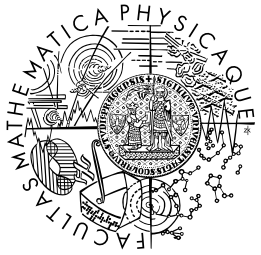

FACULTY
OF MATHEMATICS
AND PHYSICS
Charles University

DOCTORAL THESIS

Martin Böhm

Online Bin Stretching: Algorithms and Computer Lower Bounds

Computer Science Institute of Charles University

Supervisor of the doctoral thesis:	prof. RNDr. Jiří Sgall, DrSc.
Study programme:	Computer Science
Study branch:	Discrete Models and Algorithms

Prague 2018

I declare that I carried out this doctoral thesis independently, and only with the cited sources, literature and other professional sources.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Sb., the Copyright Act, as amended, in particular the fact that the Charles University has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 subsection 1 of the Copyright Act.

In ..... date ..... signature of the author

The first person to thank must immediately and irrevocably be my PhD advisor, Jiří Sgall. The academia has certainly many equally gifted computer scientists, but he has repeatedly shown to me that a scientist can harbor both great talent and great humanity. In low and high points of my doctoral studies alike, he has always supported me with both scientific insight as well as encouragement.

A special thanks goes to my colleague and frequent coauthor, Pavel Veselý. We have a student paper together that I am very fond of. He has submitted a PhD thesis at the same time as me on online packet scheduling; I recommend the reader to take a look at that thesis as well.

I would also like to mention Rob van Stee, a coauthor of our bin stretching work. His enthusiasm for the project has often boosted my own, and our email correspondence was a big motivation for making progress on the lower bound program.

I would like to thank my parents, Michal and Lenka, as well as my sister Kristýna, for all their support throughout the years; not to mention for the excellent Sunday lunches at home. The same goes to the rest of my family: all four of my grandparents, all four of my aunts and uncles, and all four of my cousins.

I am grateful to all my friends: Petr Onderka, Lukáš Lánský, Lukáš Mach, David Marek, Jirka Maršík, Josef Kakrda and many more. Special thanks to Petr and Tereza for proofreading parts of this thesis.

Finally, I would like to thank my life partner Tereza Hulcová for everything.

Title: Online Bin Stretching: Algorithms and Computer Lower Bounds

Author: Martin Böhm

Institute: Computer Science Institute of Charles University

Supervisor: prof. RNDr. Jiří Sgall, DrSc., Computer Science Institute of Charles University

Abstract:

We investigate a problem in semi-online algorithm design called Online Bin Stretching. The problem can be understood as an online repacking problem: the goal of the algorithm is to repack items of various sizes into m containers of identical size $R > 1$. The input items arrive one by one and the algorithm must assign an item to a container before the next item arrives.

A specialty of this problem is that there is a specific guarantee made to the algorithm: the algorithm learns at the start of the input that there exists a packing of all input items into m containers of capacity 1.

Our goal is to design algorithms for this problem which successfully pack the entire incoming sequence one by one while requiring the lowest container capacity R possible.

In this thesis, we show several new results about Online Bin Stretching: First, we design an algorithm that is able to pack the entire input into m containers of capacity 1.5 regardless of what the vale of m will be. Second, we show a specialized algorithm for the setting of just 3 containers; this algorithm is able to pack into 3 bins of capacity 1.375. Finally, we design and implement an involved search algorithm which is able to find lower bounds for Online Bin Stretching – and in fact we show the best known lower bounds for $3 \leq m \leq 8$.

Keywords: bin stretching online scheduling bin packing online algorithms computer search

Contents

1. Introduction

1.1 An example

We start with an example to illustrate the problem central to this thesis.

A number m of containers arrive at a shipping center. It is noted that all containers are at most 66 percent full. The items in the containers are too numerous to be individually labeled, yet all items must be unpacked and scanned for illicit and dangerous material. After the scanning, the items arrive at a conveyor belt one by one and they must be speedily repackaged into the original containers for further shipping.

Our question is: Does there exist a fast computer program which can repackage the items into the original containers without any reassignments?

We will settle our central question later in the thesis in the affirmative (the program exists). Before proceeding with the formal definitions, let us consider where the difficulty of this problem lies.

First, let us imagine that we are in the same shipping center, but in this case we receive two 99 percent loaded containers. Suppose that the first item arriving through the scanner is of size 50%. We can choose either container for it, and it is clearly not a wrong choice, as they are identical.

After storing the first item, a second item arrives through the scanner, this one of size 47% of the size of the container. We have a choice to make: Do we pack it with the first item or not? See Figure 1.1 for an illustration.

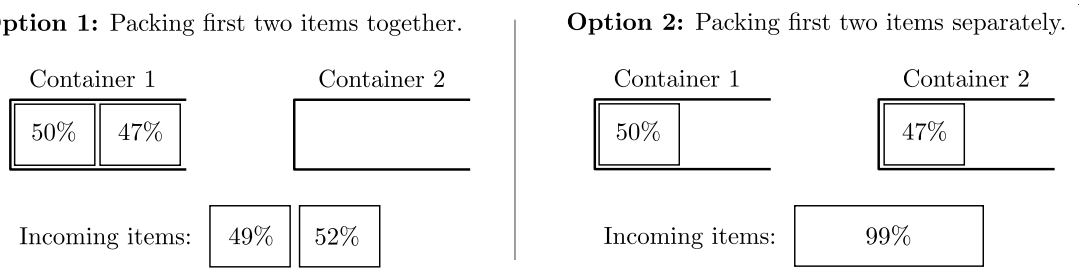

Figure 1.1: Two possible choices for repacking containers that are 99% full.

A quick thought gives us that there is no good choice here: If we pack it with the first item, we get a container that is 97 percent loaded, but then, to our dismay, we receive two items of sizes 49% and 52%, respectively – and those cannot be packed together. On the other hand, if we pack the first two items separately into two containers, we can find that the third item is a large crate taking up 99% of the container's size, and we have no more containers to choose from!

A lower bound. In the last few paragraphs, we have seen that repacking two containers that are 99% full is impossible without additional information on their contents. Can we actually get a lower number than 99% using only simple ideas? The answer is yes; we can show that we cannot repack two containers that are strictly more than 75% full. The thought process is explained in Figure 1.2.

The key decision of any algorithm (for containers loaded below 75%, where a repacking algorithm might exist) is then whether to pack sizable items together and increase the total loaded volume of one container, or pack them separately to avoid a potentially unsolvable situation.

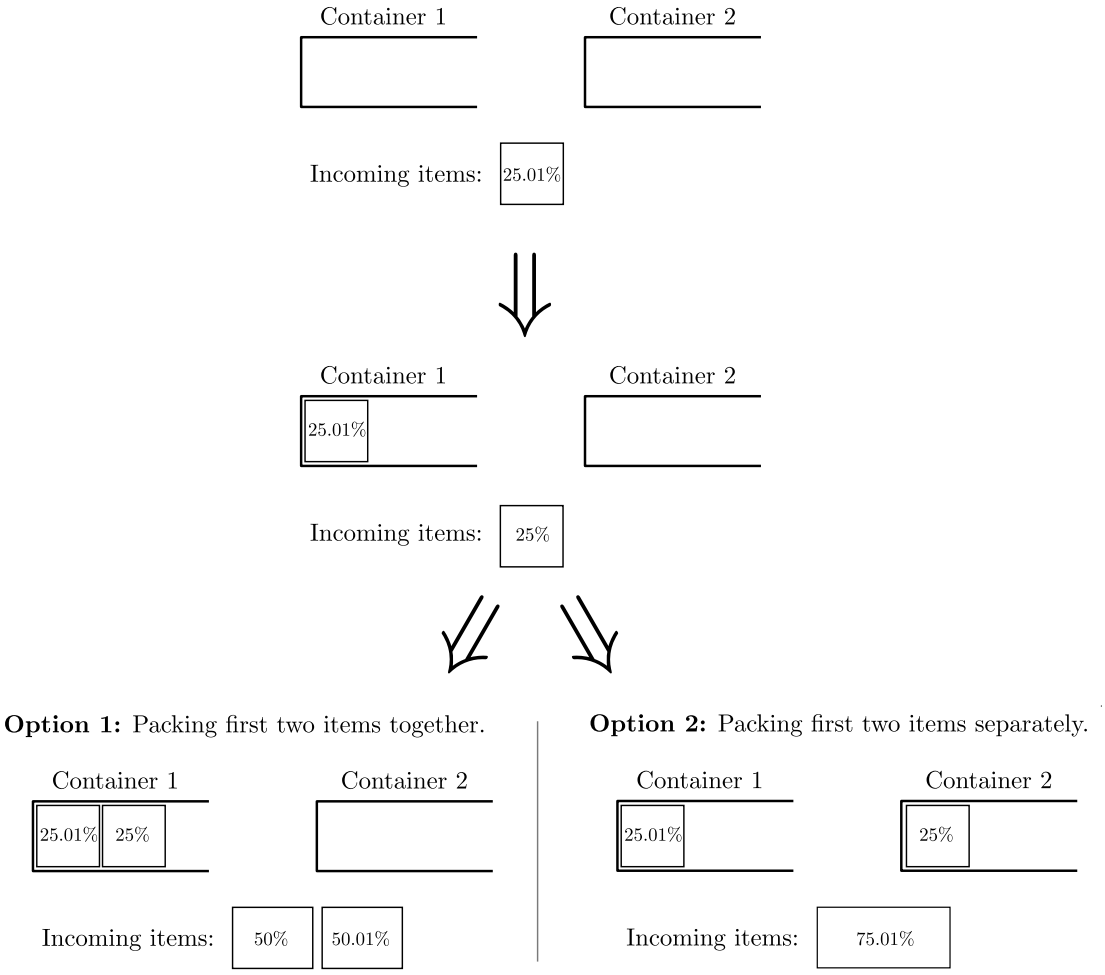

Figure 1.2: A visual proof that two containers with original load of 75.01% cannot be repackaged without any reassignments. A careful reader will observe that the number of containers is not important here; a similar visual proof can be created for any number of containers.

1.2 The online model of computation

Before we continue with the investigation of the problem of repacking containers, we specify the model of computation that we are considering.

For those not in the know of the theoretical computer science terminology, an *online algorithm* conjures an image of a program which is hosted on a server on the internet. This may indeed be so, but the definition actually predates the internet and goes to the 1970s, where the word "online" implied something arriving literally "on a line", such as on a conveyor belt.

In the online model, the input to the algorithm is not known in advance, but it arrives sequentially, one by one – exactly like luggage on the conveyor belt. After

each part of the input arrives, the algorithm must *immediately* and *irrevocably* decide what to do with this particular piece of the input. Most commonly, the decision is where to send a packet, which processor core should be selected for the current task, or which container should be used to pack the incoming item. As mentioned, after a decision is made, it cannot be reverted.

Only after a decision is made, the next item in the sequence is revealed to the algorithm, and the same decision process is repeated; of course even if two items arrive which are exactly the same, the algorithm can decide differently on them, since it knows the history of the instance.

After the entire input is processed, we look at the whole output of the algorithm and consider it a *solution* of the online problem. Naturally, the resulting solution is also a valid solution for the original non-online optimization problem, but usually is worse than what a normal polynomial-time algorithm for that optimization problem would produce.

Definition 1.1. An *online problem* is an optimization problem where some part of the input instance arrives sequentially, one by one. After each part arrives, it must be processed immediately and irrevocably, and the next part of the input instance arrives only after the processing of the previous part is complete.

An *online algorithm* is an algorithm that outputs some solution of an online problem. Given an instance I of the online problem, we can measure the performance of the online algorithm by its output's *value* on the function being optimized. We denote its value on instance I as $\mathrm{ALG}(I)$.

As examples of online problems, we can list the two that are arguably the most well-studied: scheduling and bin packing.

Problem 1.2 (Online Scheduling)**.**
Input:

- A number m – the number of available equal-speed machines.
- A sequence $j_1, j_2, \ldots, j_n$ of jobs, arriving online (one by one). Each job j has its own non-negative *processing time* $p(j)$. The processing time is revealed when the job arrives. The number of jobs n is not known in advance.
- Once a job appears, the algorithm must assign the job to one machine; the next job is revealed after the decision is made.

Output: A schedule (assignment) of jobs to machines, along with the makespan, which is the sum of the processing times on the busiest machine.
Goal: Design an online algorithm that minimizes the makespan.

Problem 1.3 (Bin Packing)**.**
Input:

- A sequence $i_1, i_2, \ldots, i_n$ of items, arriving online (one by one). Each item i has its own non-negative *size* $s(i), s(i) \leq 1$. The size is revealed when the item arrives.
- Once an item appears, it must be assigned to a *bin* such that the total load of the bin after the assignment is at most 1. The algorithm can open a new bin (with load 0) at any time, and can open any number of bins.

Output: A number of bins used by the algorithm B, as well as the assignment of all input items into these bins.
Goal: Design an online algorithm that minimizes the number of bins B.

The goal of algorithm designers is not only designing algorithms that perform well in practice, but also those that *verifiably* perform well. Since online algorithms are designed only for optimization problems, one could suggest to compute the value of the best possible online algorithm for the problem and compare our algorithm to that value. This is actually quite hard, since in almost all cases there is no good way to compute the value of the optimal online algorithm.

Instead, we compare our algorithm's performance to an optimum that actually is not online at all, but has access to the whole input at the same time. We define it as follows:

Definition 1.4. An *offline optimal algorithm* is an algorithm that optimally solves the online problem when given the entirety of the instance at the start of the program. On an instance I, we denote its value as $\mathrm{OPT}(I)$.

For a minimization problem, we say an online algorithm has *competitive ratio* c, if for all instances I,

$$\mathrm{ALG}(I) \leq c \cdot \mathrm{OPT}(I).$$

For a maximization problem, we say an online algorithm has *competitive ratio* c if for all I,

$$\mathrm{OPT}(I) \leq c \cdot \mathrm{ALG}(I).$$

It might seem confusing to some readers that the competitive ratio is differently defined for minimization and maximization problems; this is done so chiefly to make sure that the competitive ratio c is always in the interval $[1, \infty]$ of real numbers.

So far we have not spoken at all about the running time of online algorithms. This was intentional: there are no time or space restrictions on their computation. All we need is that they are algorithms, i.e., they finish in finite time. Despite this relaxed view of algorithmic runtime, almost all online algorithms turn out to run in polynomial time, many even in linear time.

1.3 Knowing a little in advance: the semi-online model

As we discussed above, an online algorithm cannot work with any assumptions about the input or its order – which means that if an online algorithm works excellently on larger input instances but poorly on a sequence of short length, its performance will be judged exactly on those short sequences.

One can easily see that "assuming nothing" is both a great advantage and at the same time a big disadvantage of the online computation model. Consider for instance the internet – while our algorithms should behave well under any scenario, it is not true that the very worst scenario will always come true. Indeed, many routing and scheduling algorithms make use of previously computed statistics and

assumptions about the data to behave well only in an average case or even just as a *heuristic*, i.e. without any provable guarantees.

If we wish to make our rule of "assuming nothing" weaker, we have several different options, all already studied in literature:

1. We could assume that the incoming items are not random, but are coming from a probabilistic distribution whose parameters we know. This is called *stochastic online optimization*.

2. We could assume that we are given some advice from an entity that knows much more about the input than we do (a likely candidate for such an oracle would be NSA). This advice can be based on our algorithm (so that the advice is exactly what our algorithm expects) as well as the input instance. The only requirement is that the information must be relatively small in order for the model to make sense. After our algorithm receives this arbitrary but limited advice, it should make use of it to solve the online problem in an effective way. This model is called *online optimization with advice*. A typical question in this model is what is the best competitive ratio which can be achieved with 1, $O(1)$ or $O(\log n)$ bits of advice.

3. There is a common sense weakening of the *optimization with advice* model from the previous point, and we describe it via an example. Suppose that our online algorithm is packing a sequence of apples into crates as best as it can. If we assume any advice at all, the algorithm could receive some information about the average curvature of the current batch of apples. The curvature may be potentially useful for packing, but clearly in real life there is no way an apple orchard can provide such an oddly specific piece of data.

   A much more reasonable request would be to provide the total number of apples, the total weight of the apples, or the estimated quality of the current season's fruit. In other words, in this model, we get a reasonable and pre-defined piece of information about the instance in advance, but our algorithms still need to solve the instance in an online way.

   This model is called *semi-online optimization*, and it is the one we focus on in this thesis.

1.4 The bin stretching problem

Let us return to our semi-online container repacking problem from Section 1.1, which we now define formally:

Problem 1.5 (Online Bin Stretching)**.**
Input:

- An integer m – the number of allowed bins. All bins are initially empty.

- A sequence of items $I = i_1, i_2, \ldots$ given online one by one. Each item has a *size* $s(i) \in [0, 1]$ and must be packed immediately and irrevocably into one of the m bins; the next item is revealed only after the previous one is packed.

Parameter: The *stretching factor* $R \geq 1$.
Output: Partitioning (packing) of I into bins $B_1, \ldots, B_m$ so that $\sum_{i \in B_j} s(i) \leq R$ for all $j = 1, \ldots, m$.
Guarantee: there exists a packing of all items in I into m bins of capacity 1.
Goal: Design an online algorithm with the stretching factor R as small as possible which packs all input sequences satisfying the guarantee.

The aforementioned stretching factor is just another name for the competitive ratio of our algorithm in the standard terminology of online algorithms (see Definition 1.4). Still, when designing algorithms for Online Bin Stretching it makes more sense to think about it as a parameter, as a maximum capacity that we want to maintain in all bins.

An attentive reader might object to the fact that we have defined Online Bin Stretching as a semi-online problem, as there is no tangible advice given to an input instance – the only thing we know is that some optimal packing exists.

The advice given is hidden in the fact that we actually only consider scaled instances where the unknown optimal packing is into m bins of capacity 1. If we go back to our example with repacking containers, we can see that the advice is actually the original load of the containers (e.g. that every container was at most 75% full).

As we can see from the nomenclature above – bins, load, packing – we will think of the problem as if it is related to Bin Packing (Problem 1.3). This is also how the literature discusses the problem. It is however important to keep in mind that the problem is not a variant of Bin Packing– the crucial difference is that we are not allowed to open as many bins as we want, which is legal in Bin Packing.

In fact, Online Bin Stretching is more closely related to the problem Online Scheduling (Problem 1.2). In the scheduling terminology, we would call it Online Scheduling with Known Optimal Makespan, and we would describe it this way:

Problem 1.6 (Online Bin Stretching/ Online Scheduling with Known Optimal Makespan)**.**

Input:

- An integer m – a number of identical (equally fast) *machines*;
- A sequence of *jobs* $I = i_1, i_2, \ldots$ given online one by one. Each job has a *processing time* $s(i) \in [0, 1]$ and must be assigned immediately and irrevocably to one of the machines. The jobs cannot be paused or reassigned later.

Output: A *schedule* assigning all input items to m machines, with the load of the busiest machine (called *makespan*) being some value R.
Guarantee: There exists a schedule which assigns all jobs so that the makespan is at most 1.
Goal: Design an online algorithm minimizing the makespan R.

A pinch of notation. For a bin B, we define the *load of the bin* $s(B) = \sum_{i \in B} s(i)$. Unlike $s(i)$, $s(B)$ can change during the course of the algorithm, as we pack more and more items into the bin. To easily differentiate between items, bins and lists of bins, we use lowercase letters for items (i, b, x), uppercase letters for bins and other sets of items (A, B, X), and calligraphic letters for lists of bins ($\mathcal{A}$, $\mathcal{C}$, $\mathcal{L}$).

Input scaling. We have defined Online Bin Stretching as a problem where all input items are of size between 0 and 1; in other words, all sizes are set up so that the guaranteed optimum packing is 1. This is natural for a generic definition, but writing and especially adding fractions such as $\frac{1}{2}$ and $\frac{7}{16}$ becomes tiresome after a while.

Therefore, in all three sections of the thesis, we rescale the instance so that the capacity of the bins in the optimum packing becomes some positive integer $S \geq 1$ and the capacity of a bin for the algorithm will also be some positive integer $R \geq S$. The resulting stretching factor will therefore be R/S.

It is worthwhile to keep in mind that our rescaling still allows fractional values; while bins have integral capacity both for the algorithm and for the optimal guarantee, the input item size $s(i)$ can be any value in the interval $(0, S]$.

1.5 Bin Stretching as a game

A basic and essential feature of any online problem – with Online Bin Stretching being no exception – is that any online algorithm is actually an algorithm for strategy in a two-player game.

What do we mean by that? If we recall the definition of the competitive ratio, we see that we measure our algorithm on the actual worst case that can happen, and this worst case is actually *adapting* to what the algorithm does for the preceding items.

Therefore, we can see the creator of the worst case as a person who plays a game with our algorithm, waiting for its move (in our case, where it packs the previous item) and then sending the next item so that the algorithm's decision is the worst possible.

In the online algorithm literature, we call this player Adversary. The other player, naturally, is called Algorithm.

Definition 1.7. A *combinatorial two-player game* is a game with two players with perfect information (the game state is fully revealed to both players). Each player has a set of moves (which do not need to be symmetric) and they alternate in selecting one move each.

A set of game states are denoted as winning the game for the first player and another set as winning for the second player. The two sets are known to both players in advance.

A player *wins* the game if it reaches its own winning state in a finite set of moves. A player has a *winning strategy* if it can reach a winning state regardless of the moves of the other player.

Theorem 1.8. *Fix a target competitive ratio c. An online problem along with c corresponds to a combinatorial two-player game between players* Algorithm *and*

Adversary. *The player* Algorithm *has a winning strategy if and only if there exists a c-competitive algorithm for the online problem. The player* Adversary *has a winning strategy if and only if no c-competitive algorithm exists.*

One can argue (in jest) that the main consequence of this theorem is that the literature on online algorithms loves to use terminology from combinatorial game theory; it is the author's personal experience that imagining an adversary and constantly thinking about its moves is useful when designing a new online algorithm.

However, the more important consequence for us is that if we can evaluate the entire decision tree for a game that corresponds to an online problem with a fixed ratio c, we can learn whether a c-competitive algorithm exists or not. There is a classical algorithm for evaluating a game tree called *the minimax algorithm* which is effective as long as the tree can be somehow bounded.

So far, everything we stated about game theory applies in full generality, but we will focus on Online Bin Stretching exclusively; therefore, we restate the game that corresponds to Online Bin Stretching:

Definition 1.9. Fix a number of bins B and a stretching factor S. The *bin stretching game* is defined as follows:

1. It is a two-player game between Algorithm and Adversary, and the player Adversary starts.

2. The player Adversary moves by sending any item $i \in (0, 1]$ with the restriction that i along with all previous items can be packed into B bins of capacity 1.

3. The player Algorithm moves by selecting a bin where the item is packed.

The winning states are defined thus:

- A state is winning for Algorithm if all bins are packed strictly below S and Adversary has cannot make any further moves.

- A state is winning for Adversary if one bin has total load at least S.

As you can see, we have formulated the bin stretching game with the player Algorithm trying to pack strictly below S. This way, a winning strategy for the player Adversary immediately implies that no online algorithm for Online Bin Stretching with stretching factor less than S exists.

The two main obstacles to implementing a search of the described two player game are the following:

1. Adversary can send an item of arbitrarily small size;

2. Adversary needs to make sure that at any time of the game, an offline optimum can pack the items arrived so far into three bins of capacity 1.

We discuss our solutions as well as the implementation details in Chapter 4.

1.6 History

1.6.1 History of Bin Stretching

Online Bin Stretching (or Online scheduling on identical machines with known makespan, if you prefer the scheduling terminology) is a natural extension of the Online scheduling problem, but it was not investigated until the 1990s.

The first result on Online Bin Stretching was a lower bound of 4/3 for two bins and a matching algorithm, which was discovered by Kellerer, Kotov, Speranza and Tuza in 1997 [22], in a paper focused primarily on semi-online partitioning problems.

The name Online Bin Stretching has been proposed by Azar and Regev in 1998 [4, 5], in the first paper dedicated to this problem. They extended the lower bound of 4/3 to any number of bins and gave an online algorithm with a stretching factor 1.625.

The first Bin Stretching algorithm. Let us dive a little bit more into the algorithms proposed by Azar and Regev, so that we can see how the ideas evolved in subsequent work. Imagine that we are designing an algorithm with stretching factor $1 + \alpha$, with α being the extra space. We can notice that whether a bin is loaded at most α is an important threshold; one reason might be because a bin of load at most α can still accept an item of size 1.

Using that threshold, Azar and Regev design two algorithms:

Algorithm 1

1: **for** the next incoming item i **do**
2: **if** there is a non-empty bin which stays below α with i, pack it there.
3: **if** there is a bin which is already above α and i fits, pack it there.
4: **if** there is an empty bin where i fits below α, pack it there.
5: Finally, try packing it into the least-loaded machine where i fits.

Algorithm 2

1: **for** the next incoming item i **do**
2: **if** there is *any* bin which stays below α with i, pack it there.
3: **if** there is bin which is already above α and i fits, pack it there.
4: Finally, try packing it into the least-loaded machine where i fits.

Both of these algorithms actually achieve a stretching factor of 5/3 (in other words, they never fail when $\alpha \geq 2/3$). A smart combination of the two leads to an algorithm with stretching factor 1.625.

The next algorithmic progress came as a consequence of the work of Cheng, Kellerer and Kotov in 2005 [8]. They design an online algorithm with a competitive ratio of 1.6 for a more general problem that also encompasses Online Bin Stretching (called Semi-Online Scheduling With Given Total Processing Time and defined below as Problem 1.10). The same algorithm therefore achieves a stretching factor of 1.6 for Online Bin Stretching.

Classification and bunching techniques. The next decrease of the upper bound on the stretching factor appeared ten years later in a result of Kellerer and Kotov [21]. Their algorithm achieves a stretching factor of $11/7 \approx 1.571$ using two ideas that appear in subsequent work as well as our thesis; we focus on them now.

The first idea is to *classify* items into groups based on their size, which in [21] are selected as follows:

Class:	Small items	Medium items	Large items
Size:	$(0, 4/7]$	$(4/7, 11/14]$	$(11/14, 1]$

Just looking at the sizes alone, we can infer some properties that may be useful; for instance, the upper bound $4/7$ for small items is precisely the value of α for an algorithm with stretching factor $11/7$. As for the medium items, we can see that a pair of them always fits together – in fact, the upper bound for medium items is the largest value for which it is true. Plus, when we pack two of them together, we get a load of at least $8/7$, which is allowed for the algorithm while these items need to be in separate bins in the optimal packing.

Similarly to item classes, Kellerer and Kotov also classify bins based on their current load, including one more class that is defined in a different way:

Class:	Tiny bins	Small bins	Medium bins	Large bins	Huge bins
Load:	$(0, 2/7]$	$(2/7, 4/7]$	$(4/7, 11/14]$	$(11/14, 1]$	$(1, 11/7]$

Class:	Large-item bins
Condition:	Contains a large item.

The algorithm works in two phases. The algorithm for the *first phase* is the following one:

Algorithm 3 First phase of Kellerer and Kotov

```
1: for the next incoming item i do
2:     if i is small then
3:         if i fits into a large-item bin, pack it there.
4:         if i fits into a small bin and the bin remains small, pack it there.
5:         Otherwise, put i into an empty bin.
6:     if i is medium then
7:         If i fits into a medium bin, pack it there.
8:         Otherwise, pack i into an empty bin.
9:     if i is large then
10:        Pack i into a small bin with the largest load.
11:        if none exist, pack i into an empty bin.
12:    if the ratio of medium & small bins to empty bins is more than 3 : 1 then
13:        End the first phase.
```

We have so far employed just the bin classification; only in the *second phase* comes the bunching technique into play. The first phase ends when the ratio of small and medium bins to non-empty bins is higher than $3 : 1$. The algorithm then creates groups (bunches) of 4 bins and packs incoming items one group at a

time; the next group is used only when it is impossible to pack into the previous one.

Roughly said, this 3:1 bunching is used to show the following: either a fourth big item arrives for this bunch (and fits into the one empty bin) or there were more than 4 units of items put in all four bins together and the algorithm must finish correctly.

The next algorithmic improvement came from a 2013 paper of Gabay, Brauner and Kotov [28]. Building upon the classification and bunching technique, they form the following item classes:

Class:	Tiny items	Small items	Medium items	Large items
Size:	$(0, 9/34]$	$(9/34, 9/17]$	$(9/17, 13/17]$	$(13/17, 1]$

Their key step is an observation that an algorithm can pack not only medium items together with only other medium items, but it can also do the same for small (non-tiny) items. Along with a more careful case analysis, they reach the stretching factor of $26/17 \approx 1.529$.

Computer lower bounds. Interestingly, the setting with a small fixed number of bins allows better lower bounds on the stretching factor. Gabay, Brauner and Kotov [15] give a new lower bound of $19/14$ for the setting where there are exactly three bins, i.e. $m = 3$. [1]

They create the lower bound instance using a computer search based on the ideas which we have described in Section 1.5 and which we will see in full detail in Chapter 4, where we also show our extensions to their approach.

The lower bounds for $m = 3$ cannot be easily translated into a lower bound for a larger m; for example, if we modify the instance by adding new bins and a corresponding number of items of size 1 (that must use exactly the new bins in the optimum), the semi-online algorithm still could use the additional capacity of α in the new bins to its advantage.

1.7 Related topics

1.7.1 Bin packing

The NP-hard offline problem Bin Packing was originally proposed by Ullman [25] and Johnson [20] in the 1970s. Since then it has seen major interest and progress, see the survey of Coffman et al. [9] for many results on classical Bin Packing and its variants.

1.7.2 Online scheduling

Alongside Bin Packing, Online Scheduling is probably the other most well-known problem for the online computational model. The first algorithm for Online Scheduling that minimizes the makespan was given by Graham [18] in the 1960s, achieving a competitive ratio of $2 - \frac{1}{m}$ for m machines.

[1] Subsequently to our work, the preprint [15] was updated to include the lower bound of $19/14$ for $m = 4$ as well [16].

This ratio was later improved to a constant factor independent on m; the currently best online algorithm for Online Scheduling is 1.9201-competitive [14], and there is a lower bound showing that no online algorithm for Online Scheduling can be better than 1.88-competitive [24].

Since Online Scheduling clearly belongs to the greater research area of scheduling theory, there are incredibly many variants that study different performance metrics, special job sizes, machines with varying properties and much more. See the survey of Albers [1] on Online Scheduling for many more results and open problems in the area.

1.7.3 Semi-online scheduling

We have already learned that Online Bin Stretching can be formulated as online scheduling on m identical machines with known optimal makespan (Problem 1.6). Algorithms for the semi-online setting are often essential in designing constant-competitive algorithms without the additional knowledge, e.g., for scheduling in the more general model of uniformly related machines [3, 6, 11].

For scheduling, other types of semi-online algorithms are studied as well. Historically first is the study of ordered sequences with non-decreasing processing times [19].

There are two variants which can be considered the closest to Online Bin Stretching.

The first is scheduling with known sum of all processing times:

Problem 1.10 (Onl. Scheduling with Known Total Processing Time)**.**

Input:

- An integer m – a number of identical machines that are available;
- A non-zero value T which represents the advice given to the algorithm in advance;
- A sequence of jobs $I = i_1, i_2, \ldots$ given online one by one. Each job has a *processing time* $s(i) \in [0, 1]$ and must be assigned immediately and irrevocably to one of the machines. The jobs cannot be paused or reassigned later.

Output: A schedule assigning all input jobs to machines.
Guarantee: The sum of processing times of all jobs is equal to the value T given to the algorithm in advance.
Goal: As in Problem 1.6, we aim to minimize the makespan (the load of the busiest machine).

Scheduling with known total processing time can be considered a sibling to Online Bin Stretching, as it is one of the problems also investigated by Kellerer, Kotov, Speranza and Tuza in [22] in their paper which contains the very first algorithm for Online Bin Stretching.

Online Scheduling with Known Total Processing Time is also the aforementioned problem investigated by Cheng, Kellerer and Kotov [8]. They

design a 1.6-competitive algorithm for it which has implications for Online Bin Stretching, as we will see in the following paragraph.

Let us now briefly consider the relationship of Online Bin Stretching and Online Scheduling with Known Total Processing Time. If we know the optimal makespan (as we do in Online Bin Stretching), we can always pad the instance by small items at the end so that the optimal makespan remains the same and the sum of processing times equals m times the optimal makespan. This means that any algorithm which is c-competitive for Online Scheduling with Known Total Processing Time has stretching factor c for Online Bin Stretching. Thus, one could conjecture that Online Bin Stretching and Online Scheduling with Known Total Processing Time might be equivalent.

However, when the sum of all processing times is known, the currently best results are a lower bound of 1.585 and an algorithm with ratio 1.6, both from [2]. This shows, somewhat surprisingly, that knowing the actual optimum gives a significantly bigger advantage to the semi-online algorithm over knowing just the sum of the processing times.

The second semi-online problem related to Online Bin Stretching has the same guarantee, but the machines have varying speeds:

Problem 1.11 (Semi-Online Scheduling on Two Related Machines)**.**
Input:

- An number $S_1 \geq 1$ denoting the speed of the first machine; the second machine has speed $S_2 = 1$.
- A sequence of jobs $I = i_1, i_2, \ldots$ given online one by one. Each job has a processing time $s(i) \in [0, 1]$ and must be assigned immediately and irrevocably to one of the machines. When a job i is assigned to a machine j, its *actual processing time* is $s(i)/S_j$.

Output: A schedule assigning all input jobs to machines.
Guarantee: There exists a schedule which assigns all jobs so that the makespan (maximum of total actual processing times per each machine) is at most 1.
Goal: Minimize the makespan (the total actual processing time of the busiest machine).

We could think about the aforementioned problem as Online Bin Stretching on two bins where the two bins have two different *compression factors*. This problem was first studied by Epstein [12] indeed as a variant of bin stretching; the paper gives upper and lower bounds on the competitive ratio (stretching factor) as a function of S_1. The upper and lower bounds of Epstein were later tightened by Dósa, Fügenschuh, Tan, Tuza and Węsek [10].

Curiously, the relation between the speed S_1 and the optimal stretching factor is quite complicated and non-monotone; contrast this with the fact that on uniform machines of equal speed, the optimal stretching factor can be easily shown to be 4/3.

See Pruhs et al. [23] for a survey of other results on (semi-)online scheduling, as well as a very recent survey by Epstein [13] that discusses Online Bin Stretching as well as all the semi-online problems mentioned above.

1.8 Contributions of this thesis

1. We present a new algorithm for Online Bin Stretching with a stretching factor of 1.5. We build on the two-phase approach which appeared previously in [21, 28]. In this approach, the first phase tries to fill some bins close to $R-1$ and achieve a fixed ratio between these bins and empty bins, while the second phase uses the bins in blocks of fixed size and analyzes each block separately.

   To reach 1.5, we needed to significantly improve the analysis using amortization techniques (represented by a weight function in our presentation) to amortize among blocks and bins of different types.

   This section uses results and text from the published paper

   Böhm, Sgall, van Stee, Veselý: "A two-phase algorithm for bin stretching with stretching factor 1.5." Journal of Combinatorial Optimization 34.3 (2017): 810-828.

2. We present a specialized algorithm for the subproblem where the number of bins is fixed to be 3. Our algorithm achieves stretching factor of $11/8 = 1.375$. This is the first improvement of the stretching factor 1.4 of Azar and Regev [4].

   This section uses results and text from the published paper

   Böhm, Sgall, van Stee, Veselý: "Online bin stretching with three bins." Journal of Scheduling 20.6 (2017): 601-621.

3. We present a new lower bound of $45/33 = 1.\overline{36}$ for Online Bin Stretching on three bins, along with a lower bound of 19/14 on four and five bins which is the first non-trivial lower bound for four and five bins. We build on the paper of Gabay et al. [15] but significantly change the implementation, both technically and conceptually. The lower bound of 19/14 for four bins is independently shown in [16].

   This section uses results and text from the conference paper:

   Böhm: "Lower Bounds for Online Bin Stretching with Several Bins." SOFSEM (Student Research Forum Papers/Posters) (2016).

4. We extend the implementation of the lower bound by adding parallelization, which allows us to go quite a bit beyond our original lower bounds. In particular, in the setting with three bins, we give a new lower bound of $112/82 \approx 1.366$. We also extend the lower bound of 19/14 to six, seven and eight bins, which strongly suggests that this might be a good candidate for a general lower bound.

   These results first appear in this thesis.

2. Algorithmic Results for Many Bins

This chapter describes an algorithm for Online Bin Stretching that achieves a stretching factor of 1.5 and works for any number of bins m. The algorithm follows the *classify and batch* approach as explained in Section 1.6; this approach has been already used in previous works.

Our main novel technique is applying the amortized analysis approach to the batching arguments. With that and other improvements, our algorithm is able to reach the stretching factor of 1.5. The goal of this chapter is to prove the following:

Theorem 2.1. *There exists an algorithm for* Online Bin Stretching *with a stretching factor of* 1.5 *for an arbitrary number of bins.*

As explained in Section 1.4, we rescale item sizes and bin capacities to avoid some fractional values in our arguments. Specifically, we scale the input so that the optimal bins have capacity 12 and the bins of the algorithm have capacity 18.

2.1 Algorithm overview

In the first phase of the algorithm we try to fill the bins so that their size is at most 6, as this leaves space for an arbitrary item in each bin. Of course, if items larger than 6 arrive, we need to pack them differently, preferably in bins of size at least 12 (since that is the size of the optimal bins). We stop the first phase when the number of non-empty bins of size at most 6 is three times the number of empty bins. In the second phase, we work in blocks consisting of three non-empty bins and one empty bin. The goal is to show that we are able to fill the bins so that the average size is at least 12, which guarantees we are able to pack the total size of $12m$ which is the upper bound on the size of all items.

The limitation of the previous results using this scheme was that the volume achieved in a typical block of four bins is slightly less than four times the size of the optimal bin, which then leads to bounds strictly above 3/2. This is also the case in our algorithm: A typical block may have three bins with items of size just above 4 from the first phase plus one item of size 7 from the second phase, while the last bin contains two items, each of size 7, from the second phase—a total of 47 instead of desired $4 \cdot 12$. However, we notice that such a block contains five items of size 7 which the optimum cannot fit into four bins. To take an advantage of this, we cannot analyze each block separately as in [21, 28]. Instead, the rough idea of our improved method is to show that a bin with no item of size more than 6 typically has size at least 13 and amortize among the blocks of different types. Technically this is done using a weight function w that takes into account both the total size of items and the number of items larger than 6. This is the main new technical idea of our proof.

There are other complications. We would like to ensure that a typical bin of size at most 6 has size at least 4 after the first phase. However, this is impossible to guarantee if the items packed there have size between 3 and 4. Larger items

are fine, as one per bin is sufficient, and the smaller ones are fine as well as we can always fit at least two of them. It is crucial to consider the items with sizes between 3 and 4 very carefully. This motivates our classification of items: Only the *regular items* of size in $(0, 3] \cup (4, 6]$ are packed in the bins filled up to size 6. The *medium items* of size in $(3, 4]$ are packed in their own bins (four or five per bin). Similarly, *large items* of size in $(6, 9]$ are packed in pairs in their own bins. Finally, the *huge items* of size larger than 9 are handled similarly as in the previous papers: If possible, they are packed with the regular items, otherwise each in their own bin.

The introduction of medium size items implies that we need to revisit the analysis of the first phase and also of the case when the first phase ends with no empty bin. These parts of the proof are similar to the previous works, but due to the new item type we need to carefully revisit it; it is now convenient to introduce another function v that counts the items according to their type; we call it a value, to avoid confusion with the main weight function w. The analysis of the second phase when empty bins are present is more complicated, as we need to take care of various degenerate cases, and it is also here where the novel amortization is used.

2.2 Classification of items and bins

We take an instance with an optimal packing into m bins of size at most 12 and, assuming that our algorithm fails, we derive a contradiction. One way to get a contradiction is to show that the size of all items is larger than $12m$. As stated above, we also use two bounds in the spirit of weight functions: weight $w(i)$ and value $v(i)$. The weight $w(i)$ is a slightly modified size to account for items of size larger than 6. The value $v(i)$ only counts the number of items with relatively large sizes. For our calculations, it is convenient to normalize the weight $w(i)$ and value $v(i)$ so that they are at most 0 for bins in the optimal packing (see Lemma 2.3). To get a contradiction, it is then sufficient to prove that the total weight or value of all bins is positive.

2.2.1 Classification of items

We classify the items based on their size $s(i)$ and define their value $v(i)$ as follows.

$s(i)$	$(9, 12]$	$(6, 9]$	$(3, 4]$	$(0, 3] \cup (4, 6]$
type	huge	large	medium	regular
$v(i)$	3	2	1	0

Definition 2.2. For a set of items A, we define the value $v(A) = (\sum_{i \in A} v(i)) - 3$.

Furthermore we define weight $w(A)$ as follows. Let $k(A)$ be the number of large and huge items in A. Then $w(A) = s(A) + k(A) - 13$.

For a set of bins $\mathcal{A}$ we define $v(\mathcal{A}) = \sum_{A \in \mathcal{A}} v(A)$, $w(\mathcal{A}) = \sum_{A \in \mathcal{A}} w(A)$ and $k(\mathcal{A}) = \sum_{A \in \mathcal{A}} k(A)$.

Lemma 2.3. *For any packing $\mathcal{A}$ of a valid input instance into m bins of an arbitrary capacity, we have $w(\mathcal{A}) \leq 0$ and $v(\mathcal{A}) \leq 0$.*

Proof. For the value $v(\mathcal{A})$, no optimal bin can contain items with the sum of their values larger than 3. The bound follows by summing over all bins and the fact that the number of bins is the same for the optimum and the considered packing.

For $w(\mathcal{A})$, we have $s(\mathcal{A}) \leq 12m$ and $k(\mathcal{A}) \leq m$, as the optimum packs all items in m bins of volume 12 and no bin can contain two items larger than 6. Thus $w(\mathcal{A}) = s(\mathcal{A}) + k(\mathcal{A}) - 13m \leq 12m + m - 13m = 0$. □

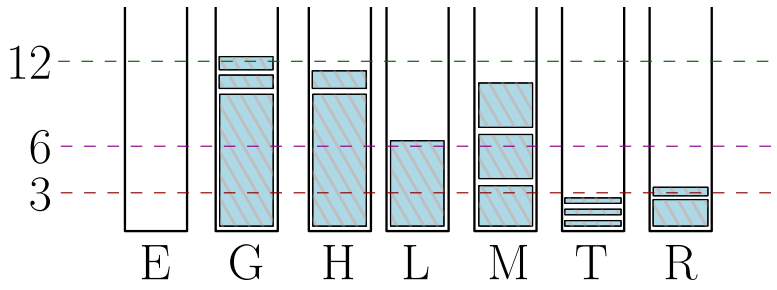

Figure 2.1: An illustration of bin types during the first phase of our algorithm with stretching factor 1.5.

2.2.2 Classification of bins

Along with items of specific types, we also wish to classify bins based on which types of items they contain as well as other properties (load, weight, value).

While item types are persistent, bin types change as items are added to the bins. As mentioned, our algorithm will work in two phases, and we make use of the bin types only during the first phase and when transitioning into the second phase.

During the first phase, our algorithm maintains the invariant that only bins of certain types exist, namely those from Definition 2.4. See Figure 2.1 for an illustration of these types.

Definition 2.4. Given a bin A, we define the following bin types and introduce letters that typically denote those bins:

- **Empty bins (E):** bins that have no item.
- **Complete bins (G):** all bins that have $w(A) \geq 0$ and $s(A) \geq 12$;
- **Huge-item bins (H):** all bins that contain a huge item (plus possibly some other items) and have $s(A) < 12$;
- **One large-item bin (L):** a bin containing only a single large item;
- **One medium-item bin (M):** a non-empty bin with $s(A) < 13$ and only medium items;
- **One tiny bin (T):** a non-empty bin with $s(A) \leq 3$;
- **Regular bins (R):** all other bins with $s(A) \in (3, 6]$;

2.3 First-phase algorithm

During the algorithm, let e be the current number of empty bins and r the current number of regular bins.

First-phase algorithm:

(1) While $r < 3e$, consider the next item i and pack it as follows, using bins of capacity 18; if more bins satisfy a condition, choose among them arbitrarily:
(2) If i is regular:
(3) If there is a huge-item bin, pack i there.
(4) Else, if there is a regular bin A with $s(A) + s(i) \leq 6$, pack it there.
(5) Else, if there is a tiny bin A with $s(A) + s(i) \leq 6$, pack it there.
(6) If i is medium and there is a medium-item bin where i fits, pack it there.
(7) If i is large and there is a large-item bin where i fits, pack it there.
(8) If i is huge:
(9) If there is a regular bin, pack i there.
(10) Else, if there is a tiny bin, pack i there.
(11) If i is still not packed, pack it in an empty bin.

First we observe that the algorithm above is properly defined. The stopping condition guarantees that the algorithm stops when no empty bin is available. Thus an empty bin is always available and each item i is packed. We now state the basic properties of the algorithm.

Lemma 2.5. *At any time during the first phase the following holds:*

(i) *All bins used by the algorithm are of the types from Definition 2.4.*
(ii) *All complete bins B have $v(B) \geq 0$.*
(iii) *If there is a huge-item bin, then there is no regular and no tiny bin.*
(iv) *There is at most one large-item bin and at most one medium-item bin.*
(v) *There is at most one tiny bin T. If T exists, then for any regular bin, $s(T) + s(R) > 6$. There is at most one regular bin R with $s(R) \leq 4$.*
(vi) *At the end of the first phase $3e \leq r \leq 3e + 3$.*

Proof. (i)-(v): We verify that these invariants are preserved when an item of each type arrives and also that the resulting bin is of the required type; the second part is always trivial when packing in an empty bin.

If a huge item arrives and a regular bin exists, it always fits there, thus no huge-item bin is created and (iii) cannot become violated. Furthermore, the resulting size is more than 12, thus the resulting bin is complete. Otherwise, if a tiny bin exists, the huge item fits there and the resulting bin is either complete or huge. In either case, if the bin is complete, its value is 0 as it contains a huge item.

If a large item arrives, it always fits in a large-item bin if it exists and makes it complete; its value is at least 1, as it contains two large items. Thus a second large-item bin is never created and (iv) is not violated.

If a medium item arrives, it always fits in a medium-item bin if it exists; the bin is then complete if it has size at least 13 and then its value is at least 1, as it contains 4 or 5 medium items; otherwise the bin type is unchanged. Again, a second medium-item bin is never created and (iv) is not violated.

If a regular item arrives and a huge-item bin exists, it always fits there, thus no regular bin is created and (iii) cannot become violated. Furthermore, if the resulting size is at least 12, the bin becomes complete and its value is 0 as it contains a huge item; otherwise the bin type is unchanged.

In the last case, a regular item arrives and no huge-item bin exists. The algorithm guarantees that the resulting bin has size at most 6, thus it is regular or tiny. We now proceed to verify (v). A new tiny bin T can be created only by packing an item of size at most 3 in an empty bin. First, this implies that no other tiny bin exists, as the item would be put there, thus there is always at most one tiny bin. Second, as the item is not put in any existing regular bin R, we have $s(R) + s(T) > 6$ and this also holds later when more items are packed into any of these bins. A new regular bin R with $s(R) \leq 4$ can be created only from a tiny bin; note that a bin created from an empty bin by a regular item is either tiny or has size in $(4, 6]$. If another regular bin with size at most 4 already exists, then both the size of the tiny bin and the size of the new item are larger than 2 and thus the new regular bin has size more than 4. This completes the proof of (v).

(vi): Before an item is packed, the value $3e - r$ is at least 1 by the stopping condition. Packing an item may change e or r (or both) by at most 1. Thus after packing an item we have $3e - r \geq 1 - 3 - 1 = -3$, i.e., $r \leq 3e + 3$. If in addition $3e \leq r$, the algorithm stops in the next step and (vi) holds. □

Terminating the first phase. If the algorithm packs all input items in the first phase, it stops. Otherwise according to Lemma 2.5(iii) we split the algorithm in two very different branches. If there is at least one huge-item bin, follow the *second phase with huge-item bins* below. If there is no huge-item bin, we follow the *second phase with regular bins*.

Any bin that is complete is not used in the second phase. In addition to complete bins and either huge-item bins, or regular and empty bins, there may exist at most three *special bins* denoted and ordered as follows: the large-item bin L, the medium-item bin M, and the tiny bin T.

2.4 Second phase with huge-item bins

In this case, we assume that a huge-item bin exists when the first phase ends. By Lemma 2.5(iii), we know that no regular and tiny bins exist. There are no empty bins either, as we end the first phase with $3e \leq r = 0$. With only a few types of bins remaining, the algorithm for this phase is very simple:

Algorithm for the second phase with huge-item bins:
Let the list of bins $\mathcal{L}$ contain first all the huge-item bins, followed by the special bins L, M, in this order, if they exist.
(1) For any incoming item i:
(2) Pack i using First Fit on the list $\mathcal{L}$, with all bins of capacity 18.

Suppose that we have an instance that has a packing into bins of capacity 12 and on which our algorithm fails. We may assume that the algorithm fails on the last item f. By considering the total volume, there always exists a bin with size at most 12. Thus $s(f) > 6$ and $v(f) \geq 2$.

If during the second phase an item n with $s(n) \leq 6$ is packed into the last bin in $\mathcal{L}$, we know that all other bins have size more than 12, thus all the remaining items fit into the last bin. Otherwise we consider $v(\mathcal{L})$. Any complete bin B has $v(B) \geq 0$ by Lemma 2.5(ii) and each huge-item bin gets nonnegative value,

too. Also $v(L) \geq -1$ if L exists. This shows that M must exist, since otherwise $v(\mathcal{L}) + v(f) \geq -1 + 2 \geq 1$, a contradiction.

Now we know that M exists, furthermore it is the last bin and thus we also know that no regular item is packed in M. Therefore M contains only medium items from the first phase and possibly large and/or huge items from the second phase. We claim that $v(M) + v(f) \geq 2$ using the fact that f does not fit into M and M contains no item a with $v(a) = 0$: If f is huge we have $s(M) > 6$, thus M must contain either two medium items or at least one medium item together with one large or huge item and $v(M) \geq -1$. If f is large, we have $s(M) > 9$; thus M contains either three medium items or one medium and one large or huge item and $v(M) \geq 0$. Thus we always have $v(\mathcal{L}) \geq -1 + v(M) + v(f) \geq 1$, a contradiction.

2.5 Second phase with regular bins

Let $\mathcal{E}$ resp. $\mathcal{R}$ be the set of empty resp. regular bins at the beginning of the second phase, and let $e = |\mathcal{E}|$. Let $\lambda \in \{0, 1, 2, 3\}$ be such that $|\mathcal{R}| = 3e + \lambda$; Lemma 2.5(vi) implies that λ exists. Note that it is possible that $\mathcal{R} = \emptyset$, in that case $e = \lambda = 0$.

We organize the remaining non-complete bins into blocks $\mathcal{B}_i$, and then order them into a list $\mathcal{L}$, as follows:

Definition 2.6. Denote the empty bins $E_1, E_2, \ldots, E_e$. The regular bins are denoted by $R_{i,j}$, $i = 1, ..., e + 1$, $j = 1, 2, 3$. The ith **block** $\mathcal{B}_i$ consists of bins $R_{i,1}, R_{i,2}, R_{i,3}, E_i$ in this order. There are several modifications to this rule:
(1) The first block $\mathcal{B}_1$ contains only λ regular bins, i.e., it contains $R_{1,1}, \ldots, R_{1,\lambda}, E_1$ in this order; in particular, if $\lambda = 0$ then $\mathcal{B}_1$ contains only E_1.
(2) The last block $\mathcal{B}_{e+1}$ has no empty bin, only exactly 3 regular bins.
(3) If $e = 0$ and $r = \lambda > 0$ we define only a single block $\mathcal{B}_1$ which contains $r = \lambda$ regular bins $R_{1,1}, \ldots, R_{1,\lambda}$.
(4) If $e = r = \lambda = 0$, there is no block, as there are no empty and regular bins.
(5) If $r > 0$, we choose as the first regular bin the one with size at most 4, if there is such a bin.
Denote the first regular bin by R_{first}. If no regular bin exists (i.e., if $r = 0$), R_{first} is undefined.

Note that R_{first} is either the first bin $R_{1,1}$ in $\mathcal{B}_1$ if $\lambda > 0$ or the first bin $R_{2,1}$ in $\mathcal{B}_2$ if $\lambda = 0$. By Lemma 2.5(v) there exists at most one regular bin with size at most 4, thus all the remaining $R_{i,j} \neq R_{\text{first}}$ have $s(R_{i,j}) > 4$.

Definition 2.7. The **list of bins** $\mathcal{L}$ we use in the second phase contains first the special bins and then all the blocks $\mathcal{B}_1, \ldots, \mathcal{B}_{e+1}$. Thus the list $\mathcal{L}$ is (some or all of the first six bins may not exist):

$$L, M, T, R_{1,1}, R_{1,2}, R_{1,3}, E_1, R_{2,1}, R_{2,2}, R_{2,3}, E_2, \ldots, E_e, R_{e+1,1}, R_{e+1,2}, R_{e+1,3}.$$

Whenever we refer to the ordering of the bins, we mean the ordering in the list $\mathcal{L}$. See Figure 2.2 for an illustration.

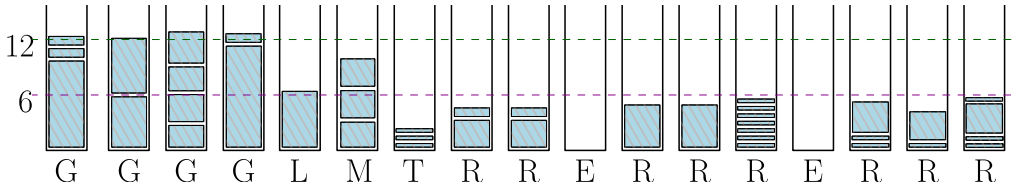

Figure 2.2: A typical state of the algorithm after the first phase. The bin labels correspond to the particular bin types. G denotes complete bins, other labels are the initial letters of the bin types. The non-complete bins (other than G) are ordered as in the list $\mathcal{L}$ at the beginning of the second phase with regular bins.

Algorithm for the second phase with regular bins:
Let $\mathcal{L}$ be the list of bins as in Definition 2.7, with all bins of capacity 18.
(1) For any incoming item i:
(2) If i is huge, pack it using First Fit on the reverse of the list $\mathcal{L}$.
(3) In all other cases, pack i using First Fit on the normal list $\mathcal{L}$.

Suppose that we have an instance that has a packing into bins of capacity 12 and on which our algorithm fails. We may assume that the algorithm fails on the last item. Let us denote this item by f. We have $s(f) > 6$, as otherwise all bins have size more than 12, contradicting the existence of optimal packing. Call the items that arrived in the second phase *new* (including f), the items from the first phase are *old*. See Figure 2.3 for an illustration of a typical final situation (and also of notions that we introduce later).

Our overall strategy is to obtain a contradiction by showing that

$$w(\mathcal{L}) + w(f) > 0\,.$$

In some cases, we instead argue that $v(\mathcal{L}) + v(f) > 0$ or $s(\mathcal{L}) + s(f) > 12|\mathcal{L}|$. Any of these is sufficient for a contradiction, as all complete bins have both value and weight nonnegative and size at least 12.

Let $\mathcal{H}$ denote all the bins from $\mathcal{L}$ with a huge item, and let $h = |\mathcal{H}| \bmod 4$. First we show that the average size of bins in $\mathcal{H}$ is large and exclude some degenerate cases; in particular, we exclude the case when no regular bin exists.

Lemma 2.8. *Let ρ be the total size of old items in R_{first} if $\mathcal{R} \neq \emptyset$ and $R_{\text{first}} \in \mathcal{H}$, otherwise set $\rho = 4$.*

(i) *The bins $\mathcal{H}$ are a final segment of the list $\mathcal{L}$ and $\mathcal{H} \subsetneq \mathcal{E} \cup \mathcal{R}$. In particular, $\mathcal{R} \neq \emptyset$ and R_{first} is defined.*
(ii) *We have $s(\mathcal{H}) \geq 12|\mathcal{H}| + h + \rho - 4$.*
(iii) *If $\mathcal{H}$ does not include R_{first}, then $s(\mathcal{H}) \geq 12|\mathcal{H}| + h \geq 12|\mathcal{H}|$.*
(iv) *If $\mathcal{H}$ includes R_{first}, then $s(\mathcal{H}) \geq 12|\mathcal{H}| + h - 1 \geq 12|\mathcal{H}| - 1$.*

Proof. First we make an easy observation used later in the proof. If $\mathcal{E} \cup \mathcal{R} \cup \{T\}$ contains a bin B with no huge item, then no bin preceding B contains a huge item. Indeed, if a huge item does not fit into B, then B must contain a new item i of size at most 9. This item i was packed using First Fit on the normal list $\mathcal{L}$, and therefore it did not fit into any previous bin. Thus the huge item also does not fit into any previous bin, and cannot be packed there.

Let $\mathcal{H}' = \mathcal{H} \cap (\mathcal{E} \cup \mathcal{R})$. We begin proving our lemma for $\mathcal{H}'$ in place of $\mathcal{H}$. That is, we ignore the special bins at this stage. The previous observation shows that $\mathcal{H}'$ is a final segment of the list.

We now prove the claims (ii)–(iv) with $\mathcal{H}'$ in place of $\mathcal{H}$. All bins $R_{i,j}$ with a huge item have size at least $4 + 9 = 13$, with a possible exception of R_{first} which has size at least $\rho + 9 = 13 + \rho - 4$, by the definition of ρ. Each E_i with a huge item has size at least 9. Thus for each i with $E_i \in \mathcal{H}'$, $s(E_i) + s(R_{i+1,1}) + s(R_{i+1,2}) + s(R_{i+1,3}) \geq 4 \cdot 12$, with a possible exception of $i = 1$ in the case when $\lambda = 0$. Summing over all i with $E_i \in \mathcal{H}'$ and the h bins in $\mathcal{R}$ from the first block intersecting $\mathcal{H}'$, and adjusting for R_{first} if $R_{\text{first}} \in \mathcal{H}'$, (ii) for $\mathcal{H}'$ follows. The claims (iii) and (iv) for $\mathcal{H}'$ are an immediate consequence as $\rho > 3$ if $R_{\text{first}} \in \mathcal{H}'$ and $\rho = 4$ otherwise.

We claim that the lemma for $\mathcal{H}$ follows if $\mathcal{H}' \subsetneq \mathcal{E} \cup \mathcal{R}$. Indeed, following the observation at the beginning of the proof, the existence of a bin in $\mathcal{E} \cup \mathcal{R}$ with no huge item implies that no special bin has a huge item, i.e., $\mathcal{H}' = \mathcal{H}$, and also $\mathcal{H}' = \mathcal{H}$ is a final segment of $\mathcal{L}$. Furthermore, the existence of a bin in $\mathcal{E} \cup \mathcal{R}$ together with $3e \leq r$ implies that there exist at least one regular bin, thus also R_{first} is defined and (i) follows. Claims (ii), (iii), and (iv) follow from $\mathcal{H}' = \mathcal{H}$ and the fact that we have proved them for $\mathcal{H}'$.

Thus it remains to show that $\mathcal{H}' \subsetneq \mathcal{E} \cup \mathcal{R}$. Suppose for a contradiction that $\mathcal{H}' = \mathcal{E} \cup \mathcal{R}$.

If T exists, let o be the total size of old items in T. If also R_{first} exists, Lemma 2.5(v) implies that $o + \rho > 6 > 4$, otherwise $o + \rho > 4$ trivially. In either case, summing with (ii) we obtain

$$o + s(\mathcal{H}') > 12|\mathcal{H}'|\,. \tag{2.1}$$

Now we proceed to bound $s(\mathcal{H})$. We have already shown claim (iv) for $\mathcal{H}'$, i.e., $s(\mathcal{H}') \geq 12|\mathcal{H}'| - 1$. If L or M has a huge item, the size of the bin is at least 12, as there is an old large or medium item in it. If T has a huge item, then (2.1) implies $s(T) + s(\mathcal{H}') > 9 + o + s(\mathcal{H}') > 9 + 12|\mathcal{H}'|$. Summing these bounds we obtain

$$s(\mathcal{H}) > 12|\mathcal{H}| - 3\,. \tag{2.2}$$

We now derive a contradiction in each of the following four cases.

Case 1: All special bins have a huge item. Then $\mathcal{L} = \mathcal{H}$ and (2.2) together with $s(f) > 6$ implies $s(\mathcal{L}) + s(f) > 12|\mathcal{L}| + 3$, a contradiction.

Case 2: There is one special bin with no huge item. Then its size together with f is more than 18, thus (2.2) together with $s(f) > 6$ implies $s(f) + s(\mathcal{L}) > 18 + 12|\mathcal{H}| - 3 > 12|\mathcal{L}|$, a contradiction.

Case 3: There are two special bins with no huge item and these bins are L and M.

Suppose first that M contains a new item n. Then $s(L) + s(n) > 18$ by the First Fit packing rule. The bin M contains at least one old medium item. Thus, using (2.2), we get $s(f) + s(\mathcal{L}) > 6 + s(L) + s(n) + 3 + 12|\mathcal{H}| - 3 > 24 + 12|\mathcal{H}| = 12|\mathcal{L}|$, a contradiction.

If M has no new item, then either f is huge and M has at least two medium items, or f is large and M has at least three items. In both cases $v(M) + v(f) \geq 2$.

Also $v(L) \geq -1$ since L has a large item, and $v(\mathcal{H}) \geq 0$ as each bin has a huge item. Altogether we get that the total value $v(\mathcal{L}) > 0$, a contradiction.

Case 4: There are two or three special bins with no huge item, one of them is T. Observe that the bin T always contains a new item n, as the total size of all old items in it is at most 3.

If there are two special bins with no huge item, denote the first one by B. As we observed at the beginning of the proof, if T exists and has no huge item, no special bin can contain a huge item, thus the third special bin cannot exist. We have $s(B) + s(n) > 18$, summing with (2.1) we obtain $s(f) + s(\mathcal{L}) \geq s(f) + s(B) + s(n) + o + s(\mathcal{H}') > 6 + 18 + 12|\mathcal{H}'| = 12|\mathcal{L}|$, a contradiction.

If there are three special bins with no huge item, we have $s(f)+s(L) > 18$ and $s(M)+s(n) > 18$. Summing with (2.1) we obtain $s(f)+s(\mathcal{L}) > 18+18+o+s(\mathcal{H}') > 36 + 12|\mathcal{H}'| = 12|\mathcal{L}|$, a contradiction. □

Having proven Lemma 2.8, we can infer existence of the following two important bins, which (as we will later see) split the instance into three logical blocks:

Definition 2.9.

- Let F, the **final bin** be the last bin in $\mathcal{L}$ before $\mathcal{H}$, or the last bin if $\mathcal{H} = \emptyset$.
- Let C, the **critical bin**, be the first bin in $\mathcal{L}$ of size at most 12.

First note that both F and C must exist: F exists by Lemma 2.8(i), which also shows that $F \in \mathcal{E} \cup \mathcal{R}$. C exists, as otherwise the total size is more than $12m$.

To make our calculations easier, we modify the packing so that f is put into F, even though it exceeds the capacity of F. Thus $s(F) > 18$ and f (a new item) as well as all the other new items packed in F or in some bin before F satisfy the property that they do not fit into any previous bin. See Figure 2.3 for an illustration of the definitions.

We start by some easy observations. Each bin, possibly with the exception of L and M, contains a new item, as it enters the phase with size at most 6, and the algorithm failed. Only items of size at most 9 are packed in bins before F; in F itself only the item f can be huge. The bin F always has at least two new items, one that did fit into it and f. All the new items in the bins after C are large, except for the new huge items in $\mathcal{H}$ and f which can be large or huge. (Note that at this point of the proof it is possible that C is after F; we will exclude this possibility soon.)

More observations are given in the next two lemmata.

Lemma 2.10. *(i) Let B be any bin before F. Then $s(B) > 9$. Furthermore, if $B \in \mathcal{E}$ then B contains at least two new items.*

(ii) Let B, B', B'' be any three bins in the ordering $\mathcal{L}$ with $B <_{\mathcal{L}} B' <_{\mathcal{L}} B'' \leq_{\mathcal{L}} F$ and let B'' contain at least two new items. Then $s(B)+s(B')+s(B'') > 36+o$, where o is the size of old items in B''.

(iii) Let B be arbitrary and let $B' \in \mathcal{R}$ be a bin such that $B <_{\mathcal{L}} B' \leq_{\mathcal{L}} F$.
If $B' \neq R_{\text{first}}$ then $s(B) + s(B') > 22$, in particular $s(B) > 11$ or $s(B') > 11$.
If $B' = R_{\text{first}}$ then $s(B) + s(B') > 21$.

Proof. F contains a new item n different from f. To prove (i), note that $s(n) \leq 9$, and n does not fit into B. It follows that if $B \in \mathcal{E}$, then B must contain at least two new items, as only items with size smaller than 9 are packed before F.

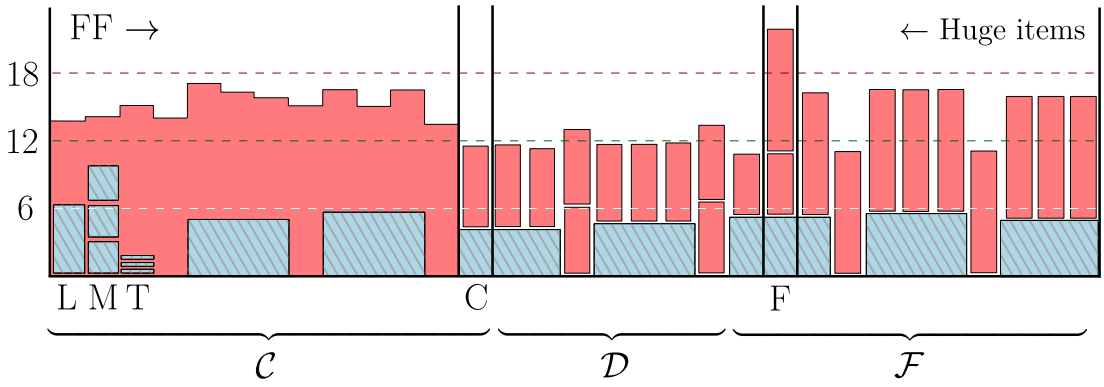

Figure 2.3: A typical state of the algorithm after the second phase with regular bins. The gray (hatched) areas denote the old items (i.e., packed in the first phase), the red (solid) regions and rectangles denote the new items (i.e., packed in the second phase). The bins that are complete at the end of the first phase are not shown. The item f on which the algorithm fails is shown as packed into the final bin F and exceeding the capacity 18, following the convention introduced after Definition 2.9.

To prove (ii), let n, n' be two new items in B'' and note that $s(B) + s(n) > 18$ and $s(B') + s(n') > 18$.

To prove (iii), observe that B' has a new item of size larger than $18 - s(B)$, and it also has old items of size at least 3 or even 4 if $B' \neq R_{\text{first}}$. □

Lemma 2.11. *The critical bin C is before F, there are at least two bins between C and F and C is not in the same block as F.*

Proof. All bins before C have size larger than 12. Using Lemma 2.8 we have

$$s(F) + s(\mathcal{H}) > 18 + 12|\mathcal{H}| - 1 = 12(|\mathcal{H}| + 1) + 5\,.$$

It remains to bound the sizes of the other bins. Note that $F \neq C$ as $s(F) > 18$. If C is after F, all bins before F have size more than 12, so all together $s(\mathcal{L}) > 12|\mathcal{L}|+5$, a contradiction. If C is just before F, then by Lemma 2.10(i), $s(C) > 9 = 12 - 3$ and the total size of bins in $s(\mathcal{L}) > 12|\mathcal{L}| + 5 - 3 > 12|\mathcal{L}|$, a contradiction.

If there is a single bin B between C and F, then $s(C) + s(B)$ plus the size of two new items in F is more than 36 by Lemma 2.10(ii). If $F \in \mathcal{E}$ then $\mathcal{H}$ starts with three bins in $\mathcal{R}$, thus $s(\mathcal{H}) \geq 12|\mathcal{H}| + 2$ using Lemma 2.8 with $h = 3$, and we get a contradiction. If $F \in \mathcal{R}$ then $R_{\text{first}} \notin \mathcal{H}$, thus Lemma 2.8 gives $s(\mathcal{H}) \geq 12|\mathcal{H}|$, and we get a contradiction as well.

The last case is when C and F are in the same block with two bins between them. Then $F \in \mathcal{E}$, so $h = 3$, and C is the first bin of the three other bins from the same block, so $R_{\text{first}} \notin \mathcal{H}$. Then $s(C) > 9$, the remaining two bins together with F have size more than 36 by Lemma 2.10(ii) and we use $s(\mathcal{H}) \geq 12|\mathcal{H}| + 3$ from Lemma 2.8 to get a contradiction. □

We now partition $\mathcal{L}$ into several parts (see Figure 2.3 for an illustration):

Definition 2.12.

- Let $\mathcal{F} = \mathcal{B}_i \cup \mathcal{H}$, where $F \in \mathcal{B}_i$.

- Let $\mathcal{D}$ be the set of all bins after C and before $\mathcal{F}$.
- Let $\mathcal{C}$ be the set of all bins before and including C.

Lemma 2.11 shows that the parts are non-overlapping. We analyze the weight of the parts separately, essentially block by block. Recall that a weight of a bin is defined as $w(A) = s(A) + k(A) - 13$, where $k(A)$ is the number of large and huge items packed in A. The proof is relatively straightforward if C is not special (and thus also $F \notin \mathcal{B}_1$), which is the most important case driving our choices for w. A typical block has nonnegative weight, we gain more weight in the block of F which exactly compensates the loss of weight in $\mathcal{C}$, which occurs mainly in C itself.

Let us formalize and prove the intuition stated in the previous paragraph in a series of three lemmata.

Lemma 2.13. *If F is not in the first block then $w(\mathcal{F}) > 5$, otherwise $w(\mathcal{F}) > 4$.*

Proof. All the new items in bins of $\mathcal{F}$ are large or huge. Each bin has a new item and the bin F has two new items. Thus $k(\mathcal{F}) \geq |\mathcal{F}| + 1$. All that remains is to show that $s(\mathcal{F}) > 12|\mathcal{F}| + 3$, and $s(\mathcal{F}) > 12|\mathcal{F}| + 4$ if F is not in the first block.

If F is the first bin in a block, the lemma follows as $s(F) > 18$ and $s(\mathcal{H}) \geq 12|\mathcal{H}| - 1$, thus $s(\mathcal{F}) = s(F) + s(\mathcal{H}) > 12|\mathcal{F}| + 5$.

In the remaining cases there is a bin in $\mathcal{R} \cap \mathcal{F}$ before F. Lemma 2.8 gives $s(\mathcal{H}) \geq 12|\mathcal{H}|$; moreover, if $F \in \mathcal{E}$, then $s(\mathcal{H}) \geq 12|\mathcal{H}| + 3$.

If F is preceded by three bins from $\mathcal{F} \cap \mathcal{R}$, then $F \in \mathcal{E}$ and thus $s(\mathcal{H}) \geq 12|\mathcal{H}|+3$. Using Lemma 2.10(iii) twice, two of the bins in $\mathcal{F} \cap \mathcal{R}$ before F have size at least 11 and using Lemma 2.10(i) the remaining one has size 9. Thus the size of these four bins is more than $11 + 11 + 9 + 18 = 4 \cdot 12 + 1$, summing with the bound for $\mathcal{H}$ we get $s(\mathcal{F}) > 12|\mathcal{F}| + 4$.

If F is preceded by two bins from $\mathcal{F} \cap \mathcal{R}$, then by Lemma 2.10(i) the total size of these two bins and two new items in F is more than 36. If $F \in \mathcal{R}$, the size of old items in F is at least 4 and with $s(\mathcal{H}) \geq 12|\mathcal{H}|$ we get $s(\mathcal{F}) > 12|\mathcal{F}| + 4$. If $F \in \mathcal{E}$, which also implies that F is in the first block, then $s(\mathcal{H}) \geq 12|\mathcal{H}| + 3$, thus $s(\mathcal{F}) > 12|\mathcal{F}| + 3$.

If F is preceded by one bin R from $\mathcal{F} \cap \mathcal{R}$, then let n be a new item in F different from f. We have $s(R) + s(n) > 18$ and $s(f) > 6$. We conclude the proof as in the previous case. □

Lemma 2.14.
If $C \in \mathcal{R}$ then $w(\mathcal{C}) \geq -6$.
If $C \in \mathcal{E}$ then $w(\mathcal{C}) \geq -5$.
If C is a special bin then $w(\mathcal{C}) \geq -4$.

Proof. For every bin B before C, $s(B) > 12$ and thus $w(B) > -1$ by the definition of C. Let $\mathcal{C}'$ be the set of all bins B before C with $w(B) \leq 0$. This implies that for $B \in \mathcal{C}'$, $s(B) \in (12, 13]$ and B has no large item. It follows that any new item in any bin after the first bin in $\mathcal{C}'$ has size more than 5. We have

$$w(\mathcal{C}) \geq w(\mathcal{C}') + w(C) \geq -|\mathcal{C}'| + w(C)\,. \tag{2.3}$$

First we argue that either $|\mathcal{C}'| \leq 1$ or $\mathcal{C}' = \{M, T\}$. Suppose that $|\mathcal{C}'| > 1$, choose $B, B' \in \mathcal{C}'$ so that B is before B'. If $B' \in \mathcal{E}$, either B' has at most two

(new) non-large items and $s(B') \le 6+6=12$, or it has at least three items and $s(B') > 5+5+5=15$; both options are impossible for $B' \in \mathcal{C}'$. If $B' \in \mathcal{R}$, it has old items of total size in $(3,6]$. Either B' has a single new item and $s(B') \le 6+6=12$, or it has at least two new items and $s(B') > 3+5+5=13$; both options are impossible for $B' \in \mathcal{C}'$. The only remaining option is that B' is a special bin. Since L has a large item, $L \notin \mathcal{C}'$ and $\mathcal{C}' = \{M, T\}$.

By Lemma 2.10(i), we have $w(C) \ge -4$. The lemma follows by summing with (2.3) in the following three cases: (i) $C \in \mathcal{R}$, (ii) if $\mathcal{C}' = \emptyset$ and also (iii) if both $C \in \mathcal{E}$ and $|\mathcal{C}'| = 1$.

For the remaining cases, (2.3) implies that it is sufficient to show $w(C) \ge -3$. If $C \in \mathcal{E}$ and $\mathcal{C}' = \{M, T\}$ then C contains two new items of size at least 5, thus $w(C) \ge -3$. If $C = T$ and $\mathcal{C}' = \{M\}$ then C either has a large item, or it has two new items: otherwise it would have size at most 3 of old items plus at most 6 from a single new item, total of at most 9, contradicting Lemma 2.10(i). Thus $w(C) \ge -3$ in this case as well. □

Lemma 2.15. (i) *For every block $\mathcal{B}_i \subseteq \mathcal{D}$ we have $w(\mathcal{B}_i) \ge 0$.*
(ii) *If there is no special bin in $\mathcal{D}$, then $w(\mathcal{D}) \ge 0$. If also $C \in \mathcal{R}$ then $w(\mathcal{D}) \ge 1$.*

Proof. First we claim that for each block $\mathcal{B}_i \subseteq \mathcal{D}$ with three bins in $\mathcal{R}$, we have

$$w(\mathcal{B}_i) \ge 0\,. \tag{2.4}$$

By Lemma 2.10(iii), one of the bins in $\mathcal{R} \cap \mathcal{B}_i$ has size at least 11. By Lemma 2.10(ii), the remaining three bins have size at least 36. We get (2.4) by observing that $k(\mathcal{B}_i) \ge 5$, as all the new items placed after C and before F are large, each bin contains a new item and E_i contains two new items.

Next, we consider an incomplete block, that is, a set of bins $\mathcal{B}$ with at most two bins from $\mathcal{R} \cap \mathcal{D}$ followed by a bin $E \in \mathcal{E} \cap \mathcal{D}$. We claim

$$w(\mathcal{B}) \ge 1\,. \tag{2.5}$$

The bin E contains two large items, since it is after C. In particular, $w(E) \ge 1$ and (2.5) follows if $|\mathcal{B}| = 1$. If $|\mathcal{B}| = 2$, the size of one item from E plus the previous bin is more than 18, the size of the other item is more than 6, thus $s(\mathcal{B}) \ge 24$; since $k(B) \ge 3$, (2.5) follows. If $|\mathcal{B}| = 3$, by Lemma 2.10(ii) we have $s(\mathcal{B}) \ge 36$; $k(\mathcal{B}) \ge 4$ and (2.5) follows as well.

By definition, $\mathcal{D}$ ends by a bin in $\mathcal{E}$ (if nonempty). Thus the lemma follows by using (2.5) for the incomplete block, i.e., for $C \in \mathcal{R}$ or for $\mathcal{B}_1$ if it does not have three bins in $\mathcal{R}$, and adding (2.4) for all the remaining blocks. Note that $C \in \mathcal{R}$ implies $\mathcal{D} \neq \emptyset$. □

We are now ready to derive the final contradiction.

If $\mathcal{D}$ does not contain a special bin, we add the appropriate bounds from Lemmata 2.14, 2.15 and 2.13. If $C \in \mathcal{R}$ then F is not in the first block and $w(\mathcal{L}) = w(\mathcal{C}) + w(\mathcal{D}) + w(\mathcal{F}) > -6+1+5 = 0$. If $C \in \mathcal{E}$ then F is not in the first block and $w(\mathcal{L}) = w(\mathcal{C}) + w(\mathcal{D}) + w(\mathcal{F}) > -5+0+5 = 0$. If C is the last special bin then $w(\mathcal{L}) = w(\mathcal{C}) + w(\mathcal{D}) + w(\mathcal{F}) > -4+0+4 = 0$. In all subcases $w(\mathcal{L}) > 0$, a contradiction.

The rest of the proof deals with the remaining case when $\mathcal{D}$ does contain a special bin. This implies there are at least two special bins and C is not the last

special bin. Since T is always the last special bin (if it exists), it must be the case that $C \neq T$ and thus $C = L$ or $C = M$. We analyze the special bins together with the first block, up to F if F belongs to it. First observe that the only bin possibly before C is L and in that case $w(L) \geq 0$, so $w(\mathcal{C}) \geq w(C)$.

Let A denote F if $F \in \mathcal{B}_1$ or E_1 if $F \notin \mathcal{B}_1$. As $A = F$ or $A \in \mathcal{E}$, we know that A contains at least two new items; denote two of these new items by n and n'. Since A is after C, we know that both n and n' are large or huge.

Let $\mathcal{A}$ be the set containing C and all bins between C and A, not including A. Thus $\mathcal{A}$ contains two or three special bins followed by at most three bins from $\mathcal{R}$. We have $k(\mathcal{A}) \geq |\mathcal{A}| - 1$ as each bin in $\mathcal{A}$ contains a large item, with a possible exception of C (if $C = M$). Furthermore $k(A) \geq 2$. The bound on $k(\mathcal{A})$ and $k(A)$ imply that

$$w(\mathcal{A}) + w(A) \geq s(\mathcal{A}) + s(A) - 12|\mathcal{A}| - 12 \tag{2.6}$$

and thus it is sufficient to bound $s(\mathcal{A}) + s(A)$.

The precise bound we need depends on what bin A is. In each case, we first determine a sufficient bound on $s(\mathcal{A})+s(A)$ and argue that it implies contradiction. Afterwards we prove the bound. Typically, we bound the size by creating pairs of bins of size 21 or 22 by Lemma 2.10(iii). We also use that $s(B) > 9$ for any $B \in \mathcal{A}$ by Lemma 2.10(i) and that n, n' together with any two bins in $\mathcal{A}$ have size at least 36 by Lemma 2.10(ii).

Case $A \neq F$: Then $F \notin \mathcal{B}_1$ and $A = E_1$. We claim that

$$s(\mathcal{A}) + s(A) \geq 12|\mathcal{A}| + 7\,. \tag{2.7}$$

First we show that (2.7) implies a contradiction. Indeed, (2.7) together with (2.6) yields $w(\mathcal{A}) + w(A) \geq -5$ and summing this with all the other bounds, namely $w(\mathcal{F}) > 5$ from Lemma 2.13 and $w(\mathcal{B}_i) \geq 0$ for whole blocks $\mathcal{B}_i \in \mathcal{D}$ from Lemma 2.15, leads to $w(\mathcal{L}) > 0$, which is a contradiction.

Now we prove (2.7). The items n and n' from A together with the first two special bins in $\mathcal{A}$ have size more than 36. Let $\mathcal{A}'$ be the set of the remaining bins; it contains possibly T and at most three bins from $\mathcal{R}$. It remains to show $s(\mathcal{A}') \geq 12|\mathcal{A}'| - 5$.

For $|\mathcal{A}'| = 0$ it holds trivially.

If $|\mathcal{A}'| = 1$, the only bin in $\mathcal{A}'$ has size more than 9 and this is sufficient.

For $|\mathcal{A}'| > 1$ we apply Lemma 2.10(iii) and pair as many bins from $\mathcal{A}'$ as possible; note that all the bins in $\mathcal{A}'$ except possibly T are in $\mathcal{R}$, so the assumptions of the lemma hold. If $|\mathcal{A}'| = 2$, then $s(\mathcal{A}') > 21 = 2 \cdot 12 - 3$. For $|\mathcal{A}'| = 3$ we get $s(\mathcal{A}') > 22 + 9 = 3 \cdot 12 - 5$, since we can create a pair without R_{first}. Finally, if $|\mathcal{A}'| = 4$ then $s(\mathcal{A}') > 22 + 21 = 4 \cdot 12 - 5$.

Case $A = F$: We claim that it is sufficient to prove

$$s(\mathcal{A}) + s(n) + s(n') > 12|\mathcal{A}| + \begin{cases} 8 & \text{if } F \in \mathcal{R} \text{ and } R_{\text{first}} \in \mathcal{A}, \\ 9 & \text{if either } F \in \mathcal{R} \text{ or } R_{\text{first}} \in \mathcal{A}, \\ 10 & \text{in all cases.} \end{cases} \tag{2.8}$$

First we show that (2.7) implies a contradiction.

If $F = E_1$ we note that $h = 3$ (as $\mathcal{H}$ starts with 3 bins in $\mathcal{R}$). Thus Lemma 2.8, items (iii) and (iv), together with $w(\mathcal{H}) \geq s(\mathcal{H}) - 12|\mathcal{H}|$ yields $w(\mathcal{H}) \geq 3$ for

$R_{\text{first}} \in \mathcal{A}$ or $w(\mathcal{H}) \geq 2$ for $R_{\text{first}} \notin \mathcal{A}$. Summing this with $w(\mathcal{A}) + w(A) > -3$ or $w(\mathcal{A}) + w(A) > -2$, that are obtained from (2.6) and (2.8) in the respective cases, we obtain $w(\mathcal{L}) > 0$, a contradiction.

If $F \in \mathcal{R}$ then we know that F also contains old items of size at least 3 if $R_{\text{first}} \notin \mathcal{A}$ or even 4 if $R_{\text{first}} \in \mathcal{A}$ (and thus $F \neq R_{\text{first}}$). Summing this with the respective bound from (2.8) we obtain $s(\mathcal{A}) + s(F) > 12|\mathcal{A}| + 12$. Summing this with $s(\mathcal{H}) \geq 12|\mathcal{H}|$ from Lemma 2.8(iii) now yields $s(\mathcal{L}) > 12|\mathcal{L}|$, a contradiction.

Thus (2.7) always leads to a contradiction.

We now distinguish subcases depending on $|\mathcal{A}|$ and in each case we either prove (2.8) or obtain a contradiction directly. Note that $R_{\text{first}} \in \mathcal{A}$ whenever $|\mathcal{A}| \geq 4$.

Case $|\mathcal{A}| = 2$: The two bins together with n and n' have size more than 36. Thus $s(\mathcal{A}) + s(n) + s(n') > 36 = 12 \cdot 2 + 12$, which implies (2.8).

Case $|\mathcal{A}| = 3$: We have $s(C) > 9$ and the remaining two bins together with n and n' have size more than 36. Thus $s(\mathcal{A}) + s(n) + s(n') > 12|\mathcal{A}| + 9$, which implies (2.8) in all cases except if $F = E_1$ and $R_{\text{first}} \notin \mathcal{A}$.

In the remaining case, $\mathcal{A} = \{L, M, T\}$ and $C = L$, as $\mathcal{A}$ contains no bin from $\mathcal{R}$ and $|\mathcal{A}| = 3$. We prove a contradiction directly. Let o be the size of old items in T. We apply Lemma 2.8(ii), using the fact that $o + \rho > 6$ by Lemma 2.5(v), where ρ is the total size of old items in $R_{\text{first}} \in \mathcal{H}$, and $h = 3$. We get $o + s(\mathcal{H}) \geq o + 12|\mathcal{H}| + h + \rho - 4 > 12|\mathcal{H}| + 5$. Let n'' be a new item in T. Since n'' does not fit into M, $s(M) + s(n'') > 18$; also $s(L) > 9$ and $s(F) > 18$. Summing all the bounds, we have $s(\mathcal{L}) \geq o + s(\mathcal{H}) + s(M) + s(n'') + s(L) + s(F) > 12|\mathcal{H}| + 5 + 18 + 9 + 18 = 12|\mathcal{L}| + 2$, a contradiction.

Case $|\mathcal{A}| = 4$: The last bin $R \in \mathcal{A}$ is in $\mathcal{R}$. Together with any previous bin it has size more than 21, the remaining two bins together with n and n' have size more than 36 by Lemma 2.10(ii). Thus $s(\mathcal{A}) + s(n) + s(n') > 21 + 36 = 4 \cdot 12 + 9$ which implies (2.8), since $R_{\text{first}} \in \mathcal{A}$.

Case $|\mathcal{A}| = 5$: First consider the case $F = E_1$. The last two bins of $\mathcal{A}$ are in $\mathcal{R}$, we pair them with two previous bins to form pairs of size more than $21 + 22$. The remaining bin has size at least 9, since n does not fit into it and $s(n) < 9$. We also have $s(F) > 18$. Thus $s(\mathcal{A}) + s(A) > 21 + 22 + 9 + 18 = 5 \cdot 12 + 10$, which implies (2.8).

If $F \in \mathcal{R}$ then one of the last two bins of $\mathcal{A}$ has size more than 11 and the other forms a pair of size more than 21 with one special bin. The remaining two bins together with n and n' have size more than 36 by Lemma 2.10(ii). Thus $s(\mathcal{A}) + s(n) + s(n') > 11 + 21 + 36 = 5 \cdot 12 + 8$ which implies (2.8), because $R_{\text{first}} \in \mathcal{A}$.

Case $|\mathcal{A}| = 6$: Then $\mathcal{A}$ contains all three special bins and three bins from $\mathcal{R}$, therefore also $F = E_1$. We form three pairs of a special bin with a bin from $\mathcal{R}$ of total size more than $21 + 22 + 22$. Since $s(F) > 18$, we have $s(\mathcal{A}) + s(F) > 21 + 22 + 22 + 18 = 6 \cdot 12 + 11$. Since in this case $A = F = E_1$, we have $s(\mathcal{H}) \geq 12|\mathcal{H}| + 3$ and $s(\mathcal{L}) > 12|\mathcal{L}|$, a contradiction.

In all of the cases we can derive a contradiction, which implies that our algorithm cannot fail. This concludes the proof of Theorem 2.1. □

2.6 Tightness of the analysis

We note that the analysis of our algorithm is tight, i.e., if we reduce the capacity of the bins below 18, the algorithm fails. Consider the following instance. Send two items of size 6 which are in the first phase packed separately into two bins. Then send $m - 1$ items of size 12. One of them must be put into a bin with an item of size 6, i.e., one bin receives items of size 18, while all the items can be packed into m bins of size 12.

To decrease the upper bound below 1.5 seems challenging. In particular, the instance above and its modifications with more items of size 6 or slightly smaller items at the beginning shows that these items need to be packed in pairs. This in turns creates difficulties that, in the current approach, lead to new item and bin types; at this point we do not know if such an approach is feasible.

3. An Algorithm Fine-Tuned for Three Bins

In this section, we focus on instances with exactly three bins. We prove the following:

Theorem 3.1. *There exists an algorithm that solves* Online Bin Stretching *for three bins with stretching factor* $1 + 3/8 = 1.375$.

Our final online algorithm uses several subroutines, one of which is the classical online algorithm First Fit:

Subroutine First Fit**:**
(1) Set an ordering of your bins.
(2) For every incoming item i:
(3) Pack i into the first bin where i fits below or to the limit t.
(4) If no such bin exists, report failure.

The three bins of our setting are named A, B, and C. We exchange the names of bins sometimes during the course of the algorithm.

Throughout the proof, we will need to argue about loads of the bins A, B, C before various items arrived. The following notation will help us in this endeavour:

Suppose that A is a bin and x is an item that gets packed at some point of the algorithm (not necessarily into A). Then $A_{\leftarrow x}$ will indicate the set of items that are packed into A just before x arrived.

We scale the input sizes by 16. The stretched bins in our setting therefore have capacity 22 and the optimal offline algorithm can pack all items into three bins of capacity 16 each.

3.1 Algorithm overview

If we were to design a new algorithm from scratch, we would probably start with trying to pack first all items in a single bin, as long as possible. In general, this is the strategy that the final algorithm will also follow. However, somewhat surprisingly, it turns out that from the very beginning we need to put items in two bins even if the items as well as their total size are relatively small.

It is clear that we have to be very cautious about exceeding a load of 6. For instance, if we put 7 items of size 1 in bin A, and 7 such items in B, then if two items of size 16 arrive, the algorithm will have a load of at least 23 in some bin. Similarly, we cannot assign too much to a single bin: putting 20 items of size 0.5 all in bin A gives a load of 22.5 somewhere if three items of size 12.5 arrive next. (Starting with items of size 0.5 guarantees that there is an optimal solution with bins of capacity 16.)

On the other hand, it is useful to keep one bin empty for some time; many problematic instances end with three large items such that one of them has to be placed in a bin that already has high load. Keeping one bin free ensures that

such items must have size more than 11 (on average), which limits the adversary's options, since all items must still fit into bins of size 16.

Deciding when exactly to start using the third bin and when to cross the threshold of 6 for the first time was the biggest challenge in designing this algorithm: both of these events should preferably be postponed as long as possible, but obviously they come into conflict at some point.

3.2 Good situations

Before stating the algorithm itself, we list a number of *good situations* (GS). These are configurations of the three bins which allow us to complete the packing regardless of the following input.

It is clear that the identities of the bins are not important here; for instance, in the first good situation, all we need is that *any* two bins together have items of size at least 26. We have used names only for clarity of presentation and of the proofs.

Definition 3.2. A *partial packing* of an input sequence S is a function $p : S_1 \to \{A, B, C\}$ that assigns a bin to each item from a prefix S_1 of the input sequence S.

Good Situation 1. *Given a partial packing such that $s(A) + s(B) \geq 26$, there exists an online algorithm that can finish the packing with capacity* 22*.*

Proof. Since the optimum can pack into three bins of size 16, the total size of items in the instance is at most $3 \cdot 16 = 48$. If two bins have size $s(A) + s(B) \geq 26$, all the remaining items (including the ones already placed on C) have size at most 22. Thus we can pack them all into bin C. □

Good Situation 2. *Given a partial packing such that $s(A) \in [4, 6]$, there exists an online algorithm that can finish the packing with capacity* 22*.*

Proof. Let A be the bin with size between 4 and 6 and B be one of the other bins (choose arbitrarily). Put all the items greedily into B. When an item x does not fit, put it into A, where it fits, as $s(A_{\leftarrow x}) \leq 6$. Now $s(B_{\leftarrow x}) + s(x) > 22$. In addition, $s(A_{\leftarrow x}) \geq 4$ by the assumption. Together we have $s(A_{\leftarrow x}) + s(B_{\leftarrow x}) + s(x) \geq 26$, which is GS1. □

From now on, we assume that each bin $X \in \{A, B, C\}$ satisfies $s(X) \notin [4, 6]$, otherwise we are in GS2.

Good Situation 3. *Given a partial packing such that $s(A) \geq 15$ and either* (i) *$s(B) + s(C) \geq 22$ or* (ii) *$s(C) < 4$ and $s(B)$ is arbitrary, there exists an online algorithm that can finish the packing with capacity* 22*.*

Proof. (i) We have $\max(s(B), s(C)) \geq 11$, so we are in GS1 on bins A and B or on bins A and C.

(ii) We pack arriving items into B. If $s(B) \geq 11$ at any time, we apply GS1 on bins A and B. Thus we can assume $s(B) < 11$ and we cannot continue packing into B any further. This implies that an item i arrives such that $s(i) > 11$. As $s(C_{\leftarrow i}) < 4$, we pack i into it and apply GS1 on bins A and C. □

Good Situation 4. *Given a partial packing such that* $s(A) + s(B) \geq 15 + \frac{1}{2}s(C)$, $s(B) < 4$, *and* $s(C) < 4$, *there exists an online algorithm that can finish the packing with capacity* 22.

Proof. Let c be the value of $s(C)$ when the conditions of this good situation hold for the first time. We run the following algorithm until we reach GS1 or GS3:

(1) If the incoming item i has $s(i) \geq 11 - \frac{1}{2}c$, pack i into B.
(2) Else, if i fits on A, pack it there.
(3) Otherwise pack i into C.

If at any time an item is to be packed into B by Step 1 (it always fits since we maintain $s(B) < 4$), then $s(A) + s(B) \geq 26$ and we reach GS1. In the event that no item is packed into B, we reach GS3 (with B in the role of C) whenever the algorithm brings the size of A to or above 15.

The only remaining case is when $s(A) < 15$ throughout the algorithm and several items with size in the interval $I := (22 - s(A), 11 - \frac{1}{2}c)$ arrive. These items are packed into C. Note that $I \subseteq (7, 11)$ and that the lower bound of I may decrease during the course of the algorithm.

The first two items with size in I will fit together, since $2(11 - \frac{1}{2}c) + c = 22$. With two such items packed into C, we know that the load $s(A) + s(C)$ is at least $s(A) + 2(22 - s(A)) = 44 - s(A) > 29$ and we have reached GS1, finishing the analysis.

□

Good Situation 5. *Given a partial packing such that a new item* a *with* $s(a) > 6$ *is packed into bin* A, $s(B_{\leftarrow a}) \in [3, 4)$, *and* $s(C_{\leftarrow a}) = 0$, *there exists an online algorithm that can finish the packing with capacity* 22.

Proof. Pack all incoming items into A as long as it is possible. If at some point $s(A) \geq 12$, we are in GS4, and so we assume the contrary. Therefore, $s(A) < 12$ and an item i arrives which cannot be packed into A.

Place i into B. If $s(i) \geq 12$, we apply GS3. We thus have $s(i) \in (10, 12)$ and $s(A_{\leftarrow i}) > 22 - s(i) > 10$. Continue with First Fit on bins B, A, and C in this order. (That is, pack an incoming item into the first bin X in which the item fits. If there is no such bin, stop.)

We claim that GS1 is reached at the latest after First Fit has packed two items, x and y, on bins other than B. If one of them (say x) is packed into bin A, this holds because $s(x) + s(B_{\leftarrow x}) > 22$ and $s(A_{\leftarrow x}) > 10$—enough for GS1. If both items do not fit in A, they are both larger than 10, since $s(A_{\leftarrow i}) < 12$ and nothing gets packed into A after item i. We will show by contradiction that this cannot happen.

As $s(A_{\leftarrow x}) < 12$ from our previous analysis, we note that $s(x), s(y) > 10$. We therefore have three items i, x, y with $s(i), s(x), s(y) > 10$ and an item $s(a) > 6$ from our initial conditions. These four items cannot be packed together by any offline algorithm into three bins of capacity 16, and so we have a contradiction with $s(x), s(y) > 10$. □

Good Situation 6. *If* $s(C) < 4$, $s(B) > 6$ *and* $s(A) \geq s(B) + 4 - s(C)$, *there exists an online algorithm that can finish the packing with capacity* 22.

Proof. Pack all items into A, until an item x does not fit. At this point $s(A_{\leftarrow x}) + s(x) > 22$. If x fits on B, we put it there and reach GS1 because $s(B_{\leftarrow x}) > 6$. Otherwise, x definitely fits on C because $s(C_{\leftarrow x}) < 4$ by assumption. By the condition on $s(A)$, we have $s(x) + s(A_{\leftarrow x}) + s(C) \geq s(x) + s(B) + 4 > 26$, and we are in GS1 again. □

Good Situation 7. *Consider the arrival of an item x. If it holds that*

- $s(A_{\leftarrow x}) < 4$,
- $s(C_{\leftarrow x}) < 4$,
- $s(B_{\leftarrow x}) \leq 9 + \frac{1}{2}(s(A_{\leftarrow x}) + s(C_{\leftarrow x}))$,
- *and* $s(B_{\leftarrow x}) + s(x) > 22$,

then there exists an online algorithm that packs all remaining items into three bins of capacity 22.

Proof. We have $s(B_{\leftarrow x}) > 22 - s(x) > 6$ and

$$\begin{aligned} s(x) > 22 - s(B_{\leftarrow x}) &\geq 13 - \frac{1}{2}(s(A_{\leftarrow x}) + s(C_{\leftarrow x})) \\ &\geq s(B_{\leftarrow x}) + 4 - s(A_{\leftarrow x}) - s(C_{\leftarrow x}). \end{aligned}$$

Placing x on A we increase $s(A)$ to at least $s(B_{\leftarrow x}) + 4 - s(C_{\leftarrow x})$ and we reach GS6. □

3.2.1 Good Situation First Fit

Throughout our algorithm, we often use a special variant of First Fit which tries to reach good situations whenever possible. This variant can be described as follows:

Definition 3.3. Let $\mathcal{L} = (X|_k, Y|_l, \ldots)$ denote a list of bins $X, Y, \ldots$ where each bin X has an associated capacity k satisfying $s(X) \leq k$. GSFF($\mathcal{L}$) (Good Situation First Fit) is an online algorithm for bin stretching that works as follows:

> **Subroutine** GSFF($\mathcal{L}$)**:** For each item i:
> If it is possible to pack i into any bin (including bins not in $\mathcal{L}$, and using capacities of 22 for all bins) so that a good situation is reached, do so and continue with the algorithm of the relevant good situation.
> Otherwise, traverse the list $\mathcal{L}$ in order and pack i into the first bin X such that $X|_k \in \mathcal{L}$ and $s(X) + s(i) \leq k$. If there is no such bin, stop.

For example, GSFF($A|_4, B|_{22}$) checks whether the packing $(A \cup \{i\}, B, C)$, $(A, B \cup \{i\}, C)$ or $(A, B, C \cup \{i\})$ is a partial packing of any good situation. If this is not the case, the algorithm packs i into bin A provided that $s(A) + s(i) \leq 4$. If $s(A) + s(i) > 4$, the algorithm packs i into bin B with capacity 22. If i cannot be placed into B, GSFF($A|_4, B|_{22}$) halts and another online algorithm must be applied to pack i and subsequent items.

3.3 The algorithm

In a way, any algorithm for online bin stretching for three bins must be designed so as to avoid several *bad situations:* the two most prominent ones being either two items of size $R/2$ or three items of size $R/3$, where R is the volume of the remaining items.

Our algorithm – especially Steps 4 and 10 – are designed to primarily evade such bad situations, while making sure that no good situation is missed. This evasive nature gives it its name.

Algorithm EVASIVE:
(1) Run GSFF$(A|_4, B|_4)$.
(2) Rename the bins so that $s(A) \geq s(B)$.
(3) If the next item j satisfies $s(j) > 6$:
(4) Set $p := 6 + s(j)$; apply GSFF$(A|_p, B|_4)$.
(5) If the next item w fits into $A|_{22}$:
(6) GSFF$(A|_{22}, B|_{22}, C|_{22})$.
(7) Else:
(8) GSFF$(A|_p, B|_{22}, C|_{22})$.
(9) Else (j satisfies $s(j) < 4$):
(10) GSFF$(A|_4, B|_q, C|_4)$ where $q := 9 + \frac{1}{2}(s(A) + s(C))$. Whenever $s(A)$ or $s(C)$ change in this step, update q and continue Step 10.
(11) GSFF$(A|_4, B|_{22}, C|_{22})$.
(12) GSFF$(A|_{22}, B|_{22}, C|_{22})$.

3.4 Analysis

3.4.1 Initial steps

Let us start the analysis of the algorithm EVASIVE in Step 3, where the algorithm branches on the size of the item j.

We first observe that our algorithm can be in two very different states, based on whether $s(j) > 6$ or $s(j) < 4$. Note that the case $s(j) \in [4, 6]$ is immediately settled using GS2, and that in either case it must be true that $s(j) > 2$; an item j with a smaller size would either fit into $A|_4$, $B|_4$ or trigger GS2.

Observation 3.4. *Assume that $2 < s(j) < 4$. We have that $s(A_{\leftarrow j}) \in (3, 4)$ and $s(B_{\leftarrow j}) + s(j) \in (6, 8)$ where A and B are bins after renaming in Step 2. Thus both A and B received some items during Step 1. Moreover, there is at most one item either in $A_{\leftarrow j}$, or in $B_{\leftarrow j}$.*

Proof. Since $s(j) < 4$ and $s(B_{\leftarrow j}) < 4$, the item j is assigned to B in Step 10, which by Step 2 is the least loaded bin among A and B after Step 1. For this bin, we have $s(B_{\leftarrow j}) > 2$. If the opposite were true, we would either reach GS2 by packing j into $B_{\leftarrow j}$, or j fits into $B|_4$, a contradiction with the definition of j.

This implies that both A and B received items in Step 1, so $s(A_{\leftarrow j}) + s(B_{\leftarrow j}) > 6$, else a good situation would have been reached before j arrived. It follows that $s(A_{\leftarrow j}) \in (3, 4)$ and $s(B_{\leftarrow j}) + s(j) \in (6, 8)$.

Since any item that is put into B during Step 1 must have size of more than two (otherwise it fits into $A|_6$), only one such item can be packed into B which proves the last statement. □

Contrast the preceding observation with the next one, which considers $s(j) > 6$:

Observation 3.5. *Assume that $s(j) > 6$. Then, $s(A_{\leftarrow j}) < 3$, $s(B_{\leftarrow j}) = 0$.*

Proof. If $s(A_{\leftarrow j}) \geq 3$ we reach GS5 by packing j into B. However, if $s(A_{\leftarrow j}) < 3$ then $s(A_{\leftarrow j}) + s(B_{\leftarrow j}) < 6$ which can be true only if $s(B_{\leftarrow j}) = 0$; indeed, we would have never packed an item z into a previously empty bin B if it were true that $s(A_{\leftarrow z}) + s(z) < 6$, $s(z) < 3$ and $s(A_{\leftarrow z}) < 3$. □

Both the analysis and the algorithm differ quite a lot based on the size of j. If it holds that $s(j) > 6$, we enter the *large case* of the analysis, while $2 < s(j) < 4$ will be analyzed as the *standard case*. Intuitively, if $s(j) > 6$, the offline optimum is now constrained as well; for instance, no three items of size 10 can arrive in the future. This makes the analysis of the large case comparatively simpler.

3.4.2 The large case

We now assume that $s(j) > 6$. Our goal in both the large case and the standard case will be to show that in the near future either a good situation is reached or several large items arrive, but Evasive is able to pack them nonetheless.

Let us start by recalling the relevant steps of the algorithm:

(4) Set $p := 6 + s(j)$; apply GSFF$(A|_p, B|_4)$.
(5) If the next item w fits into $A|_{22}$:
(6) GSFF$(A|_{22}, B|_{22}, C|_{22})$.
(7) Else:
(8) GSFF$(A|_p, B|_{22}, C|_{22})$

By choosing the limit p to be $s(j) + 6$ in Step 4, we make enough room for j to be packed into A. We also ensure that any item i larger than 6 that cannot be placed into A with capacity 22 must satisfy $s(i) + s(j) > 16$ and so i cannot be with j in the same bin in the offline optimum packing.

Let us define A_S as the set of items in A of size less than 6 (packed before or after j). We note the following:

Observation 3.6.

1. *During Step 4, if B contains any item, it is true that $s(A_S) + s(B) > 6$.*
2. *If no good situation is reached, the item w ending Step 4 satisfies $s(w) > 6$.*

Proof. The first point follows immediately from our choice of p and GSFF$(A|_p, B|_4)$.

For the second part of the observation, consider the item w that ends Step 4 and assume $s(w) \leq 6$. The possibility that $s(w) \in [4, 6]$ is excluded due to GS2. The case $s(B_{\leftarrow w}) \geq 3$ is also excluded, as this would imply GS5 with j in A.

Since $s(B_{\leftarrow w}) + s(w) > 6$, the only remaining possibility is $s(B_{\leftarrow w}) \in [2, 3)$, $s(w) \in (3, 4)$. Even though w does not fit into $A|_p$, if we were to pack w into $A|_{22}$, we

can use the first point of this observation and get $s(A) + s(B) \geq s(j) + \Big(s(A_S) + s(B_{\leftarrow w})\Big) + s(w) > 6 + 6 + 3 = 15$, enough for GS4 as $s(C) = 0$. The algorithm $\mathrm{GSFF}(A|_p, B|_4)$ in Step 4 will notice this possibility and will pack w into A, where it will always fit, as $s(A_{\leftarrow w}) < 15$ by GS4. □

We now split the analysis based on which branch is entered in Step 5:

Case 1: Item w fits into bin A; we enter Step 6.

We first note that $s(A) + s(B) < 15$, else we are in GS4 since C is still empty. This inequality also implies $s(B) = 0$, otherwise we have

$$s(A) + s(B) = s(w) + s(j) + (s(A_S) + s(B)) > 18$$

via Observation 3.6 and this is enough for GS4.

We continue with Step 6 until we reach a good situation or the end of input. Suppose three items x, y, z arrive such that none of them can be packed into A and we do not reach a good situation. We will prove that this cannot happen. We make several quick observations about those items:

1. We have $s(x) > 7$ because $s(A_{\leftarrow x}) < 15$ or we reach GS4. The item x is packed into B.
2. At any point, B contains at most one item, otherwise $s(A)+s(B) > 22+7 > 26$, reaching GS1.
3. We have $s(y) > 9$ because $\min(s(A_{\leftarrow y}), s(B_{\leftarrow y})) < 13$ by GS1. The item y is packed into C.
4. The bin C contains also at most one item, similarly to B.
5. Again, we have $s(z) > 9$ similarly to y. The item z does not fit into any bin.

From our observations above, we get $s(x) + s(y) > 22$, $s(x) + s(z) > 22$, $s(y) + s(z) > 22$. Therefore, at least two of the items $\{x, y, z\}$ are of size at least 11 and the third one is larger than 6. However, both items j and w have size at least 6, and there is no way to pack j, w, x, y, z into three bins of capacity 16, a contradiction.

Case 2: Item w does not fit into bin $A|_{22}$. The choice of p gives us $s(j)+s(w) > 16$. Item w is placed on B.

The limit p gives us an upper bound on the volume of small items A_S in A, namely $s(A_S) \leq 6$. An easy argument gives us a similar bound on B, namely if $B_S := B \setminus \{w\}$, then $s(B_S) < 4$. Indeed, we have $26 > s(A) + s(B) > 22 + s(B_S)$, the first inequality implied by not reaching GS1.

In Case 2, it is sufficient to consider two items x, y that do not fit into $A|_p$ or $B|_{22}$. We have:

1. Using $s(B_S) < 4$, we have $s(x) + s(w) > 18$ and $s(y) + s(w) > 18$.
2. None of the items x, y fits into $A|_{22}$. If say x did fit, then we use the fact that x does not fit into $B|_{22}$ and get $s(B) + s(A) = \Big(s(B_{\leftarrow x}) + s(x)\Big) + s(A_{\leftarrow x}) > 22 + s(j) > 26$ and we reach GS1.

3. The choice of the limit p on $s(A)$ implies $s(x)+s(j) > 16$ and $s(y)+s(j) > 16$.

4. Since $\min(s(A), s(B)) < 13$ at all times by GS1, we have $s(x) > 9$ and $s(y) > 9$.

5. The items x and y do not fit together into C, or we would have $s(C)+s(A) > 22 + s(y) > 26$. This implies $s(x) + s(y) > 22$.

From the previous list of inequalities and using $s(j) + s(w) > 16$, we learn that no two items from the set $\{j, w, x, y\}$ can be together in a bin of size 16. Again, this is a contradiction with the assumptions of Online Bin Stretching.

3.4.3 The standard case

From now on, we can assume that $s(j) < 4$, j is packed into B and Step 10 of Evasive is reached. Recall that by Observation 3.4 $s(A_{\leftarrow j}) \in (3, 4)$, $s(B_{\leftarrow j}) + s(j) \in (6, 8)$, and there is exactly one item either in A, or in B; we denote this item by e. We repeat the steps done by Evasive in the standard case:

(10) GSFF$(A|_4, B|_q, C|_4)$ where $q := 9 + \frac{1}{2}(s(A) + s(C))$. Whenever $s(A)$ or $s(C)$ change in this step, update q and continue Step 10.
(11) GSFF$(A|_4, B|_{22}, C|_{22})$.
(12) GSFF$(A|_{22}, B|_{22}, C|_{22})$.

Recall that $s(A) > 3$ by Observation 3.4. Assuming that no good situation is reached before Step 10, we observe the following:

Observation 3.7. *In Step* 10*, as long as C is empty, packing any item of size at least* 4 *leads to a good situation. Thus while C is empty, all items that arrive in Step* 10 *and are not put on A have size in* $(6 - s(A), 4)$.

Proof. Any item with size in $[4 - s(A), 6 - s(A)] \cup [4, 6]$ leads to GS2. Any item with size more than 6 is assigned to B if it fits there, reaching GS5, and else to A or C, reaching GS7 since $s(B) \leq 9 + \frac{1}{2}(s(A) + s(C))$. The only remaining possible sizes of items that are not packed into A are $(6 - s(A), 4)$. □

Corollary 3.8. *After Step* 10*, C contains exactly one item r and $s(A)+s(C) > 6$.*

Proof. From the previous observation it is clear that C receives at least one item r in Step 10. No second item r_2 can be packed into $C|_4$ in Step 10 as $s(r) + s(r_2)$ would be at least $2(6 - s(A)) > 4$. □

Step 10 terminates with a new item x which fits into $B|_{22}$ (otherwise we would reach GS7), but not below the limit $q = 9 + \frac{1}{2}(s(A) + s(C))$. We pack x into B in Step 11, getting $s(B) > 9 + \frac{1}{2}(s(A) + s(C)) > 12$.

A possible bad situation for our current packing is when three items b_1, b_2, b_3 arrive, where the items are such that no two items of this type fit together into any bin, and no single item of this type fits on the largest bin, which is B in our case. In fact, we will prove later that this is the only possible bad situation.

We claim that this potential bad situation cannot occur:

Claim 3.9. *Suppose that the algorithm* Evasive *reaches no good situation in the standard case. Then,* $s(C) \geq s(r) > 2.8$ *and after placing* x *into* B *in Step* 11 *it holds that*

$$s(B) < 12.8.$$

Furthermore, suppose that among items that arrive after x*, there are three items* b_1, b_2, b_3 *such that*

$$\min(s(b_1), s(b_2), s(b_3)) > 8.$$

Then, it holds that

$$\min(s(b_1), s(b_2), s(b_3)) < 9.2.$$

We now show how Claim 3.9 finishes the analysis of Evasive. After that we show the claim using linear programming; a formal proof is in Section 3.6.

After Step 10, assuming no good situation is reached, the algorithm places x into $B|_{22}$ and moves to Step 11, which is $\mathrm{GSFF}(A|_4, B|_{22}, C|_{22})$. Claim 3.9 gives us $s(B) < 12.8$ after placing x, while the fact that we exited Step 10 means that $s(B) > q = 9 + (s(A) + s(C))/2 > 12$.

Consider the first item b_1 that does not fit into $A|_4$. We have that $s(b_1) > 2$, otherwise GS2 is reached. However, any item that fits into B (as long as $s(C) \leq 4$) triggers GS4, because $s(A) + s(B) + s(b_1) \geq 6 + 12 > 15 + s(C)/2$.

We now know that the first item b_1 does not fit into both $A|_4$ and $B|_{22}$. We place it into C, noting that $s(b_1) > 22 - s(B) \geq 22 - 12.8 = 9.2$.

We keep packing items into $A|_4$, waiting for the second item b_2 that does not fit into $A|_4$ in Step 11. Again, $s(b_2) > 2$. Suppose that b_2 fits into $B|_{22}$ or $C|_{22}$. Claim 3.9 gives us $s(r) > 2.8$; we thus sum up bins B and C and get $s(B_{\leftarrow b_2}) + s(b_2) + s(r) + s(b_1) > 12 + 2 + 2.8 + 9.2 = 26$, which is enough for GS1. Our assumption was false, the item b_2 does fit into neither $B|_{22}$ nor $C|_{22}$, in particular we have that $s(b_2) > 9.2$.

We move to Step 12, pack b_2 into $A|_{22}$ and initiate $\mathrm{GSFF}(A|_{22}, B|_{22}, C|_{22})$. If at any time $s(A) \geq 14$, we enter GS1 on A and B. Otherwise, if an item b_3 does not fit into $A|_{22}$, it must satisfy $s(b_3) > 8$.

We now apply the full strength of Claim 3.9. The smallest item of b_1, b_2, b_3 must have size less than 9.2, and because of our argument, it must be b_3 – but this means it fits into B, as $s(B) < 12.8$. GS1 on bins A and B finishes the packing, since $s(A) + s(b_3) + s(B_{\leftarrow b_3}) > 22 + 12 > 26$.

Proof of Claim 3.9

Our current goal is to prove Claim 3.9. As in the large case, we would now like to appeal to the offline layout of the larger items currently packed. Unlike the large case, none of the items we have packed before Step 11 is guaranteed to be over 6.

Sidestepping this obstacle, we will argue about the offline layout of the smaller items. We now list several items that are packed before Step 12 and will be important in our analysis:

Definition 3.10. The four items e, j, r, x are defined as follows:

1. The item $e, 2 < s(e) < 4$: the only item packed into B in Step 1 by Observation 3.4. (Note that e might end up on A after renaming the bins.)

2. The item $j, 2 < s(j) < 4$, defined in Step 3.

3. The item $r, 2 < s(r) < 4$, placed into C in Step 10 by Observation 3.8; r is the only item in C until Step 11.

4. The item x which terminated Step 10.

There are four such items and only three bins, meaning that in the offline optimum layout with capacity 16, two of them are packed in the same bin. We will therefore argue about every possible pair, proving that each pair is of size more than 6.8.

Our main tool in proving the mentioned lower bounds are the inequalities that must be true during various stages of algorithm Evasive, since a good situation was not reached. We now list all the major inequalities that we will use.

- Observation 3.4:
$$s(A_{\leftarrow j}) > 3. \tag{3.1}$$

- Beginning of Step 10:
$$s(B_{\leftarrow j}) + s(j) > 6. \tag{3.2}$$

- GS2 not reached in Step 10:
$$s(A_{\leftarrow i}) + s(i) > 6. \tag{3.3}$$

- r does not fit into $B|_q$:
$$s(B_{\leftarrow r}) + s(r) > 9 + \frac{s(A_{\leftarrow r})}{2}. \tag{3.4}$$

- No GS4 if r packed into B:
$$(s(B_{\leftarrow r}) + s(r)) + s(A_{\leftarrow r}) < 15. \tag{3.5}$$

- No GS4 when x packed into $B|_{22}$:
$$(s(B_{\leftarrow x}) + s(x)) + s(r) < 15 + \frac{s(A_{\leftarrow x})}{2}. \tag{3.6}$$

- No GS4 when x packed into $B|_{22}$:
$$(s(B_{\leftarrow x}) + s(x)) + s(A_{\leftarrow x}) < 15 + \frac{s(r)}{2}. \tag{3.7}$$

- x does not fit into $B|_q$:
$$s(B_{\leftarrow x}) + s(x) > 9 + \frac{s(A_{\leftarrow x}) + s(r)}{2}. \tag{3.8}$$

- No GS6 if x packed into A:
$$s(A_{\leftarrow x}) + s(x) < s(B_{\leftarrow x}) + (4 - s(r)). \tag{3.9}$$

- No GS6 if x packed into A:
$$s(B_{\leftarrow x}) < s(A_{\leftarrow x}) + s(x) + \Big(4 - s(r)\Big). \tag{3.10}$$

- No GS6 if x packed into C:
$$s(r) + s(x) < s(B_{\leftarrow x}) + \Big(4 - s(A_{\leftarrow x})\Big). \tag{3.11}$$
- No GS6 if x packed into C:
$$s(B_{\leftarrow x}) < s(r) + s(x) + (4 - s(A_{\leftarrow x})). \tag{3.12}$$

We first show the claim using infeasible linear programming (LP) instances formed by the above inequalities. The specific instances can be found in Section 3.5 and online at `http://github.com/bohm/binstretch/`.

We write our LPs in the GNU MathProg language, thus they can be verified using GNU Linear Programming Kit. In Section 3.6 we provide formal proofs of these lemmas for completeness.

In our LPs we use a variable `i` for $s(i)$, the size of item i, and `X` for $s(X)$ where X is a bin. Instead of $s(X_{\leftarrow i})$ we write `X_i`.

Since our inequalities are mostly strict and LPs cannot contain strict inequalities, we add a non-negative variable `eps` (epsilon) which allows us to turn strict inequalities to non-strict. More precisely, we change an inequality of type `A < B` into `A + eps <= B` and we maximize the value of `eps`; we can do this, because our LPs do not need another objective function. If the optimal value of `eps` is zero, or if the LP is infeasible, then also the original system or strict inequalities is infeasible as well. Otherwise, if the optimal value of `eps` is positive, then all the strict inequalities can be satisfied.

Our first lemma establishes that j is actually the only item that is packed into B during Step 10, which intuitively means that j is not too small:

Lemma 3.11. *Assume that no good situation is reached until Step* 11*. Then it holds that during Step* 10*, only j is packed into B.*

Proof. We first prove that no two additional items j_2, j_3 can be packed into B during Step 10. Assuming the contrary, we get $s(B_{\leftarrow j_2}) + s(j_2) + s(j_3) > 6+2+2 = 10$. With that load on B, we consider the packing at the end of Step 10, when the item x arrived. If $s(x) + s(C_{\leftarrow x}) < 9$, we get GS6 by placing x into C since $s(A) > 3$, so it must be true that $s(x) + s(C_{\leftarrow x}) > 9$, which means $s(x) > 5$. This is enough for us to place x into $B|_{22}$ (where it fits, otherwise we are in GS7) and reach GS3.

This contradiction gives us that at most one additional item j_2 can be packed into B during Step 10. We will now prove that even j_2 does not exist, again by contradiction.

We split the analysis into two cases depending on which of j_2 and r arrives first.

Case 1. The item r is packed before j_2, meaning $s(B_{\leftarrow x}) = s(B_{\leftarrow r}) + s(j_2)$. We create a linear program from inequalities (3.1), (3.4), (3.7), (3.10) and $s(A_{\leftarrow r}) \geq s(A_{\leftarrow j})$ (since r arrives after j). We also add $s(r) < 4$ and $s(j_2) > 2$, since $s(A_{\leftarrow j_2}) < 4$ and j_2 did not fit into $A|_6$. We obtain LP1 for which the optimal value is 0, a contradiction.

Case 2. In the remaining case, j_2 arrives before r. We create an LP from (3.2), (3.3), (3.5), (3.7), (3.12), $s(r) < 4$, $s(A_{\leftarrow j_2}) \leq s(A_{\leftarrow r}) \leq s(A_{\leftarrow x})$, and $s(B_{\leftarrow x}) = s(B_{\leftarrow r}) = s(B_{\leftarrow j}) + s(j) + s(j_2)$ (the last two are only true here in Case 2, where r arrived later than j_2). The resulting LP2 is infeasible. □

Having established that only one item j is packed into B during Step 10, we can start deriving lower bounds on pairs of items from the set $\{e, j, r, x\}$. We will prove these bounds similarly to Lemma 3.11 by infeasible LPs from bounds that arise from evading various good situations.

Lemma 3.12. *Suppose that e and r are items as described in Definition 3.10 and suppose also that no good situation was reached during Step* 10 *of the algorithm* Evasive. *Then, $s(e) + s(r) \geq s(B_{\leftarrow j}) + s(r) > 6.8$.*

Proof. First of all, it is important to note that the item e may be packed on A or on B. Since either $B_{\leftarrow j}$, or $A_{\leftarrow j}$ contains solely e by Observation 3.4, we get that either $s(B_{\leftarrow j}) = s(e)$, or $s(B_{\leftarrow j}) \leq s(A_{\leftarrow j}) = s(e)$. Thus it is sufficient to prove $s(B_{\leftarrow j}) + s(r) > 6.8$.

We use (3.4), (3.8), (3.9), $s(j) < 4$,

$$s(B_{\leftarrow j}) \leq s(A_{\leftarrow j}) \leq s(A_{\leftarrow r}) \leq s(A_{\leftarrow x}),$$

and a converse of the claim and obtain LP3 with the optimal value equal to 0. □

Lemma 3.13. *Suppose that e and j are items as described in Definition 3.10 and suppose also that no good situation was reached by the algorithm* Evasive. *Then, $s(e) + s(j) \geq s(B_{\leftarrow j}) + s(j) > 7.6$.*

Proof. The same argument as in Lemma 3.12 gives us $s(e) + s(j) \geq s(B_{\leftarrow j}) + s(j)$. We therefore aim to prove $s(B_{\leftarrow j}) + s(j) > 7.6$. We create LP4 from (3.8), (3.11), $s(B_{\leftarrow j}) + s(r) > 6.8$ by Lemma 3.12, and $s(B_{\leftarrow j} \leq s(A_{\leftarrow x})$ for which 0 is the optimum again.

□

Lemma 3.14. *Suppose that j and r are items as described in Definition 3.10 and suppose also that no good situation was reached by the algorithm* Evasive. *Then, $s(r) + s(j) > 7$.*

Proof. Starting with (3.4):

$$s(B_{\leftarrow j}) + s(j) + s(r) > 9 + \frac{s(A_{\leftarrow r})}{2}$$

and using $s(B_{\leftarrow j}) \leq s(A_{\leftarrow j}) \leq s(A_{\leftarrow r})$ with $s(B_{\leftarrow j}) < 4$, we have:

$$\begin{aligned} s(j) + s(r) &> 9 + \left(\frac{s(A_{\leftarrow r})}{2} - s(B_{\leftarrow j}) \right) \\ &\geq 9 - \frac{s(B_{\leftarrow j})}{2} > 7. \end{aligned}$$

□

Lemma 3.15. *Suppose that x, e, j, r are items as described in Definition 3.10. Suppose also that no good situation was reached by the algorithm* Evasive. *Then,*

$$s(x) > 4 \quad \textit{and} \quad s(x) + \min(s(j), s(e), s(r)) > 6.8.$$

Proof. With all the previous lemmas in place, the proof is simple enough. We first observe that $s(x) > 4$; this is true because $s(B_{\leftarrow j}) + s(j) < 4 + 4 = 8$ and $s(B_{\leftarrow x}) + s(x) > q \geq 12$.

Since the sizes of the remaining three items $\{e, r, j\}$ are bounded from above by 4 but their pairwise sums are always at least 6.8, we have that $\min\{e, r, j\} > 2.8$, which along with $s(x) > 4$ gives us the required bound. □

From Lemmas 3.12, 3.13, 3.14 and 3.15 we get a portion of Claim 3.9: if three big items b_1, b_2, b_3 exist in the offline layout, then one of these items needs to be packed together with at least two items from the set $\{e, j, r, x\}$, and therefore $\min(s(b_1), s(b_2), s(b_3)) < 9.2$. The second bound $s(r) > 2.8$ follows from Lemma 3.12 and the fact that $s(e) < 4$.

All that remains is to prove the bound on $s(B)$, which we do in the following lemma:

Lemma 3.16. *Suppose that no good situation was reached in the algorithm* Evasive *during Step* 10*. Then, after placing x into B in Step* 11*, it holds that $s(B) < 12.8$.*

Proof. As before, we will use our inequalities to derive an LP showing the desired bound. As we have argued above, Lemma 3.12 gives us that $s(r) > 2.8$. We also use inequalities (3.7), (3.11), $s(B_{\leftarrow j}) \leq s(A_{\leftarrow j}) \leq s(A_{\leftarrow x})$ (this is true because we reorder the bins B, A in Step 2), and $s(j) < 4$ to get LP5 with 0 being the optimal value. □

With Lemma 3.16 proven, we have finished the proof of Claim 3.9 and completed the analysis of the algorithm Evasive.

3.5 Linear programs used in the proofs

LP1

```
var A_j; var A_r; var A_x; var B_r; var B_x;
var j_2; var r; var x;
var eps >= 0;

maximize obj: eps;

AjLessThanAr: A_j <= A_r;
Bx: B_x = B_r + j_2;
eq1: A_j >= 3 + eps;
eq4: B_r + r >= 9 + A_r/2 + eps;
eq7: B_x + x + A_x <= 15 + r/2 - eps;
eq10: B_x <= A_x + x + (4 - r) - eps;
j2notSmall: j_2 >= 2 + eps;
rNotBig: r <= 4 - eps;
```

LP2

```
var A_j_2; var A_r; var A_x; var B_r; var B_x; var B_j;
var r; var j_2; var x; var j;
var eps >= 0;

maximize obj: eps;

Aj2LessThanAr: A_j_2 <= A_r;
ArLessThanAx: A_r <= A_x;
BxEqBr: B_x = B_r;
BrEq: B_r = B_j + j + j_2;
eq2: B_j + j >= 6 + eps;
eq3: A_j_2 + j_2 >= 6 + eps;
eq5: B_r + r + A_r <= 15 - eps;
eq7: B_x + x + A_x <= 15 + r/2 - eps;
eq12: B_x <= r + x + (4 - A_x) - eps;
rNotBig: r <= 4 - eps;
```

LP3

```
var A_r; var A_x; var B_j; var B_r; var B_x;
var r; var x; var j;
var eps >= 0;

maximize obj: eps;

BrEq: B_r = B_j + j;
BxEq: B_x = B_r;
eq4: B_r + r >= 9 + A_r/2 + eps;
eq8: B_x + x >= 9 + (A_x + r)/2 + eps;
eq9: A_x + x <= B_x + (4 - r) - eps;
BjLessThanAx: B_j <= A_x;
BjLessThanAr: B_j <= A_r;
jNotBig: j <= 4 - eps;
contradiction: B_j + r <= 6.8;
```

LP4

```
var A_x; var B_j; var B_x;
var r; var x; var j;
var eps >= 0;

maximize obj: eps;

BxEq: B_x = B_j + j;
eq8: B_x + x >= 9 + (A_x + r)/2 + eps;
eq11: r + x <= B_x + (4 - A_x) - eps;
BjPlus_rBound: B_j + r >= 6.8 + eps;
BjLessThanAx: B_j <= A_x;
contradiction: B_j + j <= 7.6;
```

LP5

```
var A_x; var B_j; var B_x;
var r; var x; var j;
var eps >= 0;

maximize obj: eps;

BjLessThanAx: B_j <= A_x;
BxEq: B_x = B_j + j;
eq7: B_x + x + A_x <= 15 + r/2 - eps;
eq11: r + x <= B_x + (4 - A_x) - eps;
jNotBig: j <= 4 - eps;
rBound: r >= 2.8 + eps;
contradiction: B_j + j + x >= 12.8;
```

3.6 Formal proofs from Section 3.4.3

For completeness, we provide full proofs of lemmas in Section 3.4.3 which are shown using infeasible linear programs. Recall that these lemmas are parts of the proof of Claim 3.9. We use the notation introduced in Definition 3.10 and Equations (3.1)-(3.12).

Lemma 3.17 (Lemma 3.11)**.** *Assume that no good situation is reached until Step* 11*. Then it holds that during Step* 10*, only j is packed into B.*

Proof. We first prove that no two additional items j_2, j_3 can be packed into B during Step 10. Assuming the contrary, we get $s(B_{\leftarrow j_2})+s(j_2)+s(j_3) > 6+2+2 = 10$. With that load on B, we consider the packing at the end of Step 10, when the item x arrived. If $s(x) + s(C_{\leftarrow x}) < 9$, we get GS6 by placing x into C since $s(A) > 3$, so it must be true that $s(x) + s(C_{\leftarrow x}) > 9$, which means $s(x) > 5$. This is enough for us to place x into $B|_{22}$ (where it fits, otherwise we are in GS7) and reach GS3.

This contradiction gives us that at most one additional item j_2 can be packed into B during Step 10. We will now prove that even j_2 does not exist.

We split the analysis into two cases depending on which of j_2 and r arrives first.

Case 1. The item r is packed before j_2, meaning $s(B_{\leftarrow x}) = s(B_{\leftarrow r}) + s(j_2)$.

We start with inequalities (3.4), (3.7) and (3.10) in the following form:

$$9 + \frac{s(A_{\leftarrow r})}{2} < s(B_{\leftarrow r}) + s(r)$$
$$s(B_{\leftarrow r}) + s(j_2) + s(x) + s(A_{\leftarrow x}) < 15 + \frac{s(r)}{2}$$
$$s(B_{\leftarrow r}) + s(j_2) < s(A_{\leftarrow x}) + s(x) + \Big(4 - s(r)\Big)$$

We sum twice (3.4) with (3.7) and (3.10):

$$\begin{aligned}
&18 + s(A_{\leftarrow r}) + s(B_{\leftarrow r}) + s(j_2) + s(x) + s(A_{\leftarrow x}) + s(B_{\leftarrow r}) + s(j_2) \\
&< 2s(B_{\leftarrow r}) + 2s(r) + 15 + \frac{s(r)}{2} + s(A_{\leftarrow x}) + s(x) + 4 - s(r)
\end{aligned}$$

$$s(A_{\leftarrow r}) + 2s(j_2) < \frac{3s(r)}{2} + 1$$

Using $s(A_{\leftarrow r}) \geq s(A_{\leftarrow j})$ (r arrives after j) with $s(A_{\leftarrow j}) > 3$ from Observation 3.4 and $s(r) < 4$ gives us:

$$\begin{aligned}
3 + 2s(j_2) &< 7 \\
s(j_2) &< 2
\end{aligned}$$

which is a contradiction, since $s(A_{\leftarrow j_2}) < 4$ and j_2 did not fit into $A|_6$.

Case 2. In the remaining case, j_2 arrives before r, which means

$$s(B_{\leftarrow x}) = s(B_{\leftarrow r}) = s(B_{\leftarrow j}) + s(j) + s(j_2).$$

We start by summing (3.5) and (3.7). We get:

$$\begin{aligned}
s(B_{\leftarrow r}) + s(B_{\leftarrow x}) + s(x) + s(r) & \\
+ s(A_{\leftarrow x}) + s(A_{\leftarrow r}) &< 30 + \frac{s(r)}{2} \\
2s(B_{\leftarrow j}) + 2s(j) + 2s(j_2) + s(x) + s(r) & \\
+ s(A_{\leftarrow r}) + s(A_{\leftarrow x}) &< 30 + \frac{s(r)}{2}. \qquad (3.13)
\end{aligned}$$

Keeping (3.13) in mind for later use, we continue by considering (3.2), (3.3) and (3.12) in the following form:

$$\begin{aligned}
s(B_{\leftarrow j}) + s(j) &> 6 \qquad (3.14) \\
s(A_{\leftarrow j_2}) + s(j_2) &> 6 \qquad (3.15) \\
s(r) + s(x) + (4 - s(A_{\leftarrow x})) &> s(B_{\leftarrow x}) = s(B_{\leftarrow j}) + s(j) + s(j_2)
\end{aligned}$$

Summing the three inequalities gives us:

$$\begin{aligned}
s(B_{\leftarrow j}) + s(j) + s(A_{\leftarrow j_2}) + s(j_2) & \\
+ s(r) + s(x) + (4 - s(A_{\leftarrow x})) &> 12 + s(B_{\leftarrow j}) + s(j) + s(j_2) \\
s(r) + s(x) + (s(A_{\leftarrow j_2}) - s(A_{\leftarrow x})) &> 8 \\
s(r) + s(x) &> 8 \qquad (3.16)
\end{aligned}$$

Summing two times (3.14), two times (3.15) and once (3.16) gives us:

$$2s(B_{\leftarrow j}) + 2s(j) + 2s(j_2) + 2s(A_{\leftarrow j_2}) + s(r) + s(x) > 32. \qquad (3.17)$$

Using $s(A_{\leftarrow j_2}) \leq s(A_{\leftarrow r}) \leq s(A_{\leftarrow x})$ (which is only true here in Case 2, where

r arrived later) and recalling (3.13) along with (3.17), we get $30 + s(r)/2 > 32$ and $s(r) > 4$, which is a contradiction with r fitting into $C|_4$.

□

Lemma 3.18 (Lemma 3.12)**.** *Suppose that e and r are items as described in Definition 3.10 and suppose also that no good situation was reached during Step* 10 *of the algorithm* EVASIVE*. Then, $s(e) + s(r) \geq s(B_{\leftarrow j}) + s(r) > 6.8$.*

Proof. First of all, it is important to note that the item e may be packed on A or on B. Since either $B_{\leftarrow j}$, or $A_{\leftarrow j}$ contains solely e by Observation 3.4, we get that either $s(B_{\leftarrow j}) = s(e)$, or $s(B_{\leftarrow j}) \leq s(A_{\leftarrow j}) = s(e)$. Thus it is sufficient to prove $s(B_{\leftarrow j}) + s(r) > 6.8$.

We start the proof of $s(B_{\leftarrow j}) + s(r) > 6.8$ by restating (3.4), (3.8), and (3.9) in the following form:

$$\begin{aligned} s(B_{\leftarrow j}) + s(j) + s(r) &> 9 + \frac{s(A_{\leftarrow r})}{2} \\ s(B_{\leftarrow j}) + s(j) + s(x) &> 9 + \frac{s(A_{\leftarrow x}) + s(r)}{2} \\ s(B_{\leftarrow j}) + s(j) + (4 - s(r)) &> s(A_{\leftarrow x}) + s(x). \end{aligned}$$

Before summing up the inequalities, we multiply the first one by 8, the second by 2 and the third by 2. In total, we have:

$$\begin{aligned} 12s(B_{\leftarrow j}) + 12s(j) + 8 + 6s(r) + 2s(x) > 90 + 3s(A_{\leftarrow x}) + 4s(A_{\leftarrow r}) \\ + s(r) + 2s(x). \end{aligned}$$

We know that $s(B_{\leftarrow j}) \leq s(A_{\leftarrow x})$ and $s(B_{\leftarrow j}) \leq s(A_{\leftarrow r})$, allowing us to cancel out the terms:

$$5s(B_{\leftarrow j}) + 5s(r) + 12s(j) > 82.$$

Finally, using the bound $s(j) < 4$ and noting that $(82 - 48)/5 = 6.8$, we get

$$s(B_{\leftarrow j}) + s(r) > 6.8.$$

□

Lemma 3.19 (Lemma 3.13)**.** *Suppose that e and j are items as described in Definition 3.10 and suppose also that no good situation was reached by the algorithm* EVASIVE*. Then, $s(e) + s(j) \geq s(B_{\leftarrow j}) + s(j) > 7.6$.*

Proof. The same argument as in Lemma 3.12 gives us $s(e) + s(j) \geq s(B_{\leftarrow j}) + s(j)$. We therefore aim to prove $s(B_{\leftarrow j}) + s(j) > 7.6$. Summing up (3.8) and (3.11) and using $s(B_{\leftarrow x}) = s(B_{\leftarrow j}) + s(j)$, we get

$$2s(B_{\leftarrow j}) + 2s(j) + s(x) + 4 - s(A_{\leftarrow x}) > 9 + \frac{s(A_{\leftarrow x}) + s(r)}{2} + s(r) + s(x)$$

$$2s(B_{\leftarrow j}) + 2s(j) > 5 + \frac{3}{2}\Big(s(A_{\leftarrow x}) + s(r)\Big).$$

We now apply the bound $s(A_{\leftarrow x}) + s(r) \geq s(B_{\leftarrow j}) + s(r) > 6.8$, the second inequality being Lemma 3.12. We get:

$$2s(B_{\leftarrow j}) + 2s(j) > 5 + 10.2,$$

and finally $s(B_{\leftarrow j}) + s(j) > 7.6$, completing the proof.

□

Lemma 3.20 (Lemma 3.16)**.** *Suppose the algorithm* Evasive *reaches no good situation during Step* 10*. Then, after placing x into B in Step* 11*, it holds that $s(B) < 12.8$.*

Proof. As before, we will use our inequalities to derive the desired bound. As we have argued above, Lemma 3.12 gives us that $s(r) > 2.8$.

We sum up inequalities (3.7) and (3.11), getting:

$$\begin{aligned} s(B_{\leftarrow j}) + s(j) + 2s(x) + s(A_{\leftarrow x}) + s(r) &< 15 + \frac{s(r)}{2} + s(B_{\leftarrow j}) \\ &\quad + s(j) + 4 - s(A_{\leftarrow x}) \\ 2s(x) + 2s(A_{\leftarrow x}) &< 19 - \frac{s(r)}{2}. \\ s(x) + s(A_{\leftarrow x}) &< 9.5 - \frac{s(r)}{4}. \end{aligned}$$

To finish the bound we need $s(B_{\leftarrow j}) \leq s(A_{\leftarrow j}) \leq s(A_{\leftarrow x})$ (this is true because we reorder the bins B, A in Step 2), $s(r) > 2.8$ and $s(j) < 4$. Plugging them in, we get:

$$\begin{aligned} s(B) = s(B_{\leftarrow j}) + s(j) + s(x) &\leq s(A_{\leftarrow x}) + s(j) + s(x) \\ &< 9.5 - \frac{s(r)}{4} + s(j) < 9.5 - 0.7 + 4 < 12.8. \end{aligned}$$

□

4. Computer lower bounds

In this chapter, we focus on the algorithmic approach to computing lower bounds for Online Bin Stretching. We explain the technique of Gabay, Brauner and Kotov [15] as well as our extensions and its implementation.

An intermezzo with an anecdote. Before we continue with the explanation of the central ideas of our computer search, let us pause to explain the motivation of the thesis author to work on the lower bound algorithms.

While working together with Jiří Sgall, Rob van Stee and Pavel Veselý on the algorithmic results of Chapters 2 and 3, we have also investigated lower bounds for Online Bin Stretching. With some effort, the author of this thesis was able to prove a lower bound of $4/3 + \varepsilon$ for 3 bins and a very small value of ε.

To our surprise, another paper on Online Bin Stretching appeared while we were working on the problem, namely the one by Gabay, Brauner and Kotov [15] showing a much better lower bound of 19/14 using a clever minimax computer search and using a CSP solver to verify the optimum guarantee in every step. Gabay et al. were able to check all integer denominators up to 20, after which their computer program ran out of memory.

In this paper of Gabay et al. (but not in its current, updated version) we could find the following fateful sentence:

"The combinatorial explosion is very well illustrated in column `#nodes` *where we can see that even with many efficient cuts, we cannot tackle much larger problems."* [15]

Even though the sentence itself might be considered incorrect (seeing as we *can* tackle much larger problems, and in fact we will tackle problems up to denominator 82), the author of this thesis considers it quite ingenious. That one sentence turned a reasonably good result into a challenge and ignited the author's spark for computer lower bounds.

4.1 Minimax algorithm

Let us recall the bin stretching game, as defined in the introduction:

Definition 4.1. Fix a number of bins m and a stretching factor R. The *bin stretching game* is defined as follows:

1. It is a two-player game between Algorithm and Adversary, and the player Adversary starts.
2. The player Adversary moves by sending any item $i \in (0, 1]$ with the restriction that i along with all previous items can be packed into m bins of capacity 1.
3. The player Algorithm moves by selecting a bin where the item is packed.

The winning states are defined thus:

- A state is winning for ALGORITHM if all bins are packed strictly below R and ADVERSARY has cannot make any further moves.
- A state is winning for ADVERSARY if one bin has total load at least R.

Our main goal is learning whether there exists an algorithm for ONLINE BIN STRETCHING with stretching factor strictly below R, or whether every algorithm needs a stretching factor of at least R. This directly corresponds to existence of a winning strategy for the player ALGORITHM or ADVERSARY, respectively.

A "gold standard" algorithm for evaluating a given combinatorial two-player game is the *minimax algorithm*, which we can summarize as follows:

Algorithm 4 Algorithm MINIMAX(s) for a state s

```
1: if the state is defined as winning for any player, return.
2: for every move m of the current player do
3:     Construct the state t by applying the move m to the current state s.
4:     Run MINIMAX(t).
5:     if t is winning for the current player, return.
6: return the fact that the state s is winning for the other player.
```

As we also mentioned in the introduction, the two main obstacles to implementing MINIMAX for ONLINE BIN STRETCHING are the following:

1. ADVERSARY can send an item of arbitrary size;
2. ADVERSARY needs to make sure that at any time of the game, an offline optimum can pack the items arrived so far into m bins of capacity 1.

To overcome the first problem, it makes sense to create a sequence of games based on the granularity of the items that can be packed. The second problem increases the complexity of every game turn of the ADVERSARY, as it needs to run a subroutine to verify the guarantee for the next item it wishes to place.

4.2 Description of the finite game

We now formulate precisely the scaling parameter and other main concepts of the game we wish to solve. First of all, for scaling purposes, we rescale the guaranteed load of the optimum packing from 1 in the original definition to $S \in \mathbb{N}$. Therefore, if the algorithm must fill at least one bin to capacity R, the lower bound on the stretching factor will be R/S.

Next, we formalize what will be one state of our game:

Definition 4.2. Assume we fix three integral parameters: m (the number of bins), R (capacity of bins of the algorithm) and S (size of the optimal bin). Then, we define a *bin configuration* to be a pair $(\mathcal{L}, \mathcal{I})$, where

- $\mathcal{L}$ is a tuple of m non-negative integers, sorted in descending order. These denoting the current sorted loads of the bins.
- $\mathcal{I}$ is a multiset with ground set $\{1, 2, \ldots, S\}$ which contains the items used in the bins.

Additionally, for every bin configuration, it must hold:

- that there exists a packing of items from $\mathcal{I}$ into m bins with loads exactly in $\mathcal{L}$;
- that there exists a packing of items from $\mathcal{I}$ into m bins that does not exceed capacity S in any bin.

To illustrate the definition, consider a configuration $((4, 3, 2, 0), \{1, 1, 2, 2, 3\})$ – a situation where four items arrived, one bin is loaded to size 4, one to size 3 and one to size 2, leaving one more bin empty. Notice that the bin configuration does not capture the entire history of the game, but only its current state – indeed, we do not know in which order the items arrived, or even whether the bin of size 4 is filled as an item of size 3 plus an item of size 1, or 2 plus 2.

Losing information about history might seem like a bad idea at first, but it becomes more advantageous later, as it makes caching more efficient.

It is clear that every bin configuration is a valid state of the game with Adversary as the next player. We may also observe that the existence of an online algorithm for Online Bin Stretching implies an existence of an oblivious algorithm with the same stretching factor that has access only to the current bin configuration m and the incoming item i.

In the following easy observation, we make sure that bin configuration captures all we need for both players:

Observation 4.3. *Fix parameters m, R and S.*

1. *There exists a lower bound adversarial strategy for* Online Bin Stretching *if and only if there exists a strategy that receives a bin configuration and returns the next item in the lower bound.*

2. *There exists an algorithm for* Online Bin Stretching *if and only if there exists an algorithm which receives a bin configuration and a new item and returns a new bin configuration where the new item is packed.*

We are now able to formally define the game we investigate:

Definition 4.4. For a given $m \in \mathbb{N}, R \in \mathbb{N}$ and $S \in \mathbb{N}$, the *bin stretching game* $\mathrm{BSG}(m, R, S)$ is the following two-player game:

1. There are two players named Adversary and Algorithm. The player Adversary starts.

2. Each turn of the player Adversary is associated with a bin configuration $C = (\mathcal{L}, \mathcal{I})$. The start of the game is associated with the bin configuration $((0, 0, \ldots, 0), \emptyset)$.

3. The player Adversary receives a bin configuration C. Then, Adversary selects an integer i such that the multiset $\mathcal{I} \cup \{i\}$ can be packed by an offline optimum into m bins of capacity S. The pair (C, i) is then sent to the player Algorithm.

4. The player Algorithm receives a pair (C, i). The player Algorithm has to pack the item i into the m bins as described in C so that each bin has load *strictly less than* R. Algorithm then updates the configuration C into a new bin configuration, denoted C'. Algorithm then sends C' to the player Adversary.

5. If the player ALGORITHM receives a pair (C, i) such that it cannot pack the item according to the rules, the bin configuration C is won for player ADVERSARY.

6. If the player ADVERSARY has no more items i that it can send from a configuration C, the bin configuration C is won for player ALGORITHM.

Definition 4.5. We say that a game BSG(m, R, S) is a *lower bound* if the player ADVERSARY has a winning strategy when starting from the bin configuration $((0, 0, \ldots, 0), \emptyset)$.

4.3 The sequential algorithm

Our implemented algorithm is a parallel, multi computer implementation of the minimax game search algorithm. We now describe a pseudocode of its sequential version. The peculiarities of our algorithm (caching, pruning, parallelization) are described in the following sections.

One of the differences between our algorithm and the algorithm of Gabay et al. [15] is that our algorithm makes no use of alpha-beta pruning – indeed, as either ALGORITHM or ADVERSARY have a winning strategy from each bin configuration, there is no need to use this type of pruning.

Algorithm 5 Procedure EVALUATEADVERSARY

Input is a bin configuration $C = (\mathcal{L}, \mathcal{I})$.

1: **if** the configuration is cached (Section 4.5), **return** the value found in cache.
2: Create a list L of items which can be sent as the next step of the player ADVERSARY (Section 4.4).
3: **for** every item size i in the list L **do**
4: Recurse by running EVALUATEALGORITHM(C, i).
5: **if** EVALUATEALGORITHM(C, i) returns 0 (the configuration is winning for player ADVERSARY), stop the cycle and **return** 0.
6: **if** the evaluation reaches this step, store the configuration in the cache and **return** 1 (player ALGORITHM wins).

Algorithm 6 Procedure EVALUATEALGORITHM

Input is a bin configuration $C = (\mathcal{L}, \mathcal{I})$ and item i.

1: Prune the tree using known algorithms (Section 4.6.1).
2: **for** any one of the m bins **do**
3: **if** i can be packed into the bin so that its load is at most $R - 1$ **then**
4: Create a configuration C' that corresponds to this packing.
5: Run EVALUATEADVERSARY(C').
6: **if** EVALUATEADVERSARY(C') returns 1, **return** 1 as well.
7: **if** we reach this step, no placement of i results in victory of ALGORITHM; **return** 0.

Algorithm 7 Procedure SEQUENTIAL

Input is a bin configuration $C = (\mathcal{L}, \mathcal{I})$.

Fix parameters m, R, S.
Run EVALUATEADVERSARY(C).
if EVALUATEADVERSARY(C) returns 0 **then**
    return success (a lower bound exists).
else
    return failure.

4.4 Verifying the offline optimum guarantee

When we evaluate a turn of the ADVERSARY, we need to create the list $L = \{0, 1, \ldots, y\} \subseteq \{0, 1, \ldots, T\}$ of items that ADVERSARY can actually send while satisfying the ONLINE BIN STRETCHING guarantee. We employ the following steps:

Algorithm 8 Procedure MAXFEAS

Input is a bin configuration $C = (\mathcal{L}, \mathcal{I})$ with n items and m bins.

Calculate an upper bound UB on the value y.
Calculate a lower bound LB on the value y using an online best fit algorithm (Section 4.4.1).
if $LB < UB$ **then**
    Do a linear search on the interval $\{UB, UB-1, \ldots, LB\}$ using a procedure QUERY($\mathcal{I} \cup \{i\}$) that queries the cache whether $\mathcal{I} \cup \{i\}$ is feasible or not (Section 4.5).
    Update LB, UB to be the best values confirmed feasible/infeasible by QUERY.
if $LB < UB$ **then**
    Update LB using the (offline) algorithm BEST FIT DECREASING.
if $LB < UB$ **then**
    Compute the exact value of y using a dynamic program DYNPROGMAX (Section 4.4.2).
return $y = LB = UB$.

4.4.1 Upper and lower bounds

If we wish to compute the value y directly, we need to call the dynamic program DYNPROGMAX, whose complexity is $O(n \cdot S^m)$ in the worst case (and that is if we ignore potential slowdowns via hashing).

This is polynomial when m is a constant, but already for $m = 3$ and especially when $4 \leq m \leq 9$ such a call per every game state becomes prohibitively expensive. Therefore we employ several tricks that help us avoid computing DYNPROGMAX in some game states.

First, we compute a lower and upper bound on the maximum feasible item size y; if they match, we have found the correct value of y. The upper bound will be $UB := \min(y', m \cdot S - t)$, where y' is the maximum feasible value that was

computed in the previous vertex of ADVERSARY's turn, and t is the total size of all items in the instance. The second term is therefore the sum of all items that can arrive in this instance.

Online Best Fit. To find the first lower bound on y quickly, we employ an online bin packing algorithm ONLINE BEST FIT. This algorithm maintains a packing of items $\mathcal{I}$ to m bins of size S during the evaluation of the algorithm SEQUENTIAL, packing each item as it is selected by the player ADVERSARY. The algorithm ONLINE BEST FIT packs each item i into the most-loaded bin where the item fits.

Once the algorithm SEQUENTIAL selects a different item i' and evaluates a different branch of the game tree, the online algorithm removes i from its bin and inserts i' to the most-loaded bin where i' fits.

As ONLINE BEST FIT maintains just one packing, which may not be optimal, it can happen that ONLINE BEST FIT is unable to pack the next item i even though i is a feasible item. In that case, we mark the packing as inconsistent and do not use the lower bound from ONLINE BEST FIT until its online packing becomes feasible again.

If the packing maintained by ONLINE BEST FIT is still feasible, we return as the lower bound value LB the amount of unused space on the least-loaded bin.

The main advantage of ONLINE BEST FIT is that it takes at most $O(m)$ time per each step, and especially for the earlier stages of the evaluation its returned value can match the value of y.

Checking the cache. Next, if a gap still remains between LB and UB, we try to tighten it by calling a procedure QUERY which queries the cache of feasible and infeasible item multisets. The procedure has a ternary answer – either an item multiset $\mathcal{I} \cup \{j\}$ was previously computed to be feasible, or it was computed to be infeasible, or this item set is not present in the cache at all.

We update LB to be the largest value which is confirmed to be feasible, and update UB to be 1 less than the smallest value confirmed to be infeasible.

Best Fit Decreasing. If the values LB and UB are still unequal, we employ a standard offline bin packing algorithm called BEST FIT DECREASING. BEST FIT DECREASING takes items from $\mathcal{I}$ and first sorts them in decreasing order of their sizes. After that it considers each item one by one in this order, packing it into a bin where it "fits best" – where it minimizes the empty space of a bin. We can also interpret it as first sorting the items in decreasing order and then applying the algorithm ONLINE BEST FIT defined above.

As for its complexity, BEST FIT DECREASING takes in the worst case $O(m \cdot \mathcal{I})$ time. It does not need to sort items in $\mathcal{I}$, as the internal representation of $\mathcal{I}$ keeps the items sorted.

As with ONLINE BEST FIT, the lower bound LB will updated to the maximum empty space over all m bins, after BEST FIT DECREASING has ended packing. Such an item can always be sent without invalidating the ONLINE BIN STRETCHING guarantee.

4.4.2 Procedure DynprogMax

Procedure DynprogMax is a sparse modification of the standard dynamic programming algorithm for Knapsack. Given a multiset $\mathcal{I}, |\mathcal{I}| = n$ on input, our task is to find the largest item y which can be packed together with $\mathcal{I}$ into m bins (knapsacks) of capacity S each.

We use a queue-based algorithm that generates a queue Q_i of all valid m-tuples $(a, b, c, \ldots)$ that can arise by packing the first i items. We do not need to remember where the items are packed, only the loads of the bins represented by the m-tuple.

To generate a queue Q_{i+1}, we initialize it to be an empty queue. Next, we traverse the old queue Q_i and add the new item $\mathcal{I}[i+1]$ to all bins as long as it fits, creating up to m new tuples that need to be added to Q_{i+1}.

Unsurprisingly, we wish to make sure that we do not add the same tuple several times during one step. We can use an auxiliary $\{0, 1\}$ array for this purpose, but we have ultimately settled on a hash-based approach.

We use a small array A of 64-bit integers (of approximately $2^{10} - 2^{13}$ elements). When considering a tuple t' that arises from adding i to one of the bins in the tuple t, we first compute the hash $h(t')$ of the tuple t'. Since we use Zobrist hashing (see Section 4.5), this operation takes only constant time.

Next, we consider adding t' to the queue Q_{i+1}. We use the first $10 - 13$ bits of $h(t')$ (let f denote their value) and add t' to Q_{i+1} when $A[f] \neq h(t')$ – in other words, when the small array A contains something other than the hash of t' at the position f. We update $A[f]$ to contain $h(t')$ and continue.

While our hashing technique clearly can lead to duplicate entries in the queue, note that this does not hurt the correctness of our algorithm, only its running time in the worst case.

We continue adding new items to the tuples until we do n steps and all items are packed. In the final pass of the queue, we look at the empty space e in the least-loaded bin. The output of DynprogMax and the value of y is the maximum value of e over all tuples in the final pass.

Ignoring the collisions of the hashing scheme (which can happen but will not play a big role if we compute the expected running time based on our randomized hashing function), the time complexity of the procedure MaxFeas is quite high in the worst case: $\mathcal{O}(|\mathcal{I}| \cdot S^m)$.

Nonetheless, we are convinced that our approach is much faster than implementing MaxFeas using integer linear programming or using a CSP solver (which has been done in [15]) and contributes to the fact that we can solve much larger instances.

4.5 Caching

Our minimax algorithm employs extensive use of caching. We cache solutions of the dynamic programming procedure MaxFeas as well as any evaluated bin configuration C (as a hash) with its value.

Hash table properties. We store a large hash table of fixed size with each entry being a 64-bit integer corresponding to 63 bits of hash and a binary value.

The hash table is addressed by a prefix of the hash, usually between $20 - 30$ bits (depending on the computer used).

We solve the collisions by a simple linear probing scheme of a fixed length (say 4). In it, when a value needs to be inserted to an occupied position, we check the following 4 slots for an empty space and we insert the value there, should we find it. If all 4 slots are occupied, we replace one value at random.

Hash function. Our hash function is based on Zobrist hashing [29], which we now describe.

For each bin configuration, we count occurrences of items, creating pairs (i, f) belonging to $\{1, \ldots, S\} \times \{0, 1 \ldots, m \cdot S\}$, where i is the item type and f its frequency (the number of items of this size packed in all m bins).

As for the loads of the m bins, we maintain that they are sorted in descending order. We also think of them as ordered pairs (j, g), with j being the position of the bin in the ordering (e.g. 1 – largest, m – smallest) and g the actual value of the load.

For example, we can think of bin configuration $((3, 3, 2), \{1, 1, 1, 2, 3\})$ as a set of load pairs $(1, 3)$, $(2, 3)$, $(3, 2)$ along with pairs for items: $(1, 3)$, $(2, 1)$, $(3, 1)$, $(4, 0)$, $(5, 0)$ and so on.

At the start of our program, we associate a 64-bit number with each pair (i, f). We also associate a 64-bit number for each possible load of one bin. These two sets of numbers are stored as a matrix of size $S \times (m \cdot S)$ and a matrix of size $(R - 1) \times m$.

The Zobrist hash function is then simply a `XOR` of all associated numbers for a particular bin configuration.

The main advantage of this approach is fast computation of new hash values. Suppose that we have a bin configuration m with hash H. After one round of the player Adversary and one round of the player Algorithm, a new bin configuration B' is formed, with one new item placed.

Calculating the hash H' of B' can be done in time $\mathcal{O}(m)$, provided we remember the hash H; the new hash is calculated by applying XOR to H, the new associated values, and the previous associated values which have changed.

Caching of the procedure MaxFeas. We use essentially the same approach for caching results in the procedure MaxFeas, except only the m-tuple of loads needs to be hashed.

We also remark upon the values being cached in the procedure MaxFeas. At first glance, it seems that it might be best to store the value of y with each input multiset $\mathcal{I}$. However, this is a very bad idea, as we would lose upon a lot of symmetry.

Indeed, if we set i to be any item from the list $\mathcal{I}$, we would lose out on the fact that we know a lower bound on the largest value that can be sent for a multiset $\mathcal{I} \setminus \{i\} \cup \{y\}$ – namely $s(i)$, the value we know is compatible.

Instead, it is much better to cache binary feasibilities or infeasibilities for a specific multiset $\mathcal{I}$. We use these results to improve the values of LB and UB for other calls of procedure MaxFeas.

4.6 Tree pruning

Alongside the extensive caching described in Subsection 4.5, we also prune some bin configurations where it is possible to prove that a simple online algorithm is able to finalize the packing. Such a bin configuration is then clearly won for player Algorithm, as it can follow the output of the online algorithm.

4.6.1 Algorithmic pruning

Recall that in the game BSG(m, R, S), the player Algorithm is trying to pack all items into m bins with load at most $R-1$. If the search algorithm can quickly deduce that a bin configuration leads to a successful packing, we can immediately evaluate the configuration as winning for the player Algorithm and thus prune the tree.

This may remind us of the *good situations* of Chapter 3 (specifically Section 3.2). Indeed, in the case of 3 bins, we can use the good situations of Section 3.2 directly.

For general m, we restate some of the good situations for a general instance of BSG(m, R, S) such that $(R-1)/S \geq 1/3$. Recall that α is the extra space in a stretched bin; in the situations below, $\alpha = (R-1) - S$, as the player Algorithm is packing only to capacity $R-1$.

We omit the proofs of the good situations; they follow the same arguments as in the proofs in Section 3.2.

Good Situation 1. *Given a bin configuration $(\mathcal{L}, \mathcal{I})$ such that the total load of all but the last bin is at least $(m-1) \cdot S - \alpha$, there exists an online algorithm that packs all remaining items into m bins of capacity $R-1$.*

Good Situation 2. *Given a bin configuration $(\mathcal{L}, \mathcal{I})$ such that there exist two bins A, B such that $s(A) \leq \alpha$ and $s(\mathcal{L} \setminus \{A, B\}) \geq (m-2) \cdot S - 2\alpha - 1$, there exists an online algorithm that packs all remaining items into m bins of capacity $R-1$.*

Good Situation 3. *Consider a bin configuration $(\mathcal{L}, \mathcal{I})$. Define the following sizes:*

1. *Let s be the sum of loads of all bins excluding the last two.*
2. *Let r (the* last bin load requirement*) be the smallest load such that if the currently last bin B_m has load at least r, GS1 is reached (after reordering the bins).*
3. *Let o (the* overflow*) be defined as $R - r$.*

Then, if $r \leq R-1$ and:

- *either the second-to-last bin B_{m-1} has load at most α and above $(m-1) \cdot S - \alpha - o - s$;*
- *or the last bin B_m has load at most α but above $(m-1) \cdot S - \alpha - o - s$;*

there exists an online algorithm that packs all remaining items into m bins of capacity $R-1$.

Good Situation 4. *Suppose that for a bin configuration $(\mathcal{L}, \mathcal{I})$ it holds that the last two bins have load at most α.*

Define the following sizes:

1. *Let $g = (m-1) \cdot S - \alpha$ be the limit for GS1.*
2. *Let c (*critical value for B_{m-1}*) be equal to $g - (s(\mathcal{I}) - s(B_m))$. In other words, if an item of size in $[c, S]$ is packed into B_{m-1}, GS1 is reached (with B_m as the last bin).*

Next, for every item size i in $[1, c-1]$, define the following values

1. *For every bin $B_1, B_2, \ldots, B_{m-2}$, define the* virtual load *on a bin B_k as $\max(s(B_k), S + \alpha - (i-1))$.*
2. *For the last bin B_m, define the virtual load v on it as follows: If $S + \alpha - (c-2) \geq s(B_m) + s(i)$, we set $v = S + \alpha - (c-2)$. Otherwise, we set $v = s(B_m) + 2 \cdot s(i)$.*

Finally, if it is true that for every item size i in the interval $[1, c-1]$ the sum of virtual loads of bins $1, 2, \ldots, m-2, m$ is at least g, there exists an online algorithm that packs all remaining items into m bins of capacity $R - 1$.

We do realize that the last good situation is slightly more complicated than the rest; we now briefly comment on it.

First, the central idea of GS4 is that we can delay the computation of the total load as required by GS1 until there is an item that (seemingly) cannot be packed into any bin. If we can prove that at that time, GS1 is reached, we reach a contradiction and are able to finalize the packing. This motivates the definition of the virtual loads.

Second, the good situation GS4 is the first good situation with no fixed threshold, unlike its original form (GS4). Instead, it is *algorithmic* in the sense that we can check it for a bin configuration algorithmically and reach a good situation if all of its conditions are met.

4.6.2 Adversarial pruning

Compared to our fairly strong algorithmic pruning, we have only few tools to quickly detect that a bin configuration is winning for the player Adversary. More specifically, we use only the following two criteria:

Large item heuristic. Once any bin has load at least $R - S$, an item of size S packed into that bin would cause it to reach load R, which is a victory for the player Adversary. Suppose that the k-th bin reaches load $l \geq R - S$. We compute the size of the smallest item i such that

1. $s(i) + l \geq R$;
2. For any bin b in the interval $[(k+1), m]$ it holds that $s(i) + 2l \geq R$; in other words, Algorithm cannot pack two items of size l into any bin starting from the $(k+1)$-st.

Finally, we check if Adversary can send $m - k + 1$ copies of the item of size l. If so, it is a winning bin configuration for this player and we prune the tree.

Notice that there may be multiple different values of l for one bin configuration; for instance, in the setting of 19/14, for three bins with loads $11, 9, 0$, we should check whether we can send 2 items of size 10 or 3 items of size 8. Therefore, in the implementation, we compute for each bin its own candidate value of l and then check whether at least one is feasible using the dynamic programming test described in Section 4.4.

Five/nine heuristic. We use a specific heuristic for the case of 19/14, as it is a good candidate for a general lower bound. This heuristic was experimentally observed to slightly compress the size of the output tree in this setting.

This heuristic comes into play once there is a bin of load at least 5 and once all bins are non-empty (even load 1 is sufficient). The item sizes 5 and 9 are complementary in the sense that one of each can fit together in the optimal packing of capacity 14, but the two of them cannot be packed together into a bin that already has load at least 5.

A pair of items of size 9 also cannot fit together into any other bin – as all the bins have already load at least 1.

Finally, if there are too many bins of load at least 5 but not much more, a subsequent input of several items of size 14 will again force a bin of load at least 19.

We apply this heuristic only when it is true that at all times, m items of size 9 can arrive on input without breaking the adversarial guarantee. While this is true, it must be true that all bins are of load strictly less than 10.

Our heuristic considers repeatedly sending items of size 5. If at any point there are only p bins left with load strictly less than 5 and at the same time $p + 1$ items of size 14 can arrive on input, the configuration is winning for the player Adversary. On the other hand, if at any point there is a bin of load at least 10 and the invariant that m items of size 9 can still arrive holds, we are also in a winning state for Adversary.

If it is true that by repeatedly sending items of size 5 we eventually reach at least one of the aforementioned two situations, we mark the initial bin configuration as winning for the player Adversary.

A note on performance. While both of our heuristics reduce the number of tasks in our tree and the number of considered vertices, we were unable to evaluate them in every single vertex of the game tree without a performance penalty. Even the large item heuristic, which can be implemented with just one additional call to the dynamic programming procedures of Section 4.4 slows the program down considerably.

This is likely due to the fact that caching outputs of the dynamic programming calls of Section 4.4 leads to some vertices that do not need to call any dynamic programming procedure, and with our heuristics they are forced to call at least one.

4.7 Monotonicity

One of the new heuristics that enables us to go from a lower bound of 19/14 on 5 bins to 9 bins is iterating on lower bounds by monotonicity. We define it as follows:

Definition 4.6. A winning strategy for ADVERSARY has *monotonicity* k if it is true that for any two items i, j such that j is sent immediately after i, we have $s(j) \geq s(i) - k$.

Using this concept, we can iterate over k from 0 (non-decreasing instances) to $S - 1$ (full generality) to find the smallest value of monotonicity which leads to a lower bound, if any.

A potential downside of iterating over monotonicity is that it can introduce an S-fold increase in elapsed time in the case that no lower bound exists. Additionally, it is quite likely that monotonicity becomes less useful as the scaling factor T increases, as the item of relative size 1 gets smaller and smaller.

Still, solving decision trees of low monotonicity is much faster than solving the full tree, and we have empirically observed that lower bounds of lower monotonicity are fairly common; see Tables 4.1 and 4.2 for our empirical results.

Monotonicity caveats. There are two important notes regarding monotonicity that need to be discussed. First, it is now true that bin configuration is not enough to describe one state of the bin stretching game. To see this, consider monotonicity 1. If the first three input items are $1, 2, 3$, the next item needs to be of size 2 or larger. However, if the three input items are $1, 3, 2$ (which is permissible for monotonicity 1), the next item on input can be of size 1 and above. This means that the two states are not equivalent, even though their bin configuration is the same.

Notice that this problem is absent in the general case (where any item can be sent at any time, assuming it satisfies the optimum packing guarantee) and also in the case of monotonicity 0 (where the last item can be inferred from the bin configuration, as it is always the item with the largest size).

To remedy this, we mark in the bin configuration which item arrived last in the input sequence, which is sufficient for a fixed value of the monotonicity.

When increasing monotonicity by one and re-running our algorithm, we need to update our cached results (Section 4.5) as some previously losing bin configurations may become winning for the player ADVERSARY with a less monotone input. We could erase the entire cache, but it is not necessary – as any feasible bin configuration stays feasible and any bin configuration that is winning for ADVERSARY stays winning when monotonicity increases. Therefore, we only erase the infeasible configurations and the bin configurations winning for the player ALGORITHM.

The second caveat concerns combining other adversarial pruning (defined later in Section 4.6.2) with monotonicity.

When pruning the game tree, we enable all adversarial pruning tests including those that send items that would break the monotonicity constraints.

There is no structural problem when doing so, since monotonicity is only restricting ADVERSARY to reduce the tree size; giving ADVERSARY some power

back is therefore allowed. However, we must take this into account when reading the results of Tables 4.1 and 4.2. For example, a lower bound of monotonicity 1 may actually allow sending any item in the last few steps of the instance, provided this leads immediately to ALGORITHM packing one bin fully to its capacity S.

4.8 Parallelization

Up until now, we have described a single-threaded minimax algorithm with caching and pruning. We have used this single-threaded version in [26] to improve on the results of Gabay, Brauner and Kotov [15] and reach a lower bound of $45/33 = 1.\overline{36}$ for 3 bins and a lower bound of 19/14 for up to 5 bins.

In the pursuit of an improved bound for 3 bins as well as a general lower bound for m bins, we have subsequently implemented a parallel version of the minimax search algorithm.

Tasks. Our evaluation of the game tree proceeds in the following way: First, we start evaluating the game tree on the main computer (which we internally call *queen*) until a vertex corresponding to ADVERSARY's next move meets a certain threshold (for instance, sufficient depth). After that, we designate this adversarial vertex as a *task*.

Alongside the queen, we have processes whose job is to evaluate the tasks – we call them the *workers*. Workers which run on the same machine will have a common cache that they access via atomic primitives in order to maintain consistency. Workers on separate machines do not share information.

Due to the mixed environment of standard Unix threads and MPI processes, we also have a single *overseer* per each physical machine. This overseer handles the MPI communication as well as spawning the individual worker threads.

The tasks are all generated in advance by the queen. After that, their bin configurations are synchronized with all overseers running. The queen then assign tasks to overseers online, namely by assigning a batch of 250-500 tasks to an overseer. The overseer reports each value of a finished task immediately to the queen. When an overseer is finished processing a batch, it requests and receives a new one.

We have selected this communication strategy for two reasons:

1. To minimize congestion in the processing phase through the fact that the bin configurations are synchronized beforehand and only identifiers are shared in the online assignment phase.

2. To allow the queen to evaluate and prune unfinished tasks and therefore avoid some unnecessary processing by the workers.

Task selection. As mentioned above, an important decision to be made by the lower bound algorithm designer is where to split a vertex of the game tree into a task and send it to be processed in the parallel environment.

Based on our experiments, it seems that maintaining a right balance of the number of tasks as well as their running time is crucial to good performance. When the tasks are too shallow, the performance of the algorithm is dominated

by the elapsed time of the most difficult task in the list, which diminishes the gains coming from the parallel implementation.

On the other hand, if there are millions of tasks, the algorithms will still work correctly but we might lose performance from diminishing advantages of individual caching as well as due to pruning happening later in the process.

Previously, we have only used *task depth* as the principal guideline – when k items arrived on input (with k usually in the range of $\{4, 5, 6\}$), we mark the bin configuration as a new task.

However, experimenting with running time has shown us that the presence of a larger item makes the evaluation process faster a lot more than we assumed it will. Therefore, we have ultimately settled an a mixed task threshold function which takes into account both the task depth k and also the *task load* l, which is the sum of sizes of all items arrived so far in the instance. We split off a task when its task load is above l, and failing that when its task depth is below k.

We have settled on setting $k \in \{5, 6, 7\}$ and l to be around $20 - 40\%$ of the optimal bin capacity T. This way we get deeper bin configurations for very small items which experimentally seems to imply a shorter running time and a similar amount of tasks.

Saplings. Our implementation also allows us to pre-select some initial strategy for the player Adversary in advance. This way we can use our (so far limited) intuitive understanding of what is a good initial move and decrease the time needed to evaluate the whole tree.

A particularly good strategy for the lower bound of 19/14 seems to be sending an item of size 5 as the first item, followed by $m-1$ items of size 1. This adversarial strategy leads to a lower bound instance for $6, 7, 8$ and 9 bins.

We have therefore implemented a way to pre-select items to be sent in the first few rounds of the game. Given such a list of items, we compute all possible moves of the player Algorithm and create a queue of bin configurations that we each evaluate sequentially. We call such configurations *saplings*, as they are used to grow a full game tree.

The fact that already this linear, non-adaptive strategy of sending $5, 1, 1, 1, \ldots$ is enough to get a lower bound of 19/14 for 9 bins was a pleasant surprise to us. We believe this fact is due to the size of the sequence being already non-trivial (the item 5 alone occupies slightly more than 25% of one stretched bin).

A natural extension is to allow the user to input a partial game tree (an adaptive strategy for the player Adversary) and have the algorithm evaluate it sequentially; this can be easily added to our implementation once we learn more about which items should be the among the first to send.

Generating the game tree. So far we have only described how to learn whether a game tree is winning for the player Adversary (and thus is a valid lower bound) or winning for Algorithm. Naturally, our program does more than that: When a lower bound is found, it generates the game tree corresponding to the right choices of the player Adversary.

To facilitate this, our main evaluation process (the queen) creates a vertex representing all states in the top of the tree which it has in memory. Unlike the

worker processes, the queen does not use caching, and so the top of the tree is represented in its entirety.

As the tasks are evaluated by the workers, the queen receives information about winning and losing tasks, it prunes the top of the tree and removes the moves which are winning for ALGORITHM, meaning that at the end of the evaluation, only a small part of the original tree top remains – the one directly corresponding to a winning strategy for the player ADVERSARY.

After the evaluation is finished, we mark all the remaining vertices in the top of the tree as *fixed*, meaning we will not remove them or edit them anymore. We increase the values of k and l (the task load and the task depth above) and call the generating process again on the root of the tree.

When the generating process reaches a fixed vertex with outdegree non-zero, it will just move along the created out-edges and will not create a new ones.

Once it reaches a vertex with outdegree zero (a task in the previous iteration) it will again start generating all possible items allowed to be sent by the player ADVERSARY, up until the new task thresholds are reached. We send these tasks are sent to the worker processes again and we repeat our trimming and fixing process above.

Technology. We have settled on using a combination of OpenMPI [17] and standard thread library as provided by the C++ programming language. In our setting, OpenMPI is used to provide inter-computer communication API for sending and receiving tasks as described above. We employ the standard Unix threads to spawn the worker processes themselves; this way they can easily share one large cache for evaluated bin configurations.

We have originally considered using only OpenMPI processes for both inter-computer communication as well as memory sharing on one physical computer; this functionality is present in the latest version of the MPI standard, MPI-3.0. However, after implementing the shared memory functionality, we have noticed some slowdown of the worker processes when the shared memory was large (more than 1 gigabyte). This forced us into the heterogeneous model that we use right now.

4.9 Results

Tables 4.1 and 4.2 summarize our results. The paper of Gabay, Brauner and Kotov [15] contains results up to the denominator 20; we include them in the table for completeness. Results after the denominator 20 are new. Note that there may be a lower bound of size say $\frac{41}{30}$ even though none was found with this denominator; for example, some lower bound may reach 41/30 using item sizes that are not multiples of 1/30.

A note on 9 bins. We have produced a verifiable lower bound of 19/14 for the setting of 4-8 bins. The setting of 9 bins lies at the threshold of the computational power of our implementation – we have been able to successfully run the lower bound search without producing any output for $m = 9$, but as of the submission date we were not able to generate the verifiable lower bound tree.

Still, based on estimating the partial progress of the algorithm, we are convinced that we will be able to produce a verifiable lower bound of 19/14 for 9 bins within a few days. If that happens, we will update our data set at `https://github.com/bohm/binstretch/` to include it as well.

				Elapsed time	
Fraction	*Decimal*	*L. b.*	*Mon.*	*Linear*	*Parallel*
19/14	1.3571	Yes	0	2s.	
22/16	1.375	No		2s.	
26/19	1.3684	No		3s.	
30/22	$1.\overline{36}$	No		6s.	
33/24	1.375	No		5s.	
34/25	1.36	**Yes**	1	15s.	
41/30	$1.3\overline{6}$	No			
45/33	$1.\overline{36}$	**Yes**	1	1min. 48s.	
55/40	1.375	No		3min. 6s.	
56/41	1.3659	No		30min.	7s.
82/60	$1.3\overline{6}$	No			21 m. 49s.
86/63	1.36507	**Yes**	6		29s.
112/82	1.3659	**Yes**	8		3h. 21m. 31s.

Table 4.1: The results and performance of our linear and parallel computations for Online Bin Stretching with three bins. The results above the horizontal line were previously shown in [15], the rest are our results. The column *L. b.* indicates whether a lower bound was found when starting with the given stretching factor R/S as seen in column *Fraction*.
The column *Mon.* shows the lowest monotonicity that our program needs to find a lower bound. In the case of negative results, time measurements were done only using full generality, i.e. with monotonicity $S - 1$.
Some fractions below 112/82 are omitted; our lower bound computation has not found a lower bound on those.
The linear results were computed on a server with an AMD Opteron 6134 CPU and 64496 MB RAM. The size of the hash table was set to 2^{25}.
The parallel results were computed using OpenMPI on a heterogeneous cluster with 109 worker processes running.
The output of the program was not generated during the time measurements.

					Elapsed time	
Bins	*Fraction*	*Decimal*	*L. b.*	*Mon. (5)*	*Linear*	*Parallel (5)*
4	19/14	1.3571	**Yes**			18s.
4	30/24	$1.\overline{36}$	No			19s.
4	34/25	1.36	No			48s.
5	19/14	1.3571	**Yes**	2 (1)		10s.
6	19/14	1.3571	**Yes**	0 (0)		11s.
7	19/14	1.3571	**Yes**	1 (0)		2m. 13s. (16s.)
8	19/14	1.3571	**Yes**	Unk. (1)		(1h. 14s.)

Table 4.2: Results produced by our minimax algorithm for more than 3 bins. Tested on the same machine and with the same parameters as in Table 4.1, both for linear and parallel computations. In columns *Mon.* and *Parallel*, we list in brackets monotonicity and elapsed time of computation for an input having an item of size 5 at the start. Monotonicity is measured only starting with the second item.

4.10 Lower bound instance

As an illustration, we give a compact representation of our game tree for the lower bound of 45/33 for $m = 3$, which can be found on the next few pages.

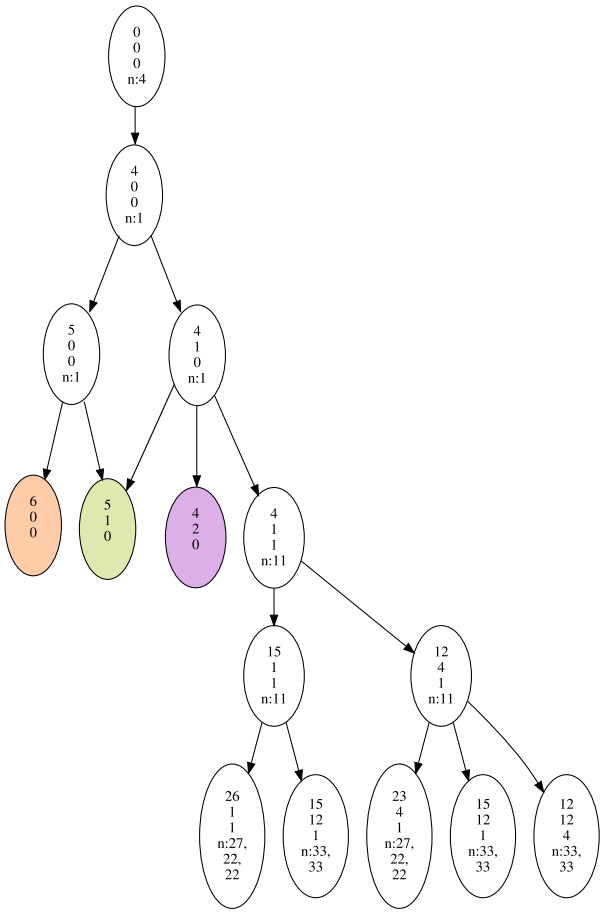

Figure 4.1: The beginning moves of the 45/33 lower bound, scaled so that $S = 33$ and $R = 45$. The vertices contain the current loads of all three bins, and a string `n:` i with i being the next item presented by the ADVERSARY. If there are several numbers after `n:`, the items are presented in the given order, regardless of packing by the player ALGORITHM. The colored vertices are expanded in later figures.

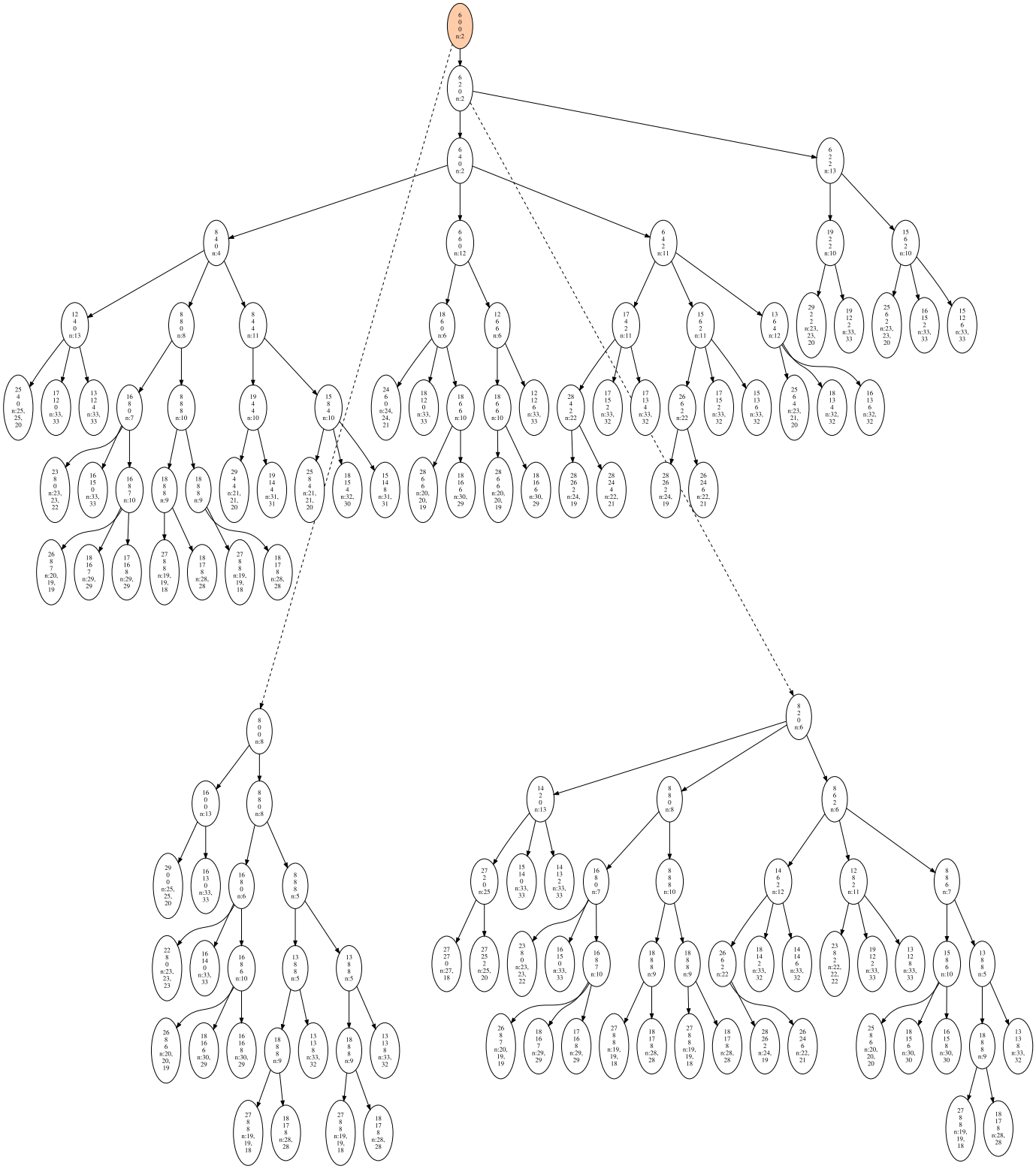

Figure 4.2: Game tree for the lower bound of 45/33, starting with the bin configuration $(6, 0, 0, \{4, 1, 1\})$.

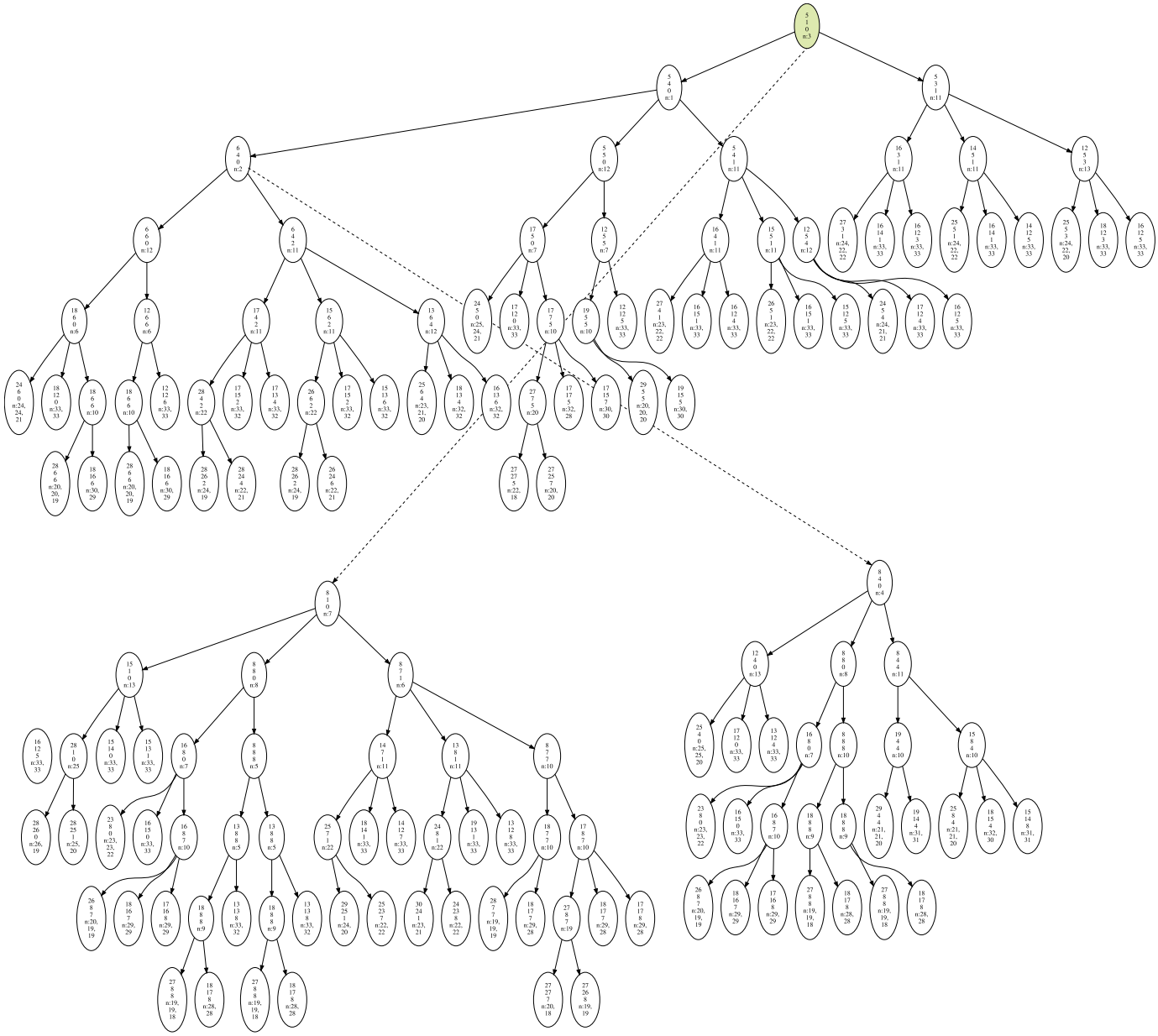

Figure 4.3: Game tree for the lower bound of 45/33, starting with the bin configuration $(5, 1, 0, \{4, 1, 1\})$.

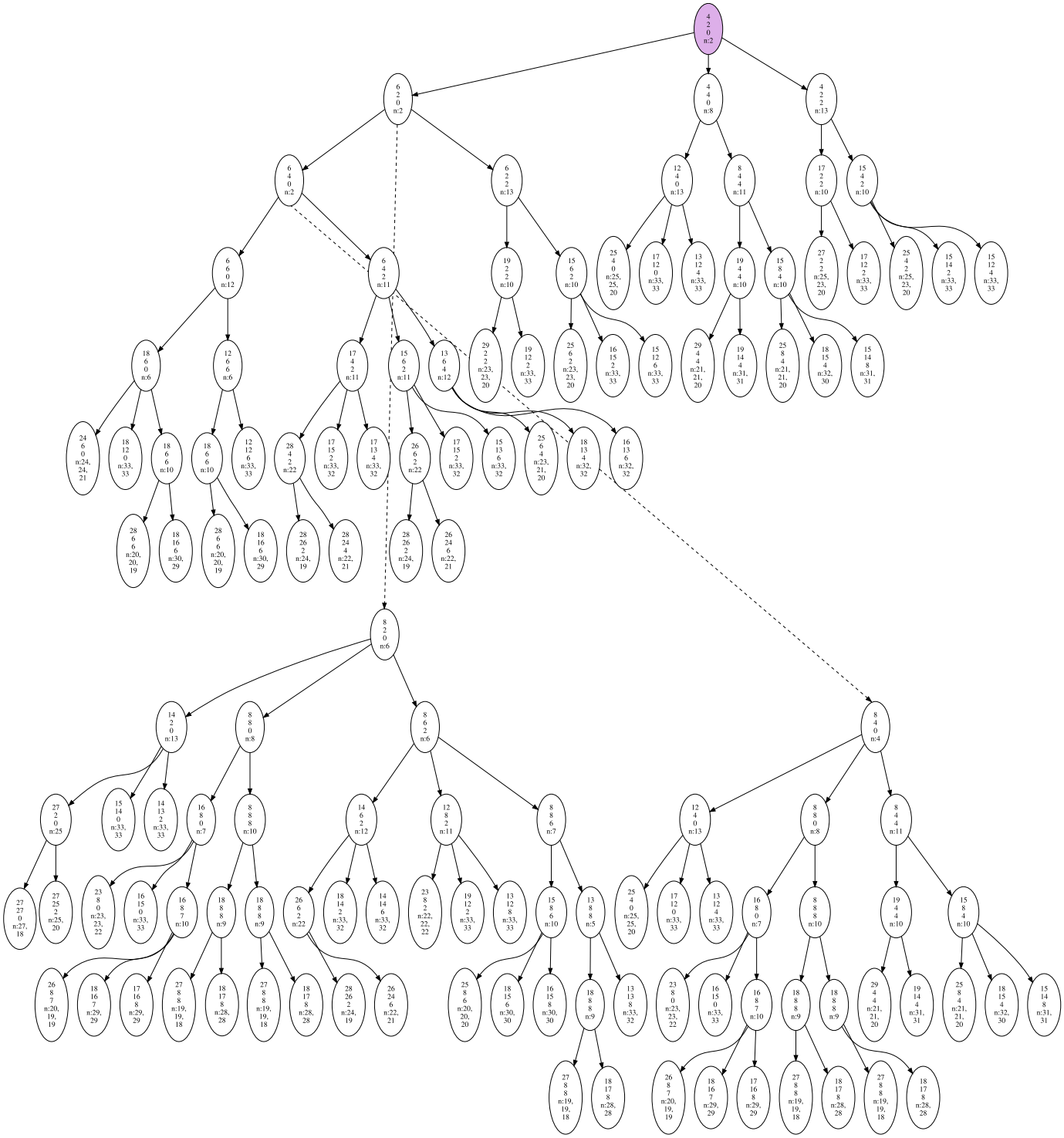

Figure 4.4: Game tree for the lower bound of 45/33, starting with the bin configuration $(4, 2, 0, \{4, 1, 1\})$.

4.11 Verification

For our lower bounds of 19/14 for $4 \leq m \leq 8$ we do not have a printable version of the game tree, as it contains too many vertices.

However, we include the lower bounds in the `graphviz` representation as part of our software appendix. We have published the lower bound software and the lower bound data at `https://github.com/bohm/binstretch/`.

Our lower bound implementation is quite large in terms of number of lines of code (8174 lines) and we cannot in good faith vouch for it to not contain any errors. At the same time, we do not expect the reader to check all the lower bound trees by hand, even though it is possible for a smaller tree such as the one in the next section.

To that end, we have implemented a simple independent C++ program which verifies that a given game tree is valid winning strategy for the player ADVERSARY and that all possible options of player ALGORITHM are present. While verifying our lower bound manually may be laborious, verifying the correctness of the C++ program should be manageable. The verifier is available along with the lower bound search code and lower bound data.

4.11.1 Game tree format

We now describe our output format for the lower bound directed acyclic graph that our implementation produces. This may be of use to those who wish to write an independent verification software.

The output format is a valid `graphviz` file format, which means it can be converted directly into a graphical representation using the open-source graphviz library using a command such as:

```
$ dot -Tpdf 45_33_3bins.dot > 45_33_3bins.pdf
```

The starting lines of the lower bound tree are the following:

```
1 strict digraph sapling1 {
2 overlap = none;
3 // 6: 5 1 1 1 1 1
```

The third line is technically a comment in the `graphviz` format; it contains the number of items already packed into the root of the game tree (or sapling, see Section 4.6.2). The number of items is then followed by an enumeration of such items.

The rest of the graph is the description of vertices (game states) and edges (transitions between game states). Every vertex in the output graph corresponds to a game state for the player ADVERSARY; the game states for the player ALGORITHM correspond to outgoing edges.

A typical vertex representation (in this case, a root) might be:

```
1 66 [label="5 2 1 1 1 0 0 0 n:4"];
2 66 -> 41530
3 66 -> 41529
4 66 -> 41528
5 66 -> 41248
```

The first number on each line is an *id* of the vertex. As these come from our lower bound algorithm (which will inevitably prune some vertices), the numbers might not be contiguous or starting from 0.

Notice that the root can also have an arbitrary id number as well; we assume that the root is always the first listed vertex in any output tree.

The first line of the vertex defines the loads of the bins of player ALGORITHM along with the next item to be sent (4 in our case).

The following lines define all outgoing edges from the vertex; as mentioned above, these correspond to the possible packings of the next item by the player ALGORITHM. It is guaranteed that the target vertex for every edge exists in the output description.

If the vertex is a heuristic vertex, its description may be one of the following:

```
1 107982 [label="9 5 5 4 4 4 2 1 h:FN (1)"];
2 107981 [label="9 5 5 5 4 4 2 0 h:(14,5)"];
```

These are the only two adversarial pruning heuristics present in the tree; see Section 4.6.2 for their specification. In the example above, the next move for the first vertex describes a *five-nine* heuristic that will send 1 item of size 5 and after that items of size either 9 or 14, as explained in Section 4.6.2).

The strategy listed in the vertex on line 2 is the *large item heuristic*. The two numbers describe the size of the incoming items and their amount. In our example, we see that sending 5 items of size 14 will force one bin to be loaded to at least 19.

As can be expected, the final line of the input is closing the curly bracket at the start.

Our output format is inherently designed for converting to a graphical form. A consequence of that is that there are no item lists in any of the vertices, only the next item that will be sent. Any verification algorithm is required to reconstruct the item lists from the tree structure.

5. Conclusion

5.1 Summary of results

We have focused single-mindedly on the online problem Online Bin Stretching which concerns itself with packing items into m bins of size R with additional knowledge that there exists some packing of the entire input into m bins of capacity 1. The problem is interesting by lacking any non-trivial general lower bounds, even though reasonably simple algorithms for it exist.

On the algorithmic side, we have seen a combination of the *classification and bunching* approach to design an online algorithm for any number of bins m with stretching factor 1.5 (Chapter 2). The analysis of our algorithm is tight, but we suspect that even lower stretching factor can be reached probably using different tools than just classification and bunching.

We have also shown a specialized algorithm for the case of the setting where there are always only three bins. This algorithm achieves a stretching factor $11/8 = 1.375$ by carefully balancing bins and reaching good situations as soon as possible.

Finally, we have designed and implemented a computer program which is able to search for lower bounds for Online Bin Stretching. We have built on the core idea of [15] but we add a plethora of new tools and implementation tricks which allowed us to far surpass the results of the previous teams, including a parallel implementation that can be run on a cluster of physical computers.

Using results from all three sections we have tightened the bounds on the best possible value of the stretching factor:

- For the case of arbitrary many bins, the lower bound/upper bound interval is now $[1.\overline{3}, 1.5]$;
- For the case of $m = 3$, it is $[1.369, 1.375]$;
- For the case of $4 \leq m \leq 8$, we have a new lower bound of 1.357.

Unfortunately (or fortunately for the future researchers in the area), the optimal stretching factor value for any fixed number of bins $m \geq 3$ still remains unknown.

5.2 What next?

We now move on to discuss several open questions related to Online Bin Stretching and problem areas which the author of this thesis believes to be interesting. If you are interested in any one of them, feel free to contact the author of this thesis for collaboration or for answering any questions.

5.2.1 The low-hanging fruit

First, let us list a few problems which we believe can be achieved in a month or two of work by an undergraduate or a graduate student.

1. *Use more computing power to get better lower bounds.* Our choice of a cluster of 109 cores was limited by the processing capabilities of our institute. Additionally, several of the used machines were desktops and we did not want to use them longer than overnight when they were not in use by colleagues.

   Why isn't it included in this thesis? We needed to stop tinkering with the lower bound search program and start writing the thesis at some point. We are quite convinced that one can get tighter bounds on the stretching factor for $3 \leq m \leq 10$ by adding more processing power or a few more pruning rules.

2. *Determine the exact stretching factor for three bins.* We now know that the best stretching factor achieved by any algorithm for Online Bin Stretching lies somewhere in the interval $[1.3659, 1.375]$. It would be nice to discover the best algorithm for this setting as well as a matching lower bound.

   It is always hard to guess what the right number should be, but based on the sequence of positive lower bound results, a good candidate (in the author's opinion) for the optimal stretching factor is $\frac{41}{30} = 1.3\overline{6}$.

   Why isn't it included in this thesis? To design an algorithm with a stretching factor below 1.375 will probably require either a new good situation or a fresh look at the algorithm design for Online Bin Stretching on 3 bins, perhaps using something other than the good situations that we introduce.

   We were originally hopeful that a lower bound of $\frac{41}{30}$ can be found within a reasonable time, but since we were not able to show it, we have not tried improving the algorithmic part either.

5.2.2 The sweeter fruit

We now list open questions which, in our opinion, are much more interesting to the online algorithm community as a whole, even though their solution probably requires more time or manpower compared to the ones above.

1. *Design a lower bound higher than 4/3 for any number of bins.* It is still true that the lower bound of 4/3 is the best known lower bound for the stretching factor of Online Bin Stretching with m bins (m being part of the input). Our experimental results seem to indicate that 19/14 is a good candidate, as it holds for $m \in \{3, 4, \ldots, 8\}$.

   Why isn't it included in this thesis? This problem was the main motivation for our extensions to the lower bound search algorithm. As it stands, the fact that we can get results for $m \leq 8$ is promising, but we need many more adversarial cuts and simplifications of the game tree before the lower bound tree for $m = 8$ became human-readable. Even then, we expect more fresh ideas to be needed before the general lower bound can be shown.

2. *Apply the same lower bound minimax approach to more problems.* We are not convinced that Online Bin Stretching is the only online problem where computer search can help finding better lower bounds. It can be

argued that many other online problems are notorious for the configuration space being very large and no reasonable simplification being known.

Why isn't it included in this thesis? Simply put, we have not yet found the right problem where our lower bound search technique would clearly produce new lower bounds. We will keep this technique in mind for other problems, and we kindly ask the reader to keep an eye for an application of this lower bound technique as well.

3. *Can machine learning help design good online algorithms?* There is much excitement in the broader programming community regarding the recent progress of machine learning / deep learning in designing winning strategies for positional and other two-player games (e.g. chess or poker).

   Already at the very beginning of this thesis (in Section 1.5) we have observed that every online algorithm corresponds to a strategy for a two-player game between Algorithm and Adversary. Our questions are: Can we use the recent improvements to machine learning to design either algorithms that perform well for an online problem such as Online Bin Stretching? Also, can we use the good (if not perfect) strategies produced by the machine learning algorithms to prune a lower bound tree faster, similar to what we do in Section 4.6?

   Why isn't it included in this thesis? We did not have enough machine learning/AI expertise to confidently answer this question in a positive or a negative way. We still believe it may be possible, but it certainly requires a researcher well-versed in current machine learning techniques.

Bibliography

[1] S. Albers. Recent advances for a classical scheduling problem. In International Colloquium on Automata, Languages, and Programming (pp. 4-14). Springer, Berlin, Heidelberg, 2013.

[2] S. Albers and M. Hellwig. Semi-online scheduling revisited. *Theor. Comput. Sci.*, 443:1–9, 2012.

[3] J. Aspnes, Y. Azar, A. Fiat, S. Plotkin, and O. Waarts. On-line load balancing with applications to machine scheduling and virtual circuit routing. *J. ACM*, 44:486–504, 1997.

[4] Y. Azar and O. Regev. On-line bin-stretching. In *Proc. of Randomization and Approximation Techniques in Computer Science (RANDOM)*, pages 71–81. Springer, 1998.

[5] Y. Azar and O. Regev. On-line bin-stretching. *Theor. Comput. Sci.* 268(1):17–41, 2001.

[6] P. Berman, M. Charikar, and M. Karpinski. On-line load balancing for related machines. *J. Algorithms*, 35:108–121, 2000.

[7] M. Böhm, J. Sgall, R. van Stee, and P. Veselý. Better algorithms for online bin stretching. In *Proc. of the 12th Workshop on Approximation and Online Algorithms (WAOA 2014)*, Lecture Notes in Comput. Sci. 8952, pages 23-34, Springer, 2015.

[8] T.C Cheng, H.Kellerer, H and V. Kotov. Semi-on-line multiprocessor scheduling with given total processing time. Theoretical Computer Science, 337(1-3), 134-146. Elsevier, 2005.

[9] E. Coffman Jr., J. Csirik, G. Galambos, S. Martello, and D. Vigo. Bin Packing Approximation Algorithms: Survey and Classification, In P. M. Pardalos, D.-Z. Du, and R. L. Graham, editors, *Handbook of Combinatorial Optimization*, pages 455–531. Springer New York, 2013.

[10] G. Dósa, A. Fügenschuh, Z. Tan, Z. Tuza, and K Węsek, Tight upper bounds for semi-online scheduling on two uniform machines with known optimum. Central European journal of operations research, 26(1), 161-180, 2018.

[11] T. Ebenlendr, W. Jawor, and J. Sgall. Preemptive online scheduling: Optimal algorithms for all speeds. *Algorithmica*, 53:504–522, 2009.

[12] L. Epstein. Bin stretching revisited. Acta Informatica, 39(2), 97-117, 2003.

[13] L. Epstein. A survey on makespan minimization in semi-online environments. Journal of Scheduling, 21(3), 269-284, 2018.

[14] R. Fleischer, and M. Wahl. Online scheduling revisited. Journal of Scheduling 3, 343–353, 2000.

[15] M. Gabay, N. Brauner, V. Kotov. Computing lower bounds for semi-online optimization problems: Application to the bin stretching problem. HAL preprint hal-00921663, version 2, 2013.

[16] M. Gabay, N. Brauner, V. Kotov. Improved Lower Bounds for the Online Bin Stretching Problem. HAL preprint hal-00921663, version 3, 2015.

[17] E. Gabriel, G. E. Fagg, G. Bosilca et al. Open MPI: Goals, Concept, and Design of a Next Generation MPI Implementation. In Proceedings, 11th European PVM/MPI Users' Group Meeting, Budapest, Hungary, 2004.

[18] R. L. Graham. Bounds for certain multi-processing anomalies. Bell System Technical Journal 45, 1563–1581, 1966.

[19] R. L. Graham. Bounds on multiprocessing timing anomalies. *SIAM J. Appl. Math.*, 17:263–269, 1969.

[20] D. Johnson. *Near-optimal Bin Packing Algorithms*. Massachusetts Institute of Technology, project MAC. Massachusetts Institute of Technology, 1973.

[21] H. Kellerer and V. Kotov. An efficient algorithm for bin stretching. *Operations Research Letters*, 41(4):343–346, 2013.

[22] H. Kellerer, V. Kotov, M. G. Speranza, and Z. Tuza. Semi on-line algorithms for the partition problem. *Oper. Res. Lett.*, 21:235–242, 1997.

[23] K. Pruhs, J. Sgall, and E. Torng. Online scheduling. In J. Y.-T. Leung, editor, *Handbook of Scheduling: Algorithms, Models, and Performance Analysis*, chapter 15, pages 15–1 – 15–41. CRC Press, 2004.

[24] J. F. Rudin III. Improved bounds for the on-line scheduling problem. Ph.D. Thesis. The University of Texas at Dallas, 2001.

[25] J. Ullman. The Performance of a Memory Allocation Algorithm. *Technical Report 100*, 1971.

[26] M. Böhm. Lower Bounds for Online Bin Stretching with Several Bins. Student Research Forum Papers and Posters at SOFSEM 2016, CEUR WP Vol-1548, 2016.

[27] M. Böhm, J. Sgall, R. van Stee, and P. Veselý. A Two-Phase Algorithm for Bin Stretching with Stretching Factor 1.5. ArXiv preprint arXiv:1601.08111v2, 2016.

[28] M. Gabay, V. Kotov, N. Brauner. Semi-online bin stretching with bunch techniques. HAL preprint hal-00869858, 2013.

[29] Zobrist, Albert L. A new hashing method with application for game playing. ICCA journal 13.2: 69-73. 1970.

[30] E. Angelelli, A. B. Nagy, M. G. Speranza and Z. Tuza. The on-line multiprocessor scheduling problem with known sum of the tasks. Journal of Scheduling, 7(6), 421-428, 2004.

List of figures

List of publications

Journal papers

[1] M. Bienkowski, M. Böhm, Ł. Jeż, P. Laskoś-Grabowski, J. Marcinkowski, J. Sgall, A. Spyra, and P. Veselý. Logarithmic price of buffer downscaling on line metrics. Theoretical Computer Science, 707: 89-93, 2018.

[2] M. Böhm, P. Veselý. Online Chromatic Number is PSPACE-Complete. Theory Comput. Syst. 62(6): 1366-1391, 2018.

[3] M. Böhm, J. Sgall, R. van Stee, P. Veselý. A two-phase algorithm for bin stretching with stretching factor 1.5. J. Comb. Optim. 34(3): 810-828, 2017.

[4] M. Böhm, J. Sgall, R. van Stee, Pavel Veselý. Online bin stretching with three bins. J. Scheduling 20(6): 601-621, 2017.

[5] M. Böhm, G. Dósa, L. Epstein, J. Sgall, and P. Veselý. Colored Bin Packing: Online Algorithms and Lower Bounds. Algorithmica, Springer, 2016. ISSN: 0178-4617.

Papers in conference proceedings

[1] N. Bansal, M. Böhm, M. Eliáš, G. Koumoutsos, and S. W. Umboh. Nested Convex Bodies are Chaseable. In Proceedings of SODA 2018, pp. 1253-1260, 2018.

[2] M. Böhm, Ł Jeż, J. Sgall, and P. Veselý. On Packet Scheduling with Adversarial Jamming and Speedup. In Proc. of the 15th Workshop on Approximation and Online Algorithms (WAOA 2017), pp. 190-206, 2017.

[3] M. Bienkowski, M. Böhm, J. Byrka, M. Chrobak, C. Dürr, L. Folwarczný, Ł. Jeż, J. Sgall, N. K. Thang, and P. Veselý. Online Algorithms for Multi-Level Aggregation. In Proceedings of the 24th Annual European Symposium on Algorithms (ESA 2016), Leibniz International Proceedings in Informatics (LIPIcs), pp. 12:1 – 12:17, 2016.

[4] M. Böhm, M. Chrobak, Ł. Jeż, F. Li, J. Sgall, and P. Veselý. Online Packet Scheduling with Bounded Delay and Lookahead. In proceedings of the 27th International Symposium on Algorithms and Computation (ISAAC 2016), 2016.

[5] M. Böhm. Lower Bounds for Online Bin Stretching with Several Bins. In proceedings of SOFSEM 2016 (Student Research Forum Papers / Posters): 1-12, 2016.

[6] M. Böhm, P. Veselý. Online Chromatic Number is PSPACE-Complete. In Proceedings of IWOCA 2016: 16-28, 2016.

[7] M. Böhm, J. Sgall, and P. Veselý. Online Colored Bin Packing. In Proceedings of WAOA 2014, LNCS, Springer, 2015.

[8] M. Böhm, J. Sgall, R. van Stee, P. Veselý. Better Algorithms for Online Bin Stretching. In Proceedings of WAOA 2014: 23-34, 2015.