Fully-Dynamic Bin Packing with Limited Repacking

Anupam Gupta*1, Guru Guruganesh*1, Amit Kumar*2, and David Wajc*1

1Carnegie Mellon University
2IIT Delhi

“To improve is to change; to be perfect is to change often.”
– Winston Churchill.

Abstract

We consider the bin packing problem in a fully-dynamic setting, where new items can arrive and old items may depart. The objective here is to obtain algorithms achieving low asymptotic competitive ratio while changing the packing sparingly between updates. We associate with each item \( i \) a movement cost \( c_i \geq 0 \). We wish to achieve good approximation guarantees while incurring a movement cost of \( \beta \cdot c_i \), either in the worst case, or in an amortized sense, for \( \beta \) as small as possible. We refer to this \( \beta \) as the recourse of the algorithm.

We obtain tight or near-tight results for this problem for the following settings:

1. For general movement costs (the \( c_i \)s are unconstrained), we show that with constant recourse one can almost match the best asymptotic competitive ratio of online bin packing (i.e., incremental and without recourse). We then show a complementary lower bound: fully-dynamic bin packing with small recourse is at least as hard as online bin packing.

2. For unit movement costs (\( c_i = 1 \) for all \( i \)), we use an LP-based approach to show a sharp threshold of \( \alpha \approx 1.3871 \). Specifically, we show that for any \( \epsilon > 0 \), any algorithm with asymptotic competitive ratio \( \alpha - \epsilon \) must suffer either polynomially large additive terms or recourse. On the positive side, for any \( \epsilon > 0 \), we give an algorithm with asymptotic competitive ratio of \( \alpha + \epsilon \), with an \( \Theta(\epsilon^{-2}) \) additive term and recourse \( O(\epsilon^{-2}) \).

3. For volume movement costs (\( c_i = s_i \) for all \( i \)), we show a tight tradeoff between competitive ratio and amortized recourse (which is often called migration factor in this context). Specifically, we show that for an algorithm to have asymptotic competitive ratio \( 1 + \epsilon \), an amortized recourse of \( \Theta(\epsilon^{-1}) \) is both necessary and sufficient.

Hence, for amortized recourse, our work gives nearly-matching upper and lower bounds for all these three natural settings. Our last two results add to the small list of problems for which tight or nearly-tight tradeoffs between amortized recourse and asymptotic competitive ratio are known.

*This work was done in part while the authors were visiting the Simons Institute for the Theory of Computing.
1 Introduction

An instance $\mathcal{I}$ of the Bin Packing problem consists of a list of $n$ items of sizes $s_1, s_2, \ldots, s_n \in [0, 1]$. The objective of a bin packing algorithm is to pack the items of $\mathcal{I}$ into a small number of unit-sized bins. Let $OPT(\mathcal{I})$ be the minimum number of unit-sized bins needed to pack all these items. This classic NP-hard optimization problem has been studied since the 1950s, with thousands of papers addressing this problem, variations and generalizations; see [17] Chapter 2 for an early survey, and [7] for a more up-to-date one. Much of this work (e.g. [3, 15, 20, 22, 23, 24, 26, 30, 31, 32]), starting with the seminal work of Ullman [29], studies the online setting, where items arrive sequentially and must be packed into bins immediately and irrevocably. However, the online problem is strictly harder than the offline version, due to the lack of information about the future: while the offline setting can be approximated to within a small additive term of $O(\log OPT)$ (see, e.g. [16]), in the online setting there is at least a 1.5403-multiplicative gap between the algorithm and $OPT$ in the worst case, even as $OPT \rightarrow \infty$ [3].

Given the wide applicability of the online problem, researchers have considered the problem where a “small” number of repackings is allowed. I.e., how well can we perform if we allow items to be moved between bins when new items arrive and old ones depart? Clearly, some repacking is necessary; to make this question non-trivial, we demand bounded “recourse”, i.e., items should be moved sparingly. Formally, a fully-dynamic bin packing algorithm maintains at every time $t$, a feasible solution to the bin packing instance $\mathcal{I}_t$ given by items inserted and not yet deleted until time $t$. Every item $i$ has a size $s_i \in [0, 1]$, and a movement cost $c_i$ which the algorithm pays every time item $i$ is moved between bins.

**Definition 1.1.** A fully-dynamic algorithm $\mathcal{A}$ has (i) an asymptotic competitive ratio (a.c.r) $\alpha$, (ii) additive term $a$ and (iii) recourse $\beta$, if at each time $t$ it packs the instance $\mathcal{I}_t$ using at most $\alpha \cdot OPT(\mathcal{I}_t) + a$ bins, with an independent of $OPT(\mathcal{I}_t)$, while the total movement cost until time $t$ is at most $\beta \cdot \sum_{i=1}^{t} c_i$. If at each time $t$ algorithm $\mathcal{A}$ incurs at most $\beta \cdot c_i$ movement cost, where $c_i$ is the cost of the element added or deleted at time $t$, we say algorithm $\mathcal{A}$ has worst case recourse $\beta$, otherwise we say it has amortized recourse $\beta$.

The goal is now to design algorithms which simultaneously have low asymptotic competitive ratios (a.c.r), additive terms, and recourse. There is a natural tension between the a.c.r and the recourse, so we want to find the optimal a.c.r-to-recourse trade-offs.

Gambosi et al. [13, 14] were the first to study dynamic bin packing algorithms. They gave a $4/3$-a.c.r algorithm for the insertion-only setting which only moves every item a constant number of times throughout the algorithm’s run, which in our terminology translates to constant amortized recourse for general movement costs. For unit movement costs, Ivković and Lloyd [15] and Balogh et al. [4, 5] gave lower bounds of $4/3$ and 1.3871 on the a.c.r of algorithms with constant recourse, respectively; Balogh et al. [5] also presented an algorithm with a.c.r tending to $3/2$ as the worst-case number of movements increases to infinity. In 2009, following Sanders et al. [27] who studied makespan minimization with bounded recourse, motivated by questions in sensitivity analysis, Epstein et al. [10] re-introduced the dynamic bin packing problem for the case where the movement cost $c_i$ of each item equals its size $s_i$ (the weight cost setting). They showed that bounded (though exponential in $\epsilon^{-1}$) recourse suffices to maintain a solution with a.c.r $(1 + \epsilon)$. This was improved by Jansen and Klein [19] who showed that for insertion-only algorithms, a polynomial dependence on $\epsilon^{-1}$ suffices; Berndt et al. [6] showed the same for fully-dynamic algorithms. They further showed that any $(1 + \epsilon)$ a.c.r algorithm must have worst-case recourse of $\Omega(\epsilon^{-2})$. While these give strong, nearly-tight results in the case of weight costs $c_j = s_j$, the unit costs $c_i = 1$ and general costs cases are not so well understood.

\footnote{Some work in this area uses the term migration factor instead of recourse in the setting of weight movement cost; we use the term recourse for brevity, and to emphasize the broader set of movement costs considered here.}
1.1 Our Results

We fully characterize the recourse to asymptotic competitive ratio trade-off for fully-dynamic BIN PACKING under general movement costs, unit movement costs and weight costs. Our results are summarized in the following five theorems.\(^2\) In all these results, we use the term online BIN PACKING to refer to the classical insertion-only model without any repacking, and fully-dynamic BIN PACKING (with recourse) to refer to the model with both insertions and deletions where we can repack items.

1.1.1 General Costs

The results of \([6, 10, 19]\) show that in the weight cost model a little recourse can circumvent the negative effects of online arrivals, and allow for arbitrarily good a.c.r. Does such a claim hold even for general costs? Our first result shows that even allowing for repacking, the fully-dynamic BIN PACKING problem is no easier than the arrival-only online problem (with no repacking).

**Theorem 1.2** (Fully Dynamic as Hard as Online). Any fully-dynamic BIN PACKING algorithm with bounded recourse under general movement costs has asymptotic competitive ratio at least as high as that of any online BIN PACKING algorithm. Given current bounds, this is at least 1.54037.

Now the concern is: since elements can both arrive and depart, is it conceivable that the fully-dynamic model is *harder* than the online one, even allowing for recourse? We show this is likely not the case, as we can almost match the a.c.r of the current-best algorithm for online BIN PACKING.

**Theorem 1.3** (Fully Dynamic as Easy as Online). Any algorithm in the Super Harmonic family of algorithms can be implemented in the fully-dynamic setting under general movement costs, yielding the same a.c.r. This implies an algorithm with a.c.r of 1.58889 using \([26]\).

We note that if all items have size at least some \(\Omega(1)\), our recourse bound becomes worst case, without harming the approximation ratio. Alternatively, we can obtain the same asymptotic competitive ratio with worst case recourse if we allow for the additive approximation term to increase to \(O(\log n)\).

The current best online BIN PACKING algorithm, \([15]\), is not from the Super Harmonic family, but is closely related to them. It has an a.c.r of 1.5815, so our results for fully-dynamic BIN PACKING are within a hair’s width of the best bounds known for online BIN PACKING.\(^3\)

1.1.2 Unit Costs

We now consider the most natural of all movement costs, that of *unit costs*; i.e., \(c_i = 1\) for all items \(i\). For this model we give tight upper and lower bounds. Let \(\alpha = 1 - \frac{1}{W_{-1}(-2/e^2)+1} \approx 1.3871\) (here \(W_{-1}\) is the lower real branch of the Lambert \(W\)-function \([8]\)). Balogh et al. \([1]\) showed \(\alpha\) is a lower bound on the a.c.r of all fully-dynamic BIN PACKING algorithms with constant recourse. We present a simpler proof of this lower bound, and show that surpassing this a.c.r requires either polynomial additive term or recourse (or both). Moreover, we present an algorithm proving \(\alpha\) is tight for this problem.

**Theorem 1.4** (Unit Costs Tight Bounds). For any \(\varepsilon > 0\), there exists a fully-dynamic BIN PACKING algorithm with a.c.r \((\alpha + \varepsilon)\), additive term \(O(\varepsilon^{-2})\) and amortized recourse \(O(\varepsilon^{-2})\) under unit movement costs. Conversely, for any fully-dynamic BIN PACKING algorithm with a.c.r \((\alpha - \varepsilon)\), the product of the additive term and the amortized recourse cost must be \(\Omega(\varepsilon^4 \cdot n)\) under unit movement costs.

\(^{2}\)A tabular listing of our results contrasted with the best previous results appears in Appendix \(A\).

\(^{3}\)Recently, a new paper presenting an 1.578-algorithm was posted on ArXiv \([2]\) (and questioning previous improvements on the Super Harmonic family of algorithms). It remains an open question as to whether our techniques can be extended to achieve the improved a.c.r bounds while maintaining constant recourse.
These complementary results give us a sharp threshold on the a.c.r of any algorithm for the unit costs model. This is the technical heart of the paper: we show both upper and lower bounds using linear programming techniques. We give a linear program that completely captures the performance of the algorithm. Indeed, we first use this LP as a gap-revealing LP, to prove that a certain family of instances give an a.c.r of at least $\approx \alpha$. Then we use it as a factor-revealing LP to show that our algorithm achieves an a.c.r at most $\approx \alpha$.

If all the items are small then our bounds are worst case recourse bounds; however, currently our solution for large items requires amortized recourse bounds.

### 1.1.3 Weight Costs

Finally, we give an extension of the already strong results known for the weight cost model. We show that the lower bound known in the worst-case recourse model extends to the amortized model as well, for which it is tight.

**Theorem 1.5 (Weight Costs Tight Bounds).** For any $\varepsilon > 0$, there exists a $(1 + \varepsilon)$-a.c.r algorithm with constant additive term and $O(\varepsilon^{-1})$ amortized recourse under weight movement costs.

Conversely, there exist infinitely many $\varepsilon > 0$ such that any $(1 + \varepsilon)$-a.c.r algorithm requires $\Omega(\varepsilon^{-1})$ amortized recourse under weight movement costs.

Note that the hardness result of the last theorem was only known for worst-case recourse ([6]); this previous lower bound consists of a hard instance which effectively disallowed any recourse, while lower bounds against amortized recourse can of course not do so. In the context of sensitivity analysis, our lower bound shows that in order to maintain a near-optimal, i.e. a $(1 + \varepsilon)$-asymptotically competitive solution at least $\Omega(\varepsilon^{-1})$ volume times the volume of items added and removed must be moved, even in an amortized sense.

### 1.2 Techniques and Approaches

**General Costs.** To show that any online lower bound implies a lower bound for general costs, we consider an online lower bound, and give items movement costs which drop sharply as the number of elements in the system increase. This drop in element costs means the total recourse budget essentially does not grow. Now, if the algorithm moves an element $e$ from the past from bin $i$ to bin $j$, we can delete all the elements given since $e$ was added, and proceed with the lower bound based on the assignment $e \mapsto j$. A careful analysis implies that for some list of items the dynamic algorithm effectively moves no items, effectively making this algorithm an online algorithm, yielding the claimed bound.

For our algorithmic results, we show how to emulate any algorithm in the Super Harmonic class of algorithms [26]. A super harmonic (SH) algorithm partitions large items into two classes (red and blue), and packs similarly-sized items in groups, with at most two groups per bin, one of each color. Finally, an SH algorithm packs small items using FirstFit into bins which are $1 - \epsilon$ full. Our simulation of SH algorithms stems from the observation that at its heart, the analysis for SH relies on maintaining a stable matching in an appropriate compatibility graph. We show that these invariants can be maintained, in a fully-dynamic setting with $O(1)$-recourse. Our algorithm for large items uses a subroutine similar to the Gale-Shapley algorithm.

Our simulation of SH algorithms also requires an efficient solution for packing similarly-sized items. The idea of this simple algorithm is to sort items by their cost (perhaps rounding costs to powers of two), such that any insertion or deletion of an item of this size can be “fixed” by moving at most one

---

4 Independently of this work, Feldkord et al. [12] obtained worst case bounds also for large items, yielding a worst case bound matching the optimal asymptotic competitive ratio.
item of each smaller cost, yielding constant worst case recourse. The final ingredient of our algorithm is packing of small items into \((1 - \epsilon)\)-full bins. Unlike the online algorithm, which does so trivially by using FirstFit, it is not so easy in the fully-dynamic case. Our algorithm for small items extends the ideas of Berndt et al. [6] for the weight cost case, namely bucketing bins into buckets of \(\Theta(1/\epsilon)\) many bins, such that all but the last bin in a bucket are \(1 - O(\epsilon)\) full, which guarantees these bins are \(1 - O(\epsilon)\) full on average. For weight costs, bucketing bins and sorting the small items by size readily yields an \(O(\epsilon^{-1})\) recourse bound. For general costs, this is more intricate, as size and cost need not be commensurate. To overcome this, in addition to this bucketing scheme, we maintain the small items in sorted order according to their size to movement cost ratio (also known as their smith ratio). Only moving items to/from a bin once it has \(\Omega(\epsilon)\)'s worth of volume removed/inserted (keeping track of erased, or “ghost” items), allows us to achieve amortized \(O(\epsilon^{-2})\) recourse cost for small items under general movement costs, completing our fully-dynamic simulation of SH algorithms.

**Unit Costs.** For the unit costs model, our lower bound is based on a natural instance, previously used by Balogh et al. [4], consisting of small items, to which large items of various sizes can be repeatedly added and removed. This is done so that the optimal solutions oscillate between instances where near-optimal solutions have most bins containing both large and small items, and instances where near-optimal solutions have most bins containing only small items. The largest lower bound that arises from such instances can then be phrased as a gap-revealing linear program; we exhibit a dual solution bounding the value of this LP, showing this lower bound is at least \(\alpha - \epsilon\).

For the upper bound, we use the same LP to guide our algorithm, so the gap-revealing LP now becomes a factor-revealing LP. Indeed, the LP solution tells us how the small items should be packed in order to “prepare” for arrival of large items. An important building block of our algorithms is the ability to deal with (sub)instances made up solely of small items. In particular, we give fully-dynamic BIN PACKING algorithms with a.c.r \((1 + \epsilon)\) and only \(O(\epsilon^{-2})\) worst case recourse if all items have size \(O(\epsilon)\). The underlying idea here is similar to our approach for small items for general costs. We further extend these ideas to allow for (approximately) packing items according to a “target curve”, which in our case is the solution to the above LP. For large items, we use ideas similar to the classic algorithm of de la Vega and Lueker [9] to pack these items near-optimally “on top” of these small items, changing the packing of these large items lazily.

**Weight Costs.** For the weight movement costs model, our algorithm is relatively straightforward, and relies on lazily computing a near-optimal solution, e.g. using de la Vega and Lueker’s algorithm [9], and lazily dealing with updates until the overall weight changes by some \(1 \pm \epsilon\) factor. The technical challenge in this model is in proving that the asymptotic competitive ratio to amortized recourse trade-off is inherent. We do this by considering item sizes which are roughly the reciprocals of the Sylvester sequence [28]. We show that for these item sizes, for an algorithm \(A\) to have \((1 + \epsilon)\)-a.c.r on an instance containing \(N\) items of each size, \(A\) must use some \(\Omega(N)\) many bins with one item of each size, whereas on an instance containing \(N\) items of each size except for the smallest size, \(\epsilon\), algorithm \(A\) must have some (smaller) \(O(N)\) many bins with more than one distinct size in them. This implies that repeatedly adding and removing the \(N\) items of size \(\epsilon\) either incurs a high a.c.r or a high amortized recourse.

Unlike previous uses of the Sylvester sequence in the context of BIN PACKING algorithms (and perhaps further afield), we obtain our sharper lower bound by relying explicitly on the divisibility properties of this sequence. To the best of our knowledge, these properties have not been used in the past in the context of analysis of dynamic and online algorithms, and these might be of use in tightening other lower bounds based on this sequence.

**Roadmap.** We present our results for the unit cost model in §2, followed by the general cost model in §3 and finally address the weight costs model in §4. A tabular view of our results and previous results is presented in §A.
2 \ Unit Movement Costs

We consider the natural unit movement costs model. We omit proofs here for brevity; a full version of this section appears as Appendix 2. First, we show that a competitive ratio better than $\alpha$ implies either a polynomial additive term or polynomial recourse. Here $\alpha = 1 - \frac{1}{W_{-1}(-2/e^3)+1} \approx 1.3871$ is such that $1 - 1/\alpha$ is a solution to

$$3 + \ln(1/2) = \ln(x) + 1/x,$$

where $W_{-1}$ is the lower real branch of the Lambert $W$-function. Next, we give an algorithm with a.c.r $\alpha + \epsilon$, with both the additive term and the recourse being polynomial in $1/\epsilon$. This shows that our lower bound is tight. A key ingredient in our algorithm is to write an LP that acts both as a gap-revealing LP to prove our lower bound, and also as a factor-revealing LP for the upper bound.

2.1 Impossibility results

We begin by observing that any algorithm with absolute competitive ratio better than $3/2$ cannot have $o(n)$ recourse cost $o(n)$. Consider two instances, both having $2n - 4$ items of size $1/n$ and one instance also having 4 items of size $1/2 + 1/2n$. Alternating between these two instance by inserting/deleting the latter 4 items, we get

**Observation 2.1.** For all $\epsilon > 0$, any $(3/2 - \epsilon)$-competitive online bin packing algorithm must have $\Omega(n)$ recourse, where $n$ is the maximum number of items encountered by the algorithm.

Alternating between two instances, where both have $2n - 4$ items of size $1/n$ and one instance also has 4 items of size $1/2 + 1/2n$, we get that any $(3/2 - \epsilon)$-competitive online bin packing algorithm must have $\Omega(n)$ recourse. However, this only rules out algorithms with a zero additive term. Balogh et al. showed that any constant-recourse algorithm (under unit movement costs) with $o(n)$ additive term must have a.c.r at least $\alpha \approx 1.3871$. We now strengthen both these impossibility results by showing the need for a large additive term to achieve any a.c.r below $\alpha$. Specifically, the following theorem implies that arbitrarily small polynomial additive terms imply recourse cost that is arbitrarily close to linear. Moreover, our proof will be shorter and simpler than that of Balogh et al. As an added benefit, the LP we use to bound the competitive ratio will also inspire our algorithm in the next section.

**Theorem 2.2 (Unit Costs: Main Lower Bound).** For any $\epsilon > 0$ and $\frac{1}{2} > \delta > 0$, for any algorithm $A$ with a.c.r $(\alpha - \epsilon)$ with additive term $T = o(\epsilon \cdot n^\delta)$, there exists an input for dynamic bin packing with $n$ items on which $A$ has recourse at least $\Omega(e^2 \cdot n^{1-\delta})$ under unit movement cost.

**Theorem 2.2 (Unit Costs: Main Lower Bound).** For any $\epsilon > 0$ and $\frac{1}{2} > \delta > 0$, for any algorithm $A$ with a.c.r $(\alpha - \epsilon)$ with additive term $T = o(\epsilon \cdot n^\delta)$, there exists an input for dynamic bin packing with $n$ items on which $A$ has recourse at least $\Omega(e^2 \cdot n^{1-\delta})$ under unit movement cost.

The Instances: The set of instances is the most natural one, and was also considered by Balogh et al. Let $1/2 > \delta > 0$, $c = 1/\delta - 1$, and let $B \geq 1/c$ be a large integer. Our hard instances consist of small items of size $1/B^c$, and large items of size $x$ for all $x \in N_\epsilon \triangleq \{x = 1/2 + i \cdot \epsilon \mid i \in \mathbb{N}_{\geq 0}, x \leq 1/\alpha\}$. Specifically, input $I_\delta$ consists of $\lfloor B^{c+1} \rfloor$ small items. And for each $x \in N_\epsilon$, the input $I_x$ consists of $\lfloor B^{c+1} \rfloor$ small items followed by $\lfloor \frac{B}{1-x} \rfloor$ size-x items.

The optimal bin packings for $I_\delta$ and $I_x$ require precisely $OPT(I_\delta) = B$ and $OPT(I_x) = \lfloor \frac{B}{1-x} \rfloor$ bins respectively. Consider any fully-dynamic bin packing algorithm $A$ with limited recourse and bounded additive term. When faced with input $I_\delta$, Algorithm $A$ needs to distribute the small items in the available bins almost uniformly. And if this input is extended to $I_x$ for some $x \in N_\epsilon$, $A$ needs to move many small items to accommodate these large items (or else use many new bins). Since $A$ does not know the value of $x$ beforehand, it cannot “prepare” simultaneously for all possible values of $x \in N_\epsilon$. 

As a warm-up we show that the linear program $[\text{LP}_\varepsilon]$ below gives a lower bound on the absolute competitive ratio $\alpha_\varepsilon$ of any algorithm $A$ with no recourse. Indeed, instantiate the variables as follows. On input $I_\varepsilon$, let $N_x$ be the number of bins in which algorithm $A$ keeps free space in the range $[x, x + \varepsilon)$ for each $x \in \{0\} \cup N_\varepsilon$. Hence the total volume packed is at most $N_0 + \sum_{x \in N_\varepsilon} (1 - x) \cdot N_x$. This must be at least $\text{Vol}(I_\varepsilon) = [B^{c+1}] \cdot 1/B^c \geq B - 1/B^c$, implying constraint $[\text{Vol}_\varepsilon]$.

Moreover, as $\text{OPT}(I_\varepsilon) = B$, the $\alpha_\varepsilon$-competitiveness of $A$ implies constraint $[\text{small}_\varepsilon]$. Now if instance $I_\varepsilon$ is extended to $I_{\varepsilon t}$, since we assumed $A$ moves no items, these $t$-sized items are placed either in bins counted by $N_x$ for $x \geq t$ or in new bins. Again, since $A$ is $\alpha_\varepsilon$-competitive and $\text{OPT}(I_{\varepsilon t}) \leq \lceil \frac{B}{1-\delta} \rceil$, we get constraint $[\text{CR}_\varepsilon]$. Hence the optimal value of $[\text{LP}_\varepsilon]$ is a valid lower bound on the competitive ratio $\alpha_\varepsilon$.

\[
\text{minimize } \alpha_\varepsilon \quad \text{(LP}_\varepsilon)\\
\text{s.t. } N_0 + \sum_{x \in N_\varepsilon} (1 - x) \cdot N_x & \geq B - 1/B^c \quad \text{(Vol}_\varepsilon)\\
N_0 + \sum_{x \in N_\varepsilon} N_x & \leq \alpha_\varepsilon \cdot B \quad \text{(small}_\varepsilon)\\
N_0 + \sum_{x \in N_\varepsilon, x \leq t-\varepsilon} x \cdot N_x + \left[ \frac{B}{1-\delta} \right] & \leq \alpha_\varepsilon \cdot \left[ \frac{B}{1-\delta} \right] \quad \forall t \in N_\varepsilon \quad \text{(CR}_\varepsilon)\\
N_x & \geq 0
\]

\[\text{2.1.1 } \text{LP}_\varepsilon \text{ as a Gap-Revealing LP}\]

In order to extend this argument to the fully-dynamic case, we observe that any solution to $[\text{LP}_\varepsilon]$ below gives a lower bound on the absolute competitive ratio $\alpha_\varepsilon$ of any algorithm $A$ with no recourse. Indeed, instantiate the variables as follows. On input $I_\varepsilon$, let $N_x$ be the number of bins in which algorithm $A$ keeps free space in the range $[x, x + \varepsilon)$ for each $x \in \{0\} \cup N_\varepsilon$. Hence the total volume packed is at most $N_0 + \sum_{x \in N_\varepsilon} (1 - x) \cdot N_x$. This must be at least $\text{Vol}(I_\varepsilon) = [B^{c+1}] \cdot 1/B^c \geq B - 1/B^c$, implying constraint $[\text{Vol}_\varepsilon]$. Moreover, as $\text{OPT}(I_\varepsilon) = B$, the $\alpha_\varepsilon$-competitiveness of $A$ implies constraint $[\text{small}_\varepsilon]$. Now if instance $I_\varepsilon$ is extended to $I_{\varepsilon t}$, since we assumed $A$ moves no items, these $t$-sized items are placed either in bins counted by $N_x$ for $x \geq t$ or in new bins. Again, since $A$ is $\alpha_\varepsilon$-competitive and $\text{OPT}(I_{\varepsilon t}) \leq \lceil \frac{B}{1-\delta} \rceil$, we get constraint $[\text{CR}_\varepsilon]$. Hence the optimal value of $[\text{LP}_\varepsilon]$ is a valid lower bound on the competitive ratio $\alpha_\varepsilon$.

\[
\text{minimize } \alpha_\varepsilon \quad \text{(LP}_\varepsilon)\\
\text{s.t. } N_0 + \sum_{x \in N_\varepsilon} (1 - x) \cdot N_x & \geq B - 1/B^c \quad \text{(Vol}_\varepsilon)\\
N_0 + \sum_{x \in N_\varepsilon} N_x & \leq \alpha_\varepsilon \cdot B \quad \text{(small}_\varepsilon)\\
N_0 + \sum_{x \in N_\varepsilon, x \leq t-\varepsilon} x \cdot N_x + \left[ \frac{B}{1-\delta} \right] & \leq \alpha_\varepsilon \cdot \left[ \frac{B}{1-\delta} \right] \quad \forall t \in N_\varepsilon \quad \text{(CR}_\varepsilon)\\
N_x & \geq 0
\]
We now claim that there exists a \( N \) such that
\[
\sum_{x \in \mathcal{N}_\varepsilon} N_x \leq (\alpha^*_\varepsilon - \Omega(\varepsilon)) \cdot B \leq \alpha^*_\varepsilon \cdot B.
\]
That is, the \( \mathcal{N}_\varepsilon \)'s satisfy constraint (small) with \( \alpha = \alpha^*_\varepsilon \).

We now claim that there exists a \( t \in \mathcal{N}_\varepsilon \) such that
\[
N_0 + \sum_{x \in \mathcal{N}_\varepsilon, x \leq t - \varepsilon} N_x + \left\lfloor \frac{B}{1 - t} \right\rfloor \geq \alpha^*_\varepsilon \cdot \left\lfloor \frac{B}{1 - t} \right\rfloor
\]
holds (notice the opposite inequality sign compared to constraint (CR)). Suppose not. Then the quantities \( N_0, N_x \) for \( x \in \mathcal{N}_\varepsilon \), and \( \alpha^*_\varepsilon \) strictly satisfy the constraint (small). If they also strictly satisfy the constraint (small), then we can maintain feasibility and slightly reduce \( \alpha^*_\varepsilon \), which contradicts the definition of \( \alpha^*_\varepsilon \). Therefore assume that constraint (small) is satisfied with equality. Now two cases arise: (i) All but one variable among \( \{N_0\} \cup \{N_x \mid x \in \mathcal{N}_\varepsilon\} \) are zero. If this variable is \( N_0 \), then tightness of (small) implies that \( N_0 = \alpha^*_\varepsilon \cdot B \). But then we satisfy (Vol) with slack, and so, we can reduce \( N_0 \) slightly while maintaining feasibility. Now we satisfy all the constraints strictly, and so, we can reduce \( \alpha^*_\varepsilon \), a contradiction. Suppose this happens to be \( N_x \), where \( x \in \mathcal{N}_\varepsilon \). So, \( N_x = \alpha^*_\varepsilon \cdot B \).

We will show later in Theorem 2.6 that \( \alpha^*_\varepsilon \leq 1.4 \). Since \( (1 - x) \leq 1/2 \), it follows that \( (1 - x)N_x \leq 0.7B \), and so we satisfy (Vol) with slack. We again get a contradiction as argued for the case when \( N_0 \) was non-zero, (ii) There are at least two non-zero variables among \( \{N_0\} \cup \{N_x \mid x \in \mathcal{N}_\varepsilon\} \) – let these be \( N_t_1 \) and \( N_t_2 \) with \( x_1 < x_2 \) (we are allowing \( x_1 \) to be 0). Now consider a new solution which keeps all variables \( N_x \) unchanged except for changing \( N_t_1 \) to \( N_t_1 + \frac{\eta}{1 - x^1} \), and \( N_t_2 \) to \( N_t_2 - \frac{\eta}{1 - x^2} \), where \( \eta \) is a small enough positive constant (so that we continue to satisfy the constraints (CR) strictly). The LHS of (Vol) does not change, and so we continue to satisfy this. However LHS of (small) decreases strictly. Again, this allows us to reduce \( \alpha^*_\varepsilon \) slightly, which is a contradiction. Thus, there must exist a \( t \) which satisfies (2). We fix such a \( t \) for the rest of the proof.

Let \( B \) denote the bins which have less than \( t \) free space. So, \( |B| = N_0 + \sum_{x \in \mathcal{N}_\varepsilon, x \leq t - \varepsilon} N_x \). Now, we insert \( \left\lfloor \frac{B}{1 - t - \varepsilon} \right\rfloor \) items of size \( t + \varepsilon \). (It is possible that \( t = 1/\alpha \), and so \( t + \varepsilon \notin \mathcal{N}_\varepsilon \), but this is still a valid instance). We claim that the algorithm must move at least \( \varepsilon \) volume of small items from at least \( \varepsilon B \) bins in \( B \). Suppose not. Then the large items of size \( t + \varepsilon \) can be placed in at most \( \varepsilon B \) bins in \( B \). Therefore, the total number of bins needed for \( \mathcal{I}_t \) is at least \( N_0 + \sum_{x \in \mathcal{N}_\varepsilon, x \leq t - \varepsilon} N_x - \varepsilon B + \left\lfloor \frac{B}{1 - t - \varepsilon} \right\rfloor \), which by inequality (2), is at least \( (\alpha^*_\varepsilon - O(\varepsilon)) \cdot OPT(\mathcal{I}_{t+\varepsilon}) \), because \( OPT(\mathcal{I}_{t+\varepsilon}) = \left\lfloor \frac{B}{1 - t - \varepsilon} \right\rfloor = \left\lfloor \frac{B}{1 - t} \right\rfloor + O(\varepsilon B) \). But we know that \( \mathcal{A} \) is \((\alpha^*_\varepsilon - O(\varepsilon))\) asymptotically competitive with additive term \( o(\varepsilon \cdot n^\delta) \) (which is \( o(\varepsilon \cdot OPT(\mathcal{I}_{t+\varepsilon})) \)). So it should use at most \((\alpha^*_\varepsilon - O(\varepsilon) + o(\varepsilon))OPT(\mathcal{I}_{t+\varepsilon}) \) bins, which is a contradiction. Since each small item has size \( 1/B^c \), the total number of items moved by the algorithm is at least \( \varepsilon^2 B/B^c \). This is \( \Omega(\varepsilon \cdot n^{1-\delta}) \), because \( \varepsilon \geq 1/B \), and \( B^c \) is \( \Theta(n^{1-\delta}) \).

Finally, we bound the optimal value of (LP_\varepsilon), thereby proving Theorem 2.2 via Lemma 2.3. To prove the lower and upper bound, we provide explicit feasible dual and primal solutions to this LP. The guiding idea behind both solutions is to drop floors and ceilings (incurring marginal loss), replacing the sums by integrals, and solve the continuous versions, giving rise to Inequality (i) These solutions can then be discretized to give the primal and dual solutions.

**Lemma 2.4.** The optimal value \( \alpha^*_\varepsilon \) of (LP_\varepsilon) satisfies \( \alpha^*_\varepsilon \in [\alpha - O(\varepsilon), \alpha + O(\varepsilon)] \).

**Lemma 2.5.** The optimal value of (LP_\varepsilon), \( \alpha^*_\varepsilon \), satisfies \( \alpha^*_\varepsilon \geq \alpha - O(\varepsilon) \).

**Proof.** We slightly modify (LP_\varepsilon) to make it easier to work with—this will affect its optimal value only by \( O(\varepsilon) \).

(i) Change the \( B - 1/B^c \) term in the RHS of inequality (Vol) to \( B \), and remove the floor and ceiling in the inequalities (CR). Since \( B \geq 1/\varepsilon \), this affects the optimal value by \( O(\varepsilon) \).
\[
\begin{align*}
\text{min. } & \alpha^*_t + 1 \\
n_0 + \sum_{x \in \mathcal{N}_t} (1 - x) \cdot n_x & \geq 1 \\
n_0 + \sum_{x \in \mathcal{N}_t} n_x - \alpha^*_t & \leq 1 \\
n_0 + \sum_{x \in \mathcal{N}_t, x \leq t - \varepsilon} n_x & \leq \frac{\alpha^*_t}{1 - t} \quad \forall t \in \mathcal{N}_t \\
n_x & \geq 0
\end{align*}
\] (LPnew\_ε)

\[
\begin{align*}
\text{max. } & Z - q_0 + 1 \\
q_0 + \sum_{t \in \mathcal{N}_t} \frac{q_t}{1 - t} & \leq 1 \quad (d1) \\
q_0 + \sum_{t \in \mathcal{N}_t} q_t & \geq Z \quad (d2) \\
q_0 + \sum_{t \geq x + \varepsilon, t \in \mathcal{N}_t} q_t & \geq (1 - x) \cdot Z \quad \forall x \in \mathcal{N}_t
\end{align*}
\] (Dual\_ε)

Figure 2: The modified LP and its dual program

(ii) Divide the inequalities through by \(B\), and introduce new variables \(n_x\) for \(N_x/B\), and

(iii) Replace \(\alpha_{\varepsilon} - 1\) by a new variable \(\alpha^*_t\), and change the objective value to \(\alpha^*_t + 1\).

This gives the LP (LPnew\_ε), whose optimal value is \(\alpha^*_t + O(\varepsilon)\). We now give a feasible solution for the dual linear program (Dual\_ε) whose objective value is \(\alpha - O(\varepsilon)\). This proves the desired result.

We start with a nearly-feasible dual solution to (Dual\_ε) and later modify it to obtain a feasible solution.

Set \(Z = \alpha(\alpha - 1), q_0 = (\alpha - 1)^2, q_{\frac{1}{2} + \varepsilon} = \alpha(\alpha - 1)^2/2,\) and \(q_t = \alpha(\alpha - 1)\varepsilon\) for all \(t \in \mathcal{N}_t\) with \(t \geq \frac{1}{2} + 2\varepsilon\).

The objective value of (Dual\_ε) with respect to this solution is exactly \(\alpha(\alpha - 1) - (\alpha - 1)^2 + 1 = \alpha\). We will now show that it (almost) satisfies the constraints. For sake of brevity, we do not explicitly write that variable \(t\) takes values in \(\mathcal{N}_t\) in the limits for the sums below. First, consider constraint (d1):

\[
q_0 + \sum_{t = \frac{1}{2} + \varepsilon}^{\frac{1}{2}} \frac{q_t}{1 - t} = q_0 + \frac{q_{\frac{1}{2} + \varepsilon}}{2 - \varepsilon} + \sum_{t > \frac{1}{2} + \varepsilon} \frac{q_t}{1 - t}
\]

\[
\leq q_0 + 2q_{\frac{1}{2} + \varepsilon} + \sum_{t > \frac{1}{2} + \varepsilon} \frac{q_t}{1 - t}
\]

\[
\leq (\alpha - 1)^2 + \alpha(\alpha - 1) + \int_{\frac{1}{2}}^{\frac{1}{2} + \varepsilon} \frac{\alpha(\alpha - 1)dx}{1 - x}
\]

\[
= (\alpha - 1)^2 + \alpha(\alpha - 1) + \alpha(\alpha - 1)(\ln(\frac{1}{2}) - \ln(1 - \frac{1}{\alpha}))
\]

\[
= (\alpha - 1)^2 + \alpha(\alpha - 1) + \alpha(\alpha - 1)\left(\frac{3 - 2\alpha}{\alpha - 1}\right) = 1
\]

where used \([1]\), the definition of \(\alpha\), in the penultimate equation. Next, consider constraint (d2):

\[
q_0 + \sum_{t = \frac{1}{2} + \varepsilon}^{\frac{1}{2}} q_t = q_0 + q_{\frac{1}{2} + \varepsilon} + \sum_{t > \frac{1}{2} + \varepsilon} q_t
\]

\[
\geq (\alpha - 1)^2 + \frac{1}{2} \alpha(\alpha - 1) + \int_{\frac{1}{2}}^{\frac{1}{2} + \varepsilon} \alpha(\alpha - 1)dx - O(\varepsilon)
\]

\[
= (\alpha - 1)^2 + \frac{1}{2} \alpha(\alpha - 1) + \alpha(\alpha - 1)\left(\frac{1}{\alpha} - \frac{1}{2}\right) - O(\varepsilon) = Z - O(\varepsilon)
\]

8
Finally, consider constraint (d3) for any \( x \in \mathcal{N}_\varepsilon \):

\[
q_0 + \sum_{t \geq x+\varepsilon, t \in \mathcal{N}_\varepsilon} q_t \geq (\alpha - 1)^2 + \int_{x+\varepsilon}^{\frac{1}{\alpha}} \alpha(\alpha - 1) dx - O(\varepsilon)
\]

\[
= (\alpha - 1)^2 + \left( \frac{1}{\alpha} - x - \varepsilon \right) \alpha(\alpha - 1) - O(\varepsilon)
\]

\[
= (\alpha - 1) \left( \alpha - 1 + \left( \frac{1}{\alpha} - x - \varepsilon \right) \cdot \alpha \right) - O(\varepsilon)
\]

\[
= Z \cdot (1 - x - \varepsilon) - O(\varepsilon).
\]

Finally, increase \( q_0 \) to \( q_0 + O(\varepsilon) \) to ensure that constraints (d2) and (d3) are satisfied. This is now a feasible solution to \((\text{Dual}_\varepsilon)\) with objective value \( Z - q_0 + 1 - O(\varepsilon) = \alpha - O(\varepsilon) \). Hence the optimal value \( \alpha^*_\varepsilon \) for the LP \((\text{LPnew}_\varepsilon)\) is at least \( \alpha - O(\varepsilon) \).

Theorem 2.2 now follows from Lemmas 2.3 and 2.5. Indeed, this bound on the LP is almost tight.

Theorem 2.6. The optimal value \( \alpha^*_\varepsilon \) for \((\text{LP}_\varepsilon)\) is at most \( \alpha + O(\varepsilon) \).

Proof. As in the case of Lemma 2.5, it is more convenient to work with \((\text{LPnew}_\varepsilon)\), whose optimum value is \( \alpha^*_\varepsilon \pm O(\varepsilon) \). Again we first give a solution that is nearly feasible, and then modify it to give a feasible solution with value at most \( \alpha + O(\varepsilon) \).

Let \( C \) denote \( \alpha - 1 \). Define \( n_x := \int_{x-\varepsilon}^{x} \frac{C}{(1-y)} dy \) for all \( x \in \mathcal{N}_\varepsilon \). Define

\[
n_0 := 1 - \int_{\frac{1}{2}}^{\frac{1}{2}} \frac{C}{(1-y)} dy = 1 - C \ln \left( \frac{1}{2} \right) + C \ln \left( 1 - \frac{1}{\alpha} \right).
\]

The first constraint of \((\text{LPnew}_\varepsilon)\) is satisfied up to an additive \( O(\varepsilon) \):

\[
n_0 + \sum_{x} (1 - x)n_x = 1 - \int_{\frac{1}{2}}^{\frac{1}{2}} \frac{C}{(1-y)} dy + \sum_{x \in \mathcal{N}_\varepsilon} (1 - x) \int_{x-\varepsilon}^{x} \frac{C}{(1-y)^2} dy
\]

\[
\geq 1 - \int_{\frac{1}{2}}^{\frac{1}{2}} \frac{C}{(1-y)} dy + \int_{\frac{1}{2}}^{\frac{1}{2}} \frac{C(1-y)}{(1-y)^2} dy - O(\varepsilon)
\]

\[
= 1 - O(\varepsilon).
\]

Next, the second constraint of \((\text{LPnew}_\varepsilon)\):

\[
n_0 + \sum_{x \in \mathcal{N}_\varepsilon} n_x = 1 - \int_{\frac{1}{2}}^{\frac{1}{2}} \frac{C}{1-y} dy + \sum_{x \in \mathcal{N}_\varepsilon} \int_{x-\varepsilon}^{x} \frac{C}{(1-y)^2} dy
\]

\[
= 1 - \int_{\frac{1}{2}}^{\frac{1}{2}} \frac{C}{1-y} dy + \int_{\frac{1}{2}}^{\frac{1}{2}} \frac{C}{(1-y)^2} dy
\]

\[
= 1 + C \left( - \ln(1/2) + \ln \left( 1 - \frac{1}{\alpha} \right) - 2 + \frac{1}{1 - \alpha} \right)
\]

\[
= 1 + C = \alpha,
\]

where the penultimate step follows by (1), and the last step uses \( C = \alpha - 1 \). Performing a similar
calculation for the last set of constraints in $(\text{LP}_{\text{new}})$, we get

\[ n_0 + \sum_{x \in N_x, x < t - \varepsilon} n_x = n_0 + \sum_{x \in N_x} n_x - \sum_{x \in N_x, x \geq t - \varepsilon} n_x \]

\[ = \alpha - \sum_{x \geq t - \varepsilon, x \in N_x} \int_{x - \varepsilon}^{x} \frac{C}{(1 - y)^2} dy \]

\[ \leq \alpha - \int_{t}^{\varepsilon} \frac{C}{(1 - y)^2} dy + O(\varepsilon) \]

\[ = \alpha - C\left(\frac{\alpha}{\alpha - 1} - \frac{1}{1 - t}\right) + O(\varepsilon) = \alpha - 1 + O(\varepsilon), \]

where the second equality follows from the previous sequence of calculations. To satisfy the constraints of $(\text{LP}_{\text{new}})$, we increase $n_0$ to $n_0 + O(\varepsilon)$ and set $\alpha'$ to $\alpha - 1 + O(\varepsilon)$. Since $t$ is always $\geq 1/2$, this will also satisfy the last set of constraints. Since the optimal value of $(\text{LP}_{\text{new}})$ is $\leq \alpha - 1 + O(\varepsilon)$, which implies that $\alpha' \leq \alpha - 1$, since we had subtracted 1 from the objective function when we constructed $(\text{LP}_{\text{new}})$ from $(\text{LP}_t)$.

### 2.2 Matching Algorithmic Results

As mentioned earlier, $(\text{LP}_s)$ also guides our algorithm. (For the rest of this section, all items smaller than $\varepsilon$ are called small, and the rest are large.) To begin, imagine a total of $B$ volume of small items come first, followed by large items. Inspired by the LP analysis above, we pack the small items such that the volume of small items $\in \{0\} \cup N_x$, where the $N_x$ values are near-optimal for $(\text{LP}_s)$. We call the space profile used by the small items the “curve”; see Figure 4 In the LP analysis above, adding $B/(1-x)$ items of size $x$ would result in an a.c.r of $\alpha + O(\varepsilon)$. But what if large items of different sizes are inserted and deleted? In §2.2.1 we show that if the small items are packed according to $(\text{LP}_s)$, then the optimal packing of the large items “on top” of this curve yields an a.c.r of $\alpha + O(\varepsilon)$. We then give a configuration LP (of size depending on $\varepsilon$ only) based on the small items’ curve, which can then be used to pack the large items with the same a.c.r. Repacking large items in a lazy fashion once the number of large items changes by a multiplicative $1 \pm O(\varepsilon)$ results in bounded amortized recourse, with $\alpha + O(\varepsilon)$ a.c.r.

The next challenge is that the small items may also be inserted and deleted. Suppose only the small items change. In §2.2.3 we show how to maintain a curve prescribed by an optimal solution to $(\text{LP}_s)$ using the small items, so that the right proportion of bins have roughly $x$ free space, for every $x \in \{0\} \cup N_x$. The idea is to pack the small items into “clumps” of bins such that the volume of small items in them fit the curve perfectly, up to an $\varepsilon$ fraction on average. To do so with little recourse we extend a simple $(1 + \varepsilon)$-approximate algorithm for instances consisting of solely small items in §2.2.1. Finally, in §§2.2.4 we combine the two ideas together to obtain our fully-dynamic algorithm.

#### 2.2.1 Dealing With Large Items

As mentioned above, the large items are those of size $\geq \varepsilon$. Given input $\mathcal{I}_s$ (consisting of small items with total volume $B$), consider a solution which has $N_x$ bins with gaps in the range $[x, x + \varepsilon)$ for $x \in \{0\} \cup N_x$, where $\{\alpha, N_0, N_x : x \in N_x\}$ is a feasible solution for $(\text{LP}_s)$. By the LP definition, this packing of small items is $\alpha$-competitive for any extension $\mathcal{I}_t$ of $\mathcal{I}_s$ such that $t \in N_x$—we just pack the size $t$ items in the bins in $N_x$ for $x \geq t$ before using new bins). In fact, we can prove a stronger fact about this solution: let $\mathcal{I}_t^k$ be the input where $\mathcal{I}_s$ is followed by $k$ items of size $t$, where $t > 1/2$. Note that $t$ is no longer constrained to belong to $N_x$. Also note, the instance $\mathcal{I}_t$ is a special case of $\mathcal{I}_t^k$ with $k = \lceil B/(1 - t) \rceil$). Then the solution remains $\alpha$-competitive for all such inputs using the same algorithm: we pack the size $t$ items into the bins $N_x$ for $x \geq t$ before using new bins.
Lemma 2.7 (Large Items of Same Size). Consider the solution to $\mathcal{I}_s$ with $\alpha_\epsilon, N_0, N_x, x \in N_\epsilon$ as above. This solution induces an $(\alpha_\epsilon + O(\epsilon))$-competitive packing for all extensions $\mathcal{I}^k_t$ of $\mathcal{I}_s$ for all $t > 1/2$ and $k \in \mathbb{Z}$.

Proof. Let $N = \sum x|t \geq t, x \in N_\epsilon \cdot N_x$ be the bins with at least $t$ free space; if $t \geq 1/\alpha$, then $N = 0$. Let $N' = \sum x|t \leq t, x \in N_\epsilon \cdot N_x$. Our algorithm first packs the size-$t$ items in the $N$ bins of $B$ before using new bins, and hence uses $N' + \max(N, k)$ bins. If $k \leq N$, we are done because of the constraint \( (\text{small})_t \), so assume $k \geq N$. A volume argument bounds the number of bins in the optimal solution for $\mathcal{I}^k_t$:

$$\text{OPT}(\mathcal{I}^k_t) \geq \begin{cases} k & \text{if } k(1-t) \geq B \\ k + (B - k(1-t)) & \text{else} \end{cases}$$

We now consider two cases:

- $k(1-t) \geq B$: Using constraint $\text{[CR]}_t$, the number of bins used by our algorithm is

$$N' + k \leq \frac{\alpha_\epsilon B}{1-t} + O(\epsilon B) + \left( k - \frac{B}{1-t} \right) \leq (\alpha_\epsilon + O(\epsilon))k.$$ 

- $k(1-t) < B$: Since $k$ lies between $N$ and $\frac{B}{1-t}$, we can write it as a convex combination $\frac{\lambda_1 B}{1-t} + \lambda_2 N$, where $\lambda_1 + \lambda_2 = 1, \lambda_1, \lambda_2 \geq 0$. We can rewrite constraints $\text{[small]}_t$ and $\text{[CR]}_t$ as

$$N' + \frac{B}{1-t} \leq \frac{\alpha_\epsilon B}{1-t} + O(\epsilon B) \quad \text{and} \quad N' + N \leq \alpha_\epsilon B.$$ 

Combining them with the same multipliers $\lambda_1, \lambda_2$, we see that $N' + k$ is at most

$$\frac{\alpha_\epsilon}{1-t} \left( \frac{\lambda_1 B}{1-t} + \lambda_2 B \right) + O(\epsilon B) = \alpha_\epsilon \left( B + \frac{\lambda_1 B}{1-t} \right) + O(\epsilon B) \leq \alpha_\epsilon (B + tk) + O(\epsilon B).$$

The desired result follows because $B + tk = k + (B - k(1-t))$ and $B$ is a lower bound on $\text{OPT}(\mathcal{I}^k_t)$.

Now we remove the assumption that all large items are of the same size. Again, assume the $B$ volume of small items is placed according to the profile given by the LP solution. Let $\mathcal{I}$ denote the instance consisting of the small and the large items. We use an extension of the Gomory-Gilmore configuration LP that takes a curve representing the packing the small items, and find the packing of configurations of large items to minimize the bins used. Naturally, we use the linear rounding idea of de la Vega and Lueker [9] to reduce the number of distinct item sizes, and hence bound the number of constraints: the resulting ILP (with variable $y_c$ for each configuration $c$) is given in ILP[2]. For readers unfamiliar with this configuration ILP, details are in Appendix 2. Let $\tau^*$ be the optimal value of this ILP. Now we remove the assumption that all large items are of the same size. Again, assume the $B$ volume of small items is placed according to the profile given by the LP solution. Let $\mathcal{I}$ denote the instance consisting of the small and the large items. As in the bin packing algorithm of de la Vega and Lueker [9], we perform linear grouping on the large items, rounding their sizes to get at most $1/\epsilon^2$ distinct sizes. Let $\mathcal{L}$ be the set of new sizes with $|\mathcal{L}| \leq 1/\epsilon^2$. This rounding changes the optimal value to at most $(1 + \epsilon) \cdot \text{OPT}(\mathcal{I})$. We can now write a compact configuration LP for the large items.

Let the sizes in $\mathcal{L}$ be $\{\ell_1, \ell_2, \ldots, \ell_u\}$ for $u \leq 1/\epsilon^2$. A configuration $c$ is a tuple $(c_1, \ldots, c_u)$ which corresponds to $c_i$ large items of size $\ell_i$ for $i \in [1, u]$, whose total size $\text{size}(c) := \sum_i c_i \ell_i \leq 1$. Hence,
any set of large items that fit into any bin form a configuration. There are at most \((1/\varepsilon)^{O(1/\varepsilon^2)}\) possible configurations, independent of the number of items. Let \(X\) be the set of distinct sizes of all configurations; hence \(|X| \leq (1/\varepsilon)^{O(1/\varepsilon^2)}\). For each configuration \(c\), the variable \(y_c\) denotes the number of bins for which the corresponding configuration is \(c\). Let \(n_i\) denote the total number of large items of size \(\ell_i\) in the instance, for \(i \in [1..u]\). For \(x \in [0,1]\), let \(N_{\leq x}\) denote the number of bins with less than \(x\) free space (i.e., more than \(1 - x\) space is taken by small items). We can now write the following integer linear program to bound the optimal number of bins needed to pack all the items.

\[
\begin{align*}
\text{minimize } & \tau \\
\text{s.t. } & N_{<x} + \sum_{c : \text{size}(c) \geq x} y_c \leq \tau & \forall x \in X \cup \{\infty\} \\
& \sum_c c_i y_c \geq n_i & \forall i = 1, \ldots, u \\
& y_c \in \mathbb{Z}_{\geq 0}
\end{align*}
\]

(For the moment let us assume we solve \([ILP]_\varepsilon\) exactly; we can approximate it with an additive loss of \(O(1/\varepsilon)\) using standard ideas.) If the optimal value of \([ILP]_\varepsilon\) is \(\tau^*\), any solution for \(I\) which respects the packing of small items must use \(\geq \tau^*\) bins. We now show that we can also pack the items in \(\tau^*\) bins; this is not immediate since different bins have different amounts of free space. Consider the greedy algorithm that orders bins in increasing order of available space: let this ordering be \(b_1, b_2, \ldots\). Now repeatedly consider the smallest remaining configuration \(c\), and place it in the first bin with free space \(\geq \text{size}(c)\) that has not yet received a configuration. Call this the canonical packing, and call a bin used if it received some configuration. The following observation is immediate.

**Observation 2.8.** If a configuration \(c\) is packed in bin \(i\), then the available space in any unused bin \(i' < i\) is strictly less than \(\text{size}(c)\). Moreover, if a configuration \(c'\) is packed in a bin which appears after a configuration \(c\), then \(\text{size}(c') \geq \text{size}(c)\).

We can now prove that the packing algorithm uses a near-minimum number of bins.

**Lemma 2.9.** The algorithm to pack large items uses at most \(\tau^*\) bins. Further, any packing of \(I\) which respects the profile constraint for the small items needs at least \(\tau^*/(1+\varepsilon)\) bins.

**Lemma 2.9.** The algorithm to pack large items uses at most \(\tau^*\) bins. Further, any packing of \(I\) which respects the profile constraint for the small items needs at least \(\tau^*/(1+\varepsilon)\) bins.

**Proof.** Let \(N = N_0 + \sum_{x \in \mathbb{N}} N_x\) be the number of bins used for packing small items. If the large items do not open any new bins, we use \(N\), and \(N \leq \tau\) using constraint \([CR-ilp]\) for \(x = \infty\). So assume that we use at least one new bin. Define a block as a maximal continuous sequence of used bins (see Figure 3 for an example), and consider the last block \(B\) used by the algorithm. If the index of the first bin is \(i\) and the block has \(t\) bins, the algorithm uses \((i - 1) + t\) bins. Let \(c\) be the configuration packed in the first bin of \(B\). Since bin \(i - 1\) is unused, Observation 2.8 shows that the available space in bin \(i - 1\) is less than \(\text{size}(c)\). Since bins are packed in increasing order of available space, it follows that \(N_{<\text{size}(c)} = i - 1\). Moreover, the size of all the configurations in this block is at least \(\text{size}(c)\) using the second part of Observation 2.8). But now the constraint \([CR-ilp]\) for \(x = \text{size}(c)\) implies \((i - 1) + t \leq \tau\), which proves the first claim of the lemma. For the second claim, once we restrict the large job sizes to \(L\) where we lose an \((1 + \varepsilon)\)-factor, any packing algorithm must satisfy the constraints of \([ILP]_\varepsilon\). \(\square\)

The final crucial next step is to relate \(\tau^*\)—which fixes the profile of the small jobs and then packs the large ones near-optimally—to \(OPT(I)\), which is not required to pack the small items in any restricted manner. We first prove that the optimal solution for \(I\) can be assumed to have some nice structure.
Observation 2.10. Fix $\varepsilon \leq 1/3$. For any input $I$, there exists a solution using at most $(1 + 3\varepsilon) \cdot OPT(I)$ bins, such that each bin containing large items (except maybe one such bin) has more than $1/2$ unit of volume occupied by large items.

Proof. We modify an optimal packing $P$ of $I$ requiring $B = OPT(I)$ bins so that it satisfies the criteria. During this process, some small items will be removed from their assigned bins and put aside. We denote this set by $E$, and we pack these items at the end. Initially $E$ is empty. Whenever there are two bins $i$ and $j$ such that the space occupied by large items is $v_i, v_j \in (0, 1/2]$ respectively, we transfer all the large items in these two bins to bin $i$. The small items are then packed arbitrarily into bin $j$ and into the remaining space in bin $i$. Since small items have size at most $\varepsilon$, at most $2\varepsilon$ volume of small items will remain unassigned. These are put in the set $E$. Each operation creates one new bin with no large items, so there are at most $B$ such operations. Hence the total volume of (small) jobs in $E$ is $\leq 2\varepsilon B$, and packing them using FirstFit, say, will require at most $2\varepsilon \cdot B/(1 - \varepsilon) + 1 \leq 3\varepsilon \cdot B$ new bins.

We can now prove the main theorem for packing large items.

Theorem 2.11. The number of bins $\tau^*$ used by our algorithm is at most $(\alpha \varepsilon + O(\varepsilon)) \cdot OPT(I) + O(1)$.

Proof. We show the existence of a packing $\mathcal{P}$ of $I$ requiring $B = OPT(I)$ bins so that it satisfies the criteria. During this process, some small items will be removed from their assigned bins and put aside. We denote this set by $E$, and we pack these items at the end. Initially $E$ is empty. Whenever there are two bins $i$ and $j$ such that the space occupied by large items is $v_i, v_j \in (0, 1/2]$ respectively, we transfer all the large items in these two bins to bin $i$. The small items are then packed arbitrarily into bin $j$ and into the remaining space in bin $i$. Since small items have size at most $\varepsilon$, at most $2\varepsilon$ volume of small items will remain unassigned. These are put in the set $E$. Each operation creates one new bin with no large items, so there are at most $B$ such operations. Hence the total volume of (small) jobs in $E$ is $\leq 2\varepsilon B$, and packing them using FirstFit, say, will require at most $2\varepsilon \cdot B/(1 - \varepsilon) + 1 \leq 3\varepsilon \cdot B$ new bins.

We start with a solution using $(1 + 3\varepsilon) \cdot OPT(I)$ bins, of the form guaranteed by Observation 2.10 and ignore the bin which has non-zero-but-less-than-half volume of large jobs in it: this can only decrease the optimal value by one. Call this packing $\mathcal{P}$. Now consider a new instance $I'$ where the small items are same as those in $I$, but each large item in $I'$ is obtained by merging all the large items packed in a bin of $\mathcal{P}$ into one. Therefore, all large items in $I'$ have size $\geq 1/2$. Now $OPT(I') \leq (1 + 3\varepsilon) \cdot OPT(I)$, since $\mathcal{P}$ yields such a packing. Henceforth, we show how to pack $I'$ using small number of bins.

Pack the small items according to the profile given by $N_x$, with $N$ bins being used. Sort the large items in increasing order of size, the bins in increasing order of free space, and pack each large item in the first bin into which it fits. A block is a contiguous set of bins used by large items, and consider

![Figure 3: Canonical packing of configurations. The grey region is the curve $\{N_x\}_{x \in X}$ of small items. The boxes are configurations. The bins are shown in increasing order of free space.](image)
the last block $B$ and the first (and smallest) large item $i \in I'$ in this block. Let $i$ have size $t$, and let
the block $B$ have $k$ bins. By the greedy packing, all large items in $B$ have size $\geq t$. Now consider a
new instance $I''$ with the same set of small items as in $I'$ (or $I$), and just $k$ large items of size $t$ each.
Clearly, $OPT(I'') \leq OPT(I')$. The greedy packing again uses $OPT(I')$ bins. But recall: Lemma 2.7
showed that for an instance having small items and $k$ identical large items of size $t$ (which we called
$I_k^t$), this greedy algorithm uses at most $(\alpha_{e} + O(\varepsilon))OPT(I'')$. Unraveling the reductions, this is at
most $(\alpha_{e} + O(\varepsilon))OPT(I)$, which proves the desired result.

Inserting/Deleting Large Items. Thus, if we have a fixed set $I_s$ of small items, and a feasible
solution to $\text{LP}_{x}$ with integral $N_x$ values and $\alpha_{e}$ close to $\alpha_{x}$, we get an $(\alpha + O(\varepsilon))$-approximate solution for the instance $I$. Note that we implicitly assumed that the set of large items is fixed. This is easily
extended to the fully-dynamic case for large items—but again where the set of small items remain
fixed to $I_s$. (We handle dynamics of small items in the next sections.) The idea is to divide the input
sequence into epochs. At the beginning of each epoch, we pack the large items into the profile given
by $N_x$, as described above. Suppose we use $\tau$ bins. This epoch lasts for the next $\varepsilon \tau$ operations (of
inserts/deletes for large items). During the epoch, any (large) item inserted is put in a new bin, and
any (large) item that is delete is simply removed without repacking other items. Since we use at most $\tau(1 + \varepsilon)$ bins overall, and the optimum value can only fall by $\varepsilon \tau$, our algorithm has $(\alpha_{e} + O(\varepsilon))$-a.c.r
during this epoch. When the epoch ends, we recompute a new packing of the current set of large
items. This may move all large items, but since the number of them is at most $\tau(1 + \varepsilon)/\varepsilon \leq 2\tau/\varepsilon$, we
can charge it to the $\varepsilon \tau$ new operations to get amortized $O(1/\varepsilon^2)$-recourse.

2.2.2 Small Items: A Warm-up

Now we generalize this result to the fully-dynamic setting for small items as well. Again the idea will
be to divide the input epochs, such that we work with a fixed profile of small items during an epoch.
All small items arriving during an epoch will be packed separately, and when the epoch ends, we will
construct a new profile of small items. We begin with a warm-up where the instance only consists of
small items, i.e., items with size at most $\varepsilon \leq 1/6$.

Lemma 2.12. For all $\varepsilon \leq 1/6$ there exists BIN PACKING algorithm with a.c.r $(1 + 3\varepsilon)$, with $O(1/\varepsilon)$ additive
term and $O(1/\varepsilon)$ worst-case recourse for instances comprising only of $\varepsilon$-small items.

Proof. Keep the bins ordered and assign jobs to maintain two invariants: (i). items in earlier bins
(according to the bin ordering) are no larger than items in later bins, and (ii) bins are partitioned into
buckets of consecutive bins, with the number of bins in each bucket (except possibly the rightmost
bucket) being $\in [1/\varepsilon, 3/\varepsilon]$. All bins in a bucket (except the rightmost one) have volume $\geq 1 - \varepsilon$. The
claimed a.c.r follows from a volume argument.

We now show a simple way to maintain these invariants with $O(1/\varepsilon)$ worst-case recourse per operation.
Upon insertion of an item $j$, insert the item into the correct bin, say bin $i$, based on its size $s_j$. If bin
$i$ overflows, remove the largest item $j'$ in $i$ (fixing this overflow), insert it into bin $i+1$ in the same
bucket, and repeat. If this process cascades and the last bin $i''$ in this bucket overflows, extend this
bucket with a new bin, and put the overflowed item from $i''$ into it. In case the size of the bucket exceeds $3/\varepsilon$, split it into two buckets with (almost) the same number of bins—this is just book-keeping
and causes no recourse. Since the cascade of inserts has length at most $2/\varepsilon$, the worst-case recourse cost
for inserts is $O(1/\varepsilon)$. Deletions are similar: just “borrow” items from the next bin in the bucket if the
bin volume falls below $1 - \varepsilon$. If the last bin in the bucket becomes empty, remove it. If number
of bins in the bucket falls below $1/\varepsilon$, merge it with the next bucket. This new bucket has at most
$1/\varepsilon + 3/\varepsilon \leq 4/\varepsilon$ bins; if it has more than $3/\varepsilon$ bins, split it into evenly-sized buckets (with no recourse).
Clearly, the worse-case recourse cost is again $O(1/\varepsilon)$.
2.2.3 Dealing With Small Items: “Fitting a Curve”

We now consider the problem of packing $\varepsilon$-small items according to an approximately-optimal solution of $[LP_z]$. We abstract the problem thus.

**Definition 2.13** (Bin curve-fitting). Given a list of bin sizes $0 \leq b_0 \leq b_1 \leq \ldots, b_K \leq 1$ and relative frequencies $f_0, f_1, f_2, \ldots, f_K$, such that $f_x \geq 0$ and $\sum_{x=0}^{K} f_x = 1$, an algorithm for the bin curve-fitting problem must pack any set of items (say $m$ of them, with sizes $s_1, \ldots, s_m \leq 1$) into a minimal number of bins $N$ such that for every $x \in [0, K]$ the number of bins of size $b_x$ that are used by this packing lie in $\{[N \cdot f_x], [N \cdot f_x]\}$.

If we have $K = 0$ with $b_0 = 1$ and $f_0 = 1$, we get standard bin packing. We want to solve the problem only for (most of the) small items, in the fully-dynamic setting. We consider the special case with relative frequencies $f_x$ being multiples of $1/T$, for $T \in \mathbb{Z}$. (Think of $T = O(1/\varepsilon)$.) As with our solution for small items from §2.2.2 our our algorithm maintains bins in some order, and when we consider the bins in this order, the items in them appear in increasing order of item size. The number of bins is always in multiples of $T$. Consecutive bins are aggregated into clumps of exactly $T$ bins each, and clumps aggregated into $\Theta(1/\varepsilon)$ buckets each. Formally, each clump has $T$ bins, with $f_x \cdot T \in \mathbb{N}$ bins of size $\approx b_x$ for $x = 0, \ldots, K$. The bins in a clump are ordered according to their target $b_x$, so each clump looks like an “L”-shape. Each bucket consists of some $s \in [1/\varepsilon, 3/\varepsilon]$ consecutive clumps. E.g., see Figure 4. For each bucket, all bins except those in the last clump are full to within additive $\varepsilon$ of their target size. (And the last bucket may have fewer than $1/\varepsilon$ clumps.)

Inserting an item adds it to the correct bin according to its size. If the bin size becomes larger than the target size for the bin, the largest item overflows into the next bin, and so on. Clearly this maintains the invariant that we are within an additive $\varepsilon$ of the target size. We perform $O(T/\varepsilon)$ moves in the same bucket; if we overflow from the last bin in the last clump of the bucket, we add a new clump of $T$ new bins to this bucket, etc. If a bucket contains too many clumps, it splits into two buckets, at no movement cost. An analogous (reverse) process happens for deletes. Loosely, the process maintains that on average the bins are full to within $O(\varepsilon)$ of the target fullness—one loss of $\varepsilon$ because each bin may have $\varepsilon$ space, and another because an $O(\varepsilon)$ fraction of bins have no guarantees whatsoever.

In order to make this process formal and relate it to the value of $[LP_z]$, we first need some technical lemmas. First we need to show that setting $T = O(1/\varepsilon)$ and restricting to frequencies being multiples of $\varepsilon$ does not hurt us. Indeed, for us, $b_0 = 1$, and $b_x = (1-x)$ for $x \in \mathbb{N_\varepsilon}$. Since $[LP_{z}]$ depends on the total volume $B$ of small items, and $f_x$ may change if $B$ changes, it is convenient to work with the normalized LP $[LP_{newz}]$ a normalized version, referred to as $[LP_{newz}]$ (see Figure 2 in Appendix 2).
Lemma 2.14 (Multiples of $\varepsilon$). For any optimal solution $\{n_x\}$ to $\text{LP}_{\text{new}}$, we can construct a new solution $\{\tilde{n}_x\}$ such that (a) all $\tilde{n}_x$ are integral multiples of $\varepsilon$, and (b) the new objective value is $\alpha^*_\varepsilon + O(\varepsilon)$.

Proof. For sake of brevity, let $\mathcal{N}_\varepsilon := \mathcal{N} \cup \{0\}$. Consider the indices $x \in \mathcal{N}_\varepsilon$ in increasing order, and modify $n_x$ in this order. Let $\Delta_x := \sum_{x' \in S_i} n_{x'} - n_x$, and define $\tilde{\Delta}_x$ be the analogous expression for $\tilde{n}_x$. We maintain the invariant that $|\Delta_x - \tilde{\Delta}_x| \leq \alpha^*_\varepsilon$, which is trivially true for the base case $x = 0$. Inductively, suppose it is true for $x$, and define $\tilde{n}_x, I_x$ as $n_x$, rounded up to the nearest multiple of $\varepsilon$, otherwise it is rounded down; this maintains the invariant. If we add $O(\varepsilon)$ to the old $\alpha^*_\varepsilon$ value, this easily satisfies the second and third set of constraints, since our rounding procedure ensures that the prefix sums are maintained up to additive $\varepsilon$, and $t \in [0, \frac{1}{2}]$. Checking this for the first constraint turns out to be more subtle. We claim that

$$\tilde{n}_0 + \sum_{x \in \mathcal{N}_\varepsilon} (1-x)\tilde{n}_x \geq 1 - O(\varepsilon).$$

For an element $x \in \mathcal{N}_\varepsilon$, let $\Delta n_x$ denote $n_x - \tilde{n}_x$. It is enough to show that $|\sum_{x \in \mathcal{N}_\varepsilon} x \cdot \Delta n_x| \leq O(\varepsilon)$.

Indeed,

$$\tilde{n}_0 + \sum_{x \in \mathcal{N}_\varepsilon} (1-x)\tilde{n}_x = n_0 + \sum_{x \in \mathcal{N}_\varepsilon} (1-x)n_x - \Delta n_0 - \sum_{x \in \mathcal{N}_\varepsilon} (1-x)\Delta n_x$$

$$\geq 1 - \sum_{x \in \mathcal{N}_\varepsilon} \Delta n_x + \sum_{x \in \mathcal{N}_\varepsilon} x \cdot \Delta n_x \geq 1 - O(\varepsilon) + \sum_{x \in \mathcal{N}_\varepsilon} x \cdot \Delta n_x.$$

We proceed to bound $\sum_{x \in \mathcal{N}_\varepsilon} x \cdot \Delta n_x$. Define $\Delta n_{\leq x}$ as $\sum_{x' \in \mathcal{N}_\varepsilon : x' \leq x} \Delta n_{x'}$. Define $\Delta n_{< x}$ analogously.

Note that $\Delta n_{\leq x}$ stays bounded between $[-\varepsilon, +\varepsilon]$. Let $I$ denote the set of $x \in \mathcal{N}_\varepsilon$ such that $\Delta n_{\leq x}$ changes sign, i.e., $\Delta n_{< x}$ and $\Delta n_{\leq x}$ have different signs. We assume w.l.o.g. that $\Delta n_{\leq x} = 0$ for any $x \in I$—we can do so by splitting $\Delta n_x$ into two parts (and so, having two copies of $x$ in $\mathcal{N}_\varepsilon$). Observe that for any two consecutive $x_1, x_2 \in I$, the function $\Delta n_{\leq x}$ is unimodal as $x$ varies from $x_1$ to $x_2$, i.e., it has only one local maxima or minima. This is because $\Delta n_x$ is negative if $\Delta n_{< x}$ is positive, and vice versa.

Let the elements in $I$ (sorted in ascending order) be $x_1, x_2, \ldots, x_q$. Let $S_i$ denote the elements in $\mathcal{N}_\varepsilon$ which lie between $x_i$ and $x_{i+1}$, where we include $x_{i+1}$ but exclude $x_i$. Note that $\sum_{x \in S_i} \Delta n_x = \Delta n_{x_{i+1}} - \Delta n_{x_i} = 0$. Let $x'_i = x_i + \varepsilon$ be the smallest element in $S_i$. Now observe that

$$\sum_{x \in S_i} x \cdot \Delta n_x = x'_i \cdot \sum_{x \in S_i} \Delta n_x + \sum_{x \in S_i} (x - x'_i) \cdot \Delta n_x = \sum_{x \in S_i} (x - x'_i) \cdot \Delta n_x.$$

Because of the unimodal property mentioned above, we get $\sum_{x \in S_i} |\Delta n_x| \leq 2\varepsilon$. Therefore, the absolute value of the above sum is at most $2\varepsilon \cdot (x_{i+1} - x'_i) \leq 2\varepsilon^2 |S_i|$, using that $x_{i+1} - x'_i = (|S_i| - 1)\varepsilon$. Now summing over all $S_i$ we see that $\sum_x x \cdot \Delta n_x \leq O(\varepsilon)$ because $|\mathcal{N}_\varepsilon|$ is $O(1/\varepsilon)$. This proves the desired claim.

So to satisfy the first constraint we can increase $n_0$ by $O(\varepsilon)$. And then, increasing $\alpha^*_\varepsilon$ by a further $O(\varepsilon)$ satisfies the remaining constraints, and proves the lemma. \[\Box\]
Let \( \tilde{n}_x \) be \( i_x \cdot \varepsilon \) where \( i_x \) is an integer. Note that \( \sum_x \tilde{n}_x \leq \alpha + O(\varepsilon) \leq 2 \), so dividing through by \( \varepsilon \), \( \sum_x i_x \leq 2/\varepsilon \). Now for any index \( x \in \{0\} \cup \mathcal{N}_\varepsilon \), we define \( f_x := \frac{\tilde{n}_x}{\sum_{x'} \tilde{n}_{x'}} = \sum_{x'} \frac{i_{x'}}{\sum_{x'} i_{x'}} \). If we set \( T := \sum_x i_x \leq 2/\varepsilon \), then \( T \) is an integer at most \( 2/\varepsilon \), and \( f_x \) are integral multiplies of \( 1/T \), which satisfies the requirements of our algorithm.

Next we show that the dynamic solution maintained by our algorithm (using the clumping and bucketing scheme) corresponds to a near-optimal solution to \( \text{LP}_{\varepsilon} \).

**Lemma 2.15 (Small Items Follow the LP).** Suppose the average bin-size is \( \eta := \sum_x f_x b_x \geq 1/4 \), and let \( \varepsilon \leq 1/6 \). The above bin curve-fitting algorithm has worst-case recourse \( O(T/\varepsilon) \) for instances with only \( \varepsilon \)-small items. Moreover, barring a set of \( O(1/\varepsilon^2) \) bins, let \( \bar{N}_x \) denote the number of bins of size \( b_x := (1-x) \) used by our algorithm. Then these quantities satisfy \( \text{LP}_{\varepsilon} \) with \( \alpha_\varepsilon = \alpha + O(\varepsilon) \).

**Proof.** The recourse bound is immediate, as each insertion or deletion causes a single item to move from at most \( T \cdot 3/\varepsilon \) bins. For the rest of the argument, ignore the last bucket with \( O(1/\varepsilon^2) \) bins. Let the total volume of items in the other bins be \( B \). Since \( \eta = \sum f_x b_x \) is the average bin-size, we expect to use \( \approx B/\eta \) bins for these items. We now show that we use at most \( (1 + O(\varepsilon)) \cdot \frac{B}{\eta} \) and at least \( (1 - O(\varepsilon)) \cdot \frac{B}{\eta} \) bins of size \( b_x \) for each \( x \).

Indeed, each (non-last) bucket satisfies the property that all bins in it, except perhaps for those in the last clump, are at least \( \varepsilon \)-close to the target value. Since each bucket has at least \( 1/\varepsilon \) clumps, it follows that if there are \( N \) clumps and the target average bin-size is \( \eta \), then \( \eta \) clumps are at least \( (\eta - \varepsilon) \) full on average. The total volume of a clump is \( \eta \cdot T \), so \( N \leq \frac{B}{(1-\varepsilon)(\eta-\varepsilon)T} = \frac{B}{\eta T} (1 + O(\varepsilon)) \), where we use that \( \eta \geq 1/4 \). Therefore, the total number of bins of size \( b_x \) used is \( f_x T \cdot N \leq (1 + O(\varepsilon)) \cdot \frac{B}{\eta} \).

The lower bound for the number of bins of size \( b_x \) follows from a similar argument and the observation that if we scale the volume of small items up by a factor of \( (1 + O(\varepsilon)) \), this volume would cause each bin to be filled to its target value. This implies that we use at least \( (1 - O(\varepsilon)) \cdot \frac{B}{\eta} \) bins with size \( b_x \).

We now show that the \( \bar{N}_x \) satisfy \( \text{LP}_{\varepsilon} \) with \( \alpha_\varepsilon = \alpha + O(\varepsilon) \). Recall that we started with an optimal solution to \( \text{LP}_{\varepsilon} \) of value \( \alpha^*_\varepsilon + O(\varepsilon) \), used Lemma 2.14 to get \( f_x = \frac{\tilde{n}_x}{\sum_{x'} \tilde{n}_{x'}} \) and ran the algorithm above. By the computations above, \( \bar{N}_x \), the number of bins of size \( b_x \) used by our algorithm, is

\[
(1 + O(\varepsilon)) \cdot \frac{f_x B}{\sum_{x'} f_x' b_x'} = (1 + O(\varepsilon)) \cdot \frac{\tilde{n}_x B}{\sum_{x'} \tilde{n}_{x'} b_{x'}} \leq (1 + O(\varepsilon)) \cdot \tilde{n}_x B,
\]

where the last inequality follows from the fact that \( \sum_{x'} \tilde{n}_{x'} b_{x'} \geq 1 \) (by the first constraint of \( \text{LP}_{\varepsilon} \). Likewise, by the same argument, we find that these \( \bar{N}_x \) satisfy \( \text{CR}_{\varepsilon} \) with \( \alpha_\varepsilon = \alpha + O(\varepsilon) \). Finally, since \( \tilde{n}_x \) satisfies constraints of \( \text{LP}_{\varepsilon} \) (up to additive \( O(\varepsilon) \) changes in \( \alpha^*_\varepsilon \)), we can verify that the quantities \( \bar{N}_x \) satisfy the last two constraints of \( \text{LP}_{\varepsilon} \) (again up to additive \( O(\varepsilon) \) changes in \( \alpha^*_\varepsilon \)). To see that they also satisfy \( \text{Vol}_{\varepsilon} \) we use the following calculation:

\[
\sum_x \bar{N}_x \geq (1 - 3\varepsilon) \cdot \sum_x \frac{f_x B}{\sum_{x'} b_{x'} f_{x'}} = (1 - O(\varepsilon)) \cdot \frac{B}{\sum_x b_x f_x} \geq (1 - O(\varepsilon)) \cdot B,
\]

because \( \sum_x f_x = 1 \) and \( b_x \leq 1 \) for all \( x \). Therefore, scaling all variables with \( (1 - O(\varepsilon)) \) will satisfy constraint \( \text{Vol}_{\varepsilon} \) as well. It follows that \( \bar{N}_x \) satisfy \( \text{LP}_{\varepsilon} \) with \( \alpha_\varepsilon = \alpha^*_\varepsilon + O(\varepsilon) \). Finally, since \( \alpha^*_\varepsilon = \alpha + O(\varepsilon) \), we get the claim. □
2.2.4 Our Algorithm

To combine the solutions for small items and large items, we will divide time into epochs. At the beginning of an epoch, let \( \mathcal{I} \) be the set of items with \( \mathcal{I}^s \) and \( \mathcal{I}^l \) being the small and the large items respectively. The algorithm will satisfy the following invariants

- Small items \( \mathcal{I}^s \) are packed as in Lemma 2.15 apart from a set of \( O(1/\varepsilon^2) \) extra bins, the remaining small items of volume \( B \) are packed according to profile given by \( \bar{N}_x \).
- The large items form a canonical packing with respect to the bins above; we ignore the \( O(1/\varepsilon^2) \) extra bins for packing the large items.

If the above invariants hold, Theorem 2.11 shows that the number of bins, \( N \), used by our algorithm at this time is at most \((\alpha\varepsilon + O(\varepsilon))OPT(\mathcal{I}) + O(1/\varepsilon^2) \). This epoch ends after \( \varepsilon N \) operations (i.e., insert or delete of an item). Whenever an item is deleted, we do not do anything. If an item is inserted we create a new bin and add the item to it. Let \( \mathcal{I}' \) be the instance at the end of this epoch. When the epoch ends, we are using at most \( N(1 + \varepsilon) \) bins. Further, the optimal value \( OPT(\mathcal{I}') \) may have gone down to \( OPT(\mathcal{I}) - \varepsilon N \). Therefore,

\[
(1 + \varepsilon) N \leq (1 + O(\varepsilon))OPT(\mathcal{I}) + O(1/\varepsilon^2) \leq (1 + O(\varepsilon))(OPT(\mathcal{I}') + \varepsilon N) + O(1/\varepsilon^2),
\]

which implies that \( N \) remains \((\alpha\varepsilon + O(\varepsilon))OPT(\mathcal{I}') \). We defer the full proof which bounds the recourse cost and the running time to the appendix. These bounds, together imply the following:

**Theorem 2.16.** There is a polynomial-time fully-dynamic bin packing algorithm which achieves a competitive ratio of \( \alpha + O(\varepsilon) \) with an additive factor of \( O(1/\varepsilon^2) \) using \( O(1/\varepsilon^2) \) recourse under unit movement costs.

**Theorem 2.16.** There is a polynomial-time fully-dynamic bin packing algorithm which achieves a competitive ratio of \( \alpha + O(\varepsilon) \) with an additive factor of \( O(1/\varepsilon^2) \) using \( O(1/\varepsilon^2) \) recourse under unit movement costs.

**Proof.** We divide time into epochs. At the beginning of an epoch, let \( \mathcal{I} \) be the set of items with \( \mathcal{I}^s \) and \( \mathcal{I}^l \) being the small and the large items respectively. The algorithm will satisfy the following invariants

- Small items \( \mathcal{I}^s \) are packed as in Lemma 2.15 apart from a set of \( O(1/\varepsilon^2) \) extra bins, the remaining small items of volume \( B \) are packed according to profile given by \( \bar{N}_x \).
- The large items form a canonical packing with respect to the bins above; we ignore the \( O(1/\varepsilon^2) \) extra bins for packing the large items.

If the above invariants hold, Theorem 2.11 shows that the number of bins, \( N \), used by our algorithm at this time is at most \((1 + O(\varepsilon))OPT(\mathcal{I}) + O(1/\varepsilon^2) \). This epoch ends after \( \varepsilon N \) operations (i.e., insert or delete of an item). Whenever an item is deleted, we do not do anything. If an item is inserted we create a new bin and add the item to it. Let \( \mathcal{I}' \) be the instance at the end of this epoch. When the epoch ends, we are using at most \( N(1 + \varepsilon) \) bins. Further, the optimal value \( OPT(\mathcal{I}') \) may have gone down to \( OPT(\mathcal{I}) - \varepsilon N \). Therefore,

\[
(1 + \varepsilon) N \leq (1 + O(\varepsilon))OPT(\mathcal{I}) + O(1/\varepsilon^2) \leq (1 + O(\varepsilon))(OPT(\mathcal{I}') + \varepsilon N) + O(1/\varepsilon^2),
\]

which implies that \( N \) remains \((1 + O(\varepsilon))OPT(\mathcal{I}') \). Let \( I^s \) and \( D^s \) be the small items inserted and deleted during this epoch respectively. Let \( \mathcal{I}^l_i \) be the large items in the instance \( \mathcal{I}' \). When the epoch ends, we
use the fully-dynamic algorithm of §2.2.3 to insert/delete the items in $I^s$ and $I^d$ respectively. After this, we solve the algorithm in §2.2.1 to find a canonical packing of large items into these bins (as before, we ignore the “extra” bins in the last bucket). After these operations, it follows that both the invariants are satisfied at the beginning of the next epoch again.

Recourse cost. The recourse cost for inserting/deleting $I^s$ and $I^d$ respectively is $O(1/\varepsilon^2)$ (Lemma 2.15). For large items, notice that there were at most $N/\varepsilon$ large items in the instance $I$, and $\varepsilon N$ more may have arrived during this epoch. Our algorithm may move all the large items to different bins, but the recourse cost is at most $O(N/\varepsilon)$. Since $\varepsilon N$ operations occurred during this epoch, it follows that the amortized recourse cost for large items is also $O(1/\varepsilon^2)$.

Efficient Implementation. Computing an optimal solution to $\text{LP}_{\text{new}}^{\varepsilon}$ can be easily done in $\text{poly}(\varepsilon^{-1})$ time. For the solution of the $\text{ILP}_{\varepsilon}$, the ideas of Fernandez de la Vega and Lueker [9] can be used. The number of variables of $\text{ILP}_{\varepsilon}$ is proportional to the number of valid configurations, which is at most $(1/\varepsilon)^{O(1/\varepsilon^2)}$. The number of constraints is $O(1/\varepsilon)$, since we only have $O(1/\varepsilon)$ different sizes in $N_{\varepsilon}$. Hence there are at most $O(1/\varepsilon)$ non-zero variables in any basic feasible solution to the LP obtained by relaxing this configuration $\text{ILP}_{\varepsilon}$. Consequently, the naïve rounding of a basic solution to the LP relaxation of $\text{ILP}_{\varepsilon}$ (i.e., taking the ceiling of every value) incurs only a $O(1/\varepsilon)$ additive term compared to the optimal solution of the LP relaxation, and therefore of the optimal solution of the ILP. The dominant additive term of our algorithm remains $O(1/\varepsilon^2)$.

3 General Movement Costs

In this section we address the problem of fully-dynamic BIN PACKING with bounded recourse under general movement costs. We first show that the fully-dynamic problem under general movement costs cannot achieve a better a.c.r than the (arrival-only) online problem. Next, we show how to match the a.c.r achievable by any SUPER-HARMONIC algorithm, showing the fully-dynamic problem performs no worse than the current known upper bounds for online BIN PACKING.

3.1 Matching the Lower Bounds for Online Algorithms

Formally, an adversary process $B$ for the online BIN PACKING problem is a potentially adaptive process that, depending on the state of the system (i.e., the current set of configurations used to pack the current set of items) either adds a new item to the system, or stops the request sequence. We say that an adversary process shows a lower bound of $c$ for deterministic online algorithms for BIN PACKING if for any (deterministic) online algorithm $A$, this process starting from the empty system always eventually reaches a state where the a.c.r is at least $c$.

**Theorem 3.1.** Let $\beta \geq 2$. Any adversary process $B$ showing a lower bound of $c$ for the a.c.r of any (deterministic) online BIN PACKING algorithm can be converted into a fully-dynamic BIN PACKING instance with general movement costs such that any (deterministic) fully-dynamic BIN PACKING algorithm with amortized recourse at most $\beta$ must have a.c.r at least $c$.

Such a claim is simple for worst-case recourse. Indeed, given a recourse bound $\beta$, set the movement cost of the $i$-th item to be $(\beta + \varepsilon)^{n-i}$ for $\varepsilon > 0$, and consider a sequence of insertions. When item $i$ arrives we cannot repack any previous item because their movement costs are larger by a factor of $\beta$, and hence this becomes a regular online algorithm. This argument does not hold if we allow for amortization, and hence we give a more careful argument.

We give a proof sketch for a special case and defer the full proof to B.1 due to space constraints. Given any adversary adversary process $B$ for the online BIN PACKING, we construct an adversary process $A$ for the fully-dynamic BIN PACKING. If $B$ proceeds to give a sequence of items $e_1, e_2, \ldots, e_n$, (in that
order), then $A$ gives the elements in the same order while maintaining the invariant: when a new element $e_{k+1}$ is inserted, the fully-dynamic algorithm will not move any previous item $e_1, \ldots, e_k$. If the algorithm moves some items, and $e_j$ is the item with the smallest index that is repacked, $A$ deletes items $e_{j+1}, \ldots, e_{k+1}$. This may in turn cause elements to be repacked, so $A$ will delete all items with indices strictly higher than the smallest index item that is repacked. Eventually it stops at a state with $k' \geq 1$ elements such that only element $e_{k'}$ has been repacked.

By maintaining the invariant, $A$ ensures that any algorithm for the fully-dynamic setting can be translated into an algorithm for the online setting. Therefore, if we ever reach the end of the sequence, then we would have a sequence of moves that can be emulated by an online algorithm. We will set the movement costs to be geometrically decreasing, and use a potential function argument to show that $A$ will either reach the end of the sequence or violate the recourse bounds. See §B.1 for the details.

Given the best lower bound for online bin packing of Balogh et al. [3], Theorem 3.1 implies:

**Corollary 3.2.** No fully-dynamic bin packing algorithm with bounded recourse under general movement costs and $o(n)$ additive term is better than 1.54037-asymptotically competitive.

### 3.2 (Nearly) Matching the Upper Bounds for Online Algorithms

Now we turn to showing upper bounds. To give a sense of our techniques, we start with a simple 2-a.c.r algorithm, and then extend the 1.6901-a.c.r of the HARMONIC algorithm [22] to the fully-dynamic setting. We then summarize our ideas to extend any Super-Harmonic algorithm, with the full details deferred to §B.

#### 3.2.1 A Simple 2-approximate Solution

The main difficulty with extending the ideas used for fully-dynamic algorithms for weight costs or unit costs, is that for general costs, size and cost need not be related in any way. For example, if some large and small items share bins, with large items having small movement cost, then deleting the large items may cause the small items to occupy only a small fraction of the bins, and hence result in a packing which is far from optimal. As a warm-up, let us consider the case where items have roughly the same size, where this does not pose such a problem.

**Lemma 3.3 (Near-Uniform Sizes).** There exists a fully-dynamic bin packing algorithm with constant worst case recourse which, given an instance only of items having size $[1/k, 1/(k-1))$ for some integer $k \geq 1$, packs them into bins, of which all but one contain $k-1$ items and are hence at least $1-1/k$ full. (If all items have size exactly $1/k$, the algorithm packs $k$ items in all bins but one.)

**Proof.** We round down movement costs of each item to the next-lower power of two. We maintain all items sorted by movement cost, with the costliest items in the first bin. All bins (except perhaps the last bin) contain $k-1$ items; if all items have size $1/k$ then bins contain $k$ items. Insertion and deletion of an item of cost $c_i = 2^i$ can be assumed to be performed at the last bin containing an item of this cost (possibly incurring an extra movement cost of $c_i$, by replacing item $i$ with another item of cost $c_i$). If addition or deletion leaves a bin with one item too many or too few, we move a single item to/from the last bin containing an item of the next lower cost, and so on. Since items are sorted and the costs are powers of 2, the total movement cost is at most $c_i + c_i \cdot (1 + 1/2 + 1/4 + \ldots) = 3 \cdot c_i$; the loss due to rounding means the (worst case) recourse is $\leq 6$. The lemma follows.

Suppose we round all item sizes to powers of 2 and use Lemma 3.3 on items of size at least $1/n$, where $n$ is the maximum number of items in the instance $I_t$ over all times $t$, while packing all items of size.
less than $1/n$ into a single bin. Then, we ensure that all but $1 + \log_2 n$ bins are at least $1/2$ full, yielding the following fact.

**Fact 3.4.** There exists a fully-dynamic bin packing algorithm with a.c.r $2$ and additive term $1 + \log_2 n$, using constant worst case recourse.

As stated in Theorem 1.3, better bounds are achievable (both in the a.c.r, and in the additive term, which can be made independent of $n$). However, the idea of packing items of nearly equal sizes together is a recurring idea in online bin packing algorithms, and proves useful for dynamic algorithms, too. We now move to a simple example of this approach.

### 3.2.2 A Simple Upper Bound: The Harmonic Algorithm

The Harmonic algorithm packs all items of size in the range $[1/k, 1/(k - 1)]$ in bins containing at least $k - 1$ such items, while packing small items (of size at most $\varepsilon$) into dedicated bins which are at least $1 - \varepsilon$ full on average, using e.g., FirstFit. This algorithm uses $(1.6901 + O(\varepsilon)) \cdot \OPT + O(\varepsilon^{-1})$ bins (see Lee and Lee, [22]). To obtain the same performance with limited recourse, we use Lemma 3.3 to pack items of sizes greater than $\varepsilon$; for such items our packing would match the requirements of the Harmonic algorithm.

In §B.2.3, we show how to keep all items of size at most $\varepsilon$ in bins which are at least $1 - O(\varepsilon)$ full using $O(\varepsilon^{-2})$ amortized recourse. This requires some new ideas, as we need to handle deletions as well as insertions (unlike the online case). First, we partition the bins into into buckets of $\Theta(1/\varepsilon)$ many bins, such that all but the last bin in a bucket are $1 - O(\varepsilon)$ full, which guarantees these bins are $1 - O(\varepsilon)$ full on average. Maintaining this property is intricate as size and cost need not be commensurate. In addition to the bucketing scheme, we maintain the small items in sorted order according to their size to movement cost ratio (also known as their smith ratio). However, insertion of a small item can create a large cascade of movements throughout the bucket. We only move items to/from a bin once it has $\Omega(\varepsilon)$’s worth of volume removed/inserted (keeping track of erased, or “ghost” items); combined with a potential function, this allows us to achieve amortized $O(\varepsilon^{-2})$ recource cost for small items under general movement costs.

Combining the two, we get the following result.

**Lemma 3.5** (Simulation of Harmonic with Bounded Recourse). For all $\varepsilon > 0$, there exists a fully-dynamic $1.6901 + O(\varepsilon)$ algorithm with $O(\varepsilon^{-1})$ additive term and $O(\varepsilon^{-2})$ amortized recourse.

### 3.2.3 Seiden’s Super-Harmonic Algorithms

Seiden [26] introduced the Super Harmonic family of algorithms to capture the many extensions of Harmonic (e.g., [22, 23, 24, 26, 31]). We give a short overview of this family of algorithms and show how to generalize our approach of §3.2.2 to capture all Super Harmonic algorithms.

A Super-harmonic (abbreviated as SH) algorithm partitions the unit interval $[0, 1]$ into some $K + 1$ intervals $[0, \varepsilon], (t_0 = \varepsilon, t_1, t_2, \ldots, (t_{K-1}, t_K = 1]$. Small items (i.e., items of size at most $\varepsilon$) are packed into dedicated bins which are $1 - \varepsilon$ full. Say a large item is of type $i$ if its size is in the range $(t_{i-1}, t_i]$. The algorithm also colors items blue or red. Each bin contains items of at most two distinct item types $i$ and $j$. If a bin contains only one item type, all its items are colored the same. If a bin contains two item types $i \neq j$, all type $i$ items are colored blue and type $j$ ones are colored red (or vice versa). The SH algorithm has associated with it three sequences $(\alpha_i)^K_{i=1}, (\beta_i)^K_{i=1}, (\gamma_i)^K_{i=1}$, and an bipartite compatibility graph $G = (V, E)$ whose role will be made clear shortly. A bin with blue
(resp., red) type $i$ items contains at most $\beta_i$ (resp., $\gamma_i$) items of type $i$, and is open if it contains less than $\beta_i$ type $i$ (resp., less than $\gamma_j$ type $j$ items). The compatibility graph determines which pair of (colored) item types can share a bin. The compatibility graph $G = (V,E)$ is defined on the vertex set $V = \{b_i \mid i \in [K]\} \cup \{r_j \mid j \in [K]\}$, with an edge $(b_i,r_j) \in E$ indicating blue items of type $i$ and red items of type $j$ are compatible and may share a bin. In addition, an SH algorithm must satisfy the following invariants.

(P1) The number of open bins is $O(1)$.

(P2) If $n_i$ is the number of type-$i$ items, the number of red type-$i$ items is $\lfloor \alpha_i \cdot n_i \rfloor$.

(P3) If $(b_i,r_j) \in E$ (blue type $i$ items and red type $j$ items are compatible), there is no pair of bins with one containing nothing but blue type $i$ items and one containing nothing but red type $j$ items.

These invariants, along with the choices of $(\alpha_i)_{i=1}^K, (\beta_i)_{i=1}^K, (\gamma_i)_{i=1}^K$ and the compatibility graph $G$ allow one to bound the a.c.r of any SH algorithm. For example, one can get the following result:

**Lemma 3.6** (Seiden [26]). There exists an SH algorithm with a.c.r $1.58889$.

### Emulating SH algorithms

In a sense, SH algorithms ignore the exact size of large items, so we can take all items of some type and color, and extend Lemma 3.3 to pack at most $\beta_i$ or $\gamma_i$ of them per bin. This allows us to satisfy Properties (P1) and (P2) with a little more work. The major challenge, however, is in maintaining Property (P3). To see this, consider a bin with $\beta_i$ blue type $i$ items and $\gamma_j$ type-$j$ items, and suppose the type $i$ items are all removed. Moreover, suppose there exists an open bin with items of type $i'$ different from $i$ compatible with $j$. If the movement costs of both type $j$ and type $i'$ items are significantly higher than the cost of the type $i$ items, we cannot afford to place these groups together, potentially violating Property (P3). To avoid such a problem, we use ideas from stable matchings. We think of groups of $\beta_i$ blue type-$i$ items and $\gamma_j$ red type-$j$ items as nodes in a bipartite graph, with an edge between these nodes if $G$ contains the edge $(b_i,r_j)$. We now maintain a stable matching under updates, with priorities being the value of the most expensive item in a group of same-type items packed in the same bin. The stability of this matching implies Property (P3). We then maintain this stable matching using (a slight modification of) the Gale-Shapley algorithm. Finally, relying on our solution for packing small items as in §3.2.2, we find that we can pack the small items in bins which are $1 - \varepsilon$ full on average, which together with the above approach for packing large items in a way satisfying the properties of SH algorithms. Combined with Lemma B.1 we obtain following theorem.

**Theorem 3.7.** There exists a fully-dynamic bin packing algorithm with a.c.r $1.58889$ and constant additive term using constant amortized recourse under general movement costs.

Note that our recourse bounds here are only amortized due to our solution for small items, given in Lemma B.4. In particular, if we know that all item sizes are at least some $\delta$, then our algorithm for small items yields an $O(1/\delta)$ worst case recourse cost (see Corollary B.8). Alternatively, packing items of different size ranges separately allows us to obtain worst case recourse, at the cost of an additive $O(\log n)$ number of bins (see Corollary B.9).

**Theorem 3.8.** There exists a fully-dynamic bin packing algorithm with a.c.r $1.58889$ and constant additive term using constant worst case recourse under general movement costs, assuming items have size at least some $\Omega(1)$. 

22
4 Weight Movement Costs (Migration Factor)

In this section, we consider the problem of fully dynamic bin packing with limited recourse, where the movement cost of each item is equal to its size. That is, \( c_j = s_j \) for each item \( j \). The recourse cost in this case is referred to as migration factor, and has been studied in the context of scheduling problems (see [1] [11] [23] [27]) as well as in our context, that of (fully) dynamic bin packing (see [6] [10] [19]). Berndt et al. [6] show that under this recourse cost, one can achieve an asymptotic competitive ratio of \((1 + \varepsilon)\) using \( \tilde{O}(\varepsilon^{-4}) \) recourse cost. They also show an information-theoretic lower bound of \( \Omega(\varepsilon^{-1}) \) on the worst case recourse cost of any asymptotically \((1 + \varepsilon)\)-competitive algorithm for this problem. We show that this tradeoff is inherent, even allowing for amortization, and determine the optimal tradeoff between asymptotic competitive and recourse in this model. We start by stating the upper bound part of this result.

**Fact 4.1.** For all \( \varepsilon \leq 1/2 \), there exists an algorithm requiring \((1 + O(\varepsilon)) \cdot OPT(I_t) + O(\varepsilon^{-2})\) bins at all times \( t \) while using only \( O(\varepsilon^{-1}) \) amortized migration factor.

The above upper bound is trivial – it suffices to repack according to an AFPTAS whenever the volume changes by a multiplicative \((1 + \varepsilon)\) factor (for completeness, we give this theorem’s proof in §C). The challenge here is in showing this algorithm’s a.c.r to recourse tradeoff is tight. We do so by constructing an instance where addition or removal of small items of size \( \varepsilon \) causes every near-optimal solution to be far from every near-optimal solution to the current instance.

4.1 Matching Amortized Migration Factor Lower Bound

In this section we give a lower bound matching the upper bound of Fact 4.1, showing that any \((1 + \varepsilon)\) asymptotically competitive algorithm must have amortized recourse \( \Omega(\varepsilon^{-1}) \) under weight movement costs.

Our proof relies on the well-known Sylvester sequence [28], given by the recurrence relation \( k_1 = 2 \) and \( k_{i+1} = (\prod_{j \leq i} k_j) + 1 \).

While this sequence has been used previously in the context of bin packing, our lower bound relies on more fine-grained divisibility properties. We crucially use the property that \( k_i \) are relatively prime, yet their inverses are integer multiples of \( 1 - \sum_{i=1}^{c} 1/k_i \); see Properties (P1)-(P4) below.

**Theorem 4.2.** For infinitely many \( \varepsilon > 0 \), any fully-dynamic bin packing algorithm with asymptotic competitive ratio \((1 + \varepsilon)\) and additive term \( o(n) \) must have amortized migration factor of \( \Omega(\varepsilon^{-1}) \).

For our proof we rely on Sylvester’s sequence [28], given by the recurrence relation \( k_1 = 2 \) and \( k_{i+1} = (\prod_{j \leq i} k_j) + 1 \), or equivalently \( k_{i+1} = k_i(k_i - 1) + 1 \) for \( i \geq 0 \), the first few terms of which are 2, 3, 7, 43, 1807, \ldots. In what follows we let \( c \) be a positive integer to be specified later, and \( \varepsilon := 1/\prod_{\ell=1}^{c} k_\ell \). In our proof of Theorem 4.2 and the proofs of the lemmas building up to this proof, we shall make use of the following properties of this sequence and \( \varepsilon \):

(P1) \( \frac{1}{k_1} + \frac{1}{k_2} + \ldots + \frac{1}{k_c} = 1 - \frac{1}{\prod_{\ell=1}^{c} k_\ell} = 1 - \varepsilon \).

(P2) If \( i \neq j \), then \( k_i \) and \( k_j \) are relatively prime.

(P3) For all \( i \in [c] \), the value \( 1/k_i = \prod_{\ell \in [c]\setminus\{i\}} k_ell / \prod_{\ell=1}^{c} k_\ell \) is an integer product of \( \varepsilon = 1/\prod_{\ell=1}^{c} k_\ell \).

(P4) If \( i \neq j \in [c] \), then \( 1/k_i = \prod_{\ell \in [c]\setminus\{i\}} k_ell / \prod_{\ell=1}^{c} k_\ell \) is an integer product of \( k_j \cdot \varepsilon = k_j / \prod_{\ell=1}^{c} k_\ell \).
For notational simplicity, since our products and sums will always be taken over the range \([c]\) or \([c]\setminus\{i\}\) for some \(i \in [c]\), we will write \(\sum_{\ell} k_{\ell}\) and \(\prod_{\ell} 1/k_{\ell}\), \(\sum_{\ell \neq i} 1/k_{\ell}\), \(\prod_{\ell \neq i} 1/k_{\ell}\), etc., taking the range of \(\ell\) to be self-evident from context. We define a vector \(s \in [0,1]^{c+1}\) as follows:

\[
s_i := \begin{cases} 
\frac{1}{k_i} \cdot (1 - \frac{\varepsilon}{2}) & i \in [c] \\
\varepsilon \cdot (\frac{3}{2} - \frac{\varepsilon}{2}) & i = c + 1
\end{cases}
\]

The vector \(s\) corresponds to the different item sizes that will be present in the input instance. Let \(\bar{1}\) be the all-ones vector, and \(\bar{e}_i\) the \(i\)-th standard basis vector. Henceforth, \(\bar{x} \in \mathbb{N}^{c+1}\) is a characteristic vector of a feasibly-packed bin; i.e., \(\chi_{i}\) is the number of items of size \(s_i\) in the bin, and as the bin is not over-flowing, we have \(\bar{x} \cdot s = \sum_i \chi_i \cdot s_i \leq 1\). The following fact is easy to check from the definition of \(s\).

**Lemma 4.3.** \(\bar{1} \cdot s = 1\).

**Proof.** By property \((P1)\) we have \(\sum_{i=1}^{c} s_i = (1 - \frac{\varepsilon}{2})(1 - \varepsilon) = 1 - s_{c+1}\).

We now show that the item sizes given by \(s\) are such that in many situations, any feasible packing in a bin will leave some gap.

**Lemma 4.4.** Let \(x\) denote a feasible packing of items into a bin, i.e., \(\bar{x} \in \mathbb{N}^{c+1}\), \(\bar{x} \cdot s \leq 1\). Then,

(a) If \(\chi_{c+1} = 1\) and \(\bar{x} \neq \bar{1}\), then \(\bar{x} \cdot s \leq 1 - \varepsilon/2\).

(b) If \(\chi_{c+1} = 0\), then \(\bar{x} \cdot s \leq 1 - \varepsilon/2\).

(c) If \(\chi_{c+1} = 0\) and \(\bar{x} \neq k_i \bar{e}_i\), then \(\bar{x} \cdot s \leq 1 - \varepsilon\).

**Proof.** For ease of notation, we define a vector \(\bar{t}\), where \(t_i = 1/k_i\) for \(i = 1, \ldots, c\) and \(t_{c+1} = 0\). So, \(s_i = (1 - \varepsilon/2) t_i\) for \(i = 1, \ldots, c\).

**Observation 4.5.** The dot product \(\bar{x} \cdot \bar{t}\) is at most 1.

**Proof.** Suppose not. By our choice of \(\varepsilon\), each \(t_i\) is an integral multiple of \(\varepsilon\), by property \((P3)\) it follows that \(\bar{x} \cdot \bar{t}\) is also an integral multiple of \(\varepsilon\). Clearly, 1 is an integral multiple of \(\varepsilon\). Therefore, \(\bar{x} \cdot \bar{t}\) is at least \(1 + \varepsilon\). But then, \(\bar{x} \cdot s \geq (1 - \varepsilon/2) \bar{x} \cdot \bar{t} \geq (1 - \varepsilon/2)(1 + \varepsilon) > 1\), a contradiction.

**Part (a)** First, observe that \(\bar{x} \neq \bar{1}\). Indeed, it is at most 1, by Observation 4.5. Moreover, if \(\bar{x} \cdot \bar{t} = 1\), the fact that \(\chi_{c+1} = 1\) would imply \(\bar{x} \cdot s = \bar{x} \cdot (1 - \varepsilon/2) \bar{t} + s_{c+1} = (1 - \varepsilon/2) + s_{c+1} > 1\), a contradiction. Also, there must be some index \(i \in [c]\) such that \(\chi_i = 0\). Indeed, otherwise the fact that \(\bar{x} \neq \bar{1}\) and Lemma 4.3 imply that \(\bar{x} \cdot s > \bar{1} \cdot s = 1\). So let \(i\) be an index such that \(\chi_i = 0\). By property \((P4)\) for all \(j \neq i\), the value \(t_j = 1/k_j\) is an integral multiple of \(k_i \varepsilon\). Therefore, \(\bar{x} \cdot \bar{t}\) is a multiple of \(k_i \varepsilon\). Since \(\bar{x} \cdot \bar{t} < 1\) and 1 is an integral multiple of \(k_i \varepsilon\), it follows that \(\bar{x} \cdot s \leq 1 - k_i \varepsilon \leq 1 - 2\varepsilon\). Therefore, \(\bar{x} \cdot s \leq (1 - \varepsilon/2)(1 - 2\varepsilon) + s_{c+1} = 1 - \varepsilon + \varepsilon^2/2 \leq 1 - \varepsilon/2\).

**Part (b)** follows directly from Observation 4.5 above. Indeed, since \(\chi_{c+1} = 0\), we have that \(\bar{x} \cdot s = (1 - \varepsilon/2) \bar{x} \cdot \bar{t} \leq (1 - \varepsilon/2)\).

**Part (c)** First, observe that \(\chi_i < k_i\) for \(i = 1, \ldots, c\). Indeed, if \(\chi = k_i \bar{e}_i + \chi'\) for some index \(i\) and some non-zero non-negative vector \(\chi'\), then \(\bar{x} \cdot \bar{t} = 1 + \bar{x} \cdot \bar{t} > 1\), which contradicts Observation 4.5. Now let \(i\) be an index such that \(\chi_i \geq 1\). Since \(\chi_i < k_i\), property \((P2)\) implies that \(\chi_i t_i = \chi_i \varepsilon \prod_{\ell \neq i} k_{\ell}\) is not an integral multiple of \(k_i \varepsilon\). But, by property \((P4)\) for all \(j \neq i\) we have that \(t_j = 1/k_j\) is an integral multiple of \(k_i \varepsilon\). Thus, \(\bar{x} \cdot \bar{t}\) is not an integral multiple of \(k_i \varepsilon\). Consequently, \(\bar{x} \cdot \bar{t} \leq 1\). But, by property \((P3)\) we have that \(\bar{x} \cdot \bar{t}\) is an integral multiple of \(\varepsilon\), from which it follows that \(\bar{x} \cdot s \leq 1 - \varepsilon\). Therefore, as \(\chi_{c+1} = 0\), we have \(\bar{x} \cdot s = \bar{x} \cdot (1 - \varepsilon/2) \bar{t} \leq (1 - \varepsilon/2)(1 - \varepsilon) \leq 1 - \varepsilon\).
Equipped with the above lemma, we now proceed to proving this section’s main result.

**Proof of Theorem 4.2.** We will show that any algorithm \( A \) using at most \((1 + \varepsilon / 7) \cdot OPT(I_t) + o(n_t)\) bins at time \( t \) uses at least \( 1 / 160c \) amortized recourse, for arbitrarily large optima and instance sizes, implying our theorem. Continuing with our previous notation, we will consider two instances \( I \) and \( I' \) with item sizes in \( \{ s_i | i \in [c + 1] \} \). The two instances will look very similar but (as we shall see) algorithm \( A \) needs to pack them very differently. The instance \( I \) consists of \( N(c + 1) \) items, with \( N \) items for each of the sizes \( s_1, \ldots, s_{c+1} \). Similarly, the instance \( I' \) consists of \( Nc \) items, with \( N \) items for each of the sizes \( s_1, \ldots, s_c \). Here \( N \) is a sufficiently large multiple of \( 1 / \varepsilon = \prod_{i \in [c]} k_i \). It is easy to check that \( c = \Theta(\log \log(1 / \varepsilon)) \), and so the total number of items is \( n = \Theta(N \cdot \log \log(1 / \varepsilon)) \). Therefore the additive \( o(n) \) term, denoted by \( f(n) \), satisfies

\[
    f(n) < \varepsilon \cdot n / (42 \cdot \Theta(\log \log(1 / \varepsilon))) = \varepsilon N / 42. \tag{3}
\]

Henceforth we will assume that the algorithm \( A \) will use at most \((1 + \varepsilon / 7) \cdot OPT(I_t) + \varepsilon N / 42 \) bins for instance \( I \) at time \( t \) (and similarly for the instance \( I' \)). We now show that the algorithm \( A \) must pack items in \( I \) and in \( I' \) in very different manner.

**Observation 4.6.** Given the instance \( I \), the algorithm \( A \) must pack the items such that there at least \( 2N / 3 \) bins for which the characteristic vector \( \chi \) is \( \vec{I} \).

**Proof.** Lemma 4.3 shows that the optimal number of bins is \( N \). Therefore, \( A \) is allowed to use at most \((1 + \varepsilon / 7)N + \varepsilon N / 42 = (1 + \varepsilon / 6)N \) bins. Now suppose there more than \( N / 3 \) bins for which the characteristic vector is not \( \vec{I} \). Lemma 4.4(a) shows that each such bin must leave out at least \( \varepsilon / 2 \) space. Therefore, the total unused space in these bins is greater than \( \varepsilon N / 6 \), which implies that the algorithm must use at least \( N + \varepsilon N / 6 \) bins, a contradiction.

We now show a similar result for the instance \( I' \).

**Observation 4.7.** Given the instance \( I' \), the algorithm \( A \) must pack the items such that there at least \( N / 2 \) bins for which the characteristic vector \( \chi \) has only one non-zero entry.

**Proof.** Let us first find the optimal value \( OPT(I') \). By property \([P1]\) the total volume of all the items is equal to \( N(1 - \varepsilon)(1 - \varepsilon / 2) \). By Lemma 4.4(b), any bin can be packed to an extent of at most \((1 - \varepsilon / 2)N \) bins. Therefore the optimal number of bins is at least \( N(1 - \varepsilon) \). Furthermore, we can achieve this bound by packing \( N \) items of size \( s_i \) in \( N / k_i \) bins for each index \( i \). Therefore, the algorithm is allowed to use at most \((1 + \varepsilon / 7)(1 - \varepsilon)N + \varepsilon N / 42 \leq (1 - 35\varepsilon / 42)N \) bins when packing \( I' \).

Suppose there are at least \( N / 2 \) bins each of which is assigned items of at least two different sizes by the algorithm. By Lemma 4.4(c), the algorithm will leave at least \( \varepsilon \) unused space in such bins; moreover, by Lemma 4.4(b), the algorithm will leave at least \( \varepsilon / 2 \) unused space in every bin. Thus, the total unused space in the bins is at least \( \varepsilon N / 2 + \varepsilon N / 4 = 3 \varepsilon N / 4 \). Since the total volume of the items is equal to \( N(1 - \varepsilon)(1 - \varepsilon / 2) + 3 \varepsilon N / 4 \geq N(1 - 3 \varepsilon / 4) > (1 - 35\varepsilon / 42)N \), a contradiction.

Now, suppose we first provide instance \( I \) to \( A \), and then remove all items of size \( s_{c+1} \) to get the instance \( I' \). Let \( B_1 \) and \( B_2 \) be the sets of bins guaranteed by Observations 4.6 and 4.7 when we had the instances \( I \) and \( I' \), respectively. Notice that as algorithm \( A \) uses at most \((1 - 35\varepsilon / 42)N \) \( \leq N \) bins while packing \( I' \), and \(|B_1| \geq 2N / 3\), \(|B_2| \geq N / 2\), we have that either (i) \(|B_1 \cap B_2| \geq N / 12\), or (ii) at least \( N / 12 \) of the bins of \( B_1 \) are closed in \( A \)'s packing of \( I' \).

Consider any bin which lies in both \( B_1 \) and \( B_2 \). In instance \( I \), algorithm \( A \) had assigned one item of each size to this bin, whereas in instance \( I' \) the algorithm assigns this bin items of only one size. Therefore, the total size of items which need to go out of (or into) this bin when we transition from
$I$ to $I'$ is at least $1/2$. Therefore, the total volume of items moved during this transition is at least $N/48$. Similarly, if $N/12$ of the bins of $B_1$ are closed in $A$'s packing of $I'$, at least $1 - s_{c+1} \geq 1/2$ volume must leave each of these bins, and so the total volume of items moved during this transition is at least $N/24 > N/48$.

Repeatedly switching between $I$ and $I'$ by adding and removing the $N$ items of size $s_{c+1}$ a total of $T$ times (for sufficiently large $T$), we find that the amortized recourse is at least

$$\frac{T \cdot N/48}{N + T \cdot 3\varepsilon \cdot N} \geq \frac{1}{160\varepsilon}.$$
Appendix

A Tabular List of Prior and Current Results

Here we recap and contrast the previous upper and lower bounds for dynamic bin packing with bounded recourse, starting with the upper bounds, in Table 1.

| Costs | A.C.R | Additive | Recourse | W.C. | Notes | Reference |
|-------|-------|----------|----------|------|-------|-----------|
| General | 2 | 1.58889 | \log n | 1 | ✗ | w.c. if n known |
| 4/3 | 1 | 1 | ✗ | | | Gambosi et al. [13] |
| Unit | \(\frac{3}{2} + \epsilon\) | \(\epsilon^{-1}\) | \(\epsilon^{-1}\) | ✓ | insertions only | Balogh et al. [5] |
| \(\alpha + \epsilon\) | \(\epsilon^{-2}\) | \(\epsilon^{-2}\) | ✗ | | | Balogh et al. [5] |
| Weight | \(1 + \epsilon\) | \(\epsilon^{-2}\) | \(\epsilon^{-O(\epsilon^{-2})}\) | ✓ | insertions only | Epstein & Levin [10] |
| \(\alpha - \epsilon\) | \(\epsilon^{-3}\log(\epsilon^{-1})\) | \(\epsilon^{-4}\) | ✓ | | | Jansen & Klein [19] |
| \(\alpha - \epsilon\) | \(\epsilon^{-2} \cdot n^\delta\) | \(\Omega(\epsilon^{-2} \cdot n^{1-\delta})\) | ✗ | for all \(\delta \in (0, 1/2]\) | | Berndt et al. [6] |

We contrast these upper bounds with matching and nearly-matching lower bounds, in Table 2. Note that here an amortized bound is a stronger bound than a corresponding worst case bound.

| Costs | A.C.R | Additive | Recourse | W.C. | Notes | Reference |
|-------|-------|----------|----------|------|-------|-----------|
| General | 1.54037… | \(o(n)\) | ∞ | ✗ | as hard as online | | Theorem 3.1 |
| Unit | \(\frac{4}{3}\) | \(o(n)\) | 1 | ✓ | \(\alpha \approx 1.3871\) | Ivković & Lloyd [18] |
| \(\alpha - \epsilon\) | \(o(n)\) | 1 | ✓ | | | Balogh et al. [4] |
| \(\alpha - \epsilon\) | \(o(\epsilon^2 \cdot n^\delta)\) | \(\Omega(\epsilon^{-2} \cdot n^{1-\delta})\) | ✗ | | | Theorem 2.2 |
| Weight | \(1 + \epsilon\) | \(o(n)\) | \(\Omega(\epsilon^{-1})\) | ✓ | | | Berndt et al. [6] |
| \(1 + \epsilon\) | \(o(n)\) | \(\Omega(\epsilon^{-1})\) | ✗ | | | Theorem 4.2 |

We emphasize again the tightness and near-tightness of our upper and lower bounds for the different movement costs. For general movement costs we show that the problem is at least as hard as online bin packing (without repacking), while the problem admits a 1.58889-asymptotically competitive algorithm, nearly matching the state of the art 1.5815 online algorithm of [15]. For unit movement costs, we show the lower bound of \(\alpha\) of Balogh et al. [4] is sharp, by presenting an algorithm with a.c.r of \(\alpha + \epsilon\) and additive term and recourse polynomial in \(1/\epsilon\). We further simplify and strengthen the previous lower bound, by showing that any algorithm with a.c.r better than \(\alpha\) requires polynomial additive term times recourse. Finally, for weight movement costs, we show that an a.c.r of \(1 + \epsilon\) implies a recourse cost of \(\Omega(\epsilon^{-1})\), even allowing for amortization (strengthening the hardness result of Berndt et al. [6]). Our simple lazy algorithm which matches this bound proves this tradeoff to be optimal.
B Omitted Proofs of Section 3 (General Movement Costs)

B.1 Matching the Lower Bounds for Online Algorithms

Below we give the proof of 3.1

Proof. Take the adversary process \( B \), and use it to generate an instance for the fully-dynamic algorithm \( A \) as follows. (When there are \( k \) items in the system, let them be labeled \( e_1, e_2, \ldots, e_k \).)

I: Given system in a state with \( k \) elements, use \( B \) to generate the next element \( e_{k+1} \), having movement cost \((2\beta)^{-k-1}\).

II: If \( A \) places \( e_{k+1} \) into some bin and does not move any other item, go back to Step I to generate the next element.

III: However, if \( A \) moves some items, and \( e_j \) is the item with the smallest index that is repacked, delete items \( e_{j+1}, \ldots, e_{k+1} \). This may in turn cause elements to be repacked, so delete all items with indices strictly higher than than the smallest index item that is repacked. Eventually we stop at a state with \( k' \geq 1 \) elements such that only element \( e_{k'} \) has been repacked. Now go back to Step I. (Also, \( e_{k'+1}, \ldots, e_{k+1} \) are deemed as being undefined.)

Since the location of each item \( e_i \) is based only on the knowledge of prior elements in the sequence \( e_1, e_2, \ldots, e_{i-1} \) and their bins, the resulting algorithm is another online algorithm. So if the length of the sequence eventually goes to infinity, we are guaranteed to reach an instance for which the a.c.r of this algorithm will be at least \( c \). Hence we want to show that for any \( n \), the length of the sequence eventually reaches \( n \) (or the adversary process stops, having showed a lower bound of \( c \)). Consider a potential function \( \Phi \) which is zero when the system has no elements. When a new element is added by \( B \), we increase \( \Phi \) by \( \beta \) times the movement cost for this element. Moreover, when \( A \) moves elements, we subtract the movement costs of these elements from \( \Phi \). Since \( A \) ensures an amortized recourse bound of \( \beta \), the potential must remain non-negative.

For a contradiction, suppose the length of the sequence remains bounded by \( n \). Hence, there is some length \( k < n \) such that Step III causes the sequence to become of length \( k \) arbitrarily often. Note that the total increase in \( \Phi \) between two such events is at most \( \beta \sum_{i=k+1}^{n}(2\beta)^{-i} \leq \frac{(2\beta)^{-k-1}}{2\beta-1} \). Since \( \beta \geq 2 \), this increase is strictly less than \( (2\beta)^{-k} \), the total decrease in \( \Phi \) due to the movement of element \( k \) alone. Since the potential decreases between two such events, there can only be finitely many such events, so the length of the sequence, i.e., the number of items in the system increases over time, eventually giving us the claimed lower bound.

\( \square \)

B.2 (Nearly) Matching the Upper Bounds for Online Algorithms

Now for upper bounds: we show how to implement algorithms in the Super-Harmonic framework in a fully-dynamic setting under general movement costs. We start with a brief high-level exposition of the Super Harmonic framework of Seiden [26].

B.2.1 The Super Harmonic family of online algorithms

A Super-harmonic (abbreviated as SH) algorithm consists of a partition of the unit interval \([0,1]\) into \( K+1 \) intervals, \([0,\varepsilon], (t_0 = \varepsilon, t_1)[t_1, t_2], \ldots, (t_{K-1}, t_K = 1] \). Small items (i.e., items of size at most \( \varepsilon \)) are packed using FirstFit into dedicated bins; because the items are small, all but one of these are at least \( 1 - \varepsilon \) full. For larger items, each arriving item is of type \( i \) if its size is in the range \((t_{i-1}, t_i]\). Items are colored either blue or red by the algorithm, with each bin containing items of at most two distinct item types \( i \) and \( j \). If a bin contains only one item type, its items are all colored the same, and if a bin contains two item types \( i \) and \( j \), then all items of type \( i \) are colored blue and
items of type \(j\) are colored red (or vice versa). The SH algorithm has associated with it three number sequences \((\alpha_i)_{i=1}^{K}, (\beta_i)_{i=1}^{K}, (\gamma_i)_{i=1}^{K}\), and an underlying bipartite compatibility graph \(\mathcal{G} = (V,E)\) whose role will be made clear shortly. A bin with blue (resp., red) type \(i\) items contains at most \(\beta_i\) (resp., \(\gamma_i\)) items of type \(i\), and is open if it contains less than \(\beta_i\) type \(i\) (resp., less than \(\gamma_j\) type \(j\) items). The compatibility graph determines which pair of (colored) item types can share a bin. The compatibility graph \(\mathcal{G} = (V,E)\) is defined on the vertex set \(V = \{b_i \mid i \in [K]\} \cup \{r_j \mid j \in [K]\}\), with an edge \((b_i, r_j) \in E\) indicating blue items of type \(i\) and red items of type \(j\) are compatible; they are allowed to be placed in a common bin. Apart from the above properties, an SH algorithm must satisfy the following invariants.

(P1) The number of open bins is \(O(1)\).

(P2) If \(n_i\) is the number of type-\(i\) items, the number of red type-\(i\) items is \([\alpha_i \cdot n_i]\).

(P3) If \((b_i, r_j) \in E\) (blue type \(i\) items and red type \(j\) items are compatible), there is no pair of bins with one containing nothing but blue type \(i\) items and one containing nothing but red type \(j\) items.

The above invariants allow one to bound the asymptotic competitive ratio of an SH algorithm, depending on the choices of \((\alpha_i)_{i=1}^{K}, (\beta_i)_{i=1}^{K}, (\gamma_i)_{i=1}^{K}\) and the compatibility graph \(\mathcal{G}\). In particular, Seiden [26] showed the following.

Lemma B.1 (Seiden [26]). There exists an SH algorithm with a.c.r \(1.58889\).

A particular property the SH algorithm implied by Lemma B.1 which we will make use of later is that its parameters or inverses are all at most a constant; in particular, \(1/\varepsilon = O(1)\) and \(K = O(1)\), and similarly \(\beta_i, \gamma_i = O(1)\) for all \(i \in [K]\).

In the following sections we proceed to describe how to maintain the above invariants that suffice to bound the competitive ratio of an SH algorithm. We start by addressing the problem of maintaining the SH invariants for the large items, in Section B.2.2. We then show how to pack small items (i.e., items of size at most \(\varepsilon\) into bins such that all but a constant of these bins are at least \(1 - \varepsilon\) full, in Section B.2.3. Finally we conclude with our upper bound in Section B.2.5.

**B.2.2 SH Algorithms: Dealing with Large Items**

First, we round all movement costs to powers of 2, increasing our recourse cost by at most a factor of 2. Now, our algorithm will have recourse cost which will be some function of \(K, (\beta_i)_{i=1}^{K}, (\gamma_i)_{i=1}^{K}\); as for our usage, we have \(K = O(1)\), and \(\beta_i, \gamma_i = O(1)\) for all \(i \in [K]\), we will simplify notation and assume that these values are indeed all bounded from above by a constant. Consequently, when moving around groups of up to \(\beta_i\) blue (resp. \(\gamma_i\) red) type \(i\) items whose highest movement cost is \(c\), the overall movement cost will be \(O(c)\). The stipulation that \(K = O(1)\) will prove useful shortly. We now explain how we maintain the invariants of SH algorithms.

**Satisfying Property [P1]**. We keep all blue items (resp. red items) of type \(i\) in bins containing up to \(\beta_i\) items (resp. \(\gamma_i\) items). We sort all bins containing type-\(i\) items by the cost of the costliest type-\(i\) item in the bin, where only the last bin containing type-\(i\) items contains less than \(\beta_i\) blue (alternatively, \(\gamma_i\) red) type-\(i\) items. Therefore, if we succeed in maintaining the above, the number of open bins is \(O(1)\); i.e., Property [P1] is satisfied. We explain below how to maintain this property together with Properties [P2] and [P3].

\[\text{Strictly speaking, we will only pack small items into bins which are } 1 - \varepsilon \text{ full on average. However, redistributing these small items or portions of these items within these bins will make these bins } 1 - \varepsilon \text{ full without changing the number of bins used by the solution. The bounds on SH algorithms therefore carries through.}\]
Satisfying Property (P2). We will only satisfy Property (P2) approximately, such that the number of red type-\(i\) items is in the range \([\alpha_i \cdot n_i], [\alpha_i \cdot n_i] + \delta_i\), where \(\delta_i = \max\{\alpha_i(\beta_i - \gamma_i) + 1 + \gamma_i, 0\}\). Removing these at most \(\delta_i\) red type-\(i\) items results in a smaller bin packing instance, and a solution requiring the same number of bins, satisfying the invariants of SH algorithms. Notice that this change to the solution can change the number of open bins by at most \(\sum_i \delta_i \leq \sum_i \alpha_i\beta_i + K = O(1)\). Therefore, satisfying Property (P2) approximately suffices to obtain our sought-after a.c.r. In fact, we satisfy a stronger property: each prefix of bins containing type-\(i\) items has a number of red type-\(i\) items in the range \([\alpha_i \cdot n_i'], [\alpha_i \cdot n_i'] + \delta_i\), where \(n_i'\) is the number of type-\(i\) items in the prefix.

Maintaining the above prefix invariant on deletion is simple enough: when removing some type-\(i\) item of cost \(2^k\), we move the last type-\(i\) item of the next movement cost to this item’s place in the packing, continuing until we reach a type-\(i\) item with no cheaper type-\(i\) items. The movement cost here is at most \(2^k + 2^{k-1} + 2^{k-2} + \ldots = O(2^k)\), so the (worst case) recourse is constant. As the prefix invariant was satisfied before deletion, it is also satisfied after deletion, as we effectively only remove an item from the last bin. When inserting a type-\(i\) item, we insert the item into the last appropriate bin according to the item’s movement cost. This might cause this bin to overflow, in which case we take the cheapest item in this bin and move it into the last appropriate bin according to this item’s cost, continuing in this fashion until we reach an open bin, or are forced to open a new bin. (The recourse here is a constant, too). The choice of color for type-\(i\) items in a newly-opened bin depends on the number of type-\(i\) items before this insertion, \(n_i\), and the number of red type-\(i\) items before this insertion, \(m_i\). If \(m_i + \gamma_i \leq [\alpha_i(n_i + \gamma_i)] + \delta_i\), the new bin’s red items are colored red. By this condition, the number of red items for the following \(\gamma_i\) insertions into this bin will satisfy our prefix property. If the condition is not satisfied, the bin is colored blue. Now, as

\[
m_i + \gamma_i > [\alpha_i(n_i + \gamma_i)] + \delta_i \\
> [\alpha_i(n_i + \beta_i)] + \alpha_i(\gamma_i - \beta_i) - 1 + \delta_i \\
> [\alpha_i(n_i + \beta_i)] - 1 + \gamma_i
\]

we find that after this new bin contains \(\beta_i\) type \(i\) items, the number of red items in the prefix is \(m_i\), while the number of type-\(i\) items is \(n_i' = n_i + \beta_i\), and so this prefix too satisfies the prefix condition.

We conclude that the above methods approximately maintain Property (P2) (as well as Property (P1)) while only incurring constant worst case recourse.

Satisfying Property (P3). Finally, in order to satisfy Property (P3) we consider the groups of up to \(\beta_i\) and \(\gamma_j\) blue and red items of type \(i\) and \(j\) packed in the same bin as nodes in a bipartite graph. A blue type \(i\) (resp. red type \(j\)) group is a copy of node \(b_i\) (resp. \(r_j\)) in the compatibility graph \(G\), and copies of nodes \(b_i\) and \(r_j\) are connected if they are connected in \(G\). If a particular node \(b_i\) and \(r_j\) are placed in the same bin, then we treat that edge as matched. Each node has a cost which is simply the maximum movement cost of an item in the group which the node represents. All nodes have a preference order over their neighbors, preferring a costlier neighbor, while breaking ties consistently. We will maintain a stable matching in this subgraph, where two nodes are matched if the items they represent. This stability clearly implies Property (P3). We now proceed to describe how to maintain this bipartite graph along with a stable matching in it.

The underlying operations we will have is addition and removal of a node of red or blue type \(i\) items of cost \(2^k\); i.e., with costliest item having movement cost \(2^k\). As argued before, as each group contains \(O(1)\) items, the movement cost of moving items of such a group is \(O(2^k)\). Using this operation we can implement insertion and deletion of single items, by changing the movement cost of groups with items moved, implemented by removal of a group and re-insertion with a higher/lower cost if the cost changes (or simple insertion/removal for a new bin opened/bin closed). As argued above, the cost of items moved is during updates in order to satisfy Property (P2) is \(O(2^k)\), where \(2^k\) is the cost of
the item added/removed; consequently, the costs of removals and insertions of groups is $O(2^k)$. We therefore need to show that the cost of insertion/removal of a group of cost $2^k$ is $O(2^k)$.

Insertions and deletions of blue and red nodes is symmetric, so we consider insertions and deletions of a blue node $b$ only. Upon insertion of some node $b$ of cost $2^k$, we insert $b$ into its place in the ordering, and scan its neighbors for the first neighbor $r$ which strictly prefers $b$ to its current match, $b'$. If no such $r$ exists, we are done. If such an $r$ exists, $b'$ is unmatched from $r$ (its items are removed from its bin) and $b$ is matched to $r$ (the items of $b$ are placed in the same bin as $r$’s bin). As $b'$ has strictly less than $b$, its cost is at most $2^{k-1}$. We now proceed similarly for $b'$ as though we inserted $b'$ into the graph. The overall movement cost is at most $2^k + 2^{k-1} + 2^{k-2} + \cdots = O(2^k)$.

Upon deletion of a blue node $b$ of cost $2^k$, if it had no previous match, we are done. If $b$ did have a match $r$, this match scans its neighbors, starting at $b$, for its first neighbor $b'$ of cost at most $2^k$ which prefers $r$ to its current match. If the cost of $b'$ is $2^k$, we match $r$ to the last blue node of the same type as $b'$, denoted by $b''$. If $b''$ was previously matched, we proceed to match its match as if $b''$ were removed (i.e., as above). As every movement decreases the number of types of blue nodes of a given cost to consider, the overall movement cost is at most $K \cdot (2^k + 2^{k-1} + 2^{k-2} + \cdots) = O(K \cdot 2^k) = O(2^k)$, where here we rely the number of types being $K = O(1)$.

We conclude that Property (P3) can be maintained using constant worst case recourse.

Lemma B.2. Properties (P1), (P2) and (P3) can be maintained using constant worst case recourse.

B.2.3 SH Algorithms: Dealing with Small Items

Here we address the problem of packing small items into bins so that all but a constant number of bins are kept at least $1 - \varepsilon$ full. Lemma B.3 allows us to do just this for items of size in the range $[\varepsilon', \varepsilon]$, for $\varepsilon' = \Omega(\varepsilon)$. Specifically, considering all integer values $c$ in the range $[[\frac{1}{\varepsilon}], [\frac{1}{\varepsilon}]]$, then, as $1/\varepsilon = O(1)$ and consequently $[\frac{1}{\varepsilon'}] - [\frac{1}{\varepsilon}] = O(1)$, we obtain the following.

Corollary B.3. All items in the range $[\varepsilon', \varepsilon]$ for any $\varepsilon' = \Omega(\varepsilon)$ can be packed into bins which are all (barring perhaps $O(\varepsilon^{-1}) = O(1)$ bins) at least $1 - \varepsilon$ full.

It now remains to address the problem of efficiently maintaining a packing of items of size at most $\varepsilon'$ into bins which are at least $1 - \varepsilon$ full (again, up to some $O(1)$ possible additive term). As $\varepsilon' = \Omega(\varepsilon)$, we will attempt to pack these items into bins which are at least $1 - O(\varepsilon)$ full on average. For notational simplicity from here on, we will abuse notation and denote $\varepsilon'$ by $\varepsilon$, contending ourselves with a packing which is $1 + O(\varepsilon)$-competitive, and is therefore keeps bins $1 - O(\varepsilon)$ full on average (as $OPT$ is close to the volume bound for instance made of only small items).

Let us first give the high-level idea of the algorithm before presenting the formal details. Define the density of an item as $c_j/s_j$, the ratio of its movement cost to its size. We arrange the bins in some fixed order, and the items in each bin will also be arranged in the order of decreasing density. This means the total order on the items (consider items in the order dictated by the ordering of bins, and then by the ordering within each bin) is in decreasing order of density as well. Besides this, we want all bins, except perhaps the last bin, to be approximately full (say, at least $1 - O(\varepsilon)$ full). The latter property will trivially guarantee $(1 + O(\varepsilon))$-competitive ratio. When we insert an item $j$, we place it in the correct bin according to its density. If this bin overflows (i.e., items in it have total size more than 1), then we remove some items from this bin (the ones with least density) and transfer them to the next bin—these items will have only smaller density than $j$, and so, their movement cost will be comparable to that of $j$. If the next bin can accommodate these items, then we can stop the process, otherwise this could lead to a long cascade. To prevent such long cascades, we arrange the bins in buckets—each bucket consists of about $O(1/\varepsilon)$ consecutive bins, and all these buckets are
approximately full except for the last bin in the bucket. Again, it is easy to see that this property will ensure \((1 + O(\varepsilon))\)-competitive ratio. Note that this extends the idea of Berndt et al. Once we have these buckets, the above-mentioned cascade stops when we reach the last bin of a bucket. Consequently we have cascades of length at most \(O(1/\varepsilon)\). If the last bin also overflows, we will add another bin at the end of this bucket, and if the bucket now gets too many bins, we will split it into two smaller buckets. One proceeds similarly for the case of deletes—if an item is deleted, we borrow some items from the next bin in the bucket, and again this could cascade only until the last bin in the bucket. (If the bucket ever has too few bins, we merge it with the next bucket).

However, this cascade is not the only issue. Because items are atomic and have varying sizes, it is possible that insertion of a tiny item (say of size \(O(\varepsilon^2)\)) could lead us to move items of size \(O(\varepsilon)\). In this case, even though the density of the latter item is smaller than the inserted item, its total movement cost could be much higher. To prevent this, we ensure that whenever a bin overflows, we move out enough items from it so that it has \(\Omega(\varepsilon)\) empty space. Now, the above situation will not happen unless we see tiny items amounting to a total size of \(\Omega(\varepsilon)\). In such a case, we can charge the movement cost of the larger item to the movement cost of all such tiny items.

The situation with item deletes is similar. When a tiny item is deleted, it is possible that the corresponding bin underflows, and the item borrowed from the next bin is large (i.e., has size about \(\varepsilon\)). Again, we cannot bound the movement cost of this large item in terms of that of the item being deleted. To take care of such issues, we do not immediately remove such tiny items from the bin. We call such items ghost items—they have been deleted, but we have not removed them from the bins containing them. When a bin accumulates ghost items of total size about \(\Omega(\varepsilon)\), we can afford to remove all these from the bin, and the total movement cost of such items can pay for borrowing items (whose total size would be \(O(\varepsilon)\)) from the next bin.

The analysis of the movement cost is done via a potential function argument, to show the following result (whose proof appears in Section B.2.4):

**Lemma B.4.** For all \(\varepsilon \leq \frac{1}{6}\) there exists an asymptotically \((1 + O(\varepsilon))\)-competitive bin packing algorithm with \(O\left(\frac{1}{\varepsilon^2}\right)\) amortized recourse for instances where all items have size at most \(\varepsilon\).

We first describe the algorithm formally. Let \(B_i\) denote the items stored in a bin \(i\). As mentioned above, our algorithm maintains a solution in which items are stored in decreasing order of density, i.e., for all \(i < i'\), for every pair of jobs \(j \in B_i\) and \(j' \in B_i'\) we have \(c_j/v_j \geq c_{j'}/v_{j'}\). Recall that \(B_i\) could contain ghost jobs. Let \(A_i\) denote the jobs in \(B_i\) which have not been deleted yet (i.e., are not ghost jobs), and \(G_i\) denote the ghost jobs. We shall use \(s(B_i)\) to denote the total size of items in \(B_i\) (define \(s(A_i)\) similarly). We maintain the following invariants, satisfied by all bins \(B_i\) that are not the last bin in their bucket:

\[
P_0 : 1 - 3\varepsilon \leq s(B_i) \leq 1.
\]
\[
P_1 : s(A_i) \geq 1 - 4\varepsilon.
\]  

Finally, a bin \(B_i\) which is the last bin in its bucket has no ghost jobs. That is, \(s(G_i) = 0\). Each bucket has at most \(3/\varepsilon\) bins. Furthermore, each bucket, except perhaps for the last bucket, has at least \(1/\varepsilon\) bins.

Our algorithm is given below. We use two functions \textsc{GrowBucket}(\(U\)) and \textsc{SplitBucket}(\(U\)) in these procedures, where \(U\) is a bucket. The first function is called when the bucket \(U\) underflows, i.e., when \(U\) has less than \(1/\varepsilon\) bins. If \(U\) is the last bucket, then we need not do anything. Otherwise, let \(U'\) be the bucket following \(U\). We merge \(U\) and \(U'\) into one bucket (note that the last bin of \(U\) need not satisfy the invariant conditions above, and so, we will need to do additional processing to
ensure that the conditions are satisfied for this bin). The function \textsc{SplitBucket}(U) is called when \( U \) contains more than \( 3/\varepsilon \) bins. In this case, we split it into two buckets, each of size more than \( 1/\varepsilon \).

**Algorithm** \textsc{insert}(j)

1: Add job \( j \) into appropriate bin \( i \)
2: \textbf{if} \( s(B_i) > 1 \) \textbf{then}
3: \hspace{1em} Call \textsc{EraseGhost}(i, s_j)
4: \textbf{end if}
5: \textbf{if} \( s(B_i) > 1 \) \textbf{then}
6: \hspace{1em} \textsc{overflow}(i, s(B_i) - 1 + 2\varepsilon)
7: \textbf{end if}

**Algorithm** \textsc{delete}(j)

1: Let \( i \) be the bin containing \( j \)
2: \textbf{if} \( i \) is the last bin in its bucket \textbf{then}
3: \hspace{1em} Erase \( j \) from \( i \)
4: \textbf{else}
5: \hspace{1em} Mark \( j \) as a ghost job.
6: \hspace{1em} \textbf{if} \( s(A_i) < 1 - 4\varepsilon \) \textbf{then}
7: \hspace{2em} Erase all ghost jobs from bin \( i \)
8: \hspace{2em} \textsc{borrow}(i, 1 - 3\varepsilon - s(A_i))
9: \hspace{1em} \textbf{end if}
10: \textbf{end if}

**Algorithm** \textsc{overflow}(i, v)

1: Let \( X \) be the minimum density jobs in \( B_i \) s.t. \( s(X) \geq v \)
2: \textbf{if} \( i \) is the last bin its bucket or
3: \hspace{1em} \textbf{if} \( v \geq 1 - 3\varepsilon \) \textbf{then}
4: \hspace{2em} Add a new bin \( i' \) following \( i \) in this bucket
5: \hspace{2em} Move \( X \) from \( i \) to \( i' \).
6: \hspace{1em} Let \( U \) be the bucket containing \( i \).
7: \hspace{1em} \textbf{if} \( U \) has more than 3/\( \varepsilon \) bins \textbf{then}
8: \hspace{2em} Call \textsc{SplitBucket}(U)
9: \hspace{1em} \textbf{end if}
10: \textbf{else}
11: \hspace{1em} Move \( X \) from bin \( i \) to bin \( i + 1 \)
12: \hspace{1em} \textbf{if} \( s(B_{i+1}) > 1 - \varepsilon \) \textbf{then}
13: \hspace{2em} Call \textsc{EraseGhost}(i + 1, s(B_{i+1}) - 1 + 2\varepsilon)
14: \hspace{1em} \textbf{end if}
15: \hspace{1em} \textbf{if} \( s(B_{i+1}) > 1 - \varepsilon \) \textbf{then}
16: \hspace{2em} \textsc{overflow}(i + 1, s(B_{i+1}) - 1 + 2\varepsilon)
17: \hspace{1em} \textbf{end if}
18: \textbf{end if}
19: \textbf{end if}
Next we consider the case of deletion of an item \( j \). Let \( i \) be the bin containing \( j \). If \( i \) is the last bin in

We first describe the function \textsc{insert}(\( j \)) for an item \( j \). We insert job \( j \) into the appropriate bin \( i \) (recall that the items are ordered by their densities). If this bin overflows, then we call the procedure \textsc{EraseGhost}(\( i, s_j \)). The procedure \textsc{EraseGhost}(\( i, s \)), where \( i \) is a bin and \( s \) is a positive quantity, starts erasing ghost jobs from \( B_i \) till one of the following events happen: (i) \( B_i \) has no ghost jobs, or (ii) total size of ghost jobs removed exceeds \( s \). Since all jobs are of size at most \( \varepsilon \), this implies that the total size of ghost jobs removed is at most \( \min(s(G_i), s + \varepsilon) \). Now its possible that even after removing these jobs, the bin overflows (this will happen only if \( s(G_i) \) was at most \( s_j \)). In this case, we offload some of the items (of lowest density) to the next bin in the bucket. Recall that when we do this, we would like to create \( O(\varepsilon) \) empty space in bin \( i \). So we transfer jobs of least density in \( B_i \) of total size at least \( s(B_i) - 1 + 2\varepsilon \) to the next bin (since all jobs are small, the empty space in bin \( i \) will be in the range \([2\varepsilon, 3\varepsilon]\)). This is done by calling the procedure \textsc{overflow}(\( i, s(B_i) - 1 + 2\varepsilon \)). The procedure \textsc{overflow}(\( i, v \)), where \( i \) is a bin and \( v \) is a positive quantity, first builds a set \( X \) of items as follows – consider items in \( B_i \) in increasing order of density, and keep adding them to \( X \) till the total size of \( X \), denoted by \( s(X) \), exceeds \( v \) (so, the total size of \( X \) is at most \( v + \varepsilon \)). We now transfer \( X \) to the next bin in the bucket (note that by construction, \( X \) does not have any ghost jobs). The same process repeats at this bin (although we will say that overflow occurs at this bin if \( s(B_i) \) exceeds \( 1 - \varepsilon \)). This cascade can end in several ways – it is not difficult to show that between two consecutive calls to \textsc{overflow}, the parameter \( v \) grows by at most \( \varepsilon \). If \( v \) becomes larger than \( 1 - 3\varepsilon \), we just create a new bin and assign all of \( X \) to this new bin. If we reach the last bin, we again create a new bin and add \( X \) to it. In both these cases, the size of the bucket increases by 1, and so we may need to split this bucket. Finally, it is possible that even after transfer of \( X \), \( s(A_i) \) does not exceed \( 1 - \varepsilon \) – we can stop the cascade at this point.

Next we consider the case of deletion of an item \( j \). Let \( i \) be the bin containing \( j \). If \( i \) is the last bin in

---

**Algorithm** \textsc{borrow}(\( i, v \))

1. Let \( X \) be the minimum densest jobs in \( B_{i+1} \) s.t. \( s(X \cap A_{i+1}) \geq \min(v, s(A_{i+1})) \)
2. Remove \( X \) from \( B_{i+1} \) (erase ghosts in \( X \))
3. Move \( A \equiv X \cap A_{i+1} \) to \( B_i \)
4. if bin \( i + 1 \) is empty then
5. Remove this bin from its bucket \( U \)
6. if \( U \) has less than \( 1/\varepsilon \) bins and it is not the last bucket then
7. Call \textsc{GrowBucket}(\( U \))
8. if \( U \) has more than \( 3/\varepsilon \) bins then
9. Call \textsc{SplitBucket}(\( U \))
10. end if
11. end if
12. if \( s(A_i) < 1 - 3\varepsilon \) then
13. \textsc{borrow}(\( i, 1 - s(A_i) - 3\varepsilon \))
14. end if
15. else
16. if \( s(A_{i+1}) < 1 - 3\varepsilon \) then
17. Erase all ghost jobs from bin \( i + 1 \)
18. \textsc{borrow}(\( i + 1, 1 - 3\varepsilon - s(A_{i+1}) \))
19. end if
20. end if
21. end if

---
its bucket, then we simply remove \( j \) (recall that the last bin in a bucket cannot contain ghost jobs). Otherwise, we mark \( j \) as a ghost job. This does not change \( s(B_i) \), but could decrease \( s(A_j) \). If it violates property \( P_1 \), we borrow enough items from the next bin such that \( i \) has free space of about \( 2\varepsilon \) only. The function \( \text{BORROW}(i, v) \), where \( i \) is a bin and \( v \) is a positive quantity, borrows densest (non-ghost) items from the next bin \( i + 1 \) of total size at least \( v \). So it orders the (non-ghost) items in \( i + 1 \) in decreasing density, and picks them till the total size accumulated is at least \( v \). This process may cascade (and will stop before we reach the last bin). However there is one subtlety – between two consecutive calls to \( \text{BORROW} \), the value of the parameter \( v \) may grow (by up to \( \varepsilon \)), and so, it is possible that bin \( i + 1 \) becomes empty, and we are not able to transfer enough items from \( i + 1 \) to \( i \). In this case, we first remove \( i + 1 \), and continue the process (if the size of the bucket becomes too small, we handle this case by merging it with the next bin and splitting the resulting bucket if needed). Since we did not move enough items to \( i \), we may need to call \( \text{BORROW}(i, v') \) again with suitable value of \( v' \).

There is a worry that the function \( \text{BORROW}(i, v) \) may call \( \text{BORROW}(i, v') \) with the same bin \( i \), and so, whether this will terminate. But note that, whenever this case happens, we delete one bin, and so, this process will eventually terminate.

### B.2.4 Analysis

We begin by showing properties of the \( \text{OVERFLOW} \) and the \( \text{BORROW} \) functions.

**Lemma B.5.** Whenever \( \text{OVERFLOW}(i, v) \) is called, bin \( i \) has no ghost jobs. Furthermore, \( s(B_i) = v + 1 \) if \( \text{OVERFLOW}(i, v) \) ends before insertion of item \( j \). Finally, when \( \text{OVERFLOW}(i, v) \) ends, \( 1 - 3\varepsilon \leq s(B_i) \leq 1 - 2\varepsilon \), and \( s(G_i) = 0 \).

Similarly, whenever \( \text{BORROW}(i, v) \) is called, bin \( i \) has no ghost jobs. Furthermore, \( s(B_i) = v + 1 \). Finally, when \( \text{BORROW}(i, v) \) ends, either \( i \) is the last bin in its bucket or \( 1 - 3\varepsilon \leq s(B_i) \leq 1 - 2\varepsilon \), and \( s(G_i) = 0 \).

**Proof.** When we insert an item \( j \) in a bin \( i \), we erase ghost items from \( i \) till either (i) we erase all ghost items, or (ii) we erase ghost items of total size at least \( s_j \). If the second case happens, then the fact that \( s(B_i) \leq 1 \) before insertion of item \( j \) implies that we will not call \( \text{OVERFLOW} \) in Line 3 of \( \text{INSERT} \). Therefore, if we do call \( \text{OVERFLOW} \), it must be the case that we ended up deleting all ghost jobs in \( i \). Further we call \( \text{OVERFLOW} \) with \( v = s(B_i) - 1 + 2\varepsilon \). Similarly, before we call \( \text{OVERFLOW}(i + 1, v) \) in Line 17 we try to ensure ghost jobs of the same volume \( v \) from bin \( (i + 1) \). If we indeed manage to remove ghost jobs of total size at least \( v \), we will not make a recursive call to \( \text{OVERFLOW} \). This proves the first part of the lemma.

When \( \text{OVERFLOW}(i, v) \), terminates, we make sure that we have transferred the densest \( v \) volume out of \( i \) to the next bin. Since job sizes are at most \( \varepsilon \), we will transfer at most items of total size in the range \([v, v + \varepsilon]\) out of \( i \). Since \( s(B_i) = v + 1 - 2\varepsilon \) before calling this function, it follows that \( s(B_i) \) after end of this function lies in the range \([1 - 3\varepsilon, 1 - 2\varepsilon]\). The claim for the \( \text{BORROW} \) borrow function follows similarly.

The following corollary follows immediately from the above lemma.

**Corollary B.6.** Properties \( P_0 \) and \( P_1 \) from (4) are satisfied throughout by all bins which are not the last bins in their bucket. All bins \( B_i \) which are last in their bucket satisfy \( s(G_i) = 0 \).

**Proof of Lemma B.4.** First we bound the competitive ratio. Barring the last bucket, which has \( O(1/\varepsilon) \) bins, all other buckets have at least \( 1/\varepsilon \) bins. All bins (except perhaps for the last bin) in each of these buckets have at most \( 4\varepsilon \) space that is empty or is filled by ghost jobs. Therefore, the competitive ratio is \((1 + O(\varepsilon))\) with an additive \( O(1/\varepsilon) \) bins.
It remains to bound the recourse cost of the algorithm.

\[
\Phi(S) = \frac{4}{\varepsilon^2} \sum_{\text{bins } i} \Phi(i), \quad \text{where}
\]

\[
\Phi(i) = c(G_i) + c(\text{fractional sparsest items above } 1 - \varepsilon \text{ in } B_i).
\]

The second term in the definition of \(\Phi(i)\) is evaluated as follows: arrange the items in \(B_i\) in decreasing order of density, and consider the items occupying the last \(\varepsilon\) space in bin \(i\) (the total size of these items could be less than \(\varepsilon\) if the bin is not completely full). Observe that at most one item may be counted fractionally here. The second term is simply the sum of the movement costs of all such item, where a fractional item contributes the appropriate fraction of its movement cost.

Now we bound the amortized movement cost with respect to potential \(\Phi\). First consider the case when we insert item \(j\) in bin \(i\). This could raise \(\Phi_i\) by \(c_j\). If bin \(i\) does not overflow, we do not pay any movement cost. Further, deletion of ghost jobs from bin \(i\) can only decrease the potential. Therefore, the amortized movement cost is bounded by \(c_j\). On the other hand, suppose bin \(i\) overflows and this results in calling the function \text{OVERFLOW}(i, v)\) with a suitable \(v\). Before this function call, let \(I\) denote the set of items which are among the sparsest items above \((1 - \varepsilon)\) volume in bin \(i\) (i.e., these contribute towards \(\Phi_i\)). Let \(d\) be the density of the least density item in \(I\). Since \(s(B_i) \geq 1\), it follows that \(\Phi_i \geq d\varepsilon\). When this procedure ends, Lemma B.5 shows that \(s(B_i) \leq 1 - \varepsilon\), and so, \(\Phi_i\) would be 0. Thus, \(\Phi_i\) decreases by at least \(d\varepsilon - c_j\). Lemma B.5 also shows that if we had recursively called \text{OVERFLOW}(i', v')\) for any other bin \(i'\), then \(s(B_{i'})\) would be at most \(1 - \varepsilon\), and so, \(\Phi_{i'}\) would be 0 when this process ends. It follows that the overall potential function \(\Phi\) decreases by at least \(4(d\varepsilon - c_j)/\varepsilon^2\).

Let us now estimate the total movement cost. We transfer items of total size at most \(1\) from one bin to another. This process will clearly end when we reach the end of the bucket, and so the total size of items moved during this process is at most \(3/\varepsilon\), i.e., the number of bins in this bucket. The density of these items is at most \(d\), and so the total movement cost is at most \(3d/\varepsilon\). Thus, the amortized movement cost is at most \(c_j/\varepsilon^2\).

Now we consider deletion of an item \(j\) which is stored in bin \(i\). Again the interesting case is when this leads to calling the function \text{BORROW}. Before \(j\) was deleted, \(s(B_i)\) was at least \(1 - 3\varepsilon\) (property \(P_0\)). After we mark \(j\) as a ghost job, \(s(A_i)\) drops below \(1 - 4\varepsilon\). So the total size of ghost jobs is at least \(\varepsilon\). Since we are removing all these jobs from bin \(i\), \(\Phi_i\) decreases by at least \(d\varepsilon\), where \(d\) is the density of the least density ghost job in \(i\). As in the case of insert, Lemma B.5 shows that whenever we make a function call \text{BORROW}(i', v')\), \(\Phi_{i'}\) becomes 0 when this process ends. So, the potential \(\Phi\) decreases by at least \(4d/\varepsilon\). Now, count the total movement cost. We transfer items of size at most \(1\) between two bins, and so, we just need to count how many bins are affected during this process (note that if we make several calls to \text{BORROW} with the same bin \(i'\), the the total size of items transferred to \(i'\) is at most \(1\)). Let \(U\) be the bucket containing \(i\). If we do not call \text{SPLIT_BUCKET}(U), then we affect at most \(3/\varepsilon\) bins. If we call \text{SPLIT_BUCKET}(U), then it must be the case that \(U\) had only \(1/\varepsilon\) bins. When we merge \(U\) with the next bucket, and perhaps split this merged bucket, the new bucket \(U\) has at least \(2/\varepsilon\) bins, and so, we will not call \text{SPLIT_BUCKET}(U) again. Thus, we will touch at most \(3/\varepsilon\) buckets in any case. It follows that the total movement cost is at most \(3d/\varepsilon\) (all bins following \(i\) store items of density at most \(d\)). Therefore, the amortized movement cost is negative. This proves the desired result.

**Dealing with Small Items: Summary.** Combining Corollary B.3 and Lemma B.4 we find that we can pack small items into bins which, ignoring some \(O(1)\) many bins are \(1 - \varepsilon\) full on average. Formally, we have the following.

**Lemma B.7.** For all \(\varepsilon \leq \frac{1}{6}\) there exists a fully-dynamic bin packing algorithm with \(O(\frac{1}{\varepsilon^2})\) amortized recourse for instances where all items have size at most \(\varepsilon\) which packs items into bins which, ignoring
some $O(1)$ bins, are at least $1 - \varepsilon$ full on average.

**Worst case bounds.** Note that the above algorithm yields worst case bounds for several natural scenarios, given in the following corollaries.

**Corollary B.8.** For all $\varepsilon \leq \frac{1}{6}$ there exists a fully-dynamic bin packing algorithm with $O\left(\frac{1}{\delta - \varepsilon}\right)$ worst case recourse for instances where all items have size in the range $[\delta, \varepsilon]$ which packs items into bins which, ignoring some $O(1)$ bins, are at least $1 - \varepsilon$ full on average.

*Proof (Sketch).* The algorithm is precisely the algorithm of Lemma B.7 only now in our analysis, as each item has size $\delta$, any single removal can incur movement of at most $\varepsilon^2$ weight, by our algorithm’s definition. The worst case migration factor follows. \qed

Finally, if we group the items of size less than $(1/n, \epsilon)$ into ranges of size $(2^i, 2i + 1]$ (guessing and doubling $n$ as necessary) requires only $O(\log n)$ additive bins (one per size range), while allowing worst case recourse bounds by the previous corollary.

**Corollary B.9.** For all $\varepsilon \leq \frac{1}{6}$ there exists a fully-dynamic bin packing algorithm with $O\left(\frac{1}{\delta - \varepsilon}\right)$ worst case recourse for instances where all items have size at most $\varepsilon$ which packs items into bins which, ignoring some $O(\log n)$ bins, are at least $1 - \varepsilon$ full on average.

### B.2.5 SH Algorithms: Conclusion

Sections B.2.3 and B.2.2 show how to maintain all invariants of SH algorithms, using $O(1)$ amortized recourse, provided $1/\epsilon = O(1)$ and $K = O(1)$, and $\beta_i, \gamma_i = O(1)$ for all $i \in [K]$. As the SH algorithm implied by Lemma B.1 satisfies these conditions, we obtain this section’s main positive result.

**Theorem B.10.** There exists a fully-dynamic bin packing algorithm with a.c.r $1.58889$ and constant additive term using constant amortized recourse under general movement costs.

## C Omitted Proofs of Section 4 (Weight Movement Costs)

### C.1 Amortized Migration Factor Upper Bound

**Fact C.1.** For all $\varepsilon \leq 1/2$, there exists an algorithm requiring $(1 + O(\varepsilon)) \cdot OPT(I_t) + O(\varepsilon^{-2})$ bins at all times $t$ while using only $O(\varepsilon^{-1})$ amortized migration factor.

*Proof. We divide the input into epochs. The first epoch starts at time 0. For an epoch starting at time $t$, let $V_t$ be the total volume of items present at time $t$. The epoch starting at time $t$ ends when the total volume of items inserted or deleted during this epoch exceeds $\varepsilon V_t$. We now explain the bin packing algorithm. Whenever an epoch ends (say at time $t$), we use an offline A(F)PTAS (e.g., [9] or [21]) to efficiently compute a solution using at most $(1 + \varepsilon) \cdot OPT(I_t) + O(\varepsilon^{-2})$ bins. (Recall that $I_t$ denotes the input at time $t$.) We pack items arriving during an epoch in new bins, using the first-fit algorithm to pack them. If an item gets deleted during an epoch, we pretend that it is still in the system and continue to pack it. When an epoch ends, we remove all the items which were deleted during this epoch, and recompute a solution using an off-line A(F)PTAS algorithm as indicated above.

To bound the recourse cost, observe that if the starting volume of items in an epoch starting at time $t$ is $V_t$, the volume at the end of this epoch is at most $V_t + A_t$, where $A_t$ is the volume of items that arrived and departed during this epoch. As $A_t > \varepsilon V_t$, the cost of reassigning these items of volume at most $V_t + A_t$ can be charged to $A_t$. Specifically, the amortized migration cost of the epoch starting at time $t$ is at most $(V_t + A_t)/A_t < V_t/\varepsilon V_t + 1 = O(\varepsilon^{-1})$. Consequently, the overall amortized migration cost is $O(\varepsilon^{-1})$. \qed
To bound the competitive ratio, we use the following easy fact: the optimal number of bins to pack a bin packing instance of total volume $V$ lies between $V$ and $2V$. (E.g., the first-fit algorithm achieves this upper bound.) Consider an epoch starting at time $t$. Let $V_t$ denote the volume of the input $I_t$ at time $t$. The algorithm uses at most $(1 + \varepsilon) \cdot OPT(I_t) + O(\varepsilon^{-2})$ bins at time $t$. Consider an arbitrary time $t'$ during this epoch. As a packing of the instance $I_{t'}$ can be extended to a packing of $I_t$ by packing $I_t \setminus I_{t'}$ with the first fit algorithm, using a further $2\varepsilon V_t$ bins, we have $OPT(I_t) \leq OPT(I_{t'}) + 2\varepsilon V_t$. On the other hand, our algorithm uses at most an additional $2\varepsilon V_t$ bins at time $t'$ compared to the beginning of the epoch. Therefore, the number of bins used by the algorithm at time $t'$ is at most
\[
(1 + \varepsilon) \cdot OPT(I_t) + 2\varepsilon V_t + O(\varepsilon^{-2}) \leq (1 + \varepsilon) \cdot (OPT(I_{t'}) + 2\varepsilon V_t) + 2\varepsilon V_t + O(\varepsilon^{-2}) \\
\leq (1 + \varepsilon) \cdot OPT(I_{t'}) + 5\varepsilon V_t + O(\varepsilon^{-2}).
\]
Now observe that $OPT(I_{t'}) \geq V_t - \varepsilon V_t = (1 - \varepsilon) \cdot V_t$ because the total volume of jobs at time $t'$ is at least $V_t - \varepsilon V_t$, and so $OPT(I_{t'}) \geq V_t/2$. Therefore, $5\varepsilon V_t \leq 10\varepsilon OPT(I_{t'})$. It follows that the number of bins used by the algorithm at time $t'$ is at most $(1 + O(\varepsilon)) \cdot OPT(I_{t'}) + O(\varepsilon^{-2}).$

\section*{References}

\begin{thebibliography}{10}

\bibitem{albers2012value} \textsc{Albers, S. and Hellwig, M.} 2012. On the value of job migration in online makespan minimization. In \textit{European Symposium on Algorithms}. Springer, 84–95.

\bibitem{balogh2017new} \textsc{Balogh, J., Békési, J., Dósa, G., Epstein, L., and Levin, A.} 2017. A new and improved algorithm for online bin packing. \textit{arXiv preprint arXiv:1707.01728}.

\bibitem{balogh2012new} \textsc{Balogh, J., Békési, J., and Galambos, G.} 2012. New lower bounds for certain classes of bin packing algorithms. \textit{Theoretical Computer Science} \textbf{440}, 1–13.

\bibitem{balogh2008lower} \textsc{Balogh, J., Békési, J., Galambos, G., and Reinelt, G.} 2008. Lower bound for the online bin packing problem with restricted repacking. \textit{SIAM Journal on Computing} \textbf{38}, 1, 398–410.

\bibitem{balogh2014online} \textsc{Balogh, J., Békési, J., Galambos, G., and Reinelt, G.} 2014. On-line bin packing with restricted repacking. \textit{Journal of Combinatorial Optimization} \textbf{27}, 1, 115–131.

\bibitem{berndt2015fully} \textsc{Berndt, S., Jansen, K., and Klein, K.} 2015. Fully dynamic bin packing revisited. In \textit{Approximation, Randomization, and Combinatorial Optimization. Algorithms and Techniques, APPROX/RANDOM 2015, August 24-26, 2015, Princeton, NJ, USA}. 135–151.

\bibitem{coffman2013bin} \textsc{Coffman Jr, E. G., Csirik, J., Galambos, G., Martello, S., and Vigo, D.} 2013. Bin packing approximation algorithms: survey and classification. In \textit{Handbook of Combinatorial Optimization}. Springer, 455–531.

\bibitem{corless1996lambert} \textsc{Corless, R. M., Gonnet, G. H., Hare, D. E., Jeffrey, D. J., and Knuth, D. E.} 1996. On the Lambert-$W$ function. \textit{Advances in Computational mathematics} \textbf{5}, 1, 329–359.

\bibitem{de2001bin} \textsc{De La Vega, W. F. and Lueker, G. S.} 1981. Bin packing can be solved within $1 + \varepsilon$ in linear time. \textit{Combinatorica} \textbf{1}, 4, 349–355.

\bibitem{epstein2009robust} \textsc{Epstein, L. and Levin, A.} 2009. A robust APTAS for the classical bin packing problem. \textit{Math. Program.} \textbf{119}, 1, 33–49.

\bibitem{epstein2014robust} \textsc{Epstein, L. and Levin, A.} 2014. Robust algorithms for preemptive scheduling. \textit{Algorithmica} \textbf{69}, 1, 26–57.

\end{thebibliography}

38
[12] Feldkord, B., Feldotto, M., and Sören, R. 2017. A tight approximation for fully dynamic bin packing without bundling. arXiv preprint arXiv:1711.01231.

[13] Gambosi, G., Postiglione, A., and Talamo, M. 1990. New algorithms for on-line bin packing. In Algorithms and Complexity, Proceedings of the First Italian Conference. 44–59.

[14] Gambosi, G., Postiglione, A., and Talamo, M. 2000. Algorithms for the relaxed online bin-packing model. SIAM journal on computing 30, 5, 1532–1551.

[15] Heydrich, S. and van Stee, R. 2016. Beating the harmonic lower bound for online bin packing. In 43rd International Colloquium on Automata, Languages, and Programming. Schloss Dagstuhl.

[16] Hoberg, R. and Rothvoss, T. 2017. A logarithmic additive integrality gap for bin packing. In Proceedings of the Twenty-Eighth Annual ACM-SIAM Symposium on Discrete Algorithms. SIAM, 2616–2625.

[17] Hochbaum, D. S. 1996. Approximation algorithms for NP-hard problems. PWS Publishing Co.

[18] Ivković, Z. and Lloyd, E. L. 1996. A fundamental restriction on fully dynamic maintenance of bin packing. Information Processing Letters 59, 4, 229–232.

[19] Jansen, K. and Klein, K. 2013. A robust AFPTAS for online bin packing with polynomial migration. In Automata, Languages, and Programming - 40th International Colloquium, ICALP 2013, Riga, Latvia, July 8-12, 2013, Proceedings, Part I. 589–600.

[20] Johnson, D. S., Demers, A., Ullman, J. D., Garey, M. R., and Graham, R. L. 1974. Worst-case performance bounds for simple one-dimensional packing algorithms. SIAM Journal on Computing 3, 4, 299–325.

[21] Karmarkar, N. and Karp, R. M. 1982. An efficient approximation scheme for the one-dimensional bin-packing problem. In Proceedings of the 23rd Annual Symposium on Foundations of Computer Science. IEEE Computer Society, 312–320.

[22] Lee, C. C. and Lee, D.-T. 1985. A simple on-line bin-packing algorithm. Journal of the ACM (JACM) 32, 3, 562–572.

[23] Ramanan, P., Brown, D. J., Lee, C.-C., and Lee, D.-T. 1989. On-line bin packing in linear time. Journal of Algorithms 10, 3, 305–326.

[24] Richey, M. B. 1991. Improved bounds for harmonic-based bin packing algorithms. Discrete Applied Mathematics 34, 1-3, 203–227.

[25] Sanders, P., Sivadasan, N., and Skutella, M. 2009. Online scheduling with bounded migration. Mathematics of Operations Research 34, 2, 481–498.

[26] Seiden, S. S. 2002. On the online bin packing problem. Journal of the ACM (JACM) 49, 5, 640–671.

[27] Skutella, M. and Verschae, J. 2010. A robust ptas for machine covering and packing. In European Symposium on Algorithms. Springer, 36–47.

[28] Sylvester, J. J. 1880. On a point in the theory of vulgar fractions. American Journal of Mathematics 3, 4, 332–335.

[29] Ullman, J. 1971. The performance of a memory allocation algorithm.
[30] VAN VLIET, A. 1995. Lower and Upper Bounds for On-line Bin Packing and Scheduling Heuristics: Onder-en Bovengrenzen Voor On-line Bin Packing en Scheduling Heuristieken. Thesis Publ.

[31] WOEINGER, G. 1993. Improved space for bounded-space, on-line bin-packing. SIAM Journal on Discrete Mathematics 6, 4, 575–581.

[32] YAO, A. C.-C. 1980. New algorithms for bin packing. Journal of the ACM (JACM) 27, 2, 207–227.