to 0 and the problem becomes not competitive for reservation costs of 0.5 times the value of the item. In between those scenarios, we have upper and lower bounds that do not match but that are asymptotically similar.

When the reservation costs depend on the size of the items, we see that, regardless of the reservation costs, there is an upper and lower bound of 2 for the (non-strict) competitive ratio. It is also easy to see that the problem is non-competitive when using the strict definition of competitive ratios.

2.4.1 Reservation costs proportional to item value

In this section, we present upper and lower bounds that depend on the value of the item. We present the results using the strict version of the competitive ratio in this setting, which, of course, yields also non-strict results in the upper bounds as

well for any positive value of β. This is not as straightforward when we consider the lower bounds, but we will see that even the lower bound results will extend to non-strict competitive ratios.

In general, given an item $x \coloneqq (s, v)$ that has size s and value v, we talk about the density of an item as the ratio between its value and size, namely $d(x) = v/s$. We will sometimes omit the dependency on x through the section if it is clear by context. In a request sequence, we will often mark each item and its size and value by its index in the sequence, namely, the i-th item is $x_i \coloneqq (s_i, v_i)$.

First, we see that, when the reservation costs depend on the item value, one can translate the lower bounds from the online simple knapsack:

If the reservation costs depend on the value, an adversary can present an instance with fixed density 1, i.e., the items are, for every i, $x_i = (s_i, s_i)$ and mimic a simple knapsack instance. Thus, no algorithm for online general knapsack with reservation costs depending on the value can achieve a better competitive ratio than the bounds for the online simple knapsack with reservation costs.

This observation is valid for both the strict and non-strict versions of the competitive ratio, because, if instead of choosing density 1, we choose an arbitrary density d, one can offset any additive constant β when computing the competitive ratio. For this reason, all of the lower bounds in this section will also be calculated as strict ratios, as one can always scale the density of all of the offered items by an arbitrary value to overcome any constant β.

We cannot make a similar claim for the upper bound, as no algorithm is guaranteed to receive an instance where the item density is fixed. However, any upper bound derived in this section is also a valid upper bound for the online simple knapsack, as the competitive ratio is always calculated on a worst-case basis. Thus, any algorithm with a competitive ratio at most c for the online general knapsack will also perform just as well on simple knapsack instances (that is, instances with fixed density).

The lower bounds offered by applying the bounds found in the simple knapsack case, however, are not as good as those that can be achieved by considering instances where the items have different densities.

Lower Bound

For the lower bound, we build a set of adversarial instances such that no algorithm can achieve a better ratio than the proposed bound. A few observations will help to analyze the behavior of these algorithms:

Lemma 2.4.1. *No algorithm packing items before the end of the instance is competitive.*

Proof. If an algorithm decides to pack an item before the end of the request sequence has been announced, the adversary can always craft an item x of size 1 and arbitrary value, which will render such an algorithm uncompetitive. Because the value of x is arbitrary, this holds true for both the strict and non-strict versions of the competitive ratio. □

Thus, we do not need to consider algorithms that preemptively pack during our lower bound analysis. The same holds for algorithms that do not reserve the first item:

Lemma 2.4.2. *Any algorithm not reserving any item is uncompetitive.*

Proof. If an algorithm decides not to reserve the first item, the adversary can stop the input sequence there, which will render such an algorithm strictly uncompetitive. For the non-strict version, an adversary can offer increasingly denser items of the same size until an arbitrary value is achieved. If an algorithm does not reserve any of these items, the algorithm cannot be competitive for any fixed value of β in the non-strict sense. Thus, by scaling the density of the first reserved item, we can assume that the first item offered by the adversary is reserved. □

With these considerations, we can construct a lower bound for all other algorithms:

Theorem 2.4.1. *No algorithm for the online general knapsack can achieve a competitive ratio smaller than* $\frac{2(1-\alpha+\sqrt{(2-3\alpha)\alpha})}{(1-2\alpha)^2}$ *for a reservation factor of* $0 < \alpha < 0.5$.

Proof. Let $\epsilon_1, \epsilon_2 > 0$ and N be a natural number. The adversarial instances consist of items $x_k := (1 - k\epsilon_1, v_k)$ for $k \geq 0$, starting with $x_0 := (1, 1)$. Each value v_k is larger than its predecessor by a decreasing factor f_k, so $v_k := f_1 f_2 \dots f_k$. Here f_1 is chosen sufficiently large (e.g., by a factor of $(1 + \epsilon_2)$ larger than the aimed competitive ratio $\frac{2(1-\alpha+\sqrt{(2-3\alpha)\alpha})}{(1-2\alpha)^2}$). Furthermore, f_k decreases slowly towards a factor of 1, such that at any point $k > N$ the factor f_k is at least $(1 - \epsilon_2)f_{k-N}$. If an algorithm decides not to reserve an item x_k, the adversary stops presenting further items of this form.

If the decision to not reserve the item x_k was made for $k = 0$, the instance ends, and the algorithm is not competitive as described above.

If the algorithm decides not to reserve an item x_k for $0 < k < N$, the instance ends as well. The optimal solution can consist of x_k, which is at least $\frac{2(1-\alpha+\sqrt{(2-3\alpha)\alpha})}{(1-2\alpha)^2}$ times larger than the largest reserved item from the algorithm x_{k-1}. Therefore, the competitive ratio in this case also exceeds the claimed one. This case uses the construction that each factor f_i for $i \leq N$ is still larger than $\frac{2(1-\alpha+\sqrt{(2-3\alpha)\alpha})}{(1-2\alpha)^2}$ times it predecessor, which is also the ratio between x_k and x_{k-1}.

If an algorithm decides to not reserve an item x_k for $k \geq N$, the algorithm presents an item $y \coloneqq (k\epsilon_1, v_{k-1})$. If the algorithm decides to reserve y, the instance stops; otherwise, copies of y will be presented as long as either a copy gets reserved or more than $\frac{2(1-\alpha+\sqrt{(2-3\alpha)\alpha})}{(1-2\alpha)^2}$ copies of y got presented. If no copies of y got reserved, the optimal solution can pack all of them together, resulting in a packing worth $\frac{2(1-\alpha+\sqrt{(2-3\alpha)\alpha})}{(1-2\alpha)^2} v_{k-1}$ while the algorithm only achieves v_{k-1}. Even ignoring the reservation costs the algorithm has paid so far, the algorithm can not achieve the desired competitive ratio. Note that this assumes ϵ_1 to be sufficiently small, which can be achieved by choosing ϵ_1 accordingly for fixed N and ϵ_2. Therefore, we assume that the algorithm will reserve one copy of y.

In this case, the optimal solution consists of the item x_k together with y, achieving $v_k + v_{k-1}$. The algorithm can pack either y or x_{k-1}, both with value v_{k-1}. Furthermore, the algorithm will have to pay reservation costs of αv_{k-1} for y plus

$$\alpha \sum_{i=0}^{k-1} v_i \geq \alpha \sum_{i=k-N}^{k-1} v_i \geq \alpha v_{k-N} \sum_{i=0}^{N-1} f_k^i = \alpha v_{k-N} \frac{1-f_k^N}{1-f_k}$$

for the reserved items x_i.

Therefore, the algorithm achieves a competitive ratio of

$$\frac{v_k + v_{k-1}}{v_{k-1} - \alpha v_{k-1} - \alpha v_{k-N} \frac{1-f_k^N}{1-f_k}} = \frac{f_k + 1}{1 - \alpha - \alpha \frac{v_{k-N}}{v_{k-1}} \frac{1-f_k^N}{1-f_k}}$$

For every ϵ_3, the geometric sum can be bounded as a series for a sufficiently large chosen N, resulting in a competitive ratio of

$$\frac{f_k + 1}{1 - \alpha - \alpha(\frac{1}{1-1/f_k} - \epsilon_3)} .$$

A factor $f_k = \frac{-1+\alpha-\sqrt{2\alpha-3\alpha^2}}{-1+2\alpha}$ promises the minimal competitive ratio among all possible factors. The resulting competitive ratio for this x_k turns out to be exactly

$$\frac{2(1-\alpha+\sqrt{(2-3\alpha)\alpha})}{(1-2\alpha)^2} .$$

Note that this minimum exists as it is defined by a trade-off between high total reservation costs (in case of a smaller factor f) and of course the ratio between the algorithm's item and the large item the optimal solution can use (in case of larger factor f).

Finally, note that if the algorithm does reserve all items x_k, at some point it will reserve some item x_y with $f_y < 1 + \epsilon_2$. Then the instance ends. The

Algorithm 3 Algorithm for an upper bound in the setting of reservation costs depending on the value for a given $c \geq 1$. R_s contains the densest reserved items of combined size at least 1

$R_s \coloneqq \emptyset$; ▷ Densest reserved items up to total size 1
$d_\delta^{R_s} \coloneqq 0$; ▷ Smallest density of all items in R_s
for $k = 1, \ldots, n$ **do**
  if $s(R_s) < 1$ **then** reserve x_k;
    $R_s \coloneqq R_s \cup x_k$;
    repeat
      Define x_δ as least dense item in R_s;
      if $s(R_s \setminus x_\delta) \geq 1$ **then** $R_s \coloneqq R_s \setminus x_\delta$;
    until $s(R_s \setminus x_\delta) < 1$;
    Set $d_\delta^{R_s}$ as density of least dense item in R_s;
  else if $d(x_k) \geq c \cdot d_\delta^{R_s}$ **then** reserve x_k;
    $R_s \coloneqq R_s \cup x_k$;
    repeat
      Define x_δ as least dense item in R_s;
      if $s(R_s \setminus x_\delta) \geq 1$ **then** $R_s \coloneqq R_s \setminus x_\delta$;
    until $s(R_s \setminus x_\delta) < 1$;
    Set $d_\delta^{R_s}$ as density of least dense item in R_s;
  else reject x_k;
pack the reserved items optimally

algorithm, as well as an optimal solution, can pack at most the value v_k of the last presented item. However, from the algorithm's value we need to deduct the reservation costs of at least $\alpha N(v_k - 2\epsilon)$ for the last N items. With sufficiently large N, the algorithm does not achieve a bounded ratio. □

Upper Bounds

In this section, we present Algorithm 3, which provides an upper bound that goes to 2 for small values of α, and diverges at $\alpha = \frac{1}{2}$ at the same asymptotical rate as the lower bound.

We already defined the item density for a given item. For this bound, we want to group subsets of items together. For a given set of items X, we define its value $v(X) = \sum_{x_i \in X} v_i$ and size $s(X) = \sum_{x_i \in X} s_i$, and accordingly its density as the ratio between them $d(X) = v(X)/s(X)$. To build an algorithm for the upper bound, we will consider at all times the densest knapsack R_s, i.e., from the reserved item set, take the subset containing the densest items that add up to 1 or more. Observe that R_s might not be packable as is, namely, the least dense item

in R_s might not fully fit inside the knapsack.We also define $x_\delta^{R_s}$ or to abbreviate x_δ to be the least dense item in R_s, and $d_\delta^{R_s}$ or d_δ its density.

Let us consider Algorithm 3: This algorithm reserves items as long as they are "dense enough". On a first stage, this algorithm reserves every item until the total size of the reserved items would overflow the knapsack, as shown in line 4. Then, in line 11, the algorithm reserves items as long as they are c times more dense than d_δ. Otherwise, the offered item is rejected. At every stage, the algorithm maintains R_s containing the densest subset of reserved items with total size 1 or larger, as we can see in the loops starting in lines 6 and 13.

Theorem 2.4.2. *Algorithm 3 achieves a strict competitive ratio of at most*

$$\frac{2c}{1-\alpha(\frac{4}{1-\frac{1}{c}}-2)}\,.$$

for a given $c > 1$ and $0 < \alpha < \frac{1}{2}$ when the reservation costs depend on the item value.

Proof. Without loss of generality, assume that $d_\delta = 1$ at the end of the algorithm. All of the items with a density larger than c have been reserved and fit in R_v. Let us say all items with a density larger than c have a combined value of v and a combined size of $s < 1$. There can be items with density in the interval $(1, c]$ that are also in R_v and have a combined value of w and a size of t, with $t + s < 1$.

Algorithm 3 is able to pack half of the value of R_v. This is because, by construction, only the last item in R_v might not fit into the knapsack with the rest of them. Thus, either the last item alone contains more than half the value of R_v or the rest of the items do. Choosing the appropriate subset of items in R_v, thus, results in a packing with value at least $\frac{v+w+(1-s-t)}{2}$, but we can also be sure that the solution is at least $v + w$ by only taking the items of density larger than 1. Additionally, if the algorithm reserves an amount of x of items with density 1, the algorithm is able to pack a value of at least $\frac{x}{2}$.

The reservation costs are the sum of the reservation costs of v and w and the reservation costs of the less dense items. The costs for the dense items are simply $\alpha(v + w)$. For items of density exactly 1, we can have at most value $x \leq 2 - t$, which takes into account the overflowing of R_v at a time when the items that make up v might not have arrived. The reservation costs for the smaller items can be calculated by a geometric sum; and, as at most a size of 2 can be reserved until the marker was increased by a factor of c, the costs are at most $\alpha 2 \sum_{i=1}^{k} \frac{1}{c^i} \leq \alpha(\frac{2}{1-1/c} - 2)$. This is also tight as items of combined size $1-\varepsilon$ can be reserved without increasing the marker, followed by one item that then increases the density at the marker and can have size of at most 1.

The optimal solution can consist of reserved and unreserved items. With unreserved items having density at most c, the optimal solution can not be larger than $v + (1 - s)c$.

Together, the algorithm achieves a competitive ratio of at least

$$\frac{v + (1 - s)c}{\max\left\{\frac{v+w+1-s-t}{2}, v + w, \frac{x}{2}\right\} - \alpha(v + w + x + \frac{2}{1-\frac{1}{c}} - 2)}.$$

For an upper bound analysis, and given a factor c and α, we need to minimize this ratio. On the other hand, v, w, s, and t are chosen adversarially to maximize the ratio.

First, note that the ratio is maximized when $w = t = 0$, as w only increases the gain of the algorithm, and a higher t only contributes even negatively to the reservation costs due to the restriction of $x \leq 2 - t$. For this, observe that the algorithm packs at least $v + w$ into the knapsack, which means that by subtracting the reservation costs, the w contribution is $(1 - \alpha)w$, which is always positive. Therefore, the ratio looks like this:

$$\frac{v + (1 - s)c}{\max\left\{\frac{v+1-s}{2}, v, \frac{x}{2}\right\} - \alpha(v + x + \frac{2}{1-\frac{1}{c}} - 2)},$$

where we integrate the 2α term into the geometric sum.

Observe that if $\max\left\{\frac{v+1-s}{2}, v\right\} = v$, $v \geq 1 - s$ whereas $\max\left\{\frac{v+1-s}{2}, v\right\} = \frac{v+1-s}{2}$ when $v \leq 1 - s$. If we take the derivative with respect to v of both of these ratios, we see that it is always negative for any $c > 1$ and $0 \leq \alpha \leq \frac{1}{2}$. Thus, to maximize the ratio, every adversarial instance will not present items denser than c. One can also see this in the following way. Even when comparing just the packing of the algorithm with the optimal solution, one easily sees that having dense items favors the algorithm: Not only is the ratio between v and $v/2 - \alpha v$ smaller than the ratio between $(1 - s)c$ and $(1 - s)/2 - \alpha(1 - s)$, which alone would be a reason to set v (and therefore s) to zero when trying to achieve a worst possible ratio, it also gets more supported by the fact that the algorithm still has to pay further reservation costs (which therefore lessens the ratio even more, when we cut out profit for both the algorithm and the optimal solution). Therefore, the ratio is largest when no items denser than c appear in the request sequence (namely $v = 0$ and $s = 0$):

$$\frac{c}{\max\left\{\frac{1}{2}, \frac{x}{2}\right\} - \alpha(x + \frac{2}{1-\frac{1}{c}} - 2)}.$$

It is easy to see that the worst case for an algorithm and its competitive ratio is an instance in which the algorithm reserved items with density 1 just to overpack the knapsack slightly (for example, by two items of size $\frac{1}{2} + \varepsilon$ each). This is, as the

guaranteed gain of an algorithm increases when allowing it to reserve more items of density 1, as the algorithm can use at least half of it for its packing. This holds even if the reservation costs are deducted for twice as much as the algorithm can use in its final packing (as we only consider $\alpha < 0.5$ here). Note that x must be at least one as 1 is the smallest density in R_s at the end of the request sequence, thus if $s = w = 0$, it must mean R_s only contains items of density 1.

Therefore, given a fixed c, the competitive ratio is

$$\frac{c}{\frac{1}{2} - \alpha(1 + \frac{2}{1-\frac{1}{c}} - 2)} = \frac{2c}{1 - \alpha(\frac{4}{1-\frac{1}{c}} - 2)} .$$

□

Corollary 2.4.1. *Algorithm 3 achieves a strict competitive ratio of at most*

$$\frac{2(2\alpha + \sqrt{\alpha(1+2\alpha)})(1 + 2\alpha + 2\sqrt{\alpha(1+2\alpha)})}{(1-2\alpha)^2\sqrt{\alpha(1+2\alpha)}} .$$

for all $0 < \alpha < \frac{1}{2}$ *when* $c = \frac{2\sqrt{2\alpha^2+\alpha}+2\alpha+1}{1-2\alpha}$

Proof. By Theorem 2.4.2, we know that Algorithm 3 achieves a competitive ratio of

$$\frac{2c}{1 - \alpha(\frac{4}{1-\frac{1}{c}} - 2)}$$

for a given $c > 1$. We can fix an optimal version of Algorithm 3 by choosing the best factor c for each reservation cost α. We take the derivative with respect to c and obtain

$$\frac{2}{1 - \alpha\left(\frac{4}{1-\frac{1}{c}} - 2\right)} - \frac{8\alpha}{\left(1 - \alpha\left(\frac{4}{1-\frac{1}{c}} - 2\right)\right)^2 \left(1 - \frac{1}{c}\right)^2 c}$$

which becomes 0 for $c = \frac{2\sqrt{2\alpha^2+\alpha}+2\alpha+1}{1-2\alpha}$, thus achieving a competitive ratio of

$$\frac{2(2\alpha + \sqrt{\alpha(1+2\alpha)})(1 + 2\alpha + 2\sqrt{\alpha(1+2\alpha)})}{(1-2\alpha)^2\sqrt{\alpha(1+2\alpha)}}.$$

□

2.4.2 Reservation costs proportional to item size

When the reservation costs depend on the size of an item, every online algorithm is not competitive in the strict sense:

If items of the type $x = (s, \alpha s)$ for any size s between 0 and 1 are presented, every reservation results in a net gain of 0 if the item gets reserved and then used, and in a negative gain if the item is not used. Thus, no reasonable algorithm reserves any item.

However, any items that directly get taken into the knapsack make the algorithm strictly not competitive. This is because the lower bound approach for the online general knapsack [77] can be applied here. Namely, if the first offered item is not taken, no more items will appear, rendering such algorithms uncompetitive. However, if the first offered item is placed into the knapsack, an item $x = (1, V)$ will be offered of size 1 and arbitrary value V, which would be taken by the optimal solution but cannot be taken by an algorithm with a partially occupied knapsack, rendering all such algorithms uncompetitive as well.

In this section, we prove that the general knapsack problem with reservation has a non-strict competitive ratio of 2 for any $\alpha > 0$.

Lower bound

We can achieve a lower bound of 2 by constructing an adversarial instance as follows:

Theorem 2.4.3. *Given a reservation cost of $0 < \alpha \leq 1$ for the item size, no online algorithm solving the online knapsack problem can be c-competitive even in the non-strict meaning for any constant $c < 2$.*

Proof. Given a reservation cost α, let us consider the following adversarial request sequence.

The first item is $x_1 = (\frac{1}{2} + \epsilon, C)$ for some $\frac{1}{2} > \epsilon > 0$ and some large value of C. If the item is taken, an item $x_2 = (1, 3C)$ is presented, and the request sequence ends. Otherwise, if the item is rejected, no other item arrives, and the request sequence ends. However, the item can also be reserved; if this is the case, items $x_i = (\frac{1}{2} + \epsilon^i, C)$ are presented until either x_i is taken for some value of i and an item $(1, 3C)$ is presented, or x_i is rejected for some i, and a complementary item $x'_i = (\frac{1}{2} - \epsilon^i, C)$ arrives. The adversary will present items x_i until one is taken or rejected or until the sum of the total reservation costs adds up to C.

If the request sequence terminates after an item $(1, 3C)$, the algorithm achieves a gain of at most C, but the optimal gain is at least $3C$; this means that for any constant $\beta > 0$, we can choose C such that the competitive ratio is larger than 2. Here, take into account that C can be an arbitrarily large value that can overcome any chosen constant β.

If the request sequence terminates after an item x_i is rejected, the optimal takes x_i and x'_i into the knapsack, with an total value of $2C$, where the algorithm can only pack a total value of C, achieving again a competitive ratio as close to 2

Algorithm 4 Algorithm for reservation costs depending on the size of an item, where D and c are constants greater than 0.

1: $R_s \coloneqq \emptyset$; ▷ Densest reserved items up to total size 1
2: $d_\delta^{R_s} \coloneqq 0$; ▷ Smallest density of all items in R_s
3: **for** $k = 1, \ldots, n$ **do**
4: **if** $d(x_k) \leq D$ **then** reject x_k;
5: **else** follow Algorithm 3 from line 4 to 18
6: pack the reserved items optimally

as necessary for an arbitrary choice of C that overcomes the value of any constant β.

If the request sequence terminates because the first item is rejected, we can choose $C > 2\beta$ and get a competitive ratio larger than 2.

Otherwise, if the algorithm keeps reserving every offered item, at some point the reservation costs become than C, at which point the request sequence ends, the gain of such an algorithm is negative, and the desired competitive ratio is also achieved by choosing a $C > 2\beta$. □

Upper Bound

Let us consider Algorithm 4 for the online general knapsack. This algorithm rejects any item that is less dense than a fixed density D. For the denser items, the algorithm operates just like Algorithm 3, first reserving items until R_s is complete and then reserving items that are a factor c denser than the least dense item in R_s. Observe that in this algorithm c and D are constants and can be chosen. Thus, we prove the following theorem.

Theorem 2.4.4. *Given a reservation factor α and an $\epsilon > 0$, we can choose c and D in Algorithm 4 such that the competitive ratio of Algorithm 4 is less than $2 + \epsilon$.*

Proof. Given a reservation factor α and an $\epsilon > 0$, we want to choose c and D in Algorithm 4 such that the competitive ratio of Algorithm 4 is smaller than $2 + \epsilon$. This means that for every request sequence S

$$\text{gain}_{\texttt{OPT}}(S) \leq (2 + \epsilon) \cdot \text{gain}_{\texttt{A}}(S) + \beta \ .$$

This also means, by extension, that we need to choose a constant β to satisfy this equation for every request sequence. First, we choose $\beta = D$. Assume D is a constant only depending on ϵ and α, then β is also a constant fulfilling the requirements. Moreover, D is the maximum value of an optimal solution if the offered items have density at most D. Thus, if we reject all items of density smaller

than D, this β makes sure that Algorithm 4 is still 1-competitive if no denser items appear.

Now let us choose a reservation factor α' such that the strict competitive ratio in 2.4.1 for Algorithm 3 is less than $2+\epsilon$ for our given ϵ. This can be done because the competitive ratio in 2.4.1 goes to 2 when α' goes to 0. We first set the competitive ratio obtained in 2.4.1 to achieve the desired ratio

$$2+\epsilon = \frac{2(2\alpha' + \sqrt{\alpha'(1+2\alpha')})(1+2\alpha'+2\sqrt{\alpha'(1+2\alpha')})}{(1-2\alpha')^2\sqrt{\alpha'(1+2\alpha')}} .$$

and obtain the appropriate value of α'

$$\alpha' = \frac{\epsilon^2 - 4\sqrt{\epsilon^3+8\epsilon^2+20\epsilon+16}+10\epsilon+16}{2(\epsilon^2+4\epsilon+4)} .$$

This reservation factor α' is a constant that only depends on ϵ. The c associated with α' is $c = \frac{2\sqrt{2\alpha'^2+\alpha'}+2\alpha'+1}{1-2\alpha'}$, which will be our chosen c, is thus, also a constant.

We choose now the threshold density $D = \alpha/\alpha'$, which also is a constant depending only on ϵ and α. Any item of size s with density $d > D$ will have a reservation cost by size of $\alpha \cdot s$, when the reservation factor is α, and a reservation cost by value $\alpha' \cdot d \cdot s > \alpha' \cdot D \cdot s$, when the reservation factor is α'. Thus, if $D = \alpha/\alpha'$, the reservation costs by value exceed the reservation costs by size. That is, $\alpha \cdot s = \alpha' \cdot (\alpha/\alpha') \cdot s = \alpha' \cdot D \cdot s < \alpha' \cdot d \cdot s$.

Assume now that all offered are of density larger than D. Thus, Algorithm 4 reserves exactly the same items as Algorithm 3 would reserve when applied to a reservation factor α'. Thus, all of the reserved items have a reservation cost by size with a reservation factor α that is smaller than the reservation cost by value with a reservation factor α'. This means that, by 2.4.1, the strict competitive ratio of such a sequence is at most $2+\epsilon$.

If Algorithm 4 is offered a request sequence with a mix of items of density larger and smaller than D, we know that the gain of Algorithm 4 has at most a factor $2+\epsilon$ over an optimal algorithm choosing only elements denser than D. If the optimal algorithm also considers less dense items, it can gain at most an extra D, which is offset by the value of β. Thus, the chosen values of c, D, and β are constants depending only on ϵ and α and, when applied as described to Algorithm 4 , achieve the desired competitive ratio. □

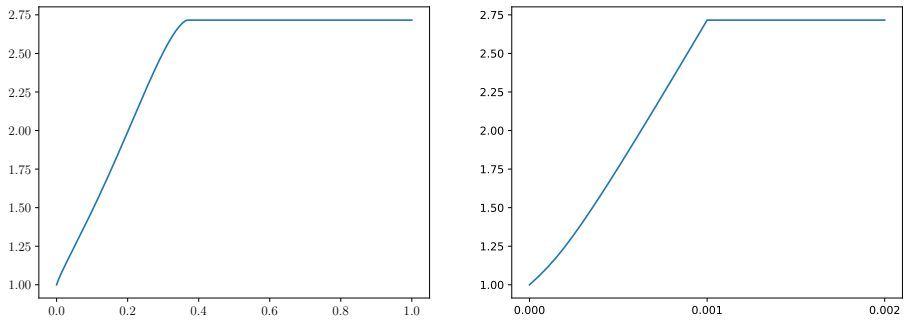


Figure 2.5: Left: Competitive ratio for $n = 1000$ in the reservation per item model. The x-axis contains the cost per reservation and the y-axis the resulting best possible competitive ratio. Right: The same for the reservation per step model.

2.5 Secretary Problem

In this section, we are applying the reservation model to the secretary problem. Instead of being forced to hire a person on the spot, we have the alternative to ask for a call-back, but we have to pay a fee for this privilege that will be deducted from the gain.

We distinguish two natural variants: In the *reservation per item*-model, we assume that a fee reserves an item forever. In the *reservation per step*-model, a fee has to be paid to reserve an item for each step in which we like to keep the reservation alive. We analyze both models precisely and prove matching upper and lower bounds for the best possible competitive ratio.

Let us say that the reservation cost is $c = \alpha/n$ per item. It turns out that in the *reservation per item*-model, the competitive ratio is basically the same as without reservation if c is asymptotically bigger than $1/e$. Otherwise, the competitive ratio beats $1/e$ and gets smaller and smaller with diminishing reservation costs. Theorems 2.5.1 and 2.5.2 contain the detailed results and Figure 2.5 shows a plot. An optimal algorithm is relatively simple and works in three phases: In the first phase, it just watches the first $\lfloor \alpha \rfloor$ items.

In the second phase, which takes a certain number of steps, it reserves every item that is the biggest seen so far. Finally, in the third phase, the algorithm looks at the remaining items. As soon as one arrives that is at least as big as the biggest seen so far, it chooses that item immediately and stops. Otherwise, at the end of the sequence, the largest reserved item is chosen.

The *reservation per step*-model is different. Here, a reservation fee has to be paid for every step in which an item is reserved. A reservation can be dropped anytime, and no costs occur from this point onward. Let us call the reservation

cost per step c. It turns out that for $c > 1/n$ again the classical algorithm without reservations is optimal. Otherwise, an optimal strategy is to ignore a certain number of items (depending on c) and then reserve all items that are the best up to this point in time until the last item arrives. Of course, only one reservation is kept at any time. The last item is chosen, if it is the best, and otherwise the reserved item, if it exists. Theorems 2.5.4 and 2.5.5 contain the details and Figure 2.5 an example.

2.5.1 Reservation per Item

The Secretary problem in the setting with reservation per items can formally be described as follows:

Definition 2.5.1 (The Secretary with Reservation per Item Problem)**.** In the Secretary with Reservation per Item problem, an algorithm receives the number of items in the instance n in the beginning, followed by n randomly ordered distinct numbers x_i. An algorithm can compare all items it has been presented so far, and has two options for each presented item:

- Take the current item, then the instance ends.
- Reserve the current item for costs of $c = \frac{\alpha}{n}$, move on to the next item (if it was not the last item)
- Reject the item, move on to the next one (if it was not the last item)

The algorithms goal is to maximize the probability to either take or reserve the best item of the instance, deducted by the expectation of paid costs for the reservation.

First, we make two key observations that are well known. They can be found, for example, in the first introduction of the marriage problem [75].

Afterwards, we will define an algorithm A and then prove that A is optimal for the reservation per item model.

Lemma 2.5.1. *Let $I = \{x_1, \ldots, x_n\}$ be an instance for the* Secretary *problem with n items ordered at random. An algorithm that accepts the first occurring locally best item after the k-th request will accept the globally best item with a probability of*

$$F(k) := \frac{k}{n} \sum_{i=k}^{n-1} \frac{1}{i} = \frac{k}{n} \ln\Big(\frac{n}{k}\Big) + O(1/n).$$

Lemma 2.5.2. *The probability $F(k) = \frac{k}{n} \sum_{i=k}^{n-1} \frac{1}{i}$ is maximal if k is the biggest integer for which $\sum_{i=k}^{n-1} \frac{1}{i} > 1$. Let m be the integer for which $F(m)$ is maximized, then $F(m) \geq \frac{m}{n}$, $F(m) = \frac{m}{n} + O(1/n)$, and $m = n/e + O(1)$.*

Now we are able to present four necessary conditions that an optimal algorithm must fulfill. Note that every item must be accepted, reserved, or neither of them. In the latter case, we will also write that an item is ignored. In any case, the algorithm remembers the value of the item.

Lemma 2.5.3 (Condition 1)**.** *An optimal algorithm must ignore an item if it is neither a locally best item nor the last item.*

Proof. Assume that an algorithm A reserves items that are not locally best items on some instance I. We can define an algorithm A' with a better expected gain as follows. A' simulates A but does not reserve or accept any item if a better item has been presented, unless the item presented is the last one. Let us look at the gain function. Because A' follows the same strategy as A, it accepts exactly the same item, or the last item if the item accepted by A was not optimal, thus if we ignore the reservation costs $r(A)$ and $r(A')$,

$$\text{gain}_A(I) + r(A) = \text{gain}_{A'}(I) + r(A') \ ,$$

but by definition of A', $r(A') < r(A)$, as it reserves fewer items. Thus, $\text{gain}_A(I) < \text{gain}_{A'}(I)$ on any instance I, and A is not optimal.

□

Lemma 2.5.4 (Condition 2)**.** *An optimal algorithm must ignore a locally best item x_k, if $k < \alpha$ and $k < m$, where m is the integer for which $F(m)$ is maximized.*

Proof. Assume that an algorithm A reserves an item x_k with $k < \alpha$ and $k < m$. The probability that k is the globally best one is k/n, which is less than the reservation cost α/n. Therefore, the expected gain of reserving this particular item is $k/n - \alpha/n < 0$.

We define an algorithm A' that just differs from A by ignoring any reservations for element x_k, when we look at the overall gain, if we take into account the linearity of expectations,

$$\text{gain}_{A'}(I) = \text{gain}_A(I) - \frac{k}{n} + \frac{\alpha}{n} > \text{gain}_A(I) \ .$$

Thus, algorithm A is not optimal.

If A takes item x_k instead, the expected gain is still k/n, which is lower than an algorithm that took the locally best item after m with expected gain $F(m) > m/n$ by Lemma 2.5.2. Thus, the optimal online algorithm for the secretary problem without reservation has a better gain, which means that A is not optimal. □

Let r denote the biggest integer such that $F(r) > c$. Note that r is only well defined if there is such an r, which is not always the case. Asymptotically, if c is smaller than n/e then r exists, and otherwise it does not.

Lemma 2.5.5 (Condition 3)**.** *An optimal algorithm must accept a locally best item x_k, if $k > r$ and $k > m$, or, if r is not well defined, if $k > m$.*

Proof. This can be proven by induction on the placement of item x_k with the second-to-last item being the base case. Assume that x_{n-1} is a locally best item, and $r < n-1$, which means that $c < F(n-1) = 1/n$. The probability that x_{n-1} is also the gbi is $n - 1/n$.

An algorithm A accepting x_{n-1} given that it is the locally best item on instance I, would have an expected gain

$$\text{gain}_A(I) = \frac{n-1}{n} - r_A(I).$$

An algorithm A' behaving identically as A but reserving x_{n-1} would have an expected gain

$$\text{gain}_{A'}(I) = 1 - r_A(I) - c.$$

Observe that $\frac{n-1}{n} > 1 - c$ when $c > \frac{1}{n}$, thus A is a better algorithm.

An algorithm A'' behaving identically as A but rejecting x_{n-1} would have an expected gain

$$\text{gain}_{A''}(I) = \frac{1}{n} - r_A(I).$$

Here it is clear that $\frac{n-1}{n} > \frac{1}{n}$, thus A is a better algorithm.

Assume that an item x_k with $k < n-1$ is a locally best item and either $k > r$ or, r is not defined but $k > m$. As an induction hypothesis, assume that an optimal algorithm must accept any locally best item $x_{k'}$ with $k' > k$.

The probability that item x_k is the globally best item, given it is a locally best item, is k/n. If an algorithm A accepts x_k, its gain is

$$\text{gain}_A(I) = \frac{k}{n} - r_A(I).$$

An algorithm A' behaving identically as A but reserving x_k would need to take the locally best item after x_k to be optimal, but the probability that the best item is the locally best item arriving after k is $F(k)$. Hence, A' would have an expected gain

$$\text{gain}_{A'}(I) = \frac{k}{n} + F(k) - r_A(I) - c.$$

Observe that $F(k) < c$ because $k < r$ and $k < m$ and thus A is a better algorithm.

An algorithm A'' behaving identically as A but rejecting x_k would still have to accept the next locally best item by induction hypothesis, thus it would have an expected gain

$$\text{gain}_{A''}(I) = F(k) - r_A(I).$$

Here it is clear that $F(k) < \frac{k}{n}$, as $k > m$, thus A is, yet again, a better algorithm.

So the best an algorithm can do is to accept the current item. □

Lemma 2.5.6 (Condition 4)**.** *If r exists, an optimal algorithm must reserve an item x_k if it is a locally best item and $k > \alpha$ or if $k > m$ and $k \leq r$.*

Proof. First, we take a look at the case that $\alpha < k < m$.

Let x_k be a locally best item with $\alpha < k < m$. Let us consider an algorithm A that ignores x_k and has a gain of $\text{gain}_A(I)$. An algorithm A' that proceeds identically but reserves x_k will have a gain of $\text{gain}_{A'}(I) = \text{gain}_A(I) + \frac{k}{n} - c$, as the probability of the globally best item being x_k—given that it is the locally best item—is k/n. Thus A is not optimal

Consider now an algorithm A that accepts x_k. Its gain is $\text{gain}_A(I) = k/n - r_A(I)$. We know that the optimal online algorithm for the secretary problem achieves an expected gain of $F(m) > m/n$ by Lemma 2.5.2. Thus, A is not optimal.

Now consider the case $m \leq k \leq r$ and that x_k is a locally best item.

An algorithm A that accepts x_k has a gain of $\text{gain}_A(I) = k/n - r_A(I)$. An algorithm A' reserving the item and accepting the next locally best item will have a gain of $\text{gain}_{A'}(I) = k/n + F(k) - c - r_A(I)$. But by construction $F(k) \geq c$, thus A' performs better than A. Lastly, an algorithm A'' that proceeds exactly as A' but ignores x_k has a gain of $\text{gain}_{A''}(I) = F(k) - c - r_A(I)$. Trivially, A' is a better algorithm. □

Theorem 2.5.1 (Optimal strategy)**.** *If $c > F(m)$, then the classic strategy without reservation is optimal.*

If $c \leq F(m)$, then the following strategy is optimal:

- *If an item x_k is either at a position $k < \alpha$ or not a locally best item, it will be ignored.*
- *If an item x_k is a locally best item and $\alpha \leq k \leq r$, it will be reserved.*
- *If an item is at a position $k > r$ and is a locally best item, it will be accepted.*

If no item was accepted, the strategy either accepts the best reserved item (if there is one) or otherwise the last item.

Proof. Observe that if $c > F(m)$, r is not defined. Thus, Conditions 1 to 3 describe exactly the classic online strategy. If $c \leq F(m)$ then r is defined, and the second algorithm follows Conditions 1 to 4. Moreover, the conditions describe strict boundaries for a strategy. If an algorithm proceeds differently it will break one of the conditions.

□

Theorem 2.5.2 (Expected gain)**.** *Given a size n and reservation costs $c = \alpha/n$, an optimal algorithm has the following properties:*

- *If $c > F(m)$, then the expected gain is $F(m)$.*
- *If $c \leq F(m)$, then the expected gain is $\frac{r - \lceil \alpha \rceil + 1}{n} - \sum_{i=\lceil \alpha \rceil}^{r} \frac{c}{i} + F(r)$.*

Proof. The expected gain for the optimal strategy is the following:

If $c > F(m)$, we are in the classical online case, the algorithm will pick the locally best item after item m, thus the expected gain is $F(m)$.

If $c \leq F(m)$, then the strategy will reserve items. The expected gain can be divided in three parts: If the globally best item is x_k then, if $k < \alpha$ the gain is 0, if $\alpha \leq k \leq r$ the gain is 1, and this happens with probability $(r - \lceil \alpha \rceil + 1)/n$ and with an expected reservation cost of $c \cdot \sum_{i=\lceil \alpha \rceil}^{r-1} \frac{1}{i}$. Finally, if $k > r$, the expected gain is $F(r)$. Thus, the total expected gain is

$$\text{gain}_A(\mathcal{I}) = 0 + \frac{r - \lceil \alpha \rceil + 1}{n} - c \cdot \sum_{i=\lceil \alpha \rceil}^{r-1} \frac{1}{i} + F(r). \tag{2.1}$$

□

Theorem 2.5.2 expresses the expected gain as an exact, but complicated and not closed formula. If the reservation costs are small enough, the following theorem provides a much nicer estimate.

Theorem 2.5.3. *The expected gain for a given $c \leq F(m)$ is*

$$e^{-c} + c \ln(c) + O(1/n) + O(c^2).$$

In particular, for $c = 1/n$ the gain is $1 - \ln(n)/n + O(1/n)$.

Proof. The only complicated term in the sum presented in Theorem 2.5.2 is r. It is defined via

$$c = \frac{r}{n}(H_n - H_r) = \frac{r}{n}(\ln(n) - \ln(r) + O(1/r)) = \frac{r}{n} \ln\left(\frac{n}{r}\right) + O(1/n).$$

With the Lambert W function we can solve this equation for r/n:

$$\frac{r}{n} = e^{W(-c+O(1/n))} = e^{-c+O(c^2)+O(1/n)} = e^{-c}(1 + O(c^2) + O(1/n))$$

We get rid of the W-function by a Taylor approximation with $W'(0) = 1$. Our goal is to approximate the gain found in (2.1) with an additive error term of $O(1/n)$. In addition to that, we have $\lceil \alpha \rceil / n = c + O(1/n)$, the sum is $\ln(r) - \ln(a)$ and $\ln(r) = -c + \ln(n) + O(1/n)$, and finally $F(r) = c + O(1/n)$. Note that $\ln(n) - \ln(\alpha) = \ln(c)$. Altogether, this yields the bound stated in the theorem. □

2.5.2 Reservation per Step

The Secretary problem in the setting with reservation per step can be defined in the following way:

Definition 2.5.2 (The Secretary with Reservation per Step Problem)**.** In the Secretary with Reservation per Step problem, an algorithm receives the number of items in the instance n in the beginning, followed by n randomly ordered distinct numbers x_i. An algorithm can compare all items it has been presented so far, and has two options for each presented item:

- Take the current item, then the instance ends.
- Reserve the current item for costs of $c = \frac{\alpha}{n}$, move on to the next item (if it was not the last item)
- Reject the item, move on to the next one (if it was not the last item)

The algorithms goal is to maximize the probability to either take or reserve the best item of the instance, deducted by the expectation of paid costs for the reservation.

Here we have the possibility to reserve an item for one step from the position k to $k+1$. In every step, we can decide if we want to reserve the item for one more step. The reservation costs are c for every reservation.

Again, the optimal strategy can be given by showing that every deviation cannot be optimal:

Lemma 2.5.7 (Condition 1)**.** *An optimal algorithm must ignore an item if it is neither a locally best item nor the last item.*

Proof. Analogous to Lemma 2.5.3. □

Lemma 2.5.8 (Condition 2)**.** *If $c > 1/n$, then no optimal algorithm reserves any item. Therefore, the optimal strategy is the same as in the basic secretary problem without reservation costs.*

Proof. We prove this by induction over the items in descending order as in Lemma 2.5.5.

If x_n is the globally best item, it is clear that only accepting the current or reserved item is optimal. For the induction step, consider a locally best item x_k. An algorithm A accepting x_k gets the globally best item with probability $\frac{k}{n}$, thus it has a gain of $\text{gain}_A(I) = \frac{k}{n}$. An alternative algorithm A' reserving it would cause additional costs of c. From the induction hypothesis, we know that in the next step, the item will be accepted. Hence, $\text{gain}_{A'}(I) = \frac{k+1}{n} - c$ and A' is not optimal. □

Lemma 2.5.9 (Condition 3)**.** *If $c < 1/n$, an optimal algorithm does not accept an item before the last one.*

Proof. An algorithm A accepting a locally best item at position k has $\text{gain}_A(I) = \frac{k}{n} - r_A(I)$. An algorithm A' reserving this item and accepting in the next step has $\text{gain}_{A'}(I) = \frac{k+1}{n} - c - r_A(I) > \frac{k}{n} - r_A(I)$. □

Lemma 2.5.10 (Condition 4)**.** *An optimal algorithm with $c \leq 1/n$ reserves the current locally best item from a point $\hat{k}(c,n)$ until the last step.*

Proof. There are only three strategies that optimal algorithms can follow at this point. An algorithm A can reserve the current locally best item from a specific point k until the end. Alternatively, an algorithm A' will reserve from point k_1 until point k_2, then reject the reserved item, and then perhaps at a future point k_3 reserve the next locally best item until the end. Finally, an algorithm A'' may refuse to reserve items.

Algorithm A'' cannot be optimal because, by Lemma 2.5.9, A'' cannot accept any item before the last step. Thus, $\text{gain}_{A''}(I) = 1/n$, which is not optimal.

Algorithm A', on the other hand, cannot be optimal because an equivalent algorithm that only reserves items from k_3 will have the exact same success chance and lower reservation costs.

Thus, we are left to analyze the gain of algorithm A. The probability that the globally best item occurs after step k is $(n-k)/n$ and the expected cost of reserving the next locally best item until the end is

$$c \cdot \sum_{i=k+1}^{n} \Pr(i \text{ is first lbi after } k) \cdot (n-i) = c \cdot \sum_{i=k+1}^{n} \frac{n-i}{i} \frac{k}{i-1}$$
$$= c\left(n - k - k\ln\left(\frac{n}{k}\right) + O(1/k)\right). \quad (2.2)$$

This means that A would have an expected gain of

$$\text{gain}_A(\mathcal{I}) = \frac{n-k}{n} - c \cdot \sum_{i=k+1}^{n} \frac{n-i}{i} \frac{k}{i-1}. \quad (2.3)$$

This expression is maximized for one value k (with a given n and c), since $\frac{n-k}{n}$ decreases linearly with growing k, while the sum increases with growing k.

We can define $\hat{k}(n,c)$ to be the integer k that maximizes $\text{gain}_A(\mathcal{I})$. Because all other strategies are worse, this algorithm must be the optimal one.

□

From these four conditions, we get an optimal online algorithm for the secretary problem in the reservation per step model.

Theorem 2.5.4 (Optimal strategy for reservation per step)**.** *If $c > 1/n$, then the strategy for the basic secretary problem without reservation is optimal. If $c \leq 1/n$, there exists a $\hat{k}(c,n)$ such that the optimal strategy is to keep the locally best item after $x_{\hat{k}(c,n)}$ reserved and accept only after the last step.*

Proof. In case of $c > 1/n$, an optimal strategy cannot reserve items (Condition 2). Therefore, the best strategy without a reservation is optimal here.

If $c \leq 1/n$, an optimal strategy will start to reserve items at some point and will not accept an item until the end (Conditions 3 and 4). The optimal point for reserving items is $\hat{k}$ (Condition 4). Every item that is not a locally optimal item will be ignored (Condition 1). □

Theorem 2.5.5 (Expected gain)**.** *If $c > 1/n$, the expected gain is $F(m) = 1/e + O(1/n)$. If $c \leq 1/n$, the expected gain is*

$$\frac{n-\hat{k}}{n} - c \cdot \sum_{i=\hat{k}+1}^{n} \frac{n-i}{i} \frac{\hat{k}}{i-1}$$

where $\hat{k}$ is chosen such that the expression gets maximal.

Proof. If $c > 1/n$, the optimal algorithm does not reserve any item. So the strategy and the expected gain is the same as in the case in Section 2.5.1 where reserving items was not optimal.

If $c \leq 1/n$, then the expected gain is calculated as in Condition 4. □

Theorem 2.5.6. *The expected gain for a given $c \leq \frac{1}{n}$ is*

$$1 - cn(1 - e^{-1/cn}) + O(1/n).$$

In particular, for $c = 1/n$ the gain becomes $1/e + O(1/n)$.

Proof. Using methods from calculus we find that the optimal $\hat{k}$ is $\hat{k} = ne^{-1/cn}(1 + O(1/\hat{k}))$. As $e^{-1/cn}$ is between $1/e$ and 1, we know that $\hat{k} = \Theta(n)$ and therefore $\hat{k} = ne^{-1/cn}(1 + O(1/n))$. Inserting this $\hat{k}$ into the estimate for the gain using a combination of (2.2) and (2.3) yields the result. □

2.6 Graph Problems

In this section, we study graph problems with the option to reserve vertices. Here we consider the node deletion problems in its *Delayed Decision* variant (see 1.2.5 and 1.2.6), where decisions can be delayed even further by reserving items (for some costs).

This reservation is not free: When computing the final competitive ratio, the algorithm has to pay a constant $\alpha \in \mathbf{R}_{\geq 0}$ for each reserved vertex; these costs are then added to the size of the chosen solution set.

We show that given bounds on the competitive ratio of a problem with delayed decisions (without reservations) implies lower and upper bounds for the same problem when adding the option of reservations. This observation allows us to give a lower bound of $\min\{1+3\alpha, 4\}$ and an upper bound of $\min\{1+5\alpha, 5\}$ for the Online Feedback Vertex Set with Reservations Problem, where $\alpha \in \mathbf{R}_{\geq 0}$ is the reservation cost per reserved vertex.

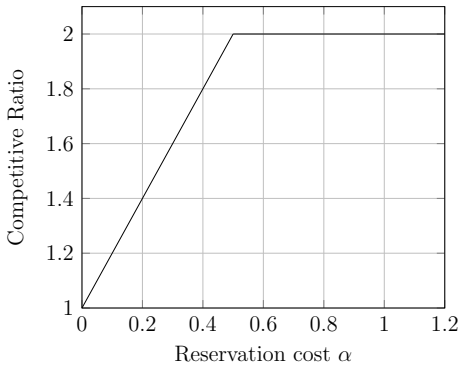
Finally, we show that the online Vertex Cover problem, when both delayed decisions and reservations are allowed, is $\min\{1+2\alpha, 2\}$-competitive.

Definition 2.6.1 (The Online Vertex Cover with Reservations Problem)**.** Let $\alpha \in \mathbf{R}_{\geq 0}$ be a constant and G an online graph induced by its vertices $V(G) = v_1, \ldots, v_n$, ordered by their occurrence in an online instance. The Online Vertex Cover with Reservations problem (VCR) is to select, for every i, vertex subsets $S_i, R_i \subseteq \{v_1, \ldots, v_i\}$ with $S_1 \subseteq \ldots \subseteq S_n$ and $R_1 \subseteq \ldots \subseteq R_n$ such that $G[\{v_1, \ldots, v_i\} \setminus (S_i \cup R_i)]$ contains no edge. The goal is to minimize the sum $|S_n|+|T|+\alpha|R_n|$, where $T \subseteq V(G)$ is a minimal vertex subset such that $G-(S_n \cup T)$ contains no edge.

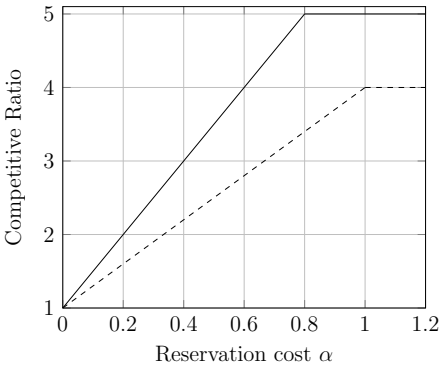
Again, the definition for the Online Feedback Vertex Set with Reservations problem (FVSR) is identical, except for replacing "contains no edge" with "is cycle-free."

For reservation costs of $\alpha = 0$, the problem becomes equivalent to the offline version, whereas for $\alpha \geq 1$, taking an element directly into the solution set becomes strictly better than reserving it, rendering this reservation option useless. The results for Online Vertex Cover with Reservations problem and Online Feedback Vertex Set with Reservations Problem are depicted in Figure 2.6.

We now extend the previous results for the delayed-decision model without reservations by presenting two general theorems that translate both upper and lower bounds to the model with reservations.



Online VC with Reservations



Online FVS with Reservations

Figure 2.6: Upper and lower bounds on the competitive ratios of VCR (left) and FVSR (right)

2.6.1 General Bounds for Reservation in Graph Problems

The delayed-decision model allows us to delay decisions free of cost as long as a valid solution is maintained. Combining delayed decisions with reservations, we have to distinguish minimization and maximization: For a minimization problem, the default is that the union of selected vertices and reserved ones constitute a valid solution at any point. For a maximization problem, in contrast, it is that the selected vertices without the reserved ones always constitute a valid solution.

Whether the goal is minimizing or maximizing, any algorithm for a problem without reservations is also an algorithm for the problem with reservations, it just never uses this third option. We present a slightly smarter approach for any *minimization* problem such as the Feedback Vertex Set with Reservation problem.

Theorem 2.6.1. *In the delayed-decision model, a c-competitive algorithm for a minimization problem without reservation yields a $\min\{c, 1 + c\alpha\}$-competitive algorithm for the variant with reservation.*

Proof. Modify the c-competitive algorithm such that it reserves whatever piece of the input, it would usually have been immediately selected until the instance ends – incurring an additional cost $c \cdot \alpha \cdot Opt$ – and then pick an optimal solution for a cost of Opt. This already provides an upper bound of $1 + c\alpha$ on the competitive ratio. But running the algorithm without modification still yields an upper bound of c of course. The algorithm can now choose the better of these two options based on the given α, yielding an algorithm that is $\max\{c, 1 + c\alpha\}$-competitive. □

Corollary 2.6.1. *There is a $\min\{5, 1+5\alpha\}$-competitive algorithm for the* FVSR.

Theorem 2.6.2. *In the delayed-decision model, a lower bound of c on the competitive ratio for a minimization problem without reservations yields a lower bound of* $\min\{1+(c-1)\alpha, c\}$ *on the competitive ratio for the problem with reservation.*

Proof. The statement is trivial for $\alpha \geq 1$; we thus consider now the case of $\alpha < 1$. Assume that we have an algorithm with reservations with a competitive ratio better than $1+(c-1)\alpha$. Even though this algorithm has the option of reserving, it must select a definitive solution, at the latest, when the instance ends. This definitive solution is, of course, at least as expensive as the optimal solution. Achieving a competitive ratio better than $1+(c-1)\alpha$ is thus possible only if the algorithm is guaranteed to reserve fewer than $c-1$ input pieces in total. But in this case, we can modify the algorithm with reservation such that it immediately accepts whatever it would have only reserved otherwise. This increases the incurred costs by $1-\alpha$ for each formerly reserved input piece, yielding an algorithm without reservations that achieves a competitive ratio better than $1+(c-1)\alpha+(c-1)(1-\alpha) = c$. □

Corollary 2.6.2. *There is no algorithm solving the* FVSR *that can achieve a lower bound better than* $\min\{1+3\alpha, 4\}$.

2.6.2 Vertex Cover

As already mentioned, VCR yields a competitive ratio of 2 without reservation. We now present tight bounds for all reservation-cost values α, beginning with the upper bound.

Corollary 2.6.3. *There is an algorithm for the* VCR *that achieves a competitive ratio of* $\min\{1+2\alpha, 2\}$ *for any reservation value.*

This immediately follows from Theorem 2.6.1.

These upper bounds, depicted in Figure 2.6, have matching lower bounds. We start by giving a lower bound for $\alpha \leq \frac{1}{2}$.

Theorem 2.6.3. *Given an $\varepsilon > 0$, there is no algorithm for the* Online Vertex Cover with Reservations Problem *achieving a competitive ratio of $1+2\alpha-\varepsilon$ for any $\alpha \leq \frac{1}{2}$.*

Proof. We present the following exhaustive set of adversarial instances as depicted in Figure 2.7. First an adversary presents two vertices u_1 and v_1 connected to each other. Any algorithm for the Online Vertex Cover with Reservations Problem is forced to cover this edge either by irrevocably choosing one vertex for the cover or by placing one of the vertices in the temporary cover (i.e., reserving it). In the first case, assume w.l.o.g. that v_1 is the chosen vertex for the cover. The adversary then presents a vertex v_2 connected only to u_1 and ends the instance.

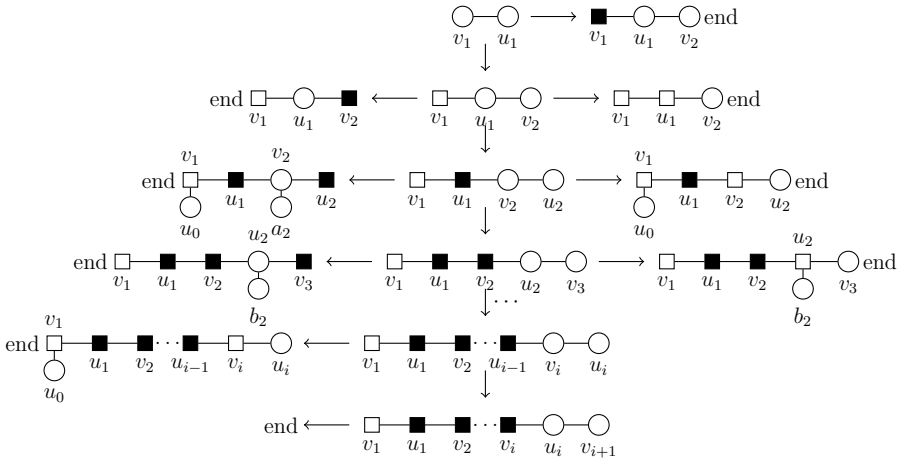


Figure 2.7: Illustration for the proof of Theorem 2.6.3. Adversarial strategy for $\alpha \leq \frac{1}{2}$. Black squares are vertices irrevocably included in the vertex cover, white square vertices are reserved to be temporarily in the vertex cover, and round vertices are not yet chosen.

Such an algorithm would have a competitive ratio of 2 as it must then cover the edge (u_1, v_2) by placing one of its endpoints in the cover. Choosing vertex u_1 alone would have been optimal, however. This is the same lower bound as given for the model without reservations.

If, again w.l.o.g., the vertex v_1 is temporarily covered instead, the adversary still presents a vertex v_2 connected to u_1. Now an algorithm has four options to cover the edge (u_1, v_2): Each of the two vertex u_1 or v_2 can be either irrevocably chosen or temporarily reserved. If u_1 or v_2 is temporarily covered, the instance will end here, and the reservation costs of the algorithm will be 2α. Both the algorithm and the optimal solution will end up choosing only vertex u_1, which implies a final competitive ratio of $1 + 2\alpha$ in this case. If vertex v_2 is chosen, the instance will also end, yielding a competitive ratio worse than 2.

Thus, the only option remaining is to irrevocably cover the vertex u_1. In this case, the adversary presents a vertex u_2 connected to v_2. An algorithm can then irrevocably or temporarily take v_2 or u_2, respectively. If an algorithm temporarily takes v_2 or u_2, the adversary will present one more vertex u_0 connected to v_1 and end the instance. This results in a graph that can be minimally covered by the vertices v_1 and v_2. The algorithm, however, will have 3 vertices in the cover and additional reservation costs of 2α for the temporarily chosen vertices. Thus, it will have a competitive ratio of $\frac{3}{2} + \alpha$, which is larger than $1 + 2\alpha$ for the considered values of α. If an algorithm irrevocably takes u_2, the same vertex u_0 will be presented, and then the instance will end with another auxiliary vertex a_2 connected to v_2. An optimal vertex cover would take vertices v_1 and v_2. Any algorithm that has already irrevocably chosen u_1 and u_2, however, will have to

choose two more vertices in order to cover the edges $\{v_1, u_0\}$ and $\{v_2, a_2\}$; thus, its competitive ratio will be worse than 2.

Again, the only remaining option is to irrevocably choose the vertex v_2, after which an adversary presents a vertex v_3 connected to u_2. An algorithm may choose to irrevocably or temporarily take the vertex u_2 or v_3. If an algorithm decides to temporarily take any vertex or irrevocably choose v_3, then the adversary presents an auxiliary vertex b_2 connected to u_2 and ends the request sequence. An optimal vertex cover in this case has size two, containing only the vertices u_1 and u_2. In the best case, however, such an algorithm has a vertex cover of size 3 and two temporary covers, thus its competitive ratio will be at best $\frac{3}{2} + \alpha$, which again is worse than $1 + 2\alpha$, as already observed.

In general, after irrevocably choosing $u_1, \ldots, u_{k-1}$ and $v_2, \ldots, v_{k-1}$, and temporarily choosing v_1, the adversary presents the vertex u_k connected to v_k. If an algorithm chooses to reserve any of the endpoints or irrevocably selects u_i, then the adversary presents the vertex u_0 and ends the request sequence. In this case, an optimal vertex cover only contains the vertices v_i for every $i = 1, \ldots k$, thus it has size k. The algorithm, however, will have to take v_1 and v_k in addition to the previously irrevocably taken vertices, thus obtaining a vertex cover of size $2k - 1$ at best, together with two temporarily taken vertices. Thus, the competitive ratio is $\frac{2k-1+2\alpha}{k} = 2 - \frac{1-2\alpha}{k} \geq 1 + 2\alpha$,

In the other case, after irrevocably choosing vertices $u_1, \ldots, u_{k-1}$ and $v_2, \ldots, v_k$, the adversary presents the vertex v_{k+1} connected to u_k. If an algorithm chooses to reserve one of the two endpoints or irrevocably choose v_{k+1}, then the adversary stops the request sequence. An optimal vertex cover of such a graph consists of the vertices u_i for every $i = 1, \ldots, k$ and it has size k. The algorithm will have to choose u_i in order to obtain a vertex cover at all, obtaining a vertex cover of size $2k - 1$ at best together with at least one reservation, thus achieving a competitive ratio of $\frac{2k-1+\alpha}{k} \geq 1 + 2\alpha - \varepsilon$ for any $k \geq \frac{1}{\varepsilon}$. □

For larger values of α, the same adversarial strategy holds, but it gives us the following lower bound.

Theorem 2.6.4. *For $\alpha > 1/2$, no algorithm for the* Online Vertex Cover with Reservations Problem *is better than 2-competitive.*

Proof. The lower bound of Theorem 2.6.3 for $\alpha = 1/2$ is $2 - \varepsilon$. For larger values of α the same adversarial strategy will give us a lower bound of 2. This is because, at all points during the analysis, either the value of the competitive ratio for each strategy was at least 2, or it had a positive correlation with the value of α, meaning that for larger values of α any algorithm following that strategy obtains strictly worse competitive ratios. □

2.7 Comments and Open Problems

In the reservation setting, decisions can be delayed by paying reservation costs. Those costs are then deducted from the value of the algorithm's solution. One can argue that this setting is most natural when the quality of the algorithm is also valued by some price itself (like for a Knapsack problem, the value of the items).

Many situations for various online problems are arguably interesting to consider in this setting.

Take, for example, the Online Bin Packing problem, with all its natural variants (see 1.2.2): In many practical situations, it is realistic to assume that items appear online. However, sometimes it is not necessary to immediately pack them into a bin. Instead, one might be able to place them at the side for a moment and wait for further items to finally decide. This variant is known as online bin packing with buffer (see [94]). However, in this variant there is no motivation not to use the buffer when there is empty space. In other situations, one may possibly have the option to put items to the side as well, but this does not necessarily come for free. This might be the case for several reasons: You have to rent additional space to temporarily store items. Or you realize that putting items to the side, and later placing them into a bin requires additional effort, which in the end might cost some money.

Additionally also various variants of the previously studied problems might be interesting:

For example, in the Online Knapsack with Removability and Reservation problem, we assume that an algorithm can only discard already packed items. A natural variant would be to again allow removing items from the knapsack, but not only to discard but also to reserve them.

Chapter 3

Algorithms with Estimates

3.1 Introduction

Even as the *advice* and *prediction* variant of online computation already model various settings in which additional information can be used to enhance the performance of online algorithms (please find an overview about some of the works in this line of research dealing with the problems covered in this thesis in the introduction), many situations are thinkable in which this is not the case.

In this section, we will describe three different motivations that might lead to the model of *estimates* as we will study within this part of the thesis. One should note that, in particular, the first two concerns are addressed for some problems, but also remain unanswered in various settings when focusing on the theoretical aspects. For example, one might consider an online coloring problem in which one predicts the colors for each vertex [6, 51].

3.1.1 Are predictions available in practice?

First, remember the section 1.3, where we mentioned the models of online algorithms with *advice* or *prediction*. Both settings have in common that they relax the assumption of blindness which a classical online algorithm has to face.

The *Advice*-setting allows an algorithm to ask a limited number of arbitrary questions to an oracle, which then answers truthfully. It is not surprising that one of the main criticisms of this (albeit theoretical) model is the question of whether the existence of such an oracle is realistic in practice.

Thus, one motivation for the *Prediction*-setting is to relax the assumption that an oracle is able to give all answers truthfully. It is not surprising that this variant of the advice setting gained a lot of traction in recent years: one can claim that machine learning methods can take previous data to learn, and then compute some kind of (possibly incorrect) predictions. The assumption that

those predictions can deviate from reality is indeed typical for results that rely on stochastic processes (which machine learning methods essentially are). Another question however, remains: Is it actually realistic that interesting questions can be answered by experience or machine learning?

This strongly depends on the kind of prediction that is given: This is not surprising, as the prediction setting, as generally referred to, does not restrict the kind of information an algorithm receives.

Given that the predictor (or oracle) in the prediction setting is usually assumed to have no restrictions with respect to computational power, everything he can rely on for his prediction is essentially the input data of prior instances. As many problems that are of interest are online versions of NP-hard problems, receiving good information about the optimal solution of such a problem might be unrealistic: Minor changes in the input can usually cause major differences in the optimal solution. Thus, one can argue that receiving those predictions and hoping for a decent quality of those might be unrealistic.

On the other hand, as the predictor likely has previous instances to base its prediction on, the best predictions we can realistically hope for are likely closely related to the input of the instance.

Thus, when trying to enhance an algorithm with some kind of predictions, we should consider a prediction on the instance instead of other properties or the solution.

3.1.2 Isn't information lost when computing a prediction?

Apart from the question of whether it is realistic to assume that good predictions exist that encode a solution due to e.g. stability issues, it might not be sensible to rely on them even if they might be available:

Given that the predictor has some data to rely its prediction on, in the process of generating a prediction some information might get lost. Usually, even when the input is predicted, the prediction will be the value that are most likely for the predictor: What is lost here however is, how "sure" the predictor is for each part of the prediction is: maybe there are some parts an algorithm can take almost for granted, and some where the available data only allows an educated guess. Knowing this, however, might be beneficial for an algorithm as it then can balance the belief in the prediction for different parts of the instance.

Thus, when trying to enhance an algorithm with some kind of prediction, we should consider a prediction that also includes some accuracy data for each part of the input.

3.1.3 Rounding Issues

Sometimes, predictions in any way is not even necessary, as the input on which an algorithm has to act was already measured. Here, however, one can almost take it for granted that the measurements do not exactly model the reality:

- Assume one receives a list of data as input in the beginning: This might be e.g., a list of items that need to be packed, a list of jobs that need to be scheduled, a map of a street network with some distances, or just anything that involves numerical data. As basically every thinkable way to measure numerical data is prone to measurement errors, an algorithm must assume that the values received are not accurate. If, for example, items of size 0.35 and one of 0.65 are measured, an algorithm cannot know whether those fit together in a bin of size 1 or if they barely will not fit together. Those issues are arguably unavoidable when dealing with any real-life data.
- Even if one assumes that the data actually can be measured in an exact way, at some point, they are handed to an electric device or a human to perform the algorithm. Latest here, we must face the fact that neither a computer nor a human will not be able to save the input with arbitrary accuracy. Thus, we need to face rounding issues, which again lead to the same problems: for example, that one can not tell whether items fit into a bin or barely not.

Both reasons have in common that an algorithm can assume that the values given as an input are indeed correct and that the actual values are within a known range of distortion (as in classical computation problems). For example, when an algorithm has to deal with numerical values, it knows that the device that measured them is prone to a five percent distortion, or that the data is cut off after four digits behind the point.

Thus, when modeling a real-life problem, we should use a model that receives data and be aware of the fact that the actual values are slightly off and are within a range given by some accuracy parameter δ.

3.1.4 Online Algorithms with Estimates

In this section, we will model various problems in the setting of online algorithms with *Estimates*. The previous motivation will finally lead to the following

Definition 3.1.1 (Online Algorithms with *Estimates*)**.** Given an online Problem Π, the online problem Π *with estimates* is defined as Π where the algorithm gets also presented an estimate $I' = (x'_1, \ldots, x'_n)$ of the instance $I = (x_1, \ldots, x_n)$ as well as an estimate accuracy parameter δ in the beginning. Each item of the instance x_i can deviate by at most δ from the estimate x'_i.

What exactly *deviate by at most* δ means is up to the respective problem: As an example, for packing problems where item sizes are announced, it is sensible to define the accuracy in a way that the actual size can differ by at most an additive constant or multiplicative factor δ from the estimate.

It is worth noting that sometimes various definitions are sensible for the distance of estimates: Take for example the Online Simple Knapsack Problem: If estimates are added with an multiplicative factor, the best possible algorithm will have unavoidable jumps in its competitive ratio as the distortion factor rises (see chapter 3.2, [55]). If the accuracy defines an absolute error, those jumps do not appear any longer (see chapter 3.3 or [13]). For seemingly related problems as the Online Simple Knapsack with Removability, this is no longer the case; here an comparably easy algorithm is optimal for both definitions (see [13] or 3.4). Note that one can study both one- and two-sided deviations of the estimates; however, both settings can easily be translated into each other.

In this section, we will study various problems in a setting with estimates, with some problems studied in an additive, some in a multiplicative accuracy variant, and some in both.

3.2 Contributions and Related Work

The first notable works related to online problems with *Estimates* study variants of the Online Scheduling Problem: On the one hand, scheduling problems in the offline variant assume that the whole input sequence (e.g. which jobs will be released when, how long do they take to complete) is known in advance, allowing a scheduler to design some optimal schedule. In classical online scheduling problems however, an algorithm only gets aware about the jobs as soon as they get released by an adversary. This is a crucial issue, as one usually cannot hope for optimal schedules then with some notable exceptions (e.g. variants which allow preemption, where an *Shortest Remaining Processing Time (SRPT)* strategy can minimize the total flow time). However, for scheduling problems it is realistic to assume that information about the jobs only gets available during the running time of an algorithm.

As for other problems, it is not surprising that many real life applications lie somewhere in between those two situations:

If, for example, a PhD-supervisor is in a program committee of some conference, their students can expect to be asked to review a paper for that conference. And while neither the actual release date (the invitation to do the review), the deadline (as they are set by the PC-member), nor the time needed for writing a good review is known prior to the invitation, realistic bounds are known months in advance: The invitation will likely come a few days after the submission deadline (but not prior to that), the review deadline will be set a few days prior to the the auther notification or rebuttal phase (but not later), and a rough guess about the workload might also be possible by experience.

The setting that is called online algorithms with *Estimates* within this thesis is able to capture those settings:

First in 2021, a scheduling variant is studied in which the *total (weighted) flow time* on a single machine must be minimized. Here the jobs appear over time in an online manner, and for each job an estimate for the processing time of the jobs was given with its release. Jobs are allowed to be preempted, and can be continued at a later point. The actual processing time is at most δ times the estimate, with δ known in advance [7].

Here two algorithms for this setting are presented: One is $O(\delta^3 \log(\delta P))$- and $O(\delta^3 \log(\delta D))$-, and one is $O(\delta^2 \log W)$-competitive. Here P denotes the ratio between the maximum to minimum processing time, D the ratio between the maximum to minimum density, and W the same ratio with respect to the weight between each item.

For the unweighted flow time scheduling problem a competitive ratio of $2\lceil\delta^2\rceil$ is proven, as well as a lower bound of 2 for an arbitrary small accuracy factor δ.

In a follow-up work it is assumed that the accuracy factor is not known in

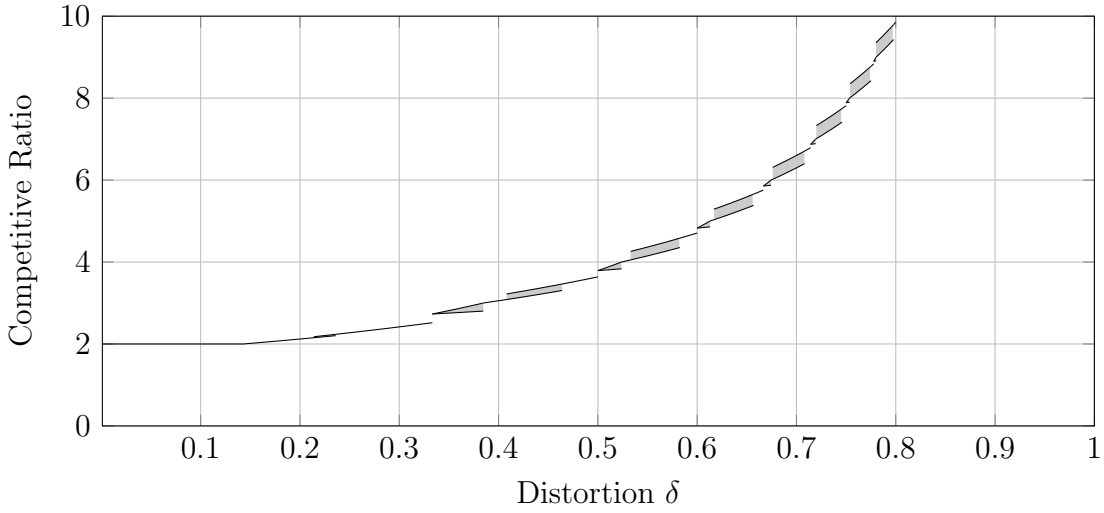


Figure 3.1: Competitive ratio of the Online Simple Knapsack with Estimates problem with multiplicate distortion δ. The gray areas signify a gap between the best known lower and upper bounds.

advance for the unweighted flow time scheduling problem on a single machine. Here an algorithm called *ZigZag* is presented, that achieves a competitive ratio of $O(\delta \log \delta)$. This of course also improves the competitive ratio of $2\lceil\delta^2\rceil$ from the previous article in case of a known accuracy factor [8].

In the same year, the results were also extended to multiple identical machines. If jobs are not allowed to be migrated between different machines but instead must be fully completed on one machine, then a $O(\delta \log P)$ ratio is achievable; also an algorithm with a competitive ratio of $O(\min(\delta \log P, \delta \log \delta + \delta \log \frac{n}{m}))$ is presented in case migration is allowed (with n denoting the number of jobs, and m the number of machines). Additionally, a variant of flow time called *total strech* in which the flow of each item is normalized by its size is also found out to be $O(\delta^2)$ competitive [9].

Note that in the mentioned scheduling papers, the parameter that is called accuracy factor δ in this thesis is instead called distortion factor μ. Apart of scheduling problems, also first results are known for the Online Simple Knapsack with Estimates Problem, where an estimate of each item is given in the beginning, and the actual value gets revealed afterwards. As found out for the scheduling problems, also in this setting no algorithm can achieve a competitive ratio of better than 2 even for arbitrary accurate estimates $\delta > 0$ (even though this construction is closer aligned to the fact that it is also not possible to beat a competitive ratio of 2 for various knapsack settings like in the setting of reservations or with sublogarithmic advice). Afterwards, the achievable competitive ratio

degrades gracefully the imprecise the estimates get [**?**].

In this thesis, we first consider a variant of the Online Simple Knapsack with Item Size Estimates problem, which differs from the model in [**?**] by the way the distortions are defined: While previously, a variant in which the actual item size can differ by some multiplicative factor was studied, in this thesis we will focus on the variant with additive distortion. It turns out, that the results differ rather significantly, which is mostly caused by the additional problems that can arise with announcements of small items: In the additive variant, an item which is announced as a small one, might be presented with size 0. This effectively gives an adversary the chance to let items disappear completely. However, this is not possible in the multiplicative variant: Here, the actual size is only off by some percentage of the announcement. In particular, this implies that the item cannot be presented with size 0.

Afterwards, we will study the Online Simple Knapsack with Removability and Item Size Estimates problem. Here we present matching results for both, the additive and multiplicative variant. Interestingly, in this variant the multiplicative and additive model are structurally the same, as the small items do not have a significant impact in this knapsack variant.

Another packing problem, for which estimates are applicable, is the Online Bin Packing Problem. Here, again, an algorithm is given a list of item sizes together with the distortion factor in the beginning. Afterwards, the items are presented iteratively like in an Online Knapsack problem. For Online Bin Packing with Item Size Estimates, we will show that accurate prediction will help to significantly undercut the competitive ratio of classical Online Bin Packing by providing a 1.5-competitive algorithm. Additionally, we show that even for tiny inaccuracy no algorithm can be better than $\frac{4}{3}$-competitive. Lastly, we also show that this ratio is achievable in a constraint variant with at most two items per bin.

Finally, we study the estimate model within the setting of Graph Exploration. Here, an algorithm receives the structure of the graph in advance. The actual edge weights then get only revealed in an online manner, whenever the agent is incident to an edge. Interestingly, for rather good accuracy, an algorithm does not benefit from the revelation of the actual values in general.

3.3 Additive Knapsack

As in the previous chapter, we start with a formal definition of the problems we investigate. In this section, notation will be slightly abused by using x_i for both the label of an item and the size of the same item, as its meaning is also clear in the given context.

Definition 3.3.1 (The Online Simple Knapsack with Item Size Estimates Problem)**.** Given an estimate accuracy $\delta \geq 0$, an instance of the Online Simple Knapsack with Item Size Estimate consists of a series of *items* $I = (x_1, \ldots, x_n)$, where each *item* is a real number in $[0, 1]$. At the beginning, the accuracy δ is announced as well as the *item size estimates* $P = (x'_1, \ldots, x'_n)$, such that for each $i \in \{1, \ldots, n\}$, $x'_i - \delta \leq x_i \leq x'_i + \delta$. At each step $i \in \{1, \ldots, n\}$, the actual item size x_i of the request sequence gets revealed. An algorithm then has the option to either pack the item in an initially empty knapsack K, if it fits, or to reject the item:

- **Pack** If $\sum_{x_k \in K} x_k + x_i \leq 1$, set $K := K \cup x_i$.
- **Reject** Do nothing.

The Online Simple Knapsack with Item Size Estimates problem (OSKE) is for an algorithm `ALG` to minimize the competitive ratio between the size of its packing and the optimal solution in the same instance.

In this section, we first show that the competitive ratio of the online simple knapsack problem with item size estimates is not less than 2 for any $\delta > 0$ and monotonously rises until it gets unbounded for any algorithm for $\delta \geq 0.5$.

Afterwards, we provide an algorithm that matches the competitive ratio of the lower bound.

We will see that the competitive ratio for all $0 < \delta < \frac{1}{2}$ is given by $\frac{1}{\min(p,q)}$, where

$$p = -\frac{0.5}{\lfloor k \rfloor} + \sqrt{\frac{1}{4\lfloor k \rfloor^2} + \frac{1-2\delta}{\lfloor k \rfloor}k} \text{ and } q = 1 - 2\delta - \frac{1}{\lceil k \rceil} \text{ , where } k = \frac{2}{1-2\delta}. \quad (3.1)$$

This ratio is visualized in Figure 3.2.

Note that p is the positive solution of the following equation.

$$\left\lfloor \frac{2}{1-2\delta} \right\rfloor \cdot \frac{p}{1-p-2\delta} = \frac{1}{p} \quad (3.2)$$

The following inequalities, which hold for $0 \leq \delta \leq 0.5$, will turn out to be useful:

$$1 - p - 2\delta \leq p \text{ and } 1 - q - 2\delta = \left\lceil \frac{2}{1-2\delta} \right\rceil^{-1} \leq q \quad (3.3)$$

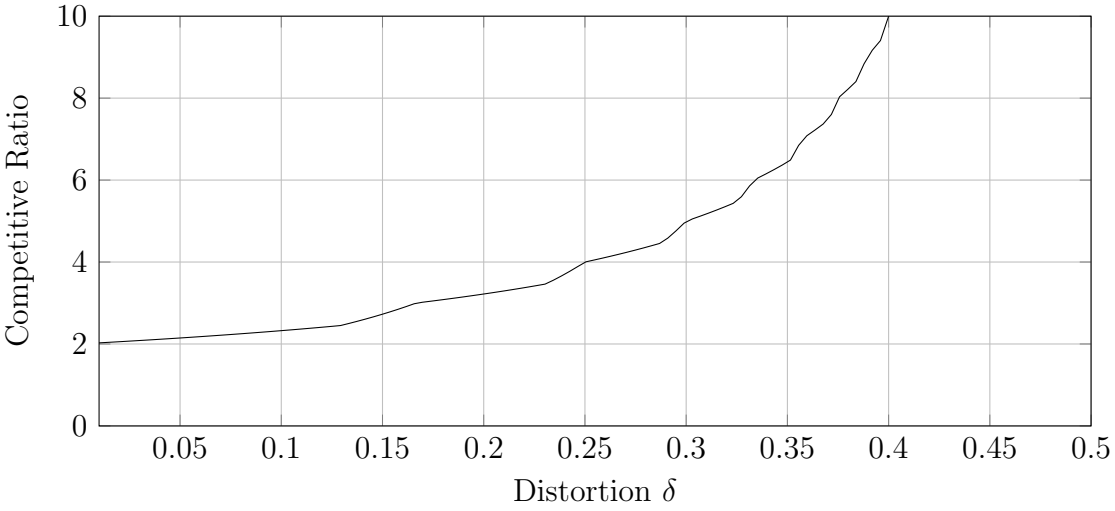


Figure 3.2: Competitive ratio in the *absolute error* model, depending on δ

3.3.1 Lower Bounds

The following two lower bounds combined are best possible for all values of δ up to $\frac{1}{2}$. Both functions have jumps induced by rounding effects and dominate one another periodically. See Equation (3.1) for the definition of p and q. In the end, we note that for $\delta \geq \frac{1}{2}$, no constant competitive online algorithm can exist.

While the construction for the first bound is slightly more involved, it defines a bound for the whole range of δ for $0 < \delta < \frac{1}{2}$. The second bound, as defined in Theorem 3.3.2, however only works starting from $\delta \geq \frac{1}{6}$. For values smaller than $\frac{1}{6}$, a different construction as in Theorem 3.3.3 is necessary.

The crucial case in the following proof is Case 1.2, where the knapsack of the optimal solution can be forced by a clever algorithm to not be completely filled.

Theorem 3.3.1. *For every $0 < \delta < 0.5$, there exists no algorithm solving the* Oskp *problem with a competitive ratio better than $\frac{1}{p}$.*

Proof. Let $0 < \varepsilon < \min(p, \delta)$ such that $1/\varepsilon \in \mathbf{N}$, and let $k = \left\lfloor \frac{2}{1-2\delta} \right\rfloor$. Let us consider an arbitrary algorithm for the Oskp problem. The following instance is announced as a prediction:

$$P = \left(\underbrace{\varepsilon,}_{1/\varepsilon \text{ many}} \quad \underbrace{\frac{1}{2},}_{k \text{ many}} \quad 1 - p - \delta \right)$$

We do a full case distinction on the possible behaviors of the algorithm. The first item is presented as ε.

Case 1: The algorithm packs ε:

The subsequent $1/\varepsilon - 1$ items are presented as 0. The next item is revealed with size p. This is possible as $p + \delta \geq 1/2$, which follows from Equation (3).

Case 1.1: The algorithm packs p:

All remaining items are presented as $1 - p$, which the algorithm can not pack due to the ε-item. As we have seen, it is possible that an item which was announced to be of size $1/2$ can be presented as p. Therefore, presenting those items as $1 - p$ is also possible due to the symmetry of the deviation.

An optimal solution can pack both p and its counterpart, while the algorithm has packed p and an item of size ε. The competitive ratio is then $1/p$ for ε converging to 0.

Case 1.2: The algorithm rejects p:

The subsequent $k-1$ items will be presented as p if the algorithm continues to reject them. If an algorithm should pack any of these items, we can use the same argumentation as in Case 1.1.

Should the algorithm reject all k items of size p, the last item is presented as $1 - p - 2\delta$. The optimal solution consists of all k items of size p, whereas the algorithm only has the item of size ε and the last item of size $1 - 2\delta - p$ in its knapsack. Since $1 - p - 2\delta \leq p$ by Equation (3.3) and $kp/(1 - p - 2\delta) = 1/p$ by Equation (3.2) is valid, the competitive ratio is at least $1/p$ for ε converging to 0.

Case 2: The algorithm rejects ε:

The subsequent $1/\varepsilon - 1$ items will be presented as ε if the algorithm continues to reject them. If the algorithm should pack any of these items, the remaining items are presented as 0, and we use the same argumentation as in Case 1.

Should the algorithm reject all $1/\varepsilon - 1$ items of size ε, the subsequent k items will be presented as $p + \varepsilon$. If such an item is packed, the argumentation is the same as in case 1.1, just with the difference that the optimal solution contains the ε-items with a total size of 1.

If no such item is packed, the last item is presented of size $1 - 2\delta - p \leq p$ (see Equation (3.3)). The optimal solution packs the full knapsack using items of size ε. Again, the competitive ratio is at least $\frac{1}{p}$. □

The following theorem gives a better bound than Theorem 3.3.1 when $q < p$ happens. Note that $q > p$ is true for $\frac{1}{6} < \delta < \frac{1}{24}(9 - 2\sqrt{3}) \approx 0.23$ and the relation $\frac{3}{16} \in [\frac{1}{6}, 0.23]$ holds. Consequently, after Theorem 3.3.2, it remains to inspect the lower q-bound for $\delta < 1/6$.

Theorem 3.3.2. *For every $\frac{3}{16} < \delta < 0.5$, there exists no algorithm solving the* Oskp *problem with a competitive ratio better than $\frac{1}{q}$.*

Proof. Let $\delta \leq \varepsilon > 0$ such that $1/\varepsilon \in \mathbf{N}$, let $k = \left\lceil \frac{2}{1-2\delta} \right\rceil$, and recall that $q = 1 - 2\delta - 1/k$. Let us consider an arbitrary algorithm for the Oskp problem. The

following prediction is announced to the algorithm:

$$P = \left(\underbrace{\varepsilon,}_{1/\varepsilon \text{ many}} \quad \underbrace{\delta,}_{k \text{ many}} \quad q + \delta \right)$$

We do a case distinction on the potential behaviors of the algorithm. The first item is presented as ε.

Case 1: The algorithm packs ε:

The subsequent $1/\varepsilon - 1$ items are presented as 0. The next presented item is of size $1/k$. This is possible if $2\delta \geq q$ since Equation (3.3) yields $q \geq 1/k$. The former holds for $\delta \geq 3/16$.

Case 1.1: The algorithm packs $1/k$:

The remaining $k - 1$ items are presented as 0, with the last item presented as $1 - 1/k = q + 2\delta$. An optimal solution can pack both $1/k$ and its counterpart, while the algorithm has packed $1/k$ and an item of size ε. The competitive ratio is then k for ε converging to 0 which yields the wished ratio by $k \geq 1/q$ thanks to Equation (3.3).

Case 1.2: The algorithm rejects $1/k$:

The subsequent $k-1$ items will be presented as $1/k$, with the last item presented as q, if the algorithm continues to reject them. If an algorithm should pack any of these items, we can use the same argumentation as in Case 1.1.

The optimal solution consists of all k items of size $1/k$, which add up to exactly 1, whereas the algorithm only has the item of size ε and the last item q in its knapsack. The competitive ratio is again at least $1/q$ for ε converging to 0.

Case 2: The algorithm rejects ε:

The subsequent $1/\varepsilon - 1$ items will be presented as ε as long as they get rejected. If an algorithm packs any of these items, we can use the same argumentation as in Case 1.

Should the algorithm reject all $1/\varepsilon - 1$ items of size ε, the subsequent k items will be presented as 0. The last item is presented as q, which directly yields the competitive ratio of $1/q$. Here, an optimal solution consists of a full knapsack with ε items. □

The crucial issue of Theorem 3.3.2 for small δ are the items of announced size δ, where an adversary must be able to present them as a 0 or as $\frac{1}{3}$ for all $\delta < \frac{1}{6}$. As this is not possible, we need to handle the setting $\delta < \frac{1}{6}$ as a special case. In particular, $q > p$ for $0 < \delta < \frac{1}{12}(4 - \sqrt{6}) \approx 0.129$ is true, and since $\frac{1}{12} < 0.129$ holds, starting with delta at $1\frac{1}{12}$ is no limitation.

Theorem 3.3.3. *For every $\frac{1}{12} < \delta < \frac{1}{6}$, there exists no algorithm solving the* Oskp *problem with a competitive ratio better than $\frac{1}{q}$.*

Proof. We define $a = \frac{1}{3} - 2\delta$, and let $\varepsilon > 0$ be arbitrary such that $1/\varepsilon \in \mathbf{N}$ and $a/\varepsilon \in \mathbf{N}$ holds. Let us consider an arbitrary algorithm for the Oskp problem. The algorithm receives the following prediction:

$$P = \left(\underbrace{\varepsilon,}_{1/\varepsilon \text{ many}} \quad \underbrace{\frac{1}{3} + \delta}_{3 \text{ many}} \right)$$

We do a similar full case distinction on the possible behaviors of the algorithm, like in Theorem 3.3.1 and Theorem 3.3.2. The first item is presented as ε, but this time we stop presenting ε-items when the algorithm has packed exactly an amount of y between a and $a + \varepsilon$.

Case 1: The algorithm packs an amount of y with ε-items:

The subsequent ε-items are presented as 0. The next item is presented of size $1/3$.

Case 1.1: The algorithm packs $1/3$:

The remaining 2 items are presented as $\frac{2}{3} - a = \frac{1}{3} + 2\delta$. An optimal solution can pack $1/3$ together with one large item and several ε-items such that it achieves a packing of at least $1 - \varepsilon$, while the algorithm has packed $1/3$ and some ε-items of total size less than $a + \varepsilon$. The competitive ratio is then $(2/3 - 2\delta)^{-1} = (1 - 1/3 - 2\delta)^{-1} = 1/q$ for ε converging to 0, which yields the wished competitive ratio.

Case 1.2: The algorithm rejects $1/3$:

The subsequent 2 items will be presented as $1/3$ until one gets accepted. If an algorithm accepts one of the first two items, we can use the same argumentation as in Case 1.1.

Otherwise, the algorithm accepts the last $1/3$-item or none of them. The optimal solution consists of all 3 items of size $1/3$, which add up to exactly 1, whereas the algorithm again has only the ε-items of total size $a + \varepsilon$ and at most the last $1/3$ item in its knapsack. The calculation of the competitive ratio is analogous to case 1.1.

Case 2: The algorithm do not pack at least $y > a$ with ε-items:

We presented all the ε-items as ε, and we know that after these items the algorithm got at most $y \leq a$ in the knapsack.

Next, we start to reveal $1/3 + (a - y) + \varepsilon$ items. This is possible because of the relation

$$\frac{1}{3} + \varepsilon \leq \frac{1}{3} + (a - y) + \varepsilon \leq \frac{1}{3} + a + \varepsilon = \frac{2}{3} - 2\delta + \varepsilon \leq \frac{1}{3} + 2\delta \quad \text{for } \delta > \frac{1}{12}.$$

Should the algorithm pick one of these 3 items, we are in the same setting as in case 1.1, but with an optimal solution consisting of just ε-items. □

As the construction of Theorem 3.3.2 cannot work for small δ, it is also worth noting that the construction of Theorem 3.3.3 is not working for larger δ. This is mainly because the items which are announced with size $\frac{1}{3}+\delta$ must be presented of size $\frac{1}{3}$ at least. Even if the 3 would be replaced with some k, it would allow an algorithm to pack multiple of those items without a chance of an adversary to hinder him.

Finally, we see that the case $\delta \geq 0.5$ can be handled using a standard construction by Marchetti-Spaccamela and Vercellis [77], thus no algorithm can achieve a bounded competitive ratio here.

Theorem 3.3.4. *For every $\delta \geq 0.5$, there exists no algorithm solving the* Oskp *problem with a constant competitive ratio.*

Proof. Consider an arbitrary algorithm and the prediction $P = (0.5, 0.5)$. The first item arrives as $\varepsilon > 0$. If the algorithm rejects it, the second item arrives as 0. Otherwise, the second item arrives as $1 - \varepsilon/2$, so the algorithm cannot pack it, whereas the optimum solution can. This construction shows that no algorithm can be competitive. □

3.3.2 Upper Bounds

We start this part with a simple algorithm that either takes the largest announced item or packs the knapsack in a greedy way.

Algorithm 5 $\frac{2}{1-2\delta}$-competitive Algorithm for $0 < \delta < \frac{1}{2}$.

if b is the largest announced item and $b \geq 0.5$ **then**
  Pack only b. END
else
  Greedily pack items. END

It turns out that this algorithm already matches the lower bound for all accuracies δ of the form $\frac{1}{2} - \frac{1}{k}$ for integers $k \geq 3$.

Theorem 3.3.5. *Given a fixed δ with $0 < \delta < \frac{1}{2}$, Algorithm 5 solves the* Oskp *problem with a competitive ratio of at most $\frac{2}{1-2\delta}$.*

Proof. Assume that there exists an announced item of size at least 0.5. Then the algorithm waits for it, packs it, and achieves a gain of $0.5 - \delta$, which gives us the claimed bound.

Thus, assume that such an item does not exist but that there exists an item that the algorithm cannot fit into its knapsack when packing greedily. This item can be at most of actual size $0.5+\delta$, meaning our gap due to not packing this item is at most $1 - (0.5 + \delta) = 0.5 - \delta$, which again gives us our wanted bound. □

In the rest of this section, we will present a more refined algorithm and prove that it matches the lower bounds of Theorems 3.3.1 and 3.3.2. Let us fix an instance (P, δ), where $0 < \delta < 0.5$ holds, and $P = (x'_1, \ldots, x'_n)$ are the announced item sizes of the Oskp problem. As in Definition 3.3.1, if x is an item, then x' denotes its announced size. Let $c = 1/\min(p, q)$ and $\overline{c} = 1/c$, see Equation (3.1) for the definition of p and q.

Algorithm 6 c-competitive algorithm for $0 < \delta < 0.5$.

```
1: if there is an item x such that x' ≥ c̄ + δ then
2:     Pack only x. END.
3: if all items have announced size ≤ 1 − c̄ − δ then
4:     Pack greedily. END.
5: Let x_l be the last item such that x'_l ∈ (1 − c̄ − δ, c̄ + δ).
6: greedy := false.
7: while there is a next item y in the queue do
8:     if y = x_l then
9:         greedy := true
10:    Let m be the size of already packed items.
11:    if greedy then
12:        Pack y if it fits into the knapsack.
13:    else if m ∈ [c̄ − (x'_l − δ), 1 − (x'_l + δ)] or y + m ∈ (1 − (x'_l + δ), c̄) then
14:        Skip y.
15:    else
16:        Pack y if it fits into the knapsack.
```

It is easy to see that the algorithm achieves the desired competitive ratio in many instances.

Lemma 3.3.1. *If the largest announced size is not in* $(1 - \overline{c} - \delta, \overline{c} + \delta)$*, then Algorithm 6 achieves the competitive ratio of at least* $\overline{c}$*.*

Proof. First, suppose there is an item x of announced size at least $\overline{c} + \delta$. In this case, x is the only item packed, see line 2. Since $x \geq \overline{c}$ is true, the competitive ratio is ensured.

Second, suppose that all items have announced size at most $1 - \overline{c} - \delta$. In this case, we pack greedily, see line 4. Let x be the first item that cannot be packed by the algorithm; if x does not exist, we have found the optimal solution. Otherwise, $x \leq 1 - \overline{c}$ holds, which means that we have packed more than $\overline{c}$ and the competitive ratio is also ensured. □

For the rest of the section, suppose that the largest announced item is in $(1-\overline{c}-\delta, \overline{c}+\delta)$, and let x_l be the last announced item such that $x'_l \in (1-\overline{c}-\delta, \overline{c}+\delta)$.

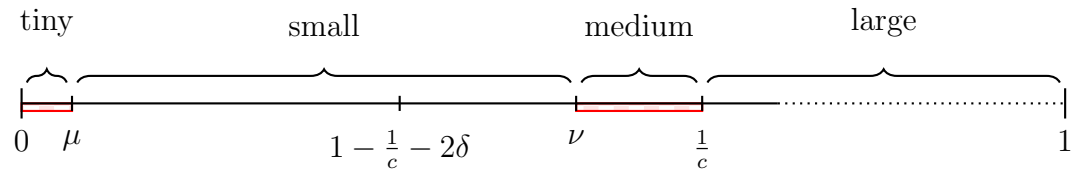


Figure 3.3: Range of item sizes. *Forbidden* ranges of item sizes to be packed are marked in red.

In this case, the algorithm can guarantee a packing of at least $\overline{c}$ for most instances. To see this, we classify each item y as either *tiny, small, medium* or *large* depending on the actual size of y and the sum of the already packed items when y arrives. We use the substitutions $\mu := \overline{c} - (x'_l - \delta)$ and $\nu := 1 - (x'_l + \delta)$. Figure 3.3 provides an overview of the variables and relations.

Definition 3.3.2. Let y be an item and let m be the sum of the already packed items when y arrives.

We say that an item y is *tiny* if $y + m \in [0, \mu)$, *small* if $y + m \in [\mu, \nu]$, *medium* if $y + m \in (\nu, \overline{c})$, and *large* otherwise.

Observe that before x_l arrives, medium items are skipped by the algorithm, see line 13. The case when a small or large item arrives is handled by the following lemma.

Lemma 3.3.2. *If a small or large item y arrives before x_l, then Algorithm 6 reaches a packing of $\overline{c}$.*

Proof. Let y be the first small or large item that arrives, and let m be the size of the packing when y arrives. Since medium items are skipped, only tiny items were packed before y, which means $m < \mu$.

First, suppose that y is small. In this case, y is packed, and after that, the knapsack's packing is of size in $[\mu, \nu]$, which means that no item is packed until x_l arrives because the condition on line 13 is satisfied. Observe that when x_l arrives, the algorithm tries to pack it. Note that $m + y + x_l \leq m + (\nu - m) + (x'_l + \delta) = 1$, which means that x_l fits into the knapsack when it arrives. Now observe that after x_l is packed, the knapsack has size at least $m + y + x_l \geq m + (\mu - m) + (x'_l - \delta) = \overline{c}$, as required.

Second, suppose that y is large. Note that $y < \overline{c} + 2\delta$ since otherwise we would be in the case handled by Lemma 3.3.1.

Now observe that $m + y \leq \mu + (\overline{c} + 2\delta) = 2\overline{c} - x'_l + 3\delta \leq 2\overline{c} - (1 - \delta - \overline{c}) + 3\delta = 3\overline{c} + 4\delta - 1 \leq 3q + 4\delta - 1$, where the last inequality follows from $\overline{c} = \min(p, q)$. Now we expand the definition of q, see Equation (3.1):

$$3q + 4\delta - 1 = 4\delta - 1 + 3(1 - 2\delta - \frac{1}{\lceil \frac{2}{1-2\delta} \rceil}) = 2 - 2\delta - \frac{3}{\lceil \frac{2}{1-2\delta} \rceil} \leq 2 - 2\delta - (1 - 2\delta) = 1$$

Hence, y fits into the knapsack. By Definition 3.3.2, $y+m \geq \overline{c}$ is valid, as required.

In both cases, the algorithm packs y and can guarantee a packing of at least $\overline{c}$. □

Recall that the instance of Oskpwe are solving is (P, δ). The following lemma says that we can assume that x_l is the last item.

Lemma 3.3.3. *Let P' be the prefix of P ending with x_l and suppose that Algorithm 6 achieves a competitive ratio of at most c on (P', δ). Then, it also achieves a competitive ratio of at most c on (P, δ).*

Proof. If all items after x_l are packed, then the competitive ratio can only decrease. Suppose that an item x comes after x_l and is not packed. By choice of x_l, we know that $x' \leq 1 - \overline{c} - \delta$, which means that $x \leq 1 - \overline{c}$. Hence, the algorithm has packed more than $\overline{c}$, and the competitive ratio is at most c. □

By Lemma 3.3.2 and 3.3.3, we may assume that x_l is the last item and all other items are either tiny or medium. We split the analysis into two cases depending on whether at most $\lfloor \frac{2}{1-2\delta} \rfloor$ non-tiny items are packed in an optimal solution.

Lemma 3.3.4. *Assume there are no small or large items, and x_l is the last item. If there is an optimal solution packing at most $k := \lfloor \frac{2}{1-2\delta} \rfloor$ non-tiny items, then Algorithm 6 achieves a competitive ratio of at most $\frac{1}{p}$.*

Proof. Recall that the algorithm packs exactly the tiny items and x_l, which means we may assume that there are no tiny items (their presence would decrease the competitive ratio). Since $x_l \geq 1 - \overline{c} - 2\delta$ holds due to the choice of x_l, we may assume $x_l = 1-\overline{c}-2\delta$ without decreasing the ratio. By Equation (3.3), $1-\overline{c}-2\delta \leq \overline{c}$ is valid, since $\overline{c} = \min(p, q)$ is defined. Hence, each item is of size at most $\overline{c}$ and the ratio is at most $\frac{k\overline{c}}{1-\overline{c}-2\delta}$. By Equation (3.2), we have

$$\frac{k\overline{c}}{1-\overline{c}-2\delta} \leq \frac{kp}{1-p-2\delta} = \frac{1}{p}.$$

Hence, the algorithm achieves a competitive ratio of $\frac{1}{p}$. This matches the lower bound given by Theorem 3.3.1 for the case $p \leq q$. □

For the other case, we can guarantee that the knapsack is filled with at least $\frac{1}{q}$.

Lemma 3.3.5. *Assume there are no small or large items, and x_l is the last item. If there is an optimal solution packing at least $k := \lceil \frac{2}{1-2\delta} \rceil$ non-tiny items, then Algorithm 6 achieves a competitive ratio of at most $\frac{1}{q}$.*

Proof. Since at least k non-tiny items are packed by an optimal solution, there is a non-tiny item of size at most $1/k$. First, suppose that there is a medium item $y \neq x_l$ such that $y \leq 1/k$. Let m be the size of the packed items immediately before y arrives. Since y is medium, we have $1/k + m \geq y + m > \nu$. Hence, after x_l is packed, the algorithm has packed

$$m + x_l \geq (\nu - 1/k) + (x_l' - \delta) = 1 - (x_l' + \delta) - 1/k + x_l' - \delta = 1 - 2\delta - 1/k = q,$$

which implies a competitive ratio of at most $1/q$.

The remaining case is $y > 1/k$ for each medium item $y \neq x_l$ and $x_l \leq 1/k$. Observe that

$$x_l' \leq 1/k + \delta = 1 - (1 - 2\delta - 1/k) - \delta = 1 - q - \delta \leq 1 - \overline{c} - \delta,$$

which is a contradiction since $x_l' > 1 - \overline{c} - \delta$. Hence, this case cannot occur, and the algorithm achieves a competitive ratio of at most $\frac{1}{q}$. □

With the previous observations, the following theorem immediately follows.

Theorem 3.3.6. *For every $\delta \in (0, \frac{1}{2})$, Algorithm 6 achieves a competitive ratio of $\frac{1}{\min(p,q)}$.*

3.4 Online Simple Knapsack with Removabilty

The online simple knapsack with removability and item size estimates can be defined analogously:

Definition 3.4.1 (The Online Simple Knapsack with Removability and Item Size Estimates Problem)**.** Given an estimate accuracy $\delta \geq 0$, an instance of the *Online Simple Knapsack with Removability and Item Size Estimate* consists of a series of *items* $I = (x_1, \ldots, x_n)$, where each *item* is a real number in $[0, 1]$. At the beginning, the accuracy δ is announced as well as the *item size estimates* $P = (x'_1, \ldots, x'_n)$, such that for each $i \in \{1, \ldots, n\}$, $x'_i - \delta \leq x_i \leq x'_i + \delta$. At each step $i \in \{1, \ldots, n\}$, the actual item size x_i of the request sequence gets revealed. An algorithm then has the option to remove a subset of items from the initially empty knapsack K, and then to either pack the current item into K, if it fits, or to reject the item:

- **Remove** Discard $S \subseteq K$ from the knapsack, set $K := K - S$.
- **Pack** If $\sum_{x_k \in K} x_k + x_i \leq 1$, set $K := K \cup x_i$.
- **Reject** Do nothing.

The Online Simple Knapsack with Removability and Item Size Estimates problem (OSKRE) is then for an algorithm `ALG` to minimize the competitive ratio between the size of its packing and the optimal solution in the same instance.

In this section, the algorithm is allowed to discard previously packed items. We consider not only additive estimate accuracy, but will later also note that multiplicative accuracy behaves very similarly.

3.4.1 Lower Bounds

Theorem 3.4.1. *For every $\delta \in (0, \frac{3}{4} - \frac{\sqrt{5}}{4}]$, there exists no algorithm solving the* Oskp *problem with removability and item size estimate with a competitive ratio better than $1/x$, where*

$$x = \frac{2 - 2\delta}{3 - 2\delta}$$

Proof. Let $\varepsilon > 0$ be small enough so that $x + 2\varepsilon \leq x + \delta$ and $1 - x - \varepsilon \geq 1 - x + \varepsilon - \delta$. An arbitrary algorithm is given the following prediction:

$$(1 - x, x + \varepsilon, x, 1 - x + \varepsilon - \delta) .$$

The first two items presented are $1-x$ and $x+\varepsilon$. The algorithm can pack either of these items, but not both.

Case 1: The algorithm packs $x+\varepsilon$:

The third item is presented as x and the last one as $1-x+\varepsilon$. Since $x > 0.5$, the algorithm can pack at most one item from the set $\{x, x+\varepsilon, 1-x+\varepsilon\}$ and its best choice is keeping $x+\varepsilon$, whereas an optimal solution is to pack x and $1-x$. Hence, the competitive ratio converges to $1/x$ as ε goes to 0.

Case 2: The algorithm packs $1-x$:

The next item is presented as $x+2\varepsilon$. Now the algorithm either keeps $1-x$ or discards it and packs $x+2\varepsilon$.

Case 2.1: The algorithm packs $x+2\varepsilon$:

The final item is presented as $1-x-\varepsilon$. Since $x+2\varepsilon \geq 1-x-\varepsilon$, the algorithm can pack at most $x+2\varepsilon$, whereas the optimal solution packs the full knapsack by taking $x+\varepsilon$ and $1-x-\varepsilon$. Hence, the competitive ratio goes to $1/x$ as ε goes to 0.

Case 2.2: The algorithm keeps $1-x$:

The final item is presented as small as possible and presented as $y := 1-x-2\delta+\varepsilon$. Now the algorithm can pack at most $1-x+y$ whereas the optimal solution packs $x+2\varepsilon+y$. Hence, the competitive ration goes to $(1-x+y)/(x+y)$ as ε goes to 0. By definition of x the algorithm cannot achieve a better competitive ratio than $\frac{1}{x} = c$ in this case. □

Note that the construction of Theorem 3.4.1 will yield a bound of Φ for $\delta = \frac{3}{4} - \frac{\sqrt{5}}{4}$, which can also be achieved without estimates as in Iwama and Taketomi [65].

Theorem 3.4.2. *For all $\delta > \frac{3}{4} - \frac{\sqrt{5}}{4}$, no algorithm for the online simple knapsack problem with removability and estimates can achieve a competitive ratio of less than Φ.*

3.4.2 Upper Bounds

In this part, we will present an algorithm that matches the lower bound of the previous part. Again, define $x = \frac{2-2\delta}{3-2\delta}$. We call items of size at most $1-x$ *small* items, items of size strictly between $1-x$ and x *medium* items, and items of size at least x *large* items.

Note that once two medium items are packed, Algorithm 7 terminates, see line 11. Hence, line 10 is well defined, as at most one medium item can be packed at that point.

Now we prove that Algorithm 7 matches the lower bound of Theorem 3.4.1.

Theorem 3.4.3. *Algorithm 7 achieves a competitive ratio at least $1/x$ for $\delta \leq \frac{3}{4} - \frac{\sqrt{5}}{4}$.*

Algorithm 7 for Online Knapsack with Removability and Estimates

1: Let x_l be the last item such that $x'_l > 1 - x - \delta$.
2: **while** there is an item y in the queue **do**
3: **if** at least x has been packed **then** END.
4: **if** y is large **then** Remove everything. Pack y. END.
5: **if** y is small **then** Pack y if it fits.
6: **if** y is medium **then**
7: **if** no medium item is packed **then**
8: Pack* y.
9: **else**
10: Let z be the medium item packed.
11: **if** $y + z \leq 1$ **then** Remove everything but z. Pack y. END.
12: **else if** $y < z$ or $y = x_l > z$ **then** Remove z. Pack* y.
13: **else** Ignore y.

* If it is not possible to pack y on lines 8 or 12, we keep removing small items until it becomes possible.

Proof. First, suppose that y is the first large item in the instance. Observe that the algorithm packs y on line 4 and terminates. Since $y \geq x$, a competitive ratio of at least $1/x$ is achieved. From now on, suppose that there are no large items.

Second, suppose that there are two medium items y_1 and y_2 such that $y_1 + y_2 \leq 1$. Observe that until x_l arrives, if there is only one medium item packed, then it is the smallest medium item that has arrived so far (see line 12). Hence, there is an iteration of the while loop in which the condition on line 11 is satisfied, i.e., two medium items are packed, and the algorithm terminates. Since $2(1 - x) > x$ holds for $\delta > 0$, the algorithm has again packed at least x as required. From now on, suppose that no two medium items fit together, no matter their position in the instance.

Third, suppose there is a small item a that is not part of the knapsack at the end, i.e., a was ignored on line 5 or removed on line 8 or 12 before packing y. Let m be the value in the knapsack at the end of the iteration in which a is discarded, and observe that $m + a > 1$ (otherwise a would be packed, resp. not removed). Since $a \leq 1 - x$, we have $m > 1 - a \geq 1 - (1 - x) = x$. Hence, the algorithm has packed at least x (and terminates in the next iteration, see line 3). From now on, we may assume that the algorithm packs all small items. In particular, if there is at most one medium item, the algorithm finds the optimal solution. Hence, assume that there are at least two medium items.

Observe that by the definition of x_l, there are no medium items after x_l. Hence, when x_l arrives, there is a medium item y in the knapsack. Let s be the size of

all small items except for x_l (note that x_l is small or medium). Assume x_l is small, i.e., the algorithm packs at least $x_l + (1 - x) + s$. Since no two medium items fit together, an optimal solution packs at most $x_l + x + s$. Observe that $x_l \geq 1 - x - \delta - \delta$, as it was announced with size at least $1 - x - \delta$. Hence, the competitive ratio is at most:

$$\frac{x_l + x + s}{x_l + (1 - x) + s} \leq \frac{x + 1 - x - 2\delta}{1 - x + 1 - x - 2\delta} = \frac{1}{x} \text{ as desired.}$$

Finally, suppose that x_l is medium. Observe that the algorithm packs $\max(y, x_l) + s > 0.5 + s$, as y and x_l combined exceed 1. Since no two medium items fit together, an optimal solution packs at most $x + s$. Hence, the competitive ratio is at most $x/0.5 < 1/x$ as desired. □

Again, if $\delta \leq \frac{3}{4} - \frac{\sqrt{5}}{4}$, an optimal algorithm does not need to consider the estimates given.

Theorem 3.4.4. *For all $\delta > 0$, there is an algorithm that achieves a competitive ratio of Φ for the online simple knapsack problem with removability and estimates [65].*

3.4.3 Multiplicative Estimate Accuracy

While the previous research was dedicated to additive estimate accuracy, it turned out that the results easily translate to the multiplicative model (as studied in [**?**]) with removability. This is not surprising, as the estimate accuracy is only needed for one item at the end in both the lower and the upper bounds, which has a medium size.

Therefore, it is possible to use an analogous lower bound construction and upper bound algorithm for the multiplicative case when the aimed competitive ratio is changed accordingly. In the multiplicative setting, x, which is defined as $\frac{2-2\delta}{3-2\delta}$ in the additive setting, can be redefined to $x = \frac{\sqrt{\delta^2+10\delta+9}+\delta-3}{4\delta}$, achiving a competitive ratio of $\frac{1}{x}$ then. As in the additive case, both bounds are tight until they reach Φ where the estimates become worthless due to the bound of Φ without estimates.

3.5 Bin Packing

We consider a slightly altered version of the Online Bin Packing problem. Here, the algorithm gets presented a list of items and their sizes in the beginning. Additionally, the precision of the estimates is given in advance.

Definition 3.5.1 (The Online Bin Packing with Item Size Estimates Problem)**.** In the Online Bin Packing with Item Size Estimates problem (OBPE), the algorithm A receives *estimated* sizes $c'(a) \in (0, 1]$ for every $a \in L$, as well as the (relative) *accuracy* $\delta \in (0, 1]$ in the beginning. Then the items are revealed with their actual sizes $c(a_i)$ and must be designated to some bin before the next item size is revealed. The actual size $c(a)$ of each item is in the interval $[c'(a)(1-\delta), \min(c'(a)(1+\delta), 1)]$.

Even as the algorithm knows more about the instance than in online bin packing, details are still only revealed in an online manner. Here, $\delta = 0$ is identical to offline bin packing, whereas $\delta = 1$ is essentially classical online bin packing (Here it receives the number of items, those can be presented with size 0, however). In some sense, the model smoothly connects the two intensively studied models.

3.5.1 Lower Bounds

Bin packing strategies without information about the input ahead have to pack items rather tightly as they know that the instance could end at any moment. This is not the case for our problem: It might in fact in some cases be wise for an algorithm to use n bins for the first n items if it knows that these bins will reach a sufficient fill level later on. Therefore, the main difficulty such algorithms face is the uncertainty about whether a given subset of items that are yet to be revealed fit together in a single bin. The most challenging input instances are the ones that provide as little information as possible for an algorithm to deal with this uncertainty.

Lower Bound of $\frac{4}{3}$ for arbitrary good accuracy

However, even if the deviation is arbitrarily small, an algorithm cannot achieve a near-optimal result. This is not particularly surprising, as small deviations already cause situations, in which in the beginning it is hidden if items will fit together or not. The following construction shows that each algorithm needs to have a competitive ratio of at least $\frac{4}{3}$ by ensuring, that the items, that are placed on already partly filled bins during the first half of the packing process, are the smallest of the entire input instance.

Theorem 3.5.1. *No strategy for* OBPE *can achieve a competitive ratio of less than* $\frac{4}{3}$*, even for arbitrary small accuracy.*

Proof. We announce a list of items L with $2n$ items with an announced size of $\frac{1}{2}$ for every item in L for some large n. The instance consists of two phases; the first one consisting of $\frac{4n}{3}$ items, the second of the remaining $\frac{2n}{3}$ items. Items of the first phase will be called an item *stacked* if they were placed upon another item by the algorithm, and *laid out* if they instead were put into a new bin. Each of the first $\frac{4n}{3}$ items will be denoted by $x_{i,k}$ where i and k are essentially counters that track the algorithm's decisions: i equals one plus the number of bins containing two items at the time when $x_{i,k}$ is presented, and k counts the number of items that were revealed after the last item was stacked. (This means that we start counting both i and k from 1). Let $t \leq \frac{2n}{3}$ be the number of items the algorithm stacks, then we observe that for every $i \leq t$ there exists exactly one stacked item which we will call x_{i,k_i}. The sizes of the items are

$$c(x_{1,k}) = \frac{1}{2} - \frac{k\delta}{2n} \text{ and } c(x_{i,k}) = c(x_{j,k_{j-1}}) - \frac{k\delta}{2n^i}$$

for $i > 1$, where $x_{j,k_{j-1}}$ is the last item that was laid out before $x_{i,k}$ was revealed. Note that all items indeed have a size of at least $\frac{1}{2}(1-\delta)$.

For all $1 \leq i \leq t$ and all k for which $x_{i+1,k}$ exists we now get

$$c(x_{i+1,k}) = c(x_{j,k_{j-1}}) - \frac{k\delta}{2n^{i+1}} > c(x_{j,k_{j-1}}) - \frac{\delta}{2n^i} \geq c(x_{i,k_i}).$$

Furthermore, if $k_i > 1$, we also get

$$c(x_{i,k_{i-1}}) > c(x_{i,k_{i-1}}) - \frac{k\delta}{2n^{i+1}} = c(x_{i+1,k}).$$

Together, these two statements imply that the stacked items $x_{1,k_1}, ..., x_{t,k_t}$ are smaller than any of the remaining $n - t$ items revealed so far.

The algorithm assigns the first $\frac{4n}{3}$ items into pn bins, where $\frac{2}{3} \leq p \leq \frac{4}{3}$, with $n(\frac{4}{3} - p)$ bins containing two items.

In the second phase, $\frac{2n}{3}$ items are presented, again all with size close to $\frac{1}{2}$. Here, $n(\frac{4}{3} - p)$ items of size slightly larger than $\frac{1}{2}$ are presented, which will fit only together with the stacked items of the first phase. The remaining $\frac{2n}{3} - n(\frac{4}{3} - p)$ items will be of size exactly $\frac{1}{2}$.

Therefore, the algorithm will use at least $pn + n(\frac{4}{3} - p) = \frac{4n}{3}$ bins, as all of the larger items from the second phase will need their own bin in addition to the pn bins of the first phase. The optimal packing however will need only n bins, as the large items of the second phase can be packed with the stacked items of the first phase, and all remaining items are of size at most $\frac{1}{2}$. Therefore, all bins of the optimal solution will consist of two items.

□

The idea of ensuring that only the smallest items are stacked was previously also used in lower bound constructions like from Babel et al., that are designed for a setting in which at most two items can be placed in a single bin [12].

Lower Bound for large prediction errors

Not surprisingly, the more the quality of the estimates degrades, it becomes less likely for an algorithm to achieve a significantly better performance than an algorithm that does not use the given estimates at all.

When the allowed deviation factor is larger than $\frac{41}{43}$, an online algorithm cannot gain substantial information out of the predictions anymore. The following proof uses a modified instance similar to Yao, where a 1.5 competitive ratio is proven for the classical online bin packing problem without any announcements[93].

Theorem 3.5.2. *No algorithm for* OBPE *achieves a competitive ratio better than* 1.5 *for a deviation* $\delta > \frac{41}{43}$

Proof. Let $L = L_1L_2L_3$ be the input instance, where each sublist L_i consists of n items for some n divisible by 12. Each item is announced with size $c'(a) = \frac{43}{168}$.

The first items of L_1 will be presented of size $\frac{1}{7} + \varepsilon$ for a small $\varepsilon > 0$, therefore an algorithm can pack one to six of them into the same bin.

As the remaining items of L_2 and L_3 can be presented as $\frac{43}{168}(1-\delta) < \frac{1}{84}$, an algorithm is forced to not use more than $\frac{n}{4}$ bins to pack the items of L_1. Otherwise, the adversary could decide to present the remaining L_2 and L_3 sufficiently small such that six items of each of the classes L_1, L_2, and L_3 fit together in one bin. Since the optimal solution then has only used $\frac{n}{6}$ bins, the algorithms solution cannot be better than 1.5-competitive. Therefore, we assume that the algorithm has not taken more than $\frac{n}{4}$ bins for the items of L_1.

Next, n items of L_2 are presented of size $\frac{1}{3} + \varepsilon$. Again, if the adversary decides to present the items of L_3 smaller than $\frac{1}{84}$ each, an optimal solution could consist of $\frac{n}{2}$ used bins (two items of each class, in each of the bins). Therefore, when trying to avoid a competitive ratio of $\frac{3}{2}$ or more, the algorithm can only use up to $\frac{3n}{4}$ bins to pack L_2 items on top of the L_1 items. Note that only two items of L_2 fit together, if zero to two items of L_1 are in a bin; and only one item of L_2 fits into a bin with three or four items of L_1. If more than half of a bin should remain empty, it can fit at most three items of L_1 or one item of L_2 (possibly combined with a single item of L_1). As the items could be revealed with the same idea as in Yao [93], we omit the details of the calculation and refer to [93]. It follows that at least $\frac{n}{2}$ bins must be filled by more than $\frac{1}{2}$, given the constraints that the L_1 items must completely be contained in $\frac{n}{4}$ bins.

When finally n L_3 items with value $\frac{1}{2}+\varepsilon$ are presented, an algorithm can only place one of them into each bin filled less than $\frac{1}{2}$. As at least $\frac{n}{2}$ bins are already filled by more than half, an algorithm will need at least $1.5n$ bins.

Therefore, no algorithm can achieve a competitive ratio better than $\frac{3}{2}$.

□

3.5.2 The Planned Harmonic Algorithm for precise Estimates

Lower bound constructions like Theorem 3.5.1 suggest that items with an announced size of around $\frac{1}{2}$ might be the hardest ones for an algorithm to deal with. This is not surprising, as an algorithm cannot know whether two such items fit into the same bin and thus risks leaving a lot of unused space if it makes a wrong decision while packing such an item.

In this section, we present an algorithm called Planned-Harmonic (PH) which achieves a competitive ratio of $\frac{3}{2}$ for rather accurate estimates.

The main idea behind PH is to pack items of size around $\frac{1}{2}$ together with smaller ones and hence avoid such difficult decisions as much as possible. The algorithm Harmonic$_4$ is used as a fallback for items that could not have been packed this way.

We will call items in the size range between $(\frac{1}{k+1}, \frac{1}{k}]$ I_k-items for $1 \leq k \leq 3$, and I_4-items otherwise. Similarly, bins that exclusively contain I_k-items are called I_k-bins for all $k \leq 4$. The algorithm PH follows the "natural" division into a planning phase which happens before any items are shown, and an update phase which is executed each time an item is revealed.

Planning Phase

Before the first item gets revealed, algorithm 8 is called. Here, PH identifies all items that might be larger than $\frac{1}{2}$ and reserves a separate bin for each of these items. It then assigns items that have a size of at most $\frac{1}{4}$ to those fixed bins in a greedy fashion.

Update Phase

Note that items, that are potentially larger than $\frac{1}{2}$ (in the code called I_{1p}), are treated differently than items that are guaranteed to be larger than $\frac{1}{2}$ (which are called I_{1g}): They were assumed to be as large as possible. The reason is that we strongly prefer I_1-items to end up in the reserved bins.

If an I_{1p}-item is revealed to have a size of at most $\frac{1}{2}$ during the update phase, it seems wiser to pack it using the standard harmonic algorithm. This is only sensible

Algorithm 8 Planning phase of PH

$I_{1g} \leftarrow \{a \in L \mid (1-\delta)c'(a) > \frac{1}{2}\}$
$I_{1p} \leftarrow \{a \in L \mid (1+\delta)c'(a) > \frac{1}{2}\} \setminus I_{1g}$
$I_{4+} \leftarrow \{a \in L \mid (1-\delta)c'(a) \leq \frac{1}{4}\}$
for $a \in I_{1g}$ **do**
    $S \leftarrow$ maximal set in I_{4+} with $(1+\delta)(c'(a) + \sum_{a' \in S} c'(a')) \leq 1$
    $I_{4+} \leftarrow I_{4+} \setminus S$
    Plan to pack a and items in S together
for $i \in \{1, ..., |I_{1p}|\}$ **do**
    Break if $I_{4+} = \emptyset$
    $S_i \leftarrow$ maximal set in $I_{4+} \setminus S \setminus S_{i-1}$ with $(1+\delta)(\frac{1}{2(1-\delta)} + \sum_{a' \in S_i} c'(a')) \leq 1$
    Designate bin B_i for S_i (note that i might not be packed into B_i)

as long as there are enough remaining I_{1p}-items to be revealed, which then can be used to fill the remaining reserved bins.

To do so, we assume that every I_{1p}-item is as large as possible in the planning phase. This allows us, to decide freely on where to place it after its revelation and is essentially the key reason why PH is $\frac{3}{2}$-competitive.

The strategy for the update phase is described in Algorithm 9. Here, a denotes the current item and $B_1, ..., B_l$ the bins which already contain an I_{1p}-item. With k we denote the number of items from I_{1p} that are assigned into a set S_i in the planning phase.

Algorithm 9 Update phase of PH

if $a \in I_{1p}$ and $c(a) > \frac{1}{2}$ **then**
    Pack a into B_{l+1}, or into a new empty bin if $k = l$
else if $a \in I_{1p}$ and $c(a) \in (\frac{1}{3}, \frac{1}{2}]$ **then**
    $m \leftarrow$ the number of items in I_{1p} that have not been packed yet
    if $m \geq k - l$ **then**
        Pack a using Harmonic$_4$
    else
        Pack a into B_{l+1}
else
    Pack a according to plan, or using Harmonic$_4$ if no plan existed for a

In this section, we will see that PH is $\frac{3}{2}$-competitive if the estimates are rather accurate.

Theorem 3.5.3. *The* Planned Harmonic *is at most $\frac{3}{2}$-competitive for a deviation $\delta \leq \frac{1}{35}$.*

To prove this theorem, we start by classifying the I_1-items, depending on how the algorithm has dealt with them:

An item of actual size greater than $\frac{1}{2}$ will be called *filled with smaller items* if it is, together with the remaining items of $S_i \subseteq I_{4+}$, packed in the same bin. There are two cases to consider: either all I_1-items are filled with smaller items, or not. In the former case, it can easily be seen that all bins except for a constant number have a fill level of at least $\frac{2}{3}$.

Lemma 3.5.1. *For $\delta \leq \frac{1}{35}$ all bins B_i are filled to at least $\frac{2}{3}$, if their respective item I_{1p} is filled with smaller items (except possibly the last one).*

Proof. Each of the bins B_i contains one I_{1p}-item as well as all members of the set S_i which was constructed by greedily adding items from I_{4+} while preserving the constraint

$$(1+\delta)\left(\frac{1}{2(1-\delta)} + c'(S_i)\right) \leq 1$$

where $c'(S_i)$ is defined as $\sum_{a' \in S_i} c'(a')$. An item in the set I_{4+} has an announced size of at most $\frac{1}{4(1-\delta)}$. This implies that for the bin B_i, the relation

$$(1+\delta)\left(\frac{1}{2(1-\delta)} + c'(S_i) + \frac{1}{4(1-\delta)}\right) > 1 \Leftrightarrow c'(S_i) > \frac{1}{1+\delta} - \frac{3}{4(1-\delta)}$$

must hold as it would otherwise be possible to augment the set S_i by another I_{4+}-item which would contradict the maximality of S_i, except of possibly the last bin when there are no items in I_{4+} left. Since any I_{1p}-item has an expected size of more than $\frac{1}{2(1+\delta)}$, for $\delta \leq \frac{1}{35}$ the fill level of B_i has to be at least

$$(1-\delta)(\frac{1}{2(1+\delta)} + c'(S_i)) > \frac{3(1-\delta)}{2(1+\delta)} - \frac{3}{4} \geq \frac{2}{3}.$$

□

Combining this with the behavior of the underlying harmonic algorithm results in the following observation:

Lemma 3.5.2. *For $\delta \leq \frac{1}{35}$ and if all I_1-items can be filled with smaller items, at most 4 bins do not reach a fill level of at least $\frac{2}{3}$.*

Proof. Lemma 3.5.1 states that the bins $B_1, ..., B_{k-1}$ all have a fill level of at least $\frac{2}{3}$, and the same holds true for all I_j-bins except for one for all $j \in \{2, 3, 4\}$ due to the nature of the algorithm Harmonic$_4$. This leaves us with at most 4 bins for which we cannot guarantee the desired fill level. □

Using this lemma, we can conclude the proof for the case that all I_1-items are paired with smaller items:

Lemma 3.5.3. *The* Planned Harmonic *algorithm achieves a competitive ratio of at most $\frac{3}{2}$ if $\delta \leq \frac{1}{35}$ for input instances where all I_1-items can be filled with smaller items.*

Proof. If an input list L can be packed into s bins by an optimal packing, all items combined can have at most a size of s. As in the situation of Lemma 3.5.2, all but four bins are filled with at least $\frac{2}{3}$ by PH, it follows that all items are distributed into at most $\frac{3s}{2} + 4$ bins. □

We now have to deal with the case where some I_1-items have not been filled up with smaller items.

Lemma 3.5.4. *The* Planned Harmonic *algorithm achieves a competitive ratio of at most $\frac{3}{2}$ if $\delta \leq \frac{1}{35}$ for input instances when not all I_1-items got combined with smaller items.*

Proof. In this situation, we notice two things:

First, all items that turned out to be in $(1/3, 1/2]$ are considered as I_2-items and are packed according to the strategy of a harmonic algorithm (e.g., they are not combined with items of other classes). In particular, no such item will be packed into a bin B_{l+1} by line 8, as the small items are already designated to get packed together with I_1 items. Second, all small items were placed in reserved bins, so the packing does not have I_4-bins.

For the rest of the proof, we can define the following weight function w:

$$w(a) = \begin{cases} \frac{1}{k}, c(a) \in (\frac{1}{k+1}, \frac{1}{k}], 1 \leq k \leq 3 \\ 0, c(a) \in (0, \frac{1}{4}] \end{cases}$$

We now observe that the weight of the whole instance $W(L) := \sum_{a \in L} w(a) \geq \mathrm{PH}(L) - 2$ holds, since at most two bins, namely one I_2- and one I_3-bin each, do not contain items with a combined weight of 1.

Let $\overline{W} = \sup\{\sum_{a \in S} w(a) \mid S \text{ is a set of items with } \sum_{a \in S} c(a) \leq 1\}$, then we can easily see that $\overline{W} \leq \frac{3}{2}$: This is already the case if S does not contain an I_1-item, and if it does contain such an item, at most one more item with a non-zero cost and a weight of at most $\frac{1}{2}$ can fit into the set. As a result, we get

$$\mathrm{PH}(L) \leq W(L) + 2 \leq \overline{W} \cdot OPT(L) + 2 \leq \frac{3}{2} \cdot OPT(L) + 2.$$

This yields the competitive ratio in this remaining case. □

In 2016, an algorithm called Reserve-Critical for online bin packing in the advice model was first introduced by Boyar et al.[35]. Since the advice model differs from the setting we are investigating, their Reserve-Critical uses a different strategy. However, their analysis uses a similar approach as PH in this setting with estimates.

3.5.3 Algorithm for at most two Items per Bin

One example for instances, when a solution can consist of at most 2 items are instances, where all items are of size larger than $\frac{1}{3}$. In the setting of online bin packing with items size estimates, one might already see that this condition holds by the given estimates.

Delayed-Best-Fit

To deal with the setting that only two items will fit together in a bin, we present an algorithm that modifies the well-known Best-Fit Algorithm. Remember that Best-Fit packs each item into the fullest bin in that it fits, or packs it in a new bin if it does not fit in any of the used bins:

Algorithm 10 Delayed Best Fit (DBF)

```
1: n ← the number of announced items
2: a ← the current item, c(a) ← the size of the item
3: if c(a) ≤ 1/2 and a in the first 1/3 n of items of size smaller 1/2 then
4:     if a fits in a bin B an item of size larger 1/2 then
5:         Pack a in bin B to the large item
6:     else
7:         Place a in an empty bin
8: else
9:     Pack a according to Best-Fit
```

The first $\frac{1}{3}n$ of items with a size smaller or equal to $\frac{1}{2}$ are also called *special items*, as they are the only ones that are not packed in a Best-Fit manner.

Theorem 3.5.4. *The* Delayed Best Fit *algorithm achieves a competitive ratio of* $\frac{4}{3}$ *for all instances where at most two items can be placed into the same bin.*

To prove this behavior, we first classify the bins used in the analysis:

Figure 3.4 depicts all possible bin configurations in a DBF-packing along with their identifiers; later we denote the number of bins of a certain type by their corresponding lowercase letter (e.g., y is the number of bins containing a single

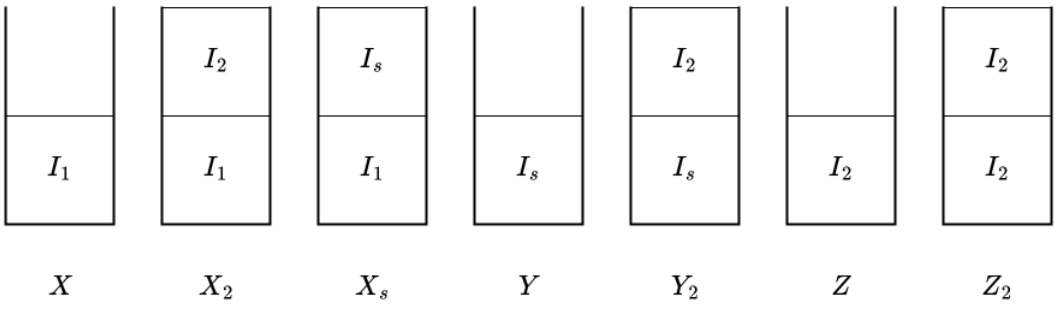


Figure 3.4: All possible bin configurations in a packing by DBF

I_s-item). Analogously to the analysis of the harmonic algorithm, we call items that are larger than $\frac{1}{2}$ I_1, items of size smaller or equal to $\frac{1}{2}$ which are no special item items of type I_2, and finally the special items I_s.

First, we need to make the following observations:

Lemma 3.5.5. *If $y > 0$ in a packing by* DBF *in an input list L, it holds* $\mathrm{DBF}(L) = n_1 + y + y_2$ *with n_1 denoting the number of I_1 items.*

Proof. DBF packs I_2-items using Best-Fit and all special items have a size of at most $\frac{1}{2}$, so before any I_2-item is placed into an empty bin it is packed together with a special item. If there exists a Y-bin in the packing there can therefore be no bins that only contain I_2-items. □

Lemma 3.5.6. *If $y > 0$ in a packing by* DBF*, then no special item that was packed in a Y- or Y_2-bin fits together with any I_1-item that lies in a X- or X_2-bin.*

Proof. We first show the statement for special items in Y-bins: For this, let a be a special item placed in a Y-bin and a' an I_1-item placed in a X- or X_2-bin in a packing by DBF. If a was revealed before a', a obviously must have been placed into an empty bin, and as a' was shown, the algorithm must have compared the two items and realized that they do not fit together into the same bin. Note that due to Lemma 3.5.5 there could not have existed any Z-bins where the algorithm could have placed a' into.

If a' was revealed before a, the two items must have again been compared when a was shown. The bin in which a' was placed cannot have additionally contained a regular I_2-item as, per definition, no such item was yet revealed.

For a special item in a Y_2-bin, we know that it must have been placed there first, and the corresponding I_2-item was packed into that bin with Best-Fit at a later point. This means that the special item must have been larger than any

I_s-item that later ended up in a Y-bin in the final packing. So when a' is presented, it cannot open a new bin as it is fitted together with all special items in bins of type Y, where Best-Fit prefers to place it over opening a new bin. □

Lemma 3.5.7. *With $y' = y + y_2$ and if $y > 0$ and $y + y_2 \geq x_s$ in a packing by* DBF*, then $OPT(L) \geq n_1 + \frac{y'-x_s}{2}$.*

Proof. From Lemma 3.5.6 we know that I_s-items that were placed in Y- or Y_2-bins can only fit together in a bin with I_1-items that were placed in X_s-bins. Only special items contained in X_s-bins can fit together with other I_1-items, but there must exist $y' - x_s$ special items that cannot have an I_1-counterpart in the optimal packing. Thus, every solution must consist of at least $n_1 + \frac{y'-x_s}{2}$ bins. □

With the previous work, we can prove the theorem for the case when $y > 0$:

Lemma 3.5.8. *The* Delayed Best Fit *algorithm achieves a competitive ratio of $\frac{4}{3}$ for all instances where at most two items can be placed into the same bin and $y > 0$.*

Proof. Let L be an arbitrary input instance with n items.

We get $\mathrm{DBF}(L) = n_1 + y'$ from Lemma 3.5.5 and also have $y' + x_s \leq \frac{n}{3}$. We make a case distinction:

- $y' \leq x_s$: This implies $y' \leq \frac{1}{6}n$ as $y' + x_s \leq \frac{1}{3}n$. If now $n_1 < \frac{1}{2}n$ we are done as $\mathrm{DBF}(L) = n_1 + y' < \frac{1}{2}n + \frac{1}{6}n = \frac{2}{3}n$. Otherwise, we know that $OPT(L) \geq n_1$ and therefore get a competitive ratio of

$$\frac{n_1 + y'}{n_1} \leq \frac{\frac{1}{2}n + \frac{1}{6}n}{\frac{1}{2}n} = \frac{4}{3}.$$

- $y' > x_s$: We now use Lemma 3.5.7 to infer a competitive ratio of

$$r \leq \frac{n_1 + y'}{n_1 + \frac{y'-x_s}{2}} = 1 + \frac{x_s + y'}{2n_1 + y' - x_s} \leq 1 + \frac{\frac{1}{3}n}{2n_1 + y' - x_s}$$

  for the case.

  If now $n_1 \geq \frac{1}{2}n$ we get $2n_1 + y' - x_s \geq n$ and thus $r \leq \frac{4}{3}$.

  If $n_1 < \frac{1}{2}n$, the inequality $\mathrm{DBF}(L) > \frac{2}{3}n$ would imply competitive ratio of $\frac{4}{3}$ as well. If however $\mathrm{DBF}(L) \geq \frac{2}{3}n$ holds, then we have

$$y' > \frac{2}{3}n - n_1 \text{ and } x_s \leq \frac{1}{3}n - y' < n_1 - \frac{1}{3}n$$

and therefore

$$y' - x_s > \frac{2}{3}n - n_1 - (n_1 - \frac{1}{3}n) = n - 2n_1.$$

But this again means $2n_1 + y' - x_s > n$ and therefore $r < \frac{4}{3}$.

□

Finally, we show that the competitive ratio is also achieved in the case $y = 0$:

Lemma 3.5.9. *The* Delayed Best Fit *algorithm achieves a competitive ratio of $\frac{4}{3}$ for all instances where at most two items can be placed into the same bin and $y = 0$.*

Proof. Again, let L be the instance. It is easy to see that we get an optimal packing if the number of I_1 items $n_1 \geq \frac{2}{3}n$, i.e., no regular I_2-items were revealed. Otherwise, we know that more than $\frac{1}{3}n$ bins contain two items and there are fewer than $\frac{1}{3}n$ items left. This means we are done as we use fewer than $\frac{2}{3}n$ bins.

The desired competitive ratio follows as we prove that the packing by DBF uses at most $\frac{2}{3}n$ bins, as an optimal packing always requires $\frac{1}{2}n$ bins. □

Taking the last two lemmas together, we have seen that DBF achieves a competitive ratio of $\frac{4}{3}$ when at most two items fit into the same bin. We close this section with the remark, that the analysis of DBF is tight and no better competitive ratio can be achieved:

The lower bound of $\frac{4}{3}$ from Theorem 3.5.1 holds for all algorithms and uses at most two items per bin, even for arbitrary accurate estimates.

3.6 Graph Exploration

In this section, we focus on a variant in which the graph's structure is known in advance, but the actual edge weights are only revealed whenever an incident vertex is visited.

Definition 3.6.1 (The Graph Exploration with Edge Weight Estimates Problem)**.** Given a graph $G = (V, E)$, an upper and lower bound $u(e)$ and $\ell(e)$ for each edge weight $w(e)$, a vertex $s \in V$ as a start-vertex and a distinct vertex $t \in V$ as an end-vertex, the Graph Exploration with Edge Weight Estimates problem (GEEWE)) is the problem of finding the shortest walk visiting all vertices, starting and ending at the designated vertices. The actual weight $w(e) \in [\ell(e), u(e)]$ of an edge e is revealed as soon as the salesman visits an incident vertex.

While in the literature usually the start- and end-vertex are identical, we will assume that they are distinct from each other as in the travelling salesman path problem. For this sake, a *walk* will refer to an open walk with distinct start and end vertices throughout the section. This helps avoiding minor technical issues like dealing with edge weights that got revealed but connect two vertices that both need to be visited (one of them being the starting vertex). The results, however, translate also to the classical travelling salesman problem, as two adjacent vertices can be chosen as a start- and end vertex. The competitive ratio proven in the upper and lower bounds stays almost the same and only deviates by one potentially optimal chosen edge in the online algorithm's solution for all of the presented algorithms and lower bound constructions.

3.6.1 Unrestricted Graph Classes

The Graph Exploration with Edge Weight Estimates problem is a generalization of the Graph Exploration problem. Therefore, lower bounds for the Graph Exploration problem also apply for the GEEWE problem, when allowing the range between upper and lower bound to be sufficiently large.

On the other hand, every upper bound for the graph exploration problem is an upper bound for the GEEWE problem, no matter how small the ratio between the upper and lower bound of each edge is.

However, the lower bound constructions used for the graph exploration model also hide the structure of the graph, for example by presenting dead ends. As those structures can only be hidden in the GEEWE problem when the factor between the upper and lower bound for some edges is very large, we focus on instances where this factor is rather small.

In this section, we will see that no algorithm can achieve a better competitive ratio than the largest factor α between the upper and lower edge weight ($u(e) \leq \alpha\ell(e)$ for all edges e), with $\alpha \leq 2$. Furthermore, we show that this ratio can easily be achieved by a rather simple algorithm.

Theorem 3.6.1. *No algorithm for the* GEEWE *problem can achieve a better competitive ratio than the maximum factor α between an upper and lower bound of an announced edge weight when $\alpha \leq 2$.*

Proof. **Construction of the Instance:**

The following instance is constructed recursively, where a component of recursion depth i consists of k connected components of recursion depth $i-1$ for each $i \geq 1$ and a constant k. Figure 3.5 depicts an example for an instance of recursion depth 2 and $k = 3$.

A component of depth 0 consists of a path with $k+1$ vertices, where each edge is announced in the interval $[1, \alpha]$ and will get presented with the actual weight 1. We will refer to the first vertex of the path as the starting vertex s_0, and the last one as the end vertex t_0, as depicted in Figure 3.5.

Each component of recursion depth $i \geq 1$ is constructed with two distinguished vertices as a start and an end vertex s_i and t_i. k components of depth $i-1$ will be placed between s_i and t_i one after another using the following construction:

The start vertex s_i is connected to the vertices s_{i-1} and t_{i-1} of the first component of depth $i-1$ with an edge of announced weight $[k^i, \alpha k^i]$. Both will be revealed with the weight of k^i.

Two adjacent components are connected with four edges of announced weight $[k^i, \alpha k^i]$, connecting the start- and end vertex of one component to the start- and end vertex of the following component. The two edges connecting the start-vertex of the first visited component to the neighboring component will be revealed with weight k^i, and the two edges incident to the end vertex will be revealed with weight αk^i.

Finally, the start- and end vertices of the last component of depth $i-1$ are connected to the end vertex t_i with edges of announced weight $[k^i, \alpha k^i]$. Again, the edge incident to the start vertex s_{i-1} is revealed as k^i, and the edge attached to the end vertex t_{i-1} has actual size αk^i.

We will call the edges that are announced with weights in the interval $[k^i, \alpha k^i]$ edges of level i.

The used construction is symmetric, thus the start- and end vertices of each component are not distinguishable from each other based on the announcements before arriving at one of them. Thus, we assume w.l.o.g. that an algorithm visits the start before the end vertex at every component.

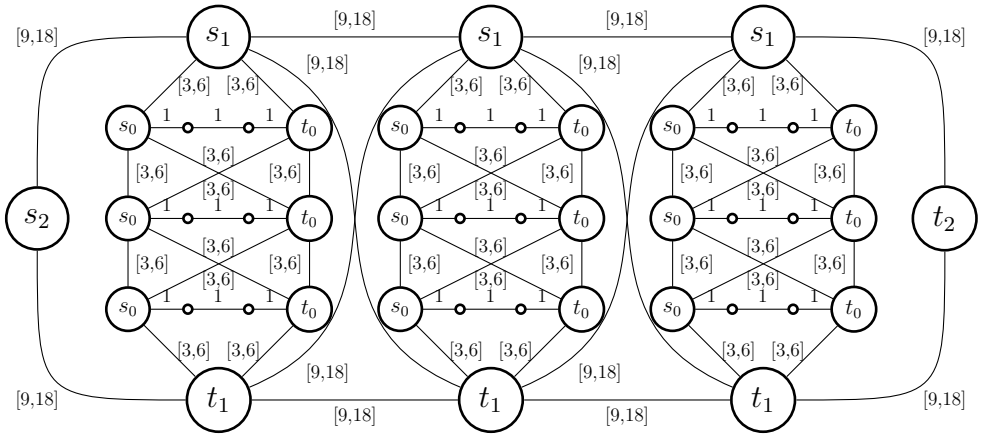


Figure 3.5: An example instance with the announcements of recursion depth 2 with $k = 3$ and $\alpha = 2$. An online algorithm needs to traverse from s_2 to t_2. The edges of level 0 are just given with the weight 1, as they will be presented with weight 1 anyway. Depending on the decisions of the algorithm, the other edges will either have a value of the upper or lower bound.

Figure 3.5 depicts an example for an instance of recursion depth 2.

As the start- and end vertices of each component are not distinguishable to the algorithm by the announcements, we can say without loss of generality that the algorithm reaches the start vertex of each component before reaching the end vertex.

How an Algorithm Traverses the Instance:
We claim that no online algorithm can traverse from the start to the end vertex of a component of recursion depth $i \geq 0$ visiting each vertex within with costs of less than $i(\alpha k + 1)k^i + k^{i+1}$. In contrast, an optimal offline solution does the traversal with total costs of $i(k + 1)k^i + k^{i+1}$.

For level $i = 0$, it is easy to see that both the online and offline algorithms can traverse through the path, resulting in total costs of k.

For each level $i \geq 1$, the algorithm needs to traverse through k components of recursion depth $i - 1$ and take at least $k + 1$ edges from the i-th level. The $k + 1$ connections between two consecutive components cost at least k^i each. Traversing each of the k components costs at least $(i - 1)(\alpha k + 1)k^{i-1} + k^i$ by induction. Thus, the algorithm must pay at least $k((i - 1)(\alpha k + 1)k^{i-1} + k^i) + (k + 1)k^i = (k+1)k^i+(i-1)(\alpha k+1)k^i+k^{i+1}$. This assumes the algorithm traverses between the components on the cheap edges with costs of k^i and traverses inside the components just the necessary amount.

To reach a new component, the algorithm has two options: leaving the previous component at the end vertex with costs of αk^i, or leaving the previous component at its start vertex with costs of k^i. In the former case, this traversal costs $(\alpha - 1)k^i$ more than in the previous calculation.

In the latter case, the algorithm needs to still visit all the vertices of the previous component. This includes its end vertex. This can be done in two ways:

- By traversing the component to the end vertex and back to the start vertex. In this case, it needs to pay for the traversal back at least the costs of an offline algorithm, so $(i-1)(k+1)k^{i-1}+k^i$ in addition to the previous calculation.
- If the algorithm does not visit all vertices from a component before leaving, the algorithm must visit the component at least twice. This causes additional costs of at least a cheap edge of level i with costs of k^i for the additional movement between components.

As we assume that $\alpha k^i \leq 2k^i$, those cases are at least as expensive for an algorithm as traversing in the intended way, visiting each component exactly once, and leaving at the respective end vertices.

Together, the algorithm must pay additional costs of at least $(\alpha-1)k^i$ for each component, resulting in total costs of at least $k(\alpha-1)k^i+(k+1)k^i+(i-1)(\alpha k+1)k^i+k^{i+1}=i(\alpha k+1)k^i+k^{i+1}$.

An optimal solution however, arrives at the end vertex of each component. Thus, it is able to leave each component of recursion depth i with the cheap k^i edge from its start vertex to the end vertex of the next component. Overall, a traversal of a depth i component costs $(k+1)k^i$ for the level i traversals, and k traversal of the depth $i-1$ components. This sums up to $k((i-1)(k+1)k^{i-1}+k^i)+(k+1)k^i = i(k+1)k^i+k^{i+1}$.

Overall, for a given k, no online algorithm can achieve a better competitive ratio than $\frac{i(\alpha k+1)k^i+k^{i+1}}{i(k+1)k^i+k^{i+1}} = \frac{i(\alpha k+1)+k}{i(k+1)+k}$ to traverse through a component of recursion depth i. It shows that there cannot be a constant competitive ratio of less than α when i and k are chosen sufficiently large. □

Remark that if $\alpha \geq 2$, we still get a lower bound of 2 by applying the proof of Theorem 3.6.1.

This competitive ratio can be achieved by a trivial algorithm that completely ignores the revealed edge weights:

Theorem 3.6.2. *An algorithm that precomputes the shortest possible walk covering all vertices from start to end vertex, based on the lower bounds of the edges and sticks to it, is at most α times worse than an optimal tour when α is the maximum factor between the announced bounds of an edge.*

Proof. As for all edges e, the actual weight $w(e)$ is bounded with $w(e) \leq u(e) \leq \alpha\ell(e)$, the precomputed walk can be at most α times more expensive than the sum of the lower bounds from its edges. The travelling salesman walk cannot be

cheaper than the sum of the lower bounds from the best tour based on the lower bounds. Thus, the precomputed walk is at most α times more expensive than an optimal walk. □

Note that the running time of this algorithm will probably be exponential in the worst case, as it needs to solve an NP-hard problem.

3.6.2 Restricted Graph Classes

As we have seen in the previous section, we cannot hope for an algorithm that beats the (from the online perspective) trivial precomputing algorithm in general. Therefore, it is interesting to see if there are algorithms that perform better on restricted graph classes, using the revealed edge weights during the algorithm's traversal throughout the graph.

A common restriction in this section is the assumption that the graph has uniform announced edge weights $[1, \alpha]$ for all edges. This allows us to achieve improved results e.g. for complete graphs, as without restrictions of the edge weights every graph structure can implicitly be constructed using edges of arbitrarily large announced weight.

3.6.3 Recalculating Algorithm

An intuitive candidate is the following algorithm, which recalculates the cheapest possible walk visiting all vertices, at each step taking the newly revealed edges into account.

Adaptive Exploration Algorithm 11 Adaptive Exploration Algorithm

Require: start vertex s, end vertex t
Agent A starts at vertex s
while Graph is not fully explored **do**
Calculate a worst case shortest walk from A to t, visiting all unvisited vertices.
Move A to the next vertex on the walk.

In every step, the algorithm can calculate an upper bound for finishing the walk by using the revealed edge weights together with the upper bound of each weight for the unrevealed edge weights. Calculating the walk is computationally expensive, as the algorithm needs to recalculate the optimal for each visited vertex.

This algorithm can be proven to be optimal for graph classes like complete graphs or complete balanced bipartite graphs.

Theorem 3.6.3. *Given a graph where each path can be extended to a Hamiltonian path from the start to a distinct end vertex and uniform announced edge weights* $[1, \alpha]$ *for* $\alpha < 2$*, the adaptive exploration algorithm reaches a competitive ratio of at most* $\frac{\alpha+1}{2}$*.*

We start with an easy observation, connecting the adaptive exploration algorithm with the nearest neighbor method.

Lemma 3.6.1. *In the setting of Theorem 3.6.3, the adaptive exploration algorithm takes the cheapest outgoing edge towards an unvisited vertex (which is not the end vertex), until it visits the end vertex in its final step.*

Note that in general the nearest neighbor method [85] is not identical to the adaptive exploration algorithm, as the nearest neighbor method does not take the global graph structure into account. For example, as the adaptive exploration algorithm computes a walk based on the worst-case announcements, it will achieve a competitive ratio of at most α, while the nearest neighbor method might only achieve a logarithmic approximation.

Proof of Lemma 3.6.1. At any point, there is a Hamiltonian extension to the current solution. Therefore, at any point, the algorithm has an option with expected worst-case costs of the next edge plus the number of remaining vertices minus one (of which the weight to reach them is unknown between 1 and α). This beats every option that involves visiting vertices twice, so no vertex will be visited twice by the adaptive exploration algorithm.

As all outgoing edges towards unvisited vertices (that are not the end vertex) are part of Hamiltonian extensions whose expected worst-case costs to the algorithm only deviate by its first edge, the decision of which vertex is visited next is determined by the cheapest outgoing edge. □

Given this observed behaviour of the adaptive exploration algorithm, we can prove 3.6.3:

Proof of 3.6.3. First, we fix an optimal walk for the analysis.

For the sake of notation, we enumerate the vertices from 1 to n in the same order as they are visited by the adaptive exploration algorithm. To prove the claim, assume that after k vertices the algorithm has paid the edge costs of $w_1, \ldots, w_{k-1}$. We will see that the optimal tour is at least twice as expensive as the sum of the $n/2$ cheapest edges of the algorithm:

Since the adaptive exploration algorithm takes the cheapest possible outgoing edge towards an unvisited vertex, we can assume that every edge leaving a vertex x towards a vertex y with $y > x$ costs at least w_x. Edges to already visited vertices

can be cheaper. As the optimal walk is a Hamiltonian one, only two edges incident to each vertex can be used in an optimal solution.

When comparing a fixed optimal solution to the walk of the adaptive exploration algorithm, there are two types of edges this optimal solution consists of: Edges from a vertex x to a vertex y with $x < y$, and edges from x to y with $x > y$.

For each edge with $x < y$, by Lemma 3.6.1 we know that the algorithm has paid at most as much as the optimal solution when leaving x.

For each edge with $x > y$, it follows by Lemma 3.6.1, that the algorithm has paid at most as much when leaving y as the optimal solution paid for leaving x.

Sadly, both cannot be combined as edges might have been counted in both cases. But as the optimal solution needs $n - 1$ edges, one of the two types of edges must occur at least $\frac{n-1}{2}$ times. Therefore, half the edges of the algorithm's solution cost as much as half the edges of the optimal solution. Let's say those edge weights total K.

The costs of an optimal solution are now at least $\frac{n-1}{2} + K$, while the algorithm's solution is at most $\frac{\alpha(n-1)}{2} + K$. The ratio between those expressions is the largest possible when K is small. However, K must be at least $\frac{n-1}{2}$. Therefore, the competitive ratio of the adaptive exploration algorithm is at most $\frac{\alpha+1}{2}$. □

Even if the restriction for the graph classes is rather strict, there are some important graph classes where 3.6.3 applies.

Corollary 3.6.1. *For every $\alpha < 2$ and every n, the adaptive exploration algorithm achieves a competitive ratio of $\frac{\alpha+1}{2}$ on the complete graphs K_n with uniform announced edge weight and distinct start- and end vertices.*

Corollary 3.6.2. *For every $\alpha < 2$ and every n, the adaptive exploration algorithm achieves a competitive ratio of $\frac{\alpha+1}{2}$ on the complete bipartite graphs $K_{n,n}$ with uniform announced edge weights when start- and end vertices are on different sides, and the complete bipartite graph $K_{n+1,n}$ with distinct start- and end vertices on the bigger side.*

Both corollaries immediately follow from 3.6.3.

Indeed, the performance of the adaptive exploration algorithm is optimal on those graph classes.

Theorem 3.6.4. *No algorithm can achieve a better competitive ratio than $\frac{\alpha+1}{2}$ even on complete graphs with uniform announced edge weights $[1, \alpha]$ and $\alpha \leq 2$.*

Proof. Let us consider any algorithm $\mathcal{A}$ on the complete graph K_{2k}. For the first k rounds, we reveal that every edge adjacent to the current vertex has a weight of 1. We call the set of vertices visited in this phase A. Then, every newly revealed

edge has a weight α (the edges going to vertices in A are still of weight 1). Let $B = V \setminus A$ denote the set of vertices visited in this second phase.

Here, we use weights 1 and α, so it never makes sense for an algorithm to go back to an already visited edge, because the triangle inequality is satisfied. Indeed, imagine that $\mathcal{A}$ follows the path $v_1 \cdots v_k$ (for $k \geq 3$) at some point but had already visited $v_2 \cdots v_{k-1}$ when it was at v_1. Then, the cost of going from v_1 to v_k directly is at most α, whereas that of the path is at least $(k-1) \geq 2 \geq \alpha$. Thus, we can assume that $|A| = |B| = k$.

The cost of the path computed by the algorithm is $(1+\alpha)k$, whereas there exists an optimal path of cost $2k$: with $A = \{a_1, \cdots, a_k\}, B = \{b_1, \cdots, b_k\}$, the path $a_1 b_1 a_2 b_2 \cdots a_k b_k a_1$ only takes edges adjacent to vertices in A, which are thus of weight 1. Thus, the algorithm cannot do better than a competitive ratio of $\frac{1+\alpha}{2}$. □

Remark that if $\alpha > 2$, we still get a lower bound of 3/2 by applying the construction used when proving 3.6.4.

A lower bound of $\frac{\alpha+1}{2}$ for complete bipartite graphs $K_{n,n}$, matching 3.6.2, can be constructed analogously. Note that the restriction for distinct start- and end vertices is only necessary for achieving a strict bound of $\frac{1+\alpha}{2}$, in case of identical start- and end vertex the ratio can decrease by $\frac{\alpha-1}{n}$ as the final edge leading towards the start/end vertex is revealed as of the beginning, so the algorithm cannot be tricked into taking the last edge as an expensive edge.

However, the adaptive exploration algorithm cannot outperform the trivial precomputing algorithm even for uniform announced edge weights and $\alpha < 2$, even on grids.

Theorem 3.6.5. *The Adaptive Exploration Algorithm does not achieve a better competitive ratio than α for graphs with announced weight $[1, \alpha]$ for all edges and $\alpha < 2$ in general.*

Proof. We can construct a grid as an example:

Assume the algorithm starts at the marked starting vertex s in the grid as depicted in Figure 3.6. As there exists a Hamiltonian path between the start vertex and the end vertex t, at each point the algorithm prefers staying on the Hamiltonian path instead of visiting a vertex twice to take a cheaper edge. This is, as taking an edge with weight α is cheaper than taking two edges even with weight 1. Also, traversing between two connected unvisited vertices will cost either the unrevealed edge of at most α, or in the best case, two revealed edges of weight 1 as well. Therefore, at every step, the algorithm will take the cheapest outgoing edge that is part of a Hamiltonian path.

In every step of the algorithm (broad path in Figure 3.6), there is either just one unvisited neighbor that needs to be visited to stay on a Hamiltonian path

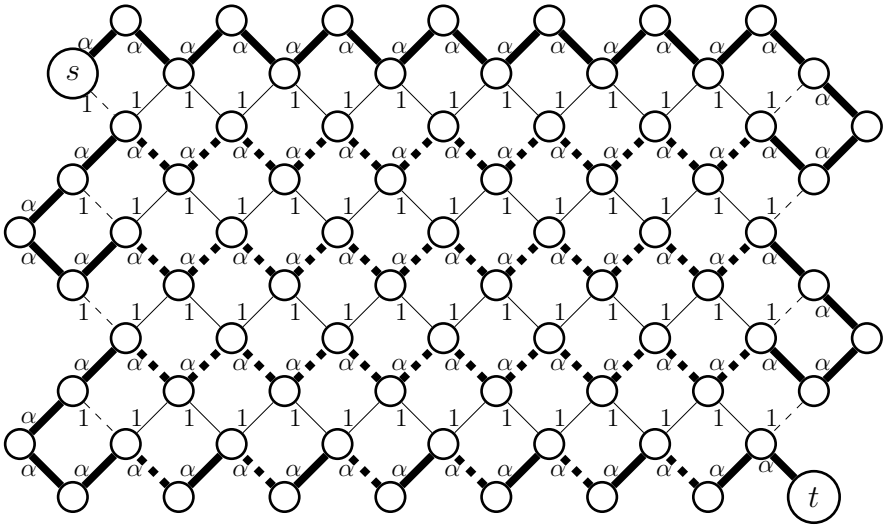


Figure 3.6: Every edge is announced with $[1, \alpha]$ and revealed with the labeled weight. The adaptive exploration algorithm will traverse on the broad edges, and will not take the thin ones (so every second layer). An alternative solution would take the continuous ones (and takes some edges twice in the bottom layer), and ignore the dashed ones. The adaptive exploration algorithm therefore walks on a Hamiltonian path, where each edge costs α. The alternative solution only pays 1 for the edges in the center but needs more edges.

or one neighbor that is adjacent to just one unvisited vertex. In both cases, this vertex needs to be visited to stay on a Hamiltonian path. Therefore, the adaptive exploration algorithm will traverse the grid in the predetermined path, even if all those edges are expensive with weight α. Note that this also seems to be the strategy that promises the cheapest path through the graph.

On the other hand, an alternative algorithm may ignore the fact that a Hamiltonian path seems cheaper for the moment. It also needs to visit the upper and lower row of the grid, thus taking edges with costs of α twice for each vertex in those rows. However, it can take the other edges for the middle part of the grid, thus paying only 1 for each edge. This comes at the cost of taking additional edges for another traversal from the right to the left (depicted by the continuous line).

Overall, for a quadratic grid of n vertices, the adaptive exploration algorithm pays $(n-1)\alpha$, while the depicted alternative solution only pays $6\sqrt{n}\alpha+(\sqrt{n}-2)\sqrt{n}$. Therefore, the competitive ratio converges to α, as n gets large. □

The fragility of the adaptive exploration algorithm on instances like a lattice is not surprising, as it allows the adversary to develop traps. Even though it follows a walk that - based on his current beliefs - seems like an algorithm's best hope, it does not value revealing edge weights. Thus, the adversary can decide at a later point whether an edge must be cheap or expensive, depending on the more

advanced tour of the algorithm. While this does not show that no $\frac{1+\alpha}{2}$-competitive algorithm can exist for uniform edge weights, it does show that focusing on the algorithm's gain alone is not promising. Thus, an algorithm must try to explore the graph in a way that forces an adversary to influence the optimal solution as well. Otherwise, it will not be able to beat the trivial competitive ratio of α even on this simple graph class.

3.6.4 Outlook on other graph classes

Many lower-bound constructions for the classical graph exploration problem make use of dead ends or the option to either present edges or not, depending on the algorithm's actions. This is not possible in the graph exploration problem with edge weight estimates. Thus, many lower bound constructions of the graph exploration problem will not work here, even for a large factor of α.

For example, Miyazaki et al. [81] proved a matching upper and lower bound of $\frac{1+\sqrt{3}}{2}$ for the graph exploration problem on a cycle graph. For a sufficiently large α, the lower bound construction will also work out in our model with edge weight estimates, as there is no structure to hide in this graph class.

For most graph classes, like tadpole graphs (a cycle with one attached path), the question of whether a vertex is on the cycle or on the path is hidden by the classical graph exploration model. Miyazaki et al. [81] and Brandt et al. [36] showed that even on unweighted tadpole graphs there cannot be an algorithm for the graph exploration problem yielding a better competitive ratio than 2. However, It is clear which edges belong to the path and thus need to be traversed exactly twice by every reasonable algorithm in the online graph exploration problem with edge weight estimates. It follows, that every edge of the path only decreases the competitive ratio. Thus, the best possible lower bound for a given α on tadpole graphs is the same construction as on a cycle, and will not exceed a competitive ratio of $\frac{1+\sqrt{3}}{2}$.

3.7 Comments and Open Problems

While the setting of online algorithms with estimates as studied in this chapter is new, it is strongly related to the setting of online algorithms with predictions (see 1.3.2). In both settings, the blindness of an online algorithm is partly relaxed by hints about the upcoming items, even though in both cases the hints are not necessarily precise.

As the last years have shown, there is a huge interest in the research community regarding everything that incorporates some kind of predictions into algorithms. This can partly be explained by the natural setting (e.g., it is realistic to assume that one does not start completely blind and without any clue when making the first decisions) and also by the intrinsic theoretical interest of the problem. One of the main reasons however, is the rising popularity of machine learning tools in recent times: Such a tool might predict what is coming based on the learned data (so essentially the same as what natural people can do), but the resulting prediction might not be completely correct. An algorithm that just follows the prediction might produce an arbitrarily bad solution for many problems. This can be seen throughout the literature, but is also very believable by simple examples (e.g. if you pack a small item into the knapsack because a large item is promised to fit together with the small one, you might end up with an almost empty knapsack in case the large item turns out to be larger and does not fit together with the small one).

Therefore, it is natural to think about robust algorithms that incorporate the prediction but are resilient against deviations. This also allows us to give worst-case guarantees (in our case for the competitive ratio), which algorithms without the robustification might not be able to provide.

Based on this use case, one of the main questions is which kind of prediction is realistic. In the literature one common variant for prediction is to predict an optimal solution (see e.g. [6, 51], where a graph appears vertex-wise and the predictor tells you about the color of each vertex in an optimal solution), which arguably might be not as realistic as a setting in which a rough guess about the instance is given (e.g. a structure of the graph, or a predicted degree of a vertex,...). As the question about a realistic setting is rather vague, one cannot hope for a final answer here.

The estimate-setting we suggest in this thesis focuses on the setting in which the predictor gives their estimate about the instance in the beginning. Whether this is realistic for a machine-learned approach naturally depends on the data with which the machine was trained. As the setting can be motivated also by real-world scenarios without machine learning, one can argue that a similar prediction is also realistic in machine learning scenarios (as both might utilize the experience of previous instances).

As this thesis only studies variants of Online Knapsack, Online Bin Packing, and Graph Exploration, and, to the best of our knowledge, related results are only known for Scheduling variants, it is reasonable to ask how different problems behave in the estimate-setting. The list of problems which have been studied in the prediction setting [1] provide almost endless motivation about which questions and problems are also interesting. Additional work is also sensible for the problems studied so far, as e.g. the Online General Knapsack problem is still unsolved, and the bounds for Online Bin Packing and the uniform case of Graph Exploration are not tight yet.

One variant might be interesting among all problem settings, however: What happens, if the predictor cannot provide an exact estimate about the accuracy? It is natural to study a setting in which the accuracy is known (e.g. if an algorithm is given a list of items that should be packed, but the values are rounded due to a cutoff at two digits after the point), also situations are thinkable in which this is not the case. Can we prove e.g., that there is an algorithm for Online Simple Knapsack with estimates but without given accuracy, which achieves a competitive ratio that is just worse by a factor of x then an algorithm that has the accuracy given?

Another natural variation is to combine the estimate setting with other aspects that make online problems semi-online. Take, for example, a combination of estimates and (truthfully answered) advice in a packing setting: Assume an algorithm gets told by the predictor that two items will appear roughly of size 0.5. Now the algorithm might be interested in finding out whether these two items will fit together or not. This might be possible by additional research. To find out whether this effort helps, a study of the problem with both the estimate and the advice setting might be interesting.

Chapter 4

Conclusion

In this thesis, we have primarily studied two different approaches that can be used to make online problems semi-online, thereby potentially making them more realistic.

As mentioned earlier, there are many real-life problems that cannot be modeled as either a classical *offline* or an *online* problem. Here, models and algorithms that ly in between are a possible solution.

In many of those applications however, it is arguably not just a single modification that makes an online algorithm a semi-online one and then models the situation perfectly. Often, one has to solve an (intrinsic) online problem, but it contains several aspects of an offline or semi-online problem.

Take, for example, the problem of booking a hotel, rental car, train, or flight; or buying financial products. Those are essentially online problems, as one has to assume that the price changes over time. Thus, one might ask about when those products are cheapest, to determine when to buy them.

As motivated in the introduction of section 2, often one can buy some kind of insurance, which allows one to decide later on. For bookings, those might come as a more expensive option that allows cancellation (which essentially includes the option to redo the booking when it is cheaper). So studying those problems within the *reservation* setting seems sensible.

On the other hand, one also is typically not completely blind when it comes to the price changes: You might know that your train is usually cheaper the earlier it is booked, so here you have a (not necessarily correct) hint on when to buy it. This would motivate studying within a *prediction* setting.

Or, you might be aware of the price structure of your local train company, and thus know the interval in which the price will move. This essentially would represent the additional information one can consider when studying online algorithms with *estimates*.

All these modifications might help to model an online problem better. In

reality, one often has several options, that combined, may model the real world as accurate as possible.

Arguably, those modifications are by far not all that are sensible in such an online setting. Thus, not only are infinitely many options of making online problem semi-online interesting, but also uncountably many combinations may be useful to model the real world.

Bibliography

[1] Algorithms with predictions. `https://algorithms-with-predictions.github.io/`. Accessed: 2025-10-28.

[2] Susanne Albers and Leon Ladewig. New results for the k-secretary problem. In *30th International Symposium on Algorithms and Computation, ISAAC*, volume 149 of *LIPIcs*, pages 18:1–18:19, 2019.

[3] Spyros Angelopoulos, Christoph Dürr, Shendan Jin, Shahin Kamali, and Marc Renault. Online Computation with Untrusted Advice. In *11th Innovations in Theoretical Computer Science Conference (ITCS 2020)*, volume 151, pages 52:1–52:15, 2020.

[4] Spyros Angelopoulos, Christoph Dürr, Shahin Kamali, Marc P Renault, and Adi Rosén. Online bin packing with advice of small size. *Theory of Computing Systems*, 62:2006–2034, 2018.

[5] Spyros Angelopoulos, Shahin Kamali, and Kimia Shadkami. Online bin packing with predictions. *Journal of Artificial Intelligence Research 78*, 2023.

[6] Antonios Antoniadis, Hajo Broersma, and Yang Meng. Incorporating predictions in online graph coloring algorithms. *Discrete Applied Mathematics*, 379:434–445, 2026.

[7] Yossi Azar, Stefano Leonardi, and Noam Touitou. Flow time scheduling with uncertain processing time. In *Proceedings of the 53rd Annual ACM SIGACT Symposium on Theory of Computing*, STOC 2021. Association for Computing Machinery, 2021.

[8] Yossi Azar, Stefano Leonardi, and Noam Touitou. Distortion-oblivious algorithms for minimizing flow time. In *Proceedings of the 2022 ACM-SIAM Symposium on Discrete Algorithms, SODA 2022*. SIAM, 2022.

[9] Yossi Azar, Eldad Peretz, and Noam Touitou. Distortion-Oblivious Algorithms for Scheduling on Multiple Machines. In *33rd International Symposium*

on Algorithms and Computation (ISAAC 2022). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2022.

[10] Moshe Babaioff, Nicole Immorlica, David Kempe, and Robert Kleinberg. A knapsack secretary problem with applications. In *Proc. of 10th International Workshop on Approximation, Randomization, and Combinatorial Optimization (APPROX)*, number 4627 in Lecture Notes in Computer Science, pages 16–28. Springer, 2007.

[11] Moshe Babaioff, Nicole Immorlica, David Kempe, and Robert Kleinberg. Online auctions and generalized secretary problems. *SIGecom Exch.*, 7(2), 2008.

[12] Luitpold Babel, Bo Chen, Hans Kellerer, and Vladimir Kotov. Algorithms for on-line bin-packing problems with cardinality constraints. *Discrete Applied Mathematics*, 143(1-3):238–251, 2004.

[13] Jakub Balabán, Matthias Gehnen, Henri Lotze, Finn Seesemann, and Moritz Stocker. Online knapsack problems with estimates. In *50th International Symposium on Mathematical Foundations of Computer Science, MFCS 2025*, volume 345 of *LIPIcs*, pages 12:1–12:19. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2025.

[14] Júlia Baligács. A lower bound of 4 for online graph exploration. In *17th Latin American Theoretical Informatics Symposium (LATIN 2026)*, Lecture Notes in Computer Science, 2026.

[15] Júlia Baligács, Yann Disser, Irene Heinrich, and Pascal Schweitzer. Exploration of graphs with excluded minors. In *31st Annual European Symposium on Algorithms, ESA 2023*. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2023.

[16] János Balogh, József Békési, György Dósa, Leah Epstein, and Asaf Levin. A new and improved algorithm for online bin packing. *26th Annual European Symposium on Algorithms (ESA 2018)*, 2018.

[17] János Balogh, József Békési, György Dósa, Leah Epstein, and Asaf Levin. Online bin packing with cardinality constraints resolved. *Journal of Computer and System Sciences*, 112:34–49, 2020.

[18] János Balogh, József Békési, György Dósa, Leah Epstein, and Asaf Levin. A new lower bound for classic online bin packing. *Algorithmica*, 83:2047–2062, 2021.

[19] János Balogh, József Békési, and Gábor Galambos. New lower bounds for certain classes of bin packing algorithms. *Theoretical Computer Science*, 440:1–13, 2012.

[20] Reuven Bar-Yehuda, Dan Geiger, Joseph Naor, and Ron M. Roth. Approximation algorithms for the feedback vertex set problem with applications to constraint satisfaction and bayesian inference. *SIAM J. Comput.*, 27(4):942–959, 1998.

[21] Ann Becker and Dan Geiger. Optimization of pearl's method of conditioning and greedy-like approximation algorithms for the vertex feedback set problem. *Artif. Intell.*, 83(1):167–188, 1996.

[22] Alexander Birx, Yann Disser, Alexander V. Hopp, and Christina Karousatou. An improved lower bound for competitive graph exploration. *Theoretical Computer Science*, 868, 2021.

[23] Hans-Joachim Böckenhauer, Elisabet Burjons, Juraj Hromkovic, Henri Lotze, and Peter Rossmanith. Online simple knapsack with reservation costs. In *38th International Symposium on Theoretical Aspects of Computer Science, STACS 2021*, volume 187 of *LIPIcs*, pages 16:1–16:18, 2021.

[24] Hans-Joachim Böckenhauer, Fabian Frei, and Peter Rossmanith. Removable Online Knapsack and Advice. In *41st International Symposium on Theoretical Aspects of Computer Science (STACS 2024)*, volume 289 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 18:1–18:17, Dagstuhl, Germany, 2024. Schloss Dagstuhl – Leibniz-Zentrum für Informatik.

[25] Hans-Joachim Böckenhauer, Janosch Fuchs, and Walter Unger. Exploring sparse graphs with advice. *Information and Computation*, 289(Part A), 2022.

[26] Hans-Joachim Böckenhauer, Matthias Gehnen, Juraj Hromkovic, Ralf Klasing, Dennis Komm, Henri Lotze, Daniel Mock, Peter Rossmanith, and Moritz Stocker. Online Unbounded Knapsack. In *Theory of Computing Systems*, volume 69, 2025.

[27] Hans-Joachim Böckenhauer, Juraj Hromkovič, Dennis Komm, Peter Rossmanith, and Moritz Stocker. A survey of online knapsack problems. *Discrete Applied Mathematics*, 378:492–507, 2026.

[28] Hans-Joachim Böckenhauer, Ralf Klasing, Tobias Mömke, Peter Rossmanith, Moritz Stocker, and David Wehner. Online knapsack with removal and recourse. In *Combinatorial Algorithms*, 2023.

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

[30] Allan Borodin and Ran El-Yaniv. *Online computation and competitive analysis*. Cambridge University Press, 1998.

[31] Joan Boyar, Stephan J. Eidenbenz, Lene M. Favrholdt, Michal Kotrbcík, and Kim S. Larsen. Online dominating set. In Rasmus Pagh, editor, *15th Scandinavian Symposium and Workshops on Algorithm Theory, SWAT 2016*, volume 53 of *LIPIcs*, pages 21:1–21:15. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2016.

[32] Joan Boyar, Lene M. Favrholdt, Michal Kotrbcík, and Kim S. Larsen. Relaxing the irrevocability requirement for online graph algorithms. In *Algorithms and Data Structures - 15th International Symposium, WADS 2017*, volume 10389 of *Lecture Notes in Computer Science*, pages 217–228. Springer, 2017.

[33] Joan Boyar, Lene M Favrholdt, Christian Kudahl, Kim S Larsen, and Jesper W Mikkelsen. Online algorithms with advice: a survey. *ACM Computing Surveys (CSUR)*, 50(2):1–34, 2017.

[34] Joan Boyar, Lene M. Favrholdt, and Kim S. Larsen. Online unit profit knapsack with untrusted predictions. In Artur Czumaj and Qin Xin, editors, *18th Scandinavian Symposium and Workshops on Algorithm Theory, SWAT 2022*, volume 227 of *LIPIcs*, pages 20:1–20:17. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2022.

[35] Joan Boyar, Shahin Kamali, Kim S Larsen, and Alejandro López-Ortiz. Online bin packing with advice. *Algorithmica*, 74:507–527, 2016.

[36] Sebastian Brandt, Klaus-Tycho Foerster, Jonathan Maurer, and Roger Wattenhofer. Online graph exploration on a restricted graph class: Optimal solutions for tadpole graphs. *Theoretical Computer Science*, 839, 2020.

[37] Donna J Brown. A lower bound for on-line one-dimensional bin packing algorithms. Technical report, Technical Report R-864, Coordinated Sci. Lab., Urbana, Illinois, 1979.

[38] Niv Buchbinder, Kamal Jain, and Mohit Singh. Secretary problems via linear programming. *Math. Oper. Res.*, 39(1):190–206, 2014.

[39] Niv Buchbinder and Joseph Naor. Online primal-dual algorithms for covering and packing. *Math. Oper. Res.*, 34(2):270–286, 2009.

[40] Elisabet Burjons, Fabian Frei, Matthias Gehnen, Henri Lotze, Daniel Mock, and Peter Rossmanith. Delaying decisions and reservation costs. In *29th International Computing and Combinatorics Conference, COCOON 2023*, volume 29 of *Lecture notes in Computer Science*, 2023.

[41] Elisabet Burjons and Matthias Gehnen. Online general knapsack with reservation costs. In *Approximation and Online Algorithms - 23rd International Workshop, WAOA 2025*, volume 16077 of *Lecture Notes in Computer Science*, pages 33–47. Springer, 2025.

[42] Elisabet Burjons, Matthias Gehnen, Henri Lotze, Daniel Mock, and Peter Rossmanith. The secretary problem with reservation costs. In *Computing and Combinatorics - 27th International Conference, COCOON 2021*, volume 13025 of *Lecture Notes in Computer Science*, pages 553–564. Springer, 2021.

[43] Elisabet Burjons, Matthias Gehnen, Henri Lotze, Daniel Mock, and Peter Rossmanith. The Online Simple Knapsack Problem with Reservation and Removability. In *48th International Symposium on Mathematical Foundations of Computer Science (MFCS 2023)*, volume 272 of *Leibniz International Proceedings in Informatics (LIPIcs)*, Dagstuhl, Germany, 2023. Schloss Dagstuhl – Leibniz-Zentrum für Informatik.

[44] T.-H. Hubert Chan, Fei Chen, and Shaofeng H.-C. Jiang. Revealing optimal thresholds for generalized secretary problem via continuous LP: Impacts on online k-item auction and bipartite k-matching with random arrival order. In *Proceedings of the Twenty-Sixth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2015*, pages 1169–1188. SIAM, 2015.

[45] Li-Hsuan Chen, Ling-Ju Hung, Henri Lotze, and Peter Rossmanith. Online node- and edge-deletion problems with advice. *Algorithmica*, 83(9):2719–2753, 2021.

[46] Edward G Coffman, János Csirik, Gábor Galambos, Silvano Martello, and Daniele Vigo. Bin packing approximation algorithms: Survey and classification. In *Handbook of combinatorial optimization*, pages 455–531. Springer, 2013.

[47] Marc Demange and Vangelis Th. Paschos. On-line vertex-covering. *Theoretical Computer Science*, 332(1):83–108, 2005.

[48] Stefan Dobrev, Rastislav Královic, and Euripides Markou. Online graph exploration with advice. In *Structural Information and Communication Complexity - 19th International Colloquium, SIROCCO 2012*. Springer, 2012.

[49] Eugene B. Dynkin. The optimum choice of the instant for stopping a markov process. *Soviet Mathematics*, 4:627–629, 1963.

[50] Franziska Eberle, Alexander Lindermayr, Nicole Megow, Lukas Nölke, and Jens Schlöter. Robustification of Online Graph Exploration Methods. *Proceedings of the AAAI Conference on Artificial Intelligence*, 36(9), 2022.

[51] Fabian Frei, Matthias Gehnen, Dennis Komm, Rastislav Královič, Richard Královič, Peter Rossmanith, and Moritz Stocker. Tree coloring with predictions. *Discrete Applied Mathematics*, 380:386–394, 2026.

[52] Hiroshi Fujiwara and Koji Kobayashi. Improved lower bounds for the online bin packing problem with cardinality constraints. *Journal of Combinatorial Optimization*, 29(1):67–87, 2015.

[53] Matthias Gehnen, Kübra Güven, Valentin Hächler, Dennis Komm, and Richard Královič. Improved Results for Knapsack with Removal. In *51st International Symposium on Mathematical Foundations of Computer Science (MFCS 2026)*, Leibniz International Proceedings in Informatics (LIPIcs), 2026.

[54] Matthias Gehnen, Ralf Klasing, and Émile Naquin. Graph exploration with edge weight estimates. In *17th Latin American Theoretical Informatics Symposium (LATIN 2026)*, Lecture Notes in Computer Science, 2026.

[55] Matthias Gehnen, Henri Lotze, and Peter Rossmanith. Online Simple Knapsack with Bounded Predictions. In *41st International Symposium on Theoretical Aspects of Computer Science (STACS 2024)*, LIPIcs, pages 37:1–37:20, 2024.

[56] Matthias Gehnen and Julius Stannat. Endgames in fog of war chess. In *13th International Conference on Fun with Algorithms (FUN 2026)*, volume 366 of *LIPIcs*, pages 21:1–21:20, 2026.

[57] Matthias Gehnen and Moritz Stocker. Stealing from the dragon's hoard: Online unbounded knapsack with removal. In *43rd International Symposium on Theoretical Aspects of Computer Science, STACS 2026*, volume 364 of *LIPIcs*, pages 43:1–43:17. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2026.

[58] Matthias Gehnen and Eberhard Triesch. Transitive Avoidance Games on Boards of Odd Size. In *The Electronic Journal of Combinatorics*, volume 28, Isse 4, 2021.

[59] Matthias Gehnen and Andreas Usdenski. Online bin packing with item size estimates, 2025.

[60] Matthias Gehnen and Luca Venier. Tetris Is Not Competitive. In *12th International Conference on Fun with Algorithms (FUN 2024)*, volume 291, pages 16:1–16:16, 2024.

[61] Xin Han, Yasushi Kawase, and Kazuhisa Makino. Online unweighted knapsack problem with removal cost. *Algorithmica*, 70(1):76–91, 2014.

[62] Xin Han, Yasushi Kawase, Kazuhisa Makino, and Haruki Yokomaku. Online Knapsack Problems with a Resource Buffer. In *30th International Symposium on Algorithms and Computation (ISAAC 2019)*, LIPIcs, pages 28:1–28:14, 2019.

[63] Xin Han and Kazuhisa Makino. Online removable knapsack with limited cuts. *Theoretical Computer Science*, 411(44):3956–3964, 2010.

[64] Sungjin Im, Ravi Kumar, Mahshid Montazer Qaem, and Manish Purohit. Online knapsack with frequency predictions. In *Advances in Neural Information Processing Systems 34: Annual Conference on Neural Information Processing Systems 2021, NeurIPS 2021, December 6-14, 2021, virtual*, pages 2733–2743, 2021.

[65] Kazuo Iwama and Shiro Taketomi. Removable online knapsack problems. In *Automata, Languages and Programming, 29th International Colloquium, ICALP 2002*, volume 2380 of *Lecture Notes in Computer Science*, pages 293–305. Springer, 2002.

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

[67] David S Johnson. *Near-optimal bin packing algorithms*. PhD thesis, Massachusetts Institute of Technology, 1973.

[68] David S. Johnson, Alan Demers, Jeffrey D. Ullman, Michael R Garey, and Ronald L. Graham. Worst-case performance bounds for simple one-dimensional packing algorithms. *SIAM Journal on computing*, 3(4):299–325, 1974.

[69] Bala Kalyanasundaram and Kirk R. Pruhs. Constructing competitive tours from local information. *Theoretical Computer Science*, 130(1), 1994.

[70] Robert D. Kleinberg. A multiple-choice secretary algorithm with applications to online auctions. In *Proceedings of the Sixteenth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2005*, pages 630–631. SIAM, 2005.

[71] Dennis Komm. *An Introduction to Online Computation - Determinism, Randomization, Advice*. Texts in Theoretical Computer Science. An EATCS Series. Springer, 2016.

[72] Dennis Komm. *An Introduction to Online Computation - Determinism, Randomization, Advice*. Texts in Theoretical Computer Science. An EATCS Series. Springer, 2016.

[73] Chan C Lee and Der-Tsai Lee. A simple on-line bin-packing algorithm. *Journal of the ACM (JACM)*, 32(3):562–572, 1985.

[74] Frank M Liang. A lower bound for on-line bin packing. *Information processing letters*, 10(2):76–79, 1980.

[75] D. V. Lindley. Dynamic programming and decision theory. *Journal of the Royal Statistical Society. Series C (Applied Statistics)*, 10(1):39–51, 1961.

[76] Henri Lotze. *Going Offline - delays, reservations and predictions in online computation*. PhD-Thesis. RWTH Aachen, 2023.

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

[78] Nicole Megow, Kurt Mehlhorn, and Pascal Schweitzer. Online graph exploration: New results on old and new algorithms. *Theoretical Computer Science*, 463, 2012.

[79] Jesper W Mikkelsen. Randomization can be as helpful as a glimpse of the future in online computation. *43rd International Colloquium on Automata, Languages, and Programming (ICALP 2016)*, 2016.

[80] Michael Mitzenmacher and Sergei Vassilvitskii. Algorithms with predictions. *Communications of the ACM*, 65(7):33–35, 2022.

[81] S. Miyazaki, Naoyuki Morimoto, and Yasuo Okabe. The online graph exploration problem on restricted graphs. *IEICE Trans. Inf. Syst.*, 92-D, 2009.

[82] Manish Purohit, Zoya Svitkina, and Ravi Kumar. Improving online algorithms via ML predictions. In *Advances in Neural Information Processing Systems*. Curran Associates, Inc., 2018.

[83] Prakash Ramanan, Donna J Brown, Chung-Chieh Lee, and Der-Tsai Lee. On-line bin packing in linear time. *Journal of Algorithms*, 10(3):305–326, 1989.

[84] Marc P Renault, Adi Rosén, and Rob van Stee. Online algorithms with advice for bin packing and scheduling problems. *Theoretical Computer Science*, 600:155–170, 2015.

[85] Daniel J. Rosenkrantz, Richard E. Stearns, and Philip M. Lewis, II. An analysis of several heuristics for the traveling salesman problem. *SIAM Journal on Computing*, 6(3), 1977.

[86] Steven S Seiden. On the online bin packing problem. *Journal of the ACM (JACM)*, 49(5):640–671, 2002.

[87] Daniel Dominic Sleator and Robert Endre Tarjan. Amortized efficiency of list update rules. In *Proceedings of the 16th Annual ACM Symposium on Theory of Computing*, pages 488–492. ACM, 1984.

[88] Erik van den Akker, Kevin Buchin, and Klaus-Tycho Foerster. Multi-agent online graph exploration on cycles and tadpole graphs. In *Structural Information and Communication Complexity - 31st International Colloquium, SIROCCO 2024*. Springer, 2024.

[89] André van Vliet. An improved lower bound for on-line bin packing algorithms. *Information processing letters*, 43(5):277–284, 1992.

[90] Heinz-Jürgen Voss. Some properties of graphs containing k independent circuits. *Theory of Graphs. Proceedings of Colloquium Tihany*, pages 321–334, 1968.

[91] Yajun Wang and Sam Chiu-wai Wong. Two-sided online bipartite matching and vertex cover: Beating the greedy algorithm. In *Automata, Languages, and Programming – 42nd International Colloquium, ICALP 2015*, volume 9134 of *Lecture Notes in Computer Science*, pages 1070–1081. Springer, 2015.

[92] Chenyang Xu and Guochuan Zhang. Learning-augmented algorithms for online subset sum. *J. Glob. Optim.*, 87(2):989–1008, 2023.

[93] Andrew Chi-Chih Yao. New algorithms for bin packing. *Journal of the ACM (JACM)*, 27(2):207–227, 1980.

[94] Minghui Zhang, Xin Han, Yan Lan, and Hing-Fung Ting. Online bin packing problem with buffer and bounded size revisited. In *Journal of Combinatorial Optimization*, volume 33, pages 530–542, 2015.

[95] Yubai Zhang, Zhao Zhang, Yishuo Shi, and Xianyue Li. Algorithm for online 3-path vertex cover. *Theory Comput. Syst.*, 64(2):327–338, 2020.

[96] Yunhong Zhou, Deeparnab Chakrabarty, and Rajan Lukose. Budget constrained bidding in keyword auctions and online knapsack problems. In *Internet and Network Economics*, 2008.