

# A New Deterministic Algorithm for Dynamic Set Cover

Sayan Bhattacharya\*

Monika Henzinger†

Danupon Nanongkai‡

## Abstract

We present a deterministic dynamic algorithm for maintaining a  $(1 + \epsilon)f$ -approximate minimum cost set cover with  $O(f \log(Cn)/\epsilon^2)$  amortized update time, when the input set system is undergoing element insertions and deletions. Here,  $n$  denotes the number of elements, each element appears in at most  $f$  sets, and the cost of each set lies in the range  $[1/C, 1]$ . Our result, together with that of Gupta et al. [STOC'17], implies that there is a deterministic algorithm for this problem with  $O(f \log(Cn))$  amortized update time and  $O(\min(\log n, f))$ -approximation ratio, which nearly matches the polynomial-time hardness of approximation for minimum set cover in the static setting. Our update time is only  $O(\log(Cn))$  away from a trivial lower bound.

Prior to our work, the previous best approximation ratio guaranteed by deterministic algorithms was  $O(f^2)$ , which was due to Bhattacharya et al. [ICALP'15]. In contrast, the only result that guaranteed  $O(f)$ -approximation was obtained very recently by Abboud et al. [STOC'19], who designed a dynamic algorithm with  $(1 + \epsilon)f$ -approximation ratio and  $O(f^2 \log n/\epsilon)$  amortized update time. Besides the extra  $O(f)$  factor in the update time compared to our and Gupta et al.'s results, the Abboud et al. algorithm is *randomized*, and works only when the adversary is *oblivious* and the sets are unweighted (each set has the same cost).

We achieve our result via the primal-dual approach, by maintaining a fractional packing solution as a dual certificate. This approach was pursued previously by Bhattacharya et al. and Gupta et al., but not in the recent paper by Abboud et al. Unlike previous primal-dual algorithms that try to satisfy some *local* constraints for individual sets at all time, our algorithm basically waits until the dual solution changes significantly *globally*, and fixes the solution only where the fix is needed.

---

\*University of Warwick, UK. Email: S.Bhattacharya@warwick.ac.uk

†University of Vienna, Austria. Email: monika.henzinger@univie.ac.at

‡KTH Royal Institute of Technology, Sweden. Email: danupon@kth.se

# Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
1.1	Technical Overview . . . . .	2
<b>2</b>	<b>Minimum Set Cover in the Static Setting</b>	<b>6</b>
<b>3</b>	<b>Our Dynamic Algorithm</b>	<b>7</b>
3.1	Classification of elements . . . . .	7
3.2	Levels and Weights of elements . . . . .	8
3.3	The shadow input and the invariants . . . . .	8
3.4	The dynamic algorithm . . . . .	9
3.5	The REBUILD( $\leq k$ ) subroutine . . . . .	10
<b>4</b>	<b>Analysis of our dynamic algorithm</b>	<b>11</b>
4.1	Bounding the update time of our dynamic algorithm . . . . .	13
4.2	Bounding the approximation ratio . . . . .	14
<b>5</b>	<b>Describing the REBUILD(<math>\leq k</math>) subroutine</b>	<b>16</b>
5.1	Justifying Properties 3.4, 3.5 and 3.6 . . . . .	19
5.2	The subroutine FIX-LEVEL( $k, \mathcal{S}', \mathcal{E}'$ ) . . . . .	19
<b>A</b>	<b>A Static Primal-Dual Algorithm for Minimum Set Cover</b>	<b>24</b>

# 1 Introduction

In the (static) set cover problem, an algorithm is given a collection  $\mathcal{S}$  of  $m$  sets over a universe  $\mathcal{E}$  of  $n$  elements such that  $\cup_{s \in \mathcal{S}} s = \mathcal{E}$ . Each set  $s \in \mathcal{S}$  has a positive *cost*  $c_s$ . After scaling these costs by some appropriate factor, we can always get a parameter  $C > 1$  such that:

$$1/C \leq c_s \leq 1 \text{ for every set } s \in \mathcal{S}. \quad (1.1)$$

For any  $\mathcal{S}' \subseteq \mathcal{S}$ , let  $c(\mathcal{S}') = \sum_{s \in \mathcal{S}'} c_s$  denote the total cost of all the sets in  $s \in \mathcal{S}'$ . We say that a set  $s \in \mathcal{S}$  *covers* an element  $e \in \mathcal{E}$  iff  $e \in s$ . Our goal is to pick a collection of sets  $\mathcal{I} \subseteq \mathcal{S}$  with minimum total cost  $c(\mathcal{I})$  so as to cover all the elements in the universe  $\mathcal{E}$ .

Set cover is a fundamental optimization problem that has been extensively studied in the contexts of polynomial-time approximation algorithms and online algorithms. In recent years, it has received significant attention in the *dynamic algorithms* community as well, where the goal is to maintain a set cover  $\mathcal{I} \subseteq \mathcal{S}$  of small cost efficiently under a sequence of element insertions/deletions in  $\mathcal{E}$ . In particular, a dynamic algorithm for set cover must support the following *update operations*.

PREPROCESS( $m$ ): Create  $m$  empty sets in  $\mathcal{S}$ . Return  $\mathcal{I} = \emptyset$ , and identifiers (e.g., integers) to the sets in  $\mathcal{S}$ .

INSERT( $\mathcal{F} = \{s_1, s_2, \dots\}$ ): Insert to  $\mathcal{E}$  a new element  $e$  which belongs to the sets  $s_1, s_2, \dots$  (their identifiers are given as parameters). Return an identifier to the new element  $e$ , and the identifiers of sets that get added to and removed from  $\mathcal{I}$ .

DELETE( $e$ ): Delete element  $e$  from  $\mathcal{E}$ . Return the identifiers of sets that get added to and removed from  $\mathcal{I}$ .

After each update, the algorithm must guarantee that  $\mathcal{I}$  is a set cover; i.e. every element  $e \in \mathcal{E}$  is in some set in  $\mathcal{I}$ . Let  $f$  and  $n$  be the maximum size of  $\mathcal{F}$  and  $\mathcal{E}$ , respectively, over all updates. The parameter  $f$  is known as the *maximum frequency*. It is usually assumed that  $f, n$  and  $m$  are known and fixed in the beginning, but note that our algorithm does not really need this assumption.<sup>1</sup> Note that dynamic set cover as defined above is a generalization of the dynamic vertex cover problem which, together with the dynamic maximum matching problem, have been studied extensively in recent years (e.g. [31, 3, 20, 13, 9, 10, 12, 35, 30, 32, 18, 6, 7, 8, 33, 36, 2, 25]).

The performance of dynamic algorithms is mainly measured by the *update time*, the time to handle each INSERT and DELETE operation. Previous works on set cover focus on the *amortized update time*, where an algorithm is said to have an amortized update time of  $t$  if, for any  $k$ , the total time it spends to process the first  $k$  updates is at most  $kt$ . We also consider only the amortized update time in this paper, and simply use “update time” to refer to “amortized update time”. The time for the PREPROCESS operation is called the *preprocessing time*. It is typically not a big concern as long as it is polynomial.<sup>2</sup>

**Perspective:** Since the static set cover problem is NP-complete, it is natural to consider approximation algorithms. An algorithm has an approximation ratio of  $\alpha$  if outputs a set cover  $\mathcal{I}$  with  $c(\mathcal{I}) \leq \alpha \cdot OPT$ , where  $OPT$  is cost of the optimal set cover. Since the tight approximation factors for polynomial-time static set cover algorithms are  $\Theta(\log n)$  and  $\Theta(f)$  (e.g. [17, 16, 15, 34, 26]), it is natural to ask if one can also obtain these same guarantees in the dynamic setting with small update time. The  $O(\log n)$  approximation ratio was already achieved in 2017 via greedy-like techniques by Gupta et al. [19]. Their algorithm is deterministic and has  $O(f \log n)$  update time. This update time is only  $O(\log n)$  away from the trivial  $\Omega(f)$  lower bound – the time needed for specifying the sets that contain a given element (which is currently being inserted). A similar lower bound holds even in some settings where updates can be specified with less than  $f$  bits [1], e.g., when elements and sets are fixed in advance, and the updates are activations and deactivations

---

<sup>1</sup>We mention that our algorithm do not really need the preprocessing step. When INSERT( $\mathcal{F} = \{s_1, s_2, \dots\}$ ) is called with a new set  $s_i$ , it can simply create  $s_i$  on the fly.

<sup>2</sup>Our algorithm requires only linear preprocessing time.

Reference	Approximation Ratio	Update Time	Deterministic?	Weighted?
[19]	$O(\log n)$	$O(f \log n)$	yes	yes
[19, 9]	$O(f^3)$	$O(f^2)$	yes	yes
[11]	$O(f^2)$	$O(f \log(m + n))$	yes	yes
[1]	$(1 + \epsilon)f$	$O(f^2 \log n / \epsilon)$	no	no
<b>Our result</b>	$(1 + \epsilon)f$	$O(f \log(Cn) / \epsilon^2)$	<b>yes</b>	<b>yes</b>

Table 1: Summary of results on dynamic set cover

of elements.<sup>3</sup> The  $O(f)$  approximation ratio was recently achieved by Abboud et al. [1] (improving upon the approximation factors of  $O(f^2)$  and higher by [11, 19, 9]). Abboud et al. show how to maintain an  $(1 + \epsilon)f$ -approximation in  $O(f^2 \log n / \epsilon)$  amortized update time. Their algorithm, however, is randomized and does not work for the weighted case (when different sets have different costs).<sup>4</sup> Like most randomized dynamic algorithms currently existing in the literature, it works only when the future updates do not depend on the algorithm’s past output – this is also known as the *oblivious adversary assumption*. Removing this assumption is a central question in this area, since it may in general make many dynamic algorithms useful as subroutines inside fast static algorithms [14, 29, 5, 4, 27, 28, 12, 23]. Accordingly, prior to our work, it was natural to ask if there is an efficient  $O(f)$ -approximation algorithm for dynamic set cover that is *deterministic* and/or can handle the *weighted case*. In this paper we answer this question positively.

**Theorem 1.1.** *We can maintain a  $(1 + \epsilon)f$ -approximate minimum-cost set cover in the dynamic setting, deterministically, with  $O(f \log(Cn) / \epsilon^2)$  amortized update time, where  $C$  is the ratio between the maximum and minimum set costs.*

Thus, we simultaneously (a) improve upon the update time of Abboud et al. [1], (b) derandomize their result, and (c) extend their result to the weighted case. Our algorithm, together with the one in Gupta et al. [19], settles an important open question in a line of work on dynamic set cover [9, 11, 19, 1]. We can now get an  $O(\min(\log n, f))$ -approximation using a deterministic algorithm with  $O(f \log(Cn))$  update time. The approximation ratio matches the one achievable by the best possible polynomial-time static algorithm, whereas the update time is only  $O(\log(Cn))$  away from a trivial lower bound of  $\Omega(f)$ .

## 1.1 Technical Overview

**Previous Approaches:** The *primal-dual schema* is a powerful tool for designing many static approximation algorithms. In recent years, it has also been the main driving force behind *deterministic* dynamic algorithms for set cover and maximum matching (e.g. [10, 12, 9, 11, 19]), including all  $O(\text{poly}(f))$ -approximations for dynamic set cover *except* the one by Abboud et al. [1].

The *dual* of minimum set cover happens to be a *fractional packing* problem, which is defined as follows. Given a set system  $(\mathcal{S}, \mathcal{E})$  as input, we have to assign a fractional weight  $w(e) \geq 0$  to every element. We want to maximize  $\sum_{e \in \mathcal{E}} w(e)$ , subject to the following constraint:

$$\sum_{e \in s} w(e) \leq c_s \text{ for every set } s \in \mathcal{S}. \quad (1.2)$$

<sup>3</sup>Abboud et al. [1] showed that, under SETH, there is no algorithm with polynomial preprocessing time and  $f^{1-\delta}$  update time for any constant  $\delta > 0$  when elements and sets are fixed in advance, and the updates are activations and deactivations of elements.

<sup>4</sup>A fundamental difficulty to extend Abboud et al.’s algorithm to the weighted case is the static algorithm it is based on. This static algorithm repeatedly picks an element  $e$  that is not yet covered, and adds all sets  $e$  belong to in the set cover solution. It is easy to prove that this algorithm returns an  $f$ -approximation in the unweighted case. It is also easy to construct an example that shows that this algorithm cannot guarantee any reasonable approximation ratio for the weighted case.

For the rest of this paper, we let  $W(s) = \sum_{e \in s} w(e)$  denote the total weight received by a set  $s \in \mathcal{S}$  from all its elements. Furthermore, define  $w(S) = \sum_{e \in S} w(e)$  for every subset of elements  $S \subseteq \mathcal{E}$ . Thus, the goal is to maximize  $w(\mathcal{E})$  subject to the constraint that  $W(s) \leq c_s$  for all sets  $s \in \mathcal{S}$ .

Let  $OPT(\mathcal{S}, \mathcal{E})$  denote the total cost of the minimum set cover in  $(\mathcal{S}, \mathcal{E})$ . We simply use  $OPT$  when  $(\mathcal{S}, \mathcal{E})$  is clear from the context. All the previous primal-dual algorithms for dynamic set cover try to maintain some invariants about *individual* sets all the time. In particular, they are based on the following lemma.

**Lemma 1.2.** *Consider any fractional packing  $w$  in  $(\mathcal{S}, \mathcal{E})$  that satisfy (1.2). Then we have  $w(\mathcal{E}) \leq OPT(\mathcal{S}, \mathcal{E})$ . In addition, if there exist a set-cover  $\mathcal{I} \subseteq \mathcal{S}$  of  $\mathcal{E}$  and an  $\alpha \geq 1$  such that*

$$W(s) \geq c_s/\alpha \text{ for all sets } s \in \mathcal{I}, \quad (1.3)$$

*then we have  $c(\mathcal{I}) \leq \alpha f \cdot w(\mathcal{E}) \leq \alpha f \cdot OPT(\mathcal{S}, \mathcal{E})$ .*

All the previous dynamic primal-dual algorithms maintain a set cover  $\mathcal{I}$  and a fractional packing  $w$  that satisfy (1.3). Needless to say, the algorithms have to change  $\mathcal{I}$  and the weights  $w(e)$  of some elements  $e$  in a carefully chosen manner after each update. For example, the algorithm of Bhattacharya et al. [11] satisfies (1.3) with  $\alpha = O(f)$ , implying an  $O(f^2)$ -approximation factor. To obtain an  $O(f)$  approximation factor we need to satisfy (1.3) with  $\alpha = O(1)$ . However, as pointed out by Abboud et al. [1], it is not clear how to maintain such a strict constraint efficiently for  $\alpha = O(1)$ . The trouble is that one update may violate (1.3) and may cause a sequence of weight changes for many elements. This creates difficulties for bounding the update time (which typically require intricate arguments via clever potential functions).

Because of this difficulty, Abboud et al. opted for a different approach that is based on the following static algorithm. (i) Pick any uncovered element  $e$  uniformly at random, called *pivot*. (ii) Include all sets containing  $e$  in the set cover solution. (iii) Repeat this process until all elements are covered. It is easy to see that this algorithm returns an  $f$ -approximation for the unweighted case (when every set has the same cost). Since this approximation ratio does not hold for the weighted case in the static setting, it seems difficult to extend the approach of Abboud et al. to the weighted case. More importantly, in the analysis of their algorithm in the dynamic setting, Abboud et al. crucially rely on randomness and the oblivious adversary assumption. This allows them to argue that before a pivot element gets deleted, many other non-pivot elements must also get deleted in expectation. Thus they can charge the time their algorithm needs in handling the deletion of a pivot to the (large number of) non-pivot elements that got deleted in the past. This type of argument was also used for maintaining a maximal matching [3, 35]. To the best of our knowledge, there was no technique to derandomize this type of argument. In fact, all the known deterministic dynamic algorithms for set cover and matching use the primal-dual schema. This leads to a basic question: *Is the primal-dual approach powerful enough to give an  $O(f)$ -approximation algorithm for dynamic set cover?*

We answer this question in the affirmative. Unlike previous dynamic primal-dual algorithms, which try to satisfy conditions like (1.3) that are *local* to individual sets, our algorithm basically waits until the dual solution changes significantly *globally*, and then it fixes the solution only where the fix is needed.

**Our Approach and the Showcase (Batch Deletion):** To appreciate our main idea, consider the *batch deletion* setting, where we have to preprocess a set system  $(\mathcal{S}, \mathcal{E})$  and then there is an update  $D \subseteq \mathcal{E}$  that changes the set system to  $(\mathcal{S}, \mathcal{E}')$ , where  $\mathcal{E}' = \mathcal{E} \setminus D$ . Our goal is to recompute an approximately minimum set cover in the new input  $(\mathcal{S}, \mathcal{E}')$  in time proportional to the size of the update (i.e.,  $|D|$ ).

Suppose that originally we have a pair  $(\mathcal{I}, w)$  that satisfies (1.3) with  $\alpha = 1$  for  $(\mathcal{S}, \mathcal{E})$ ; thus,  $c(\mathcal{I}) \leq f \cdot w(\mathcal{E}) \leq f \cdot OPT(\mathcal{S}, \mathcal{E})$ . Clearly,  $\mathcal{I}$  remains a set cover of the new set system  $(\mathcal{S}, \mathcal{E}')$ . To simplify things even further, suppose that the element-weights are *uniform*,<sup>5</sup> and  $|D| \leq \epsilon \cdot |\mathcal{E}'|$  for some  $\epsilon > 0$ . This implies

<sup>5</sup>This means that  $w(e) = \delta$  for every  $e \in \mathcal{E}$  (for some  $\delta \geq 0$ ).

that  $w(D) \leq \epsilon \cdot w(\mathcal{E}')$ . So Lemma 1.2 gives us:

$$\begin{aligned} c(\mathcal{I}) \leq f \cdot w(\mathcal{E}) &= f \cdot (w(\mathcal{E}') + w(D)) \\ &\leq f(1 + \epsilon) \cdot w(\mathcal{E}') \\ &\leq f(1 + \epsilon) \cdot OPT(\mathcal{S}, \mathcal{E}'). \end{aligned}$$

**Observation 1.3.** *If  $|D| \leq \epsilon \cdot |\mathcal{E}'|$  and the element-weights are uniform, then  $c(\mathcal{I}) \leq f(1 + \epsilon) \cdot OPT(\mathcal{S}, \mathcal{E}')$ .*

In words,  $\mathcal{I}$  remains a good approximation to  $OPT(\mathcal{S}, \mathcal{E}')$  if  $|D|$  is small. Thus, intuitively we do not need to do anything when  $|D| \leq \epsilon \cdot |\mathcal{E}'|$ . This is already different from previous primal-dual algorithms that might have to do a lot of work, since (1.3) might be violated. In contrast, when  $|D|$  becomes larger than  $\epsilon \cdot |\mathcal{E}'|$ , in  $O(f \cdot |\mathcal{E}'|)$  time we can just compute a new pair  $(\mathcal{I}', w')$  satisfying (1.3) with  $\alpha = 1$  for the set system  $(\mathcal{S}, \mathcal{E}')$ . Since we do this only after  $\epsilon \cdot |\mathcal{E}'|$  deletions, we get an amortized update time of  $O(f/\epsilon)$ .

One lesson from the above argument is this: Instead of trying to satisfy (1.3), we might benefit from dealing with  $D$  only when it is large enough, for this might help us ensure that the amortized update time remains small. Of course this is easy to argue under the uniform weight assumption, which often makes the situation too simple. The following lemma is the key towards doing something similar in the general setting.

**Lemma 1.4.** *Define  $s_{>x} = \{e \in s : w(e) > x\}$  for every set  $s \subseteq \mathcal{E}$  of elements. Suppose that:*

$$|D_{>x}| \leq \epsilon \cdot |\mathcal{E}'_{>x}| \text{ for all } x \geq 0. \quad (1.4)$$

*Then we have  $w(D) \leq \epsilon \cdot w(\mathcal{E}')$ , and thus  $c(\mathcal{I}) \leq f(1 + \epsilon) \cdot OPT(\mathcal{S}, \mathcal{E}')$ .*

*Proof sketch.* Note that  $w(e) = \int_0^\infty \mathbb{1}(x < w(e)) dx$ , where  $\mathbb{1}(x < w(e))$  is one if  $x < w(e)$ , and zero otherwise. Thus, we have:

$$\begin{aligned} \sum_{e \in D} w(e) &= \int_0^\infty \sum_{e \in D} \mathbb{1}(x < w(e)) dx \\ &= \int_0^\infty |\{e \in D : w(e) > x\}| dx \\ &= \int_0^\infty |D_{>x}| dx \\ &\leq \epsilon \cdot \int_0^\infty |\mathcal{E}'_{>x}| dx \\ &= \epsilon \cdot \int_0^\infty |\{e \in \mathcal{E}' : w(e) > x\}| dx \\ &= \epsilon \cdot \sum_{e \in \mathcal{E}'} w(e). \end{aligned}$$

□

Lemma 1.4 tells us that if  $|D_{>x}| \leq \epsilon \cdot |\mathcal{E}'_{>x}|$  for all  $x$ , then we do not have to do anything. When  $|D_{>x}| > \epsilon \cdot |\mathcal{E}'_{>x}|$  for some  $x$ , we need to “fix”  $\mathcal{I}$ . We do so by running a static algorithm on some sets and elements. The static algorithm is described in Algorithm 1.5. We describe how to use it in Algorithm 1.6.

**Algorithm 1.5** (Static uniform-increment). *Given an input  $(\hat{\mathcal{S}}, \hat{\mathcal{E}})$ , start with  $\hat{w}(e) \leftarrow 0$  for all  $e \in \hat{\mathcal{E}}$  and  $\hat{\mathcal{I}} \leftarrow \emptyset$ . Repeat the following until  $\hat{\mathcal{I}}$  covers all elements: (i) Raise weights  $\hat{w}(e)$  at the same rate for every  $e \in \hat{\mathcal{E}}$  not covered by  $\hat{\mathcal{I}}$ , until some sets  $s$  in  $\hat{\mathcal{S}} \setminus \hat{\mathcal{I}}$  are tight (i.e.  $W(s) = c_s$ ). (ii) Add such sets to  $\hat{\mathcal{I}}$ .*

**Algorithm 1.6** (Batch deletion algorithm). *Initially, compute  $(\mathcal{I}, w)$  by running Algorithm 1.5 on  $(\mathcal{S}, \mathcal{E})$ . To handle  $D$ , let  $x^*$  be the minimum  $x$  such that (1.4) is violated and do the following. (Do nothing if (1.4) is not violated at all.) Run Algorithm 1.5 to compute  $(\hat{\mathcal{I}}, \hat{w})$  for  $(\mathcal{S}_{>x^*}, \mathcal{E}'_{>x^*})$ , where  $\mathcal{S}_{>x^*} = \{s \in \mathcal{S} : s \cap \mathcal{E}_{>x^*} \neq \emptyset\}$ . Set  $\mathcal{I} \leftarrow (\mathcal{I} \setminus \mathcal{S}_{>x^*}) \cup \hat{\mathcal{I}}$ ,  $D = D \setminus D_{>x^*}$ , and  $w(e) = \hat{w}(e)$  for all  $e \in \mathcal{E}_{>x^*}$ .*

In short, Algorithm 1.6 fixes  $\mathcal{I}$  by running the static Algorithm 1.5 on input  $(\mathcal{S}_{>x^*}, \mathcal{E}'_{>x^*})$ . We can implement Algorithm 1.5 in  $O(|\mathcal{S}_{>x^*}| + |\mathcal{E}'_{>x^*}|) = O(f|\mathcal{E}_{>x^*}|) = O(f|D_{>x^*}|/\epsilon)$  time approximately.<sup>6</sup> This gives an amortized update time of  $O(f/\epsilon)$ , since we can charge the  $O(f|D_{>x^*}|/\epsilon)$  time spent in “fixing”  $\mathcal{I}$  to the elements in  $D_{>x^*}$  that got deleted. The lemma below implies the correctness of this strategy.

**Lemma 1.7.** (i) *Let  $(\mathcal{I}, w)$  denote the output of Algorithm 1.5 when given  $(\mathcal{S}, \mathcal{E})$  as input. For every  $x \geq 0$ , the subset of elements  $\mathcal{E} \setminus \mathcal{E}_{>x}$  is covered by  $\mathcal{I} \setminus \mathcal{S}_{>x}$ . (ii) After processing  $D$ , Algorithm 1.6 produces  $D, \mathcal{I}$ , and  $w$  that satisfy (1.4); which implies that  $c(\mathcal{I}) \leq (1 + \epsilon)f \cdot \text{OPT}(\mathcal{E}')$ .*

*Proof Idea.* For (i), Algorithm (1.5) stops raising the weight of some element  $e \in \mathcal{E} \setminus \mathcal{E}_{>x}$  only when some set  $s$  containing  $e$  is tight. At this point we also stop raising the weight of every element in  $s$ , making  $s \in \mathcal{I} \setminus \mathcal{S}_{>x}$ . For (ii), an intuition is that Algorithm 1.6 has subtracted from  $D$  all  $D_{>x}$  that violate (1.4). This subtraction does not increase  $D_{>x}$  for any  $x$ , and, thus, does not create any new violation to (1.4).  $\square$

Lemma 1.7(i) implies that we can add/remove to/from  $\mathcal{I}$  sets in  $\mathcal{S}_{>x^*}$  without worrying about the coverage of elements in  $\mathcal{E} \setminus \mathcal{E}_{>x^*}$  (they will be covered even when we remove all sets in  $\mathcal{S}_{>x^*}$  from  $\mathcal{I}$ ). Consequently, we can guarantee that after processing  $D$ , Algorithm 1.6 produces a set  $\mathcal{I}$  that covers all elements. Lemma 1.7(ii) immediately implies the claimed approximation guarantee. Note that it is crucial to apply our new Lemma 1.4 with appropriate  $D, \mathcal{I}$ , and  $w$ , which are changed after Algorithm 1.6 processes  $D$ .

Note that running Algorithm 1.5 at the preprocessing is crucial for the correctness of Algorithm 1.6, as otherwise  $\mathcal{I}$  might not cover all elements after Algorithm 1.6 finishes. In other words, Lemma 1.7(i) might not be true if we replace Algorithm 1.5 by some other static algorithm.<sup>7</sup> We believe that this is key that gives the running time improvement over Abboud et al.’s algorithm, since otherwise we may have to spend more time checking if other elements remain covered. (This is essentially what happened in Abboud et al.’s algorithm.) Note that Algorithm 1.5 was also used in previous dynamic algorithms [9, 11, 19], but to our knowledge it does not play a role in the correctness of those algorithms like in our algorithm.

One detail to mention is how Algorithm 1.6 finds  $x^*$ . One simple way is to round element weights to the form  $(1 + \epsilon)^i$  for integers  $i$ . (We refer to such  $i$  as the *level* of an element in the rest of this paper.) With this rounding, we simply have to search for  $O(\log(Cn))$  different choices of  $x^*$ , where  $C$  is defined in Theorem 1.1. The total update time amortized over deletions in  $D$  becomes  $O(\log(Cn) + f/\epsilon)$ .

**The Fully-Dynamic Algorithm (Sketched).** Extending the above algorithm to handle more deletions is rather straightforward: We include a newly deleted element to  $D$ , check for  $x^*$ , and then fix  $(\mathcal{S}_{>x^*}, \mathcal{E}'_{>x^*})$  as in Algorithm 1.6 if such a  $x^*$  exists. This gives a decremental (i.e. deletion-only) algorithm with  $O(\log(Cn) + f/\epsilon)$  update time. This bound is faster than the  $O(f^2/\epsilon^5)$  bound from [1] when  $f = \omega(\sqrt{\log n})$  and might be of an independent interest given that decremental algorithms have been heavily studied and lead to some applications (e.g. [14, 4, 5, 24, 21, 22]).

Handling insertions, on the other hand, is more intricate. When an element  $e$  is inserted, we set its weight  $w(e)$  to the maximum possible value to make some set  $s$  containing  $e$  tight, i.e.  $\sum_{e \in s \cap (\mathcal{E} \cup D)} w(e) = c_s$  (note

<sup>6</sup>We can implement an approximate version of Algorithm 1.5, where element weights are in the form  $(1 + \epsilon)^i$  for some  $i$ , and we say that a set  $s$  is tight if  $W(s) \geq c_s/(1 + \epsilon)$ , increasing the approximation ratio by another  $(1 + \epsilon)$  multiplicative factor. It is not hard to see that this can be done in  $O((|\mathcal{S}_{>x^*}| + |\mathcal{E}_{>x^*}|) \log(n)/\epsilon)$  time. We further show that the  $\log n$  term can be eliminated.

<sup>7</sup>Consider, e.g., when  $\mathcal{E} = \{e_1, \dots, e_{12}\}$ ,  $\mathcal{S} = \{s_1 = \{e_1, \dots, e_{10}\}, s_2 = \{e_{10}, e_{11}, e_{12}\}\}$ ,  $c_{s_1} = c_{s_2} = 10$ , and  $D = \{e_{12}\}$ . If we start with all element-weights being zero except  $w(e_1) = w(e_{12}) = 10$ , a deletion of  $e_{12}$  causes  $D_{>0} = \{e_{12}\}$  and  $\mathcal{E}'_{>0} = \{e_1\}$ , making the new set system to violate (1.4) at  $x = 0$ . But it is not enough to change only the weight of  $e_1$  which is the only element in  $\mathcal{E}'_{>0}$ . Lemma 1.7(i) guarantees that this will not happen if  $(\mathcal{I}, w)$  is computed in a certain way as in Algorithm 1.5.

that elements in  $D$  also contribute to the weights of sets). This means that if  $e$  is already in a tight set, then  $w(e) = 0$ . Otherwise, it is increased until a new tight set  $s$  is created, which will be added to  $\mathcal{I}$ . We keep the newly inserted elements in a separate set (call it  $\mathcal{E}'$  for now) because they do not get weights in the uniform way (like when we run Algorithm 1.5).<sup>8</sup> When Algorithm 1.6 calls Algorithm 1.5 on some  $(\mathcal{S}_{>x^*}, \mathcal{E}'_{>x^*})$ , it will try to include in a greedy manner elements from  $\mathcal{E}'$  in the uniform weight increment process and move them to  $\mathcal{E}$ . See Section 3 and Appendix 5 for the details.

## 2 Minimum Set Cover in the Static Setting

In this section, we describe some basic concepts about the set cover problem *in the static setting*. We use the notations that were introduced in Section 1. We start with a simple lemma that follows from LP-duality.

**Lemma 2.1.** *Consider a valid set cover  $\mathcal{S}' \subseteq \mathcal{S}$  and an assignment of nonnegative weights  $\{w(e)\}$  to every element  $e \in \mathcal{E}$  that satisfy (1.2), i.e.  $\{w(e)\}$  forms a valid fractional packing. If  $c(\mathcal{S}') \leq \alpha \cdot \sum_{e \in \mathcal{E}} w(e)$ , then  $\mathcal{S}'$  is an  $\alpha$ -approximate minimum set cover.*

In Section 1, we described a simple static primal-dual algorithm that returns an  $f$ -approximate minimum set cover (see Algorithm 1.5). We now consider a *discretized* variant of the above algorithm, which increases the weights of elements in powers of  $(1 + \epsilon)$ , instead of increasing these weights in a continuous manner. This results in a *hierarchical partition* of the set-system  $(\mathcal{S}, \mathcal{E})$ , which assigns the sets and elements to different levels. In the Appendix, we explain how the algorithm generates this hierarchical partition. Here, we only state some important properties of the partition and show how these properties imply a  $(1 + \epsilon)f$ -approximation for the minimum set cover problem. For the rest of the paper, we fix two parameters  $\epsilon, L$ .

$$0 < \epsilon < 1/2 \text{ and } L = \lceil \log_{(1+\epsilon)}(C \cdot n) \rceil + 1. \quad (2.1)$$

The algorithm outputs a *hierarchical partition* of the set-system  $(\mathcal{S}, \mathcal{E})$ , where each set  $s \in \mathcal{S}$  is assigned to some *level*  $\ell(s) \in \{0, \dots, L\}$ . The level of an element  $e \in \mathcal{E}$  is defined as the maximum level among all the sets it belongs to, i.e.,  $\ell(e) = \max\{\ell(s) : s \in \mathcal{S}, e \in s\}$ . Note that if  $e \in s$ , then  $\ell(e) \geq \ell(s)$ .

**Tight and slack sets:** Recall that  $W(s) = \sum_{e \in s} w(e)$  denotes the total weight received by a set  $s \in \mathcal{S}$  from all its elements. We say that a set  $s \in \mathcal{S}$  is *tight* if  $(1 + \epsilon)^{-1}c_s \leq W(s) \leq c_s$  and *slack* if  $0 \leq W(s) < (1 + \epsilon)^{-1}c_s$ . The hierarchical partition returned by the algorithm satisfies the following properties.

**Property 2.2.** *For every element  $e \in \mathcal{E}$ , we have  $w(e) = (1 + \epsilon)^{-\ell(e)}$ .*

**Property 2.3.** *Every set  $s \in \mathcal{S}$  has  $0 \leq W(s) \leq c_s$ . Furthermore, every set  $s \in \mathcal{S}$  that is slack has  $\ell(s) = 0$ .*

**Property 2.4.** *Every element  $e \in \mathcal{E}$  is contained in at least one tight set.*

**Lemma 2.5.** *Let  $\mathcal{S}_{tight} = \{s \in \mathcal{S} : (1 + \epsilon)^{-1}c_s \leq W(s) \leq c_s\}$  denote the collection of tight sets. They form a  $(1 + \epsilon)f$ -approximate minimum set cover of the input  $(\mathcal{S}, \mathcal{E})$ .*

*Proof.* Since each element belongs to at most  $f$  sets, a simple counting argument gives us:

$$\begin{aligned} (1 + \epsilon)f \cdot \sum_{e \in \mathcal{E}} w(e) &\geq (1 + \epsilon) \cdot \sum_{s \in \mathcal{S}} W(s) \\ &\geq \sum_{s \in \mathcal{S}_{tight}} (1 + \epsilon) \cdot W(s) \\ &\geq \sum_{s \in \mathcal{S}_{tight}} c_s \\ &= c(\mathcal{S}_{tight}) \end{aligned} \quad (2.2)$$

<sup>8</sup>In particular, we can show that weights of all elements in the set system  $(\mathcal{S}, \mathcal{E} \cup D)$  are as if we run the static algorithm (Algorithm 1.5) on this set system. We cannot say the same for  $(\mathcal{S}, \mathcal{E} \cup D \cup \mathcal{E}')$ .

By Property 2.4, every element  $e \in \mathcal{E}$  is covered by some set in  $\mathcal{S}_{tight}$ . In other words, the sets in  $\mathcal{S}_{tight}$  form a valid set cover. Furthermore, by Property 2.3, we have  $0 \leq W(s) \leq c_s$  for all sets  $s \in \mathcal{S}$ . Accordingly, the weights  $\{w(e)\}$  assigned to the elements form a valid fractional packing. From (2.2) and Lemma 2.1, we now infer that the sets in  $\mathcal{S}_{tight}$  form a  $(1 + \epsilon)f$ -approximate minimum set cover of the input  $(\mathcal{S}, \mathcal{E})$ .  $\square$

### 3 Our Dynamic Algorithm

Consider the minimum set cover problem in a dynamic setting, where the input  $(\mathcal{S}, \mathcal{E})$  keeps changing via a sequence of element insertions and deletions. Specifically, during each update, an element is either inserted into or deleted from the set system  $(\mathcal{S}, \mathcal{E})$ . When an element  $e$  is inserted, we get to know about the sets in  $\mathcal{S}$  that contain the element  $e$ . We assume that  $f$  remains an upper bound on the maximum frequency of an element throughout this sequence of updates (although our dynamic algorithm does not need to know the value of  $f$  in advance). We will present a *deterministic* dynamic algorithm for maintaining a  $(1 + \epsilon)f$ -approximate minimum set cover in this setting with  $O(f \cdot \log(Cn)/\epsilon^2)$  amortized update time.

#### 3.1 Classification of elements

The main idea behind our dynamic algorithm is simple. We maintain a relaxed version of the hierarchical partition from Section 2 in a *lazy manner*. To be more specific, in the preprocessing phase we start with a set-system  $(\mathcal{S}, \mathcal{E})$  where  $\mathcal{E} = \emptyset$ . At this point, every set  $s \in \mathcal{S}$  is at level  $\ell(s) = 0$  and has a weight  $W(s) = 0$ , and Properties 2.2, 2.3, 2.4 are vacuously true. Subsequently, while handling the sequence of updates, whenever we observe that a significant fraction of elements has been deleted from levels  $\leq i$  for some  $i \in [0, L]$ , we *rebuild* all the levels  $\{0, \dots, i\}$  in a certain natural manner. We refer to the subroutine which performs this rebuilding as  $\text{REBUILD}(\leq i)$ .

We will classify elements into three distinct types – *active*, *passive* and *dead*. Let  $A, P$  and  $D$  respectively denote the set of active, passive and dead elements. Informally, every element is *active* in the hierarchical partition described in Section 2, where we considered the static setting. To get the main intuition in the dynamic setting, consider an update at some time-step  $t$ , and suppose that this update does *not* lead to a call to the subroutine  $\text{REBUILD}(\leq i)$  for any  $i \in [0, L]$ . Recall that a set  $s \in \mathcal{S}$  is called *tight* when its weight lies in the range  $[(1 + \epsilon)^{-1}c_s, c_s]$  and *slack* when its weight lies in the range  $[0, (1 + \epsilon)^{-1}c_s)$ . As in Section 2, suppose that the tight sets in the hierarchical partition form a valid set cover just before the update at time-step  $t$  (see Lemma 2.5). Now, consider three possible cases.

*Case (a): The update at time-step  $t$  deletes an element  $e$ .* In this case, we classify the element  $e$  as *dead*. We continue to *pretend*, however, that the element  $e$  still exists and do *not* change its weight  $w(e)$ . Thus, we take the value of  $w(e)$  into account while calculating the weight of any set in the fractional packing solution. This ensures that the collection of tight sets remains a valid set cover for the current input  $(\mathcal{S}, \mathcal{E})$ .

*Case (b): The update at time-step  $t$  inserts an element  $e$  that belongs to at least one tight set.* In this case, we assign the element  $e$  to level  $\ell(e) = \max\{\ell(s) : s \in \mathcal{S}, e \in s\}$ , classify it as *passive*, and assign it a weight  $w(e) = 0$ . This ensures that the tight sets continue to remain a set cover in  $(\mathcal{S}, \mathcal{E})$ .

*Case (c): The update at time-step  $t$  inserts an element  $e$  such that all sets containing  $e$  are slack.* In this case, Property 2.3 implies that every set containing the element  $e$  lies at level 0. Hence, we assign the element  $e$  also to level  $\ell(e) = \max\{\ell(s) : s \in \mathcal{S}, e \in s\} = 0$ . Unlike in Case (b), however, here we can no longer leave the hierarchical partition unchanged, since in that event the collection of tight sets will no longer form a valid set cover. We address this issue in the following manner. Let  $\mathcal{S}_e$  denote the collection of sets containing  $e$ . Note that  $|\mathcal{S}_e| \leq f$ . Let  $\lambda > 0$  be the *minimum value* such that if we increase the weight of each set in  $\mathcal{S}_e$  by an additive  $\lambda$ , then the weight of some set  $s \in \mathcal{S}_e$  becomes equal to  $c_s$ . We classify the

element  $e$  as *passive*, and assign it a weight  $w(e) = \lambda$ . This ensures that now the collection of tight sets again forms a valid set cover. This also leads to a very important consequence, which is stated below.

**Claim 3.1.** *A passive element  $e$  receives a weight of  $w(e) \leq (1 + \epsilon)^{-\ell(e)}$  just after getting inserted.*

*Proof.* In Case (b) above, a passive element receives zero weight and the claim trivially holds. For the rest of the proof, consider the scenario described in Case (c) above. Recall that as per (1.1) we have  $c_s \leq 1$  for every set  $s \in \mathcal{S}$ . Let  $s' \in \mathcal{S}$  be a set containing  $e$  whose weight becomes equal to  $c_{s'}$  when we assign a weight of  $\lambda$  to the element  $e$  (see the description for Case (c) above). Thus, we must have  $\lambda \leq c_{s'} \leq 1 = (1 + \epsilon)^0 = (1 + \epsilon)^{\ell(e)}$ .  $\square$

## 3.2 Levels and Weights of elements

Throughout the duration of our algorithm, the level of an element  $e$  (regardless of whether it is active, passive or dead) will be defined to be  $\ell(e) = \max\{\ell(s) : s \in \mathcal{S}, e \in s\}$ . From the preceding discussion, we also conclude that the weights assigned to the elements satisfy the following conditions.

If an element  $e$  is active, then  $w(e) = (1 + \epsilon)^{-\ell(e)}$ . In contrast, if an element  $e$  is passive, then  $w(e) \leq (1 + \epsilon)^{-\ell(e)}$ . Finally, if an element  $e$  is dead, then its weight depends on its state at the time of its deletion. Specifically, if it was active at the time of its deletion, then  $w(e) = (1 + \epsilon)^{-\ell(e)}$ . If it was passive at the time of its deletion, then  $w(e) \leq (1 + \epsilon)^{-\ell(e)}$ . To summarize, a dead element  $e$  always has  $w(e) \leq (1 + \epsilon)^{-\ell(e)}$ .

## 3.3 The shadow input and the invariants

Recall that the set  $\mathcal{E}$  is partitioned into two subsets, namely  $A \subseteq \mathcal{E}$  and  $P = \mathcal{E} \setminus A$ . From the way we assign the weights to elements, it follows that our algorithm works by *pretending* as if the dead elements *were* still present in the input. Accordingly, we consider an input  $(\mathcal{S}, \mathcal{E}^*)$ , where  $\mathcal{E}^* = \mathcal{E} \cup D$ . We refer to  $(\mathcal{S}, \mathcal{E}^*)$  as the *shadow input* (as opposed to the actual input  $(\mathcal{S}, \mathcal{E})$ ). Indeed, the hierarchical partition maintained by our dynamic algorithm will be similar to the one from Section 2 on the shadow input  $(\mathcal{S}, \mathcal{E}^*)$ , barring the fact that the passive/dead elements will have weights  $w(e) \leq (1 + \epsilon)^{-\ell(e)}$ . To explain this more formally, we use the following notations. For every set  $s \in \mathcal{S}$ , we let  $W(s) = \sum_{e \in s \cap \mathcal{E}} w(e)$  and  $W^*(s) = \sum_{e \in s \cap \mathcal{E}^*} w(e)$  respectively denote the total weight of all the elements that belong to  $s$  in  $(\mathcal{S}, \mathcal{E})$  and in  $(\mathcal{S}, \mathcal{E}^*)$ . Our dynamic algorithm will satisfy the three invariants stated below. Invariant 3.1 follows from the discussion in Section 3.2. Invariant 3.2 is analogous to Property 2.3, whereas Invariant 3.3 is analogous to Property 2.4.

**Invariant 3.1.** *Consider any element  $e \in \mathcal{E}^* = A \cup P \cup D$ . The level of  $e$  is defined as  $\ell(e) = \max\{\ell(s) : s \in \mathcal{S}, e \in s\}$ . If  $e \in A$ , then we have  $w(e) = (1 + \epsilon)^{-\ell(e)}$ . Otherwise, if  $e \in P \cup D$ , then we have  $0 \leq w(e) \leq (1 + \epsilon)^{-\ell(e)}$ .*

**Invariant 3.2.** *Every set  $s \in \mathcal{S}$  satisfies  $0 \leq W^*(s) \leq c_s$ . Furthermore, every set  $s \in \mathcal{S}$  with weight  $W^*(s) < (1 + \epsilon)^{-1}c_s$  is at level  $\ell(s) = 0$ .*

**Invariant 3.3.** *Each element  $e \in \mathcal{E} \cup D$  is contained in at least one set  $s \in \mathcal{S}$  with  $(1 + \epsilon)^{-1}c_s \leq W^*(s) \leq c_s$ .*

Let  $\mathcal{S}_{tight}^* \subseteq \mathcal{S}$  be the collection of sets with weights  $(1 + \epsilon)^{-1}c_s \leq W^*(s) \leq c_s$  in the hierarchical partition maintained by our algorithm. Replacing Properties 2.3, 2.4 by Invariants 3.2, 3.3 in the proof of Lemma 2.5, we conclude that  $\mathcal{S}_{tight}^*$  gives a  $(1 + \epsilon)f$ -approximate minimum set cover in the shadow input  $(\mathcal{S}, \mathcal{E}^*)$ . Invariant 3.3 further implies that  $\mathcal{S}_{tight}^*$  is a valid set cover in the actual input  $(\mathcal{S}, \mathcal{E})$ . We will show later that  $\mathcal{S}_{tight}^*$  is in fact a  $(1 + O(\epsilon))f$ -approximate minimum set cover in the actual input  $(\mathcal{S}, \mathcal{E})$  as well. This happens because, intuitively, our dynamic algorithm ensures that the actual input  $(\mathcal{S}, \mathcal{E})$  always remains *close* to the shadow input  $(\mathcal{S}, \mathcal{E}^*)$ .

### 3.4 The dynamic algorithm

Recall that  $A, P,$  and  $D$  respectively denote the set of active, passive and deleted elements. We let  $A_i, P_i$  and  $D_i$  respectively denote the set of active, passive and dead elements at level  $i \in [0, L]$ . Let  $\mathcal{E}_i = \{e \in \mathcal{E} : \ell(e) = i\}$  denote the set of all elements in the current input that are at level  $i \in [0, L]$ . Thus, for each  $i \in [0, L]$ , the set  $\mathcal{E}_i$  is partitioned into two subsets:  $A_i$  and  $P_i$ . For each level  $i \in [0, L]$ , we also define:

$$\mathcal{E}_{\leq i} = \cup_{j \leq i} \mathcal{E}_j, \quad A_{\leq i} = \cup_{j \leq i} A_j, \quad P_{\leq i} = \cup_{j \leq i} P_j, \quad \text{and} \quad D_{\leq i} = \cup_{j \leq i} D_j. \quad (3.1)$$

For every level  $i \in [0, L]$ , we maintain a counter  $\mathcal{C}_{\leq i}$ . Each call to  $\text{REBUILD}(\leq i)$  sets  $D_{\leq i} = P_{\leq i} = \emptyset$  and  $\mathcal{C}_{\leq j} = \epsilon \cdot |\mathcal{E}_{\leq j}|$  for all  $j \leq i$ . In contrast, every time an element gets deleted from some level  $i$ , for all  $j \in [i, L]$  we decrease the counter  $\mathcal{C}_{\leq j}$  by one. Finally, to ensure that the shadow input  $(\mathcal{S}, \mathcal{E}^*)$  remains *close* to the actual input  $(\mathcal{S}, \mathcal{E})$ , we call  $\text{REBUILD}(\leq i)$  whenever  $\mathcal{C}_{\leq i}$  becomes equal to 0. If the counters of multiple levels become 0 during the same update, we call  $\text{REBUILD}(\leq i)$  for the largest such level  $i$ .

**Tight sets:** As in Section 3.3, we will let  $\mathcal{S}_{tight}^* = \{s \in \mathcal{S} : (1 + \epsilon)^{-1}c_s \leq W^*(s) \leq c_s\}$  denote the collection of tight sets with respect to the shadow input  $(\mathcal{S}, \mathcal{E}^*)$ .

**Preprocessing phase:** Initially, we have an input  $(\mathcal{S}, \mathcal{E})$  where  $\mathcal{E} = \emptyset$ . At this point,  $D = \emptyset$ , every set  $s \in \mathcal{S}$  is at level  $\ell(s) = 0$  with weight  $W^*(s) = 0$ , and hence Invariants 3.1, 3.2, 3.3 are vacuously true.

**Handling the deletion of an element:** When an element  $e$  gets deleted, we call the subroutine described in Figure 1. Steps 01 – 02 in Figure 1 were explained under *Case (a)* in Section 3.1, whereas steps 03 – 07 in Figure 1 were explained while defining the counters  $\mathcal{C}_{\leq i}$ .

01. Remove the element  $e$  from  $\mathcal{E}_{\ell(e)}$ , and from  $A_{\ell(e)} \cup P_{\ell(e)}$ .
02. Insert the element  $e$  into  $D_{\ell(e)}$ .
03. FOR  $k = L$  down to  $\ell(e)$ :
04.      $\mathcal{C}_{\leq k} \leftarrow \mathcal{C}_{\leq k} - 1$ .
05.     IF  $\mathcal{C}_{\leq k} = 0$ , THEN
06.         Call the subroutine  $\text{REBUILD}(\leq k)$ .
07.     RETURN.

Figure 1: Handling the deletion of an element  $e$ .

**Handling the insertion of an element:** When an element  $e$  gets inserted, we call the subroutine in Figure 2. Steps 02 – 04 and 05 – 09 in Figure 2 were respectively discussed under *Case (b)* and *Case (c)* in Section 3.1.

01. Let  $i = \max\{\ell(s) : s \in \mathcal{S}, e \in s\}$ .
02. IF there is at least one set  $s \in \mathcal{S} \cap \mathcal{S}_{tight}^*$  that contains  $e$ , THEN
03.      $\ell(e) \leftarrow i$ .
04.     Insert the element  $e$  into  $\mathcal{E}_i$  and into  $P_i$ , with weight  $w(e) \leftarrow 0$ .
05. ELSE
06.     Let  $\mathcal{S}_e = \{s \in \mathcal{S}, e \in s\}$  be the collection of all sets that contain  $e$ .
06.     Let  $\lambda = \min\{x : W^*(s) + x = c_s \text{ for some } s \in \mathcal{S}_e\}$ .
07.      $\ell(e) \leftarrow i$ .
08.     Insert the element  $e$  into  $\mathcal{E}_i$  and into  $P_i$ , with weight  $w(e) \leftarrow \lambda$ .
09.     FOR all sets  $s \in \mathcal{S}_e$ :  $W^*(s) \leftarrow W^*(s) + w(e)$ .

Figure 2: Handling the insertion of an element  $e$ .

**Output of our algorithm:** We maintain the collection of tight sets  $\mathcal{S}_{tight}^* = \{s \in \mathcal{S} : (1 + \epsilon)^{-1}c_s \leq W^*(s) < c_s\}$ . We show in Section 4 that  $\mathcal{S}_{tight}^*$  is a  $(1 + \epsilon)f$ -approximate minimum set cover in  $(\mathcal{S}, \mathcal{E})$ .

**Correctness of the invariants:** Suppose that Invariants 3.1, 3.2, 3.3 hold just before the deletion of an element  $e$ . This is handled by the subroutine in Figure 1. It is easy to check that steps 01 – 02 in Figure 1 do *not* lead to a violation of any invariant. This is because the element  $e$  gets moved from  $A \cup P$  to  $D$ , but its weight  $w(e)$  remains the same, and it still contributes to the weights  $W^*(s)$  of all the sets  $s$  containing  $e$ .

Similarly, suppose that Invariants 3.1, 3.2 and 3.3 hold just before the insertion of an element  $e$ . We handle this insertion by calling the subroutine in Figure 2. Consider two possible cases.

*Case (1): Steps 02 – 04 get executed in Figure 2.* In this case, the element  $e$  becomes passive with weight  $w(e) = 0$ , and it belongs to at least one tight set. Thus, the weight  $W^*(s)$  of every set  $s \in \mathcal{S}$  remains unchanged, and the three invariants continue to remain satisfied.

*Case (2): Steps 05 – 09 get executed in Figure 2.* In this case, all the sets  $s \in \mathcal{S}$  containing  $e$  have weights  $W^*(s) < (1 + \epsilon)^{-1}c_s$  and are at level 0 (see Invariant 3.2) at the time  $e$  gets inserted. Let  $\mathcal{S}'_e = \arg \min_{s \in \mathcal{S}_e} \{c_s - W^*(s)\}$ . After we assign weight  $w(e) \leftarrow \lambda$  to the element  $e$ , every set  $s \in \mathcal{S}'_e$  gets weight  $W^*(s) = c_s$ , and every other set  $s \in \mathcal{S}_e \setminus \mathcal{S}'_e$  continues to have weight  $W^*(s) < c_s$  (even though its weight increased). The weights of the sets  $s \in \mathcal{S} \setminus \mathcal{S}_e$  do not change. This ensures that Invariants 3.2 and 3.3 continue to hold. Finally, revisiting the proof of Claim 3.1, we infer that Invariant 3.1 also continues to hold, since  $e$  becomes passive with weight  $w(e) = \lambda \leq 1 = (1 + \epsilon)^0 = (1 + \epsilon)^{-\ell(e)}$ .

To summarize, we conclude that if the subroutine  $\text{REBUILD}(\leq j)$  has the property that a call to this subroutine never leads to a violation of the invariants, then the invariants continue to hold all the time.

**Data structures:** We use the following data structures. For each level  $i \in [1, L]$ , we maintain the sets  $\mathcal{E}_i, A_i, P_i$  and  $D_i$  as doubly linked lists. Each entry in each of these lists also maintains a bidirectional pointer to the corresponding element. Using these pointers, we can determine the state of a given element (e.g., whether it is active, passive or dead) and insert/delete it in a given list in  $O(1)$  time.

For every element  $e \in \mathcal{E} \cup D$ , we maintain its level  $\ell(e)$  and weight  $w(e)$ . For every set  $s \in \mathcal{S}$ , we also maintain its level  $\ell(s)$  and weight  $W^*(s)$  with respect to the shadow input  $(\mathcal{S}, \mathcal{E}^*)$ . Finally, for every level  $i \in [0, L]$ , we maintain the counter  $\mathcal{C}_{\leq i}$ .

### 3.5 The $\text{REBUILD}(\leq k)$ subroutine

A detailed description of the subroutine appears in Section 5. Here, we summarize a few key properties of this subroutine that will be heavily used in the analysis of our algorithm. Property 3.4 ensures that Invariants 3.1, 3.2, 3.3 do not get violated. Property 3.5 specifies the time taken to implement a call to the subroutine, and how the counters  $\{\mathcal{C}_{\leq i}\}$  get updated as a result of this call. Property 3.6, on the other hand, explains how the subroutine changes the states and levels of different elements in the hierarchical partition.

**Property 3.4.** *If Invariants 3.1, 3.2, 3.3 were satisfied just before a call to the subroutine  $\text{REBUILD}(\leq k)$  for any  $k \in [0, L]$ , then these invariants continue to remain satisfied at the end of that call.*

**Property 3.5.** *Consider any level  $k \in [0, L]$ . The time taken to implement a call to  $\text{REBUILD}(\leq k)$  is proportional to  $f$  times the number of elements in  $\mathcal{E}_{\leq k} \cup D_{\leq k}$  in the beginning of the call, plus  $O(\log(Cn)/\epsilon)$ . Furthermore, at the end of this call, we have  $\mathcal{C}_{\leq j} = \epsilon \cdot |\mathcal{E}_{\leq j}|$  for all levels  $j \in [0, k]$ .*

**Property 3.6.** *Consider any level  $k \in [0, L]$  and any call to the subroutine  $\text{REBUILD}(\leq k)$ .*

**(1)** *The call to  $\text{REBUILD}(\leq k)$  cleans up all the dead elements at level  $\leq k$ . Specifically, this means the following. Consider any element  $e$  that belongs to  $D_{\leq k}$  just before the call to  $\text{REBUILD}(\leq k)$ . Then that element  $e$  does not appear in any of the sets  $A, P$ , or  $D$  at the end of the call.*

(2) The call to  $\text{REBUILD}(\leq k)$  converts some of the passive elements at level  $\leq k$  to passive elements at level  $k + 1$ , and the remaining passive elements at level  $\leq k$  get converted into active elements at level  $\leq k + 1$ . Specifically, let  $Z$  denote the set of elements in  $P_{\leq k}$  just before the call to  $\text{REBUILD}(\leq k)$ . Then during the call to  $\text{REBUILD}(\leq k)$ , a subset  $Z' \subseteq Z$  of these elements gets added to  $P_{k+1}$ , and the remaining elements  $e \in Z' \setminus Z$  get added to  $A_{\leq k+1}$ .

(3) The call to  $\text{REBUILD}(\leq k)$  moves up some of the active elements at level  $\leq k$  to level  $k + 1$ , and the remaining active elements at level  $\leq k$  continue to be active at level  $\leq k$ . In other words, the elements in  $A_{\leq k}$  never go out of the set  $A_{\leq k+1}$  during the call to  $\text{REBUILD}(\leq k)$ .

(4) The call to  $\text{REBUILD}(\leq k)$  does not touch the elements at level  $\geq k + 1$ . In other words, for any  $i \geq k + 1$ , if an element  $e$  belonged to  $A_i$ ,  $P_i$  or  $D_i$  just before the call to  $\text{REBUILD}(\leq k)$ , then it continues to belong to the same set  $A_i$ ,  $P_i$  or  $D_i$  at the end of the call to  $\text{REBUILD}(\leq k)$ .

**Corollary 3.7.** At the end of any call to  $\text{REBUILD}(\leq k)$ , we have  $D_j = P_j = \emptyset$  for all  $j \in [0, k]$ .

*Proof.* Follows from parts (1), (2) of Property 3.6.  $\square$

## 4 Analysis of our dynamic algorithm

We start by proving some simple properties of our algorithm that will be useful in the subsequent analysis. These properties formalize the intuition that the fractional packing solution maintained by the algorithm does not change significantly in between two successive calls to  $\text{REBUILD}(\leq j)$  at any level  $j \in [0, L]$ . This happens because of three main reasons (see Figure 1). First, we set  $C_{\leq k} = \epsilon \cdot |\mathcal{E}_{\leq k}|$  for all  $k \in [0, j]$  at the end of each call to  $\text{REBUILD}(\leq j)$ . Second, we decrement the counter  $C_{\leq k}$  for all  $k \in [j, L]$  each time some element gets deleted from level  $j$ . Third, we call  $\text{REBUILD}(\leq k)$  whenever  $C_{\leq k}$  becomes equal to 0.

**Notation:** Throughout the rest of this section, we use the superscript  $(t)$  to denote the status of some set/counter at time-step  $t$ . For instance, the symbol  $D_{\leq j}^{(t)}$  will denote the set of dead elements at level  $\leq j$  at time-step  $t$ , and the symbol  $C_{\leq j}^{(t)}$  will denote the value of the counter  $C_{\leq j}$  at time-step  $t$ .

**Lemma 4.1.** Fix any level  $j \in [0, L]$  and consider any two time-steps  $t' < t$  that satisfy the following properties: (1) A call was made to the subroutine  $\text{REBUILD}(\leq k)$  for some  $k \in [j, L]$  just before time-step  $t'$ . (2) No call was made to  $\text{REBUILD}(\leq k)$  for any  $k \in [j, L]$  during the time-interval  $[t', t]$ . Let  $M_{\leq j}^{(t' \rightarrow t)}$  denote the set of elements that got deleted from level  $\leq j$  during the time-interval  $[t', t]$ . Then we have:

$$\left| M_{\leq j}^{(t' \rightarrow t)} \right| + C_{\leq j}^{(t)} = \epsilon \cdot \left| A_{\leq j}^{(t')} \right|.$$

*Proof.* As the subroutine  $\text{REBUILD}(\leq k)$  was called for some  $k \in [j, L]$  just before time-step  $t'$ , Property 3.5 implies that  $C_{\leq j}^{(t')} = \epsilon \cdot \left| A_{\leq j}^{(t')} \right|$ . Next, note that during the time-interval  $[t', t]$ , no call is made to the subroutine  $\text{REBUILD}(\leq k)$  for any  $k \in [j, L]$ . Hence, during this time-interval, the counter  $C_{\leq j}$  gets decremented by one iff an element gets deleted from level  $\leq j$  (see Figure 1), and the set  $M_{\leq j}^{(t' \rightarrow t)}$  consists precisely of these elements. Thus, we infer that:  $\left| M_{\leq j}^{(t' \rightarrow t)} \right| + C_{\leq j}^{(t)} = C_{\leq j}^{(t')} = \epsilon \cdot \left| A_{\leq j}^{(t')} \right|$ .  $\square$

**Corollary 4.2.** Consider any level  $j \in [1, L]$  and time-steps  $t' < t$  as defined in Lemma 4.1. Then we have:

$$\left| M_{\leq j}^{(t' \rightarrow t)} \right| \leq \epsilon \cdot \left| A_{\leq j}^{(t')} \right|.$$

*Proof.* Note that the subroutine  $\text{REBUILD}(\leq j)$  gets called whenever  $C_{\leq j} = 0$  (see Figure 1), and before finishing its execution the subroutine resets  $C_{\leq j} = \epsilon \cdot |\mathcal{E}_{\leq j}|$  (see Property 3.5). Thus, the counter  $C_{\leq j}$  always remains nonnegative, and in particular we have  $C_{\leq j}^{(t)} \geq 0$ . The corollary now follows from Lemma 4.1.  $\square$

**Corollary 4.3.** Consider any level  $j \in [1, L]$  and time-steps  $t' < t$  as defined in Lemma 4.1. Then we have:

$$\left| D_{\leq j}^{(t)} \right| \leq \epsilon \cdot \left| A_{\leq j}^{(t')} \right|.$$

*Proof.* Since the subroutine  $\text{REBUILD}(\leq k)$  was called for some  $k \in [j, L]$  just before time-step  $t'$ , Corollary 3.7 implies that  $D_{\leq j}^{(t')} = \emptyset$ . We now track how the set  $D_{\leq j}$  changes during the time-interval  $(t', t)$ .

Whenever an element  $e$  gets deleted from level  $\leq j$  during this time-interval, the element  $e$  gets added to both the sets  $D_{\leq j}$  and  $M_{\leq j}^{(t' \rightarrow t)}$ . On the other hand, whenever the subroutine  $\text{REBUILD}(\leq k)$  gets called for some  $k \in [1, j-1]$ , all the dead elements at level  $\leq k$  get removed from the hierarchical partition (see part (1) of Property 3.6). Since  $k < j$ , such a call to  $\text{REBUILD}(\leq k)$  can potentially remove some elements from the set  $D_{\leq j}$ , but no element from  $M_{\leq j}^{(t' \rightarrow t)}$  gets removed due to the call.

Since no call is made to the subroutine  $\text{REBUILD}(\leq k)$  for any  $k \in [j, L]$  during the time-interval  $[t', t]$ , and since  $D_{\leq j} = \emptyset$  at time-step  $t'$ , the preceding discussion implies that  $D_{\leq j}^{(t)} \subseteq M_{\leq j}^{(t' \rightarrow t)}$ . Thus, from Corollary 4.2, we get  $\left| D_{\leq j}^{(t)} \right| \leq \left| M_{\leq j}^{(t' \rightarrow t)} \right| \leq \epsilon \cdot \left| A_{\leq j}^{(t')} \right|$ .  $\square$

**Lemma 4.4.** Consider any level  $j \in [1, L]$  and time-steps  $t' < t$  as defined in Lemma 4.1. Then we have:

$$\left| A_{\leq j}^{(t)} \right| \geq (1 - \epsilon) \cdot \left| A_{\leq j}^{(t')} \right|.$$

*Proof.* According to part (3) of Property 3.6, a call to  $\text{REBUILD}(\leq k)$  for some  $k \in [1, j-1]$  can never decrease the size of the set  $A_{\leq j}$ . Since no call was made to  $\text{REBUILD}(\leq k)$  for any  $k \in [j, L]$  during the time-interval  $[t', t]$ , we conclude that: During the time-interval  $[t', t]$ , the set  $A_{\leq j}$  can decrease in size only via deletion of elements from level  $\leq j$ . Moreover, the set  $M_{\leq j}^{(t' \rightarrow t)}$  contains all these deleted elements. Thus, we infer that:  $A_{\leq j}^{(t')} \setminus A_{\leq j}^{(t)} \subseteq M_{\leq j}^{(t' \rightarrow t)}$ . Applying Corollary 4.2, we now get:  $\left| A_{\leq j}^{(t')} \setminus A_{\leq j}^{(t)} \right| \leq \left| M_{\leq j}^{(t' \rightarrow t)} \right| \leq \epsilon \cdot \left| A_{\leq j}^{(t')} \right|$ . In words, at most an  $\epsilon$  fraction of the elements get deleted from the set  $A_{\leq j}$  during the time-interval  $[t', t]$ . Hence, it follows that  $\left| A_{\leq j}^{(t)} \right| \geq (1 - \epsilon) \cdot \left| A_{\leq j}^{(t')} \right|$ .  $\square$

**Corollary 4.5.** At any time-step  $t$  and any level  $j \in [0, L]$  we have  $\left| D_{\leq j}^{(t)} \right| \leq 2\epsilon \cdot \left| A_{\leq j}^{(t)} \right|$ .

*Proof.* Fix any level  $j \in [0, L]$  and time-step  $t$ . We will show that the lemma holds for level  $j$  at time-step  $t$ . Let  $t' < t$  be the last time-step before  $t$  with the following property: a call was made to  $\text{REBUILD}(\leq k)$  for some  $k \in [j, L]$  just before time-step  $t'$ . Thus, during the time-interval  $[t', t]$  no call was made to  $\text{REBUILD}(\leq k)$  for any  $k \in [j, L]$ . Hence, Corollary 4.3 and Lemma 4.4 imply that:

$$\begin{aligned} \left| D_{\leq j}^{(t)} \right| &\leq \epsilon \cdot \left| A_{\leq j}^{(t')} \right| \\ &= (\epsilon/(1 - \epsilon)) \cdot (1 - \epsilon) \left| A_{\leq j}^{(t')} \right| \\ &\leq (\epsilon/(1 - \epsilon)) \cdot \left| A_{\leq j}^{(t)} \right| \\ &\leq 2\epsilon \cdot \left| A_{\leq j}^{(t)} \right|. \end{aligned}$$

The last inequality holds as long as  $\epsilon \leq 1/2$ . Thus, we infer that  $\left| D_{\leq j}^{(t)} \right| \leq 2\epsilon \cdot \left| A_{\leq j}^{(t)} \right|$  at time-step  $t$ .  $\square$

## 4.1 Bounding the update time of our dynamic algorithm

**Theorem 4.6.** *Our dynamic algorithm has an amortized update time of  $O(f \log(Cn)/\epsilon^2)$ .*

*Proof.* Recall that every element belongs to at most  $f$  sets. Hence, ignoring the potential call to  $\text{REBUILD}(\leq k)$ , it takes  $O(f + L) = O(f + \log_{(1+\epsilon)}(Cn)) = O(f + \log(Cn)/\epsilon)$  time to implement all the steps in Figure 1 and Figure 2. In other words, the update time of our dynamic algorithm is dominated by the time spent on the calls to  $\text{REBUILD}(\leq k)$ . Henceforth, we focus on bounding the time spent on these calls.

Fix any  $k \in [0, L]$  and consider any call to  $\text{REBUILD}(\leq k)$  that is made just after some time-step (say)  $t$ . Let  $t' < t$  be the last time-step before  $t$  with the following property: a call was made to  $\text{REBUILD}(\leq j)$  for some  $j \in [k, L]$  just before time-step  $t'$ . Since  $\mathcal{E}_{\leq k} = A_{\leq k} \cup P_{\leq k}$ , Corollary 3.7 states that:

$$D_{\leq k}^{(t')} = P_{\leq k}^{(t')} = \emptyset, \text{ and hence } \mathcal{E}_{\leq k}^{(t')} = A_{\leq k}^{(t)}. \quad (4.1)$$

Let  $\mathcal{I}_{\leq k}^{(t' \rightarrow t)}$  denote the set of elements that get inserted into the set-system  $(\mathcal{S}, \mathcal{E})$  at some level  $\leq k$  during the time-interval  $[t', t]$ . Furthermore, as in Lemma 4.1, let  $M_{\leq k}^{(t' \rightarrow t)}$  denote the set of elements that get deleted from some level  $\leq k$  during the time-interval  $[t', t]$ . During the same time-interval, no call was made to  $\text{REBUILD}(\leq j)$  for any  $j \in [k, L]$ . Moreover, Property 3.6 implies that a call to the subroutine  $\text{REBUILD}(\leq j)$  for some  $j \in [1, k-1]$  does not change the set of elements in  $\mathcal{E}_{\leq k}$ . Thus, during the time-interval  $[t', t]$ , the only way the set  $\mathcal{E}_{\leq k}$  can increase in size is via insertions of elements at levels  $\leq k$ . It follows that:  $\mathcal{E}_{\leq k}^{(t)} \subseteq \mathcal{E}_{\leq k}^{(t')} \cup \mathcal{I}_{\leq k}^{(t' \rightarrow t)}$ . Since  $\mathcal{E}_{\leq k}^{(t')} = A_{\leq k}^{(t)}$  according to (4.1), we get:

$$\mathcal{E}_{\leq k}^{(t)} \subseteq A_{\leq k}^{(t')} \cup \mathcal{I}_{\leq k}^{(t' \rightarrow t)} \quad (4.2)$$

We will show next that  $|D_{\leq k}^{(t)}| \leq 3\epsilon |A_{\leq k}^{(t')}|$ . Note that  $D_{\leq k}^{(t)} \subseteq D_{\leq k}^{(t')} \cup M_{\leq k}^{(t' \rightarrow t)}$ . From Corollary 4.5 it follows that  $|D_{\leq k}^{(t')}| \leq 2\epsilon |A_{\leq k}^{(t')}|$ . Furthermore, since a call was made to the subroutine  $\text{REBUILD}(\leq k)$  just after time-step  $t$ , it must be the case that  $C_{\leq k}^{(t)} = 0$ . Thus, from Lemma 4.1 we infer that

$$\left| M_{\leq k}^{(t' \rightarrow t)} \right| = \epsilon \cdot \left| A_{\leq k}^{(t')} \right| \quad (4.3)$$

It follows that:

$$\left| D_{\leq k}^{(t)} \right| \leq 3\epsilon |A_{\leq k}^{(t')}| \quad (4.4)$$

Let  $T$  denote the total “cost” (update time) we pay for calling the subroutine  $\text{REBUILD}(\leq k)$  at time-step  $t$ . From (4.2), (4.4) and Property 3.5, we get:

$$\begin{aligned} T &= O\left(f \cdot \left| \mathcal{E}_{\leq k}^{(t)} \right| + f \cdot \left| D_{\leq k}^{(t)} \right|\right) + O\left(\frac{\log(Cn)}{\epsilon}\right) \\ &= O\left(f \cdot \left| A_{\leq k}^{(t')} \right| + f \cdot \left| \mathcal{I}_{\leq k}^{(t' \rightarrow t)} \right| + \frac{\log(Cn)}{\epsilon}\right) \end{aligned} \quad (4.5)$$

After each update, our dynamic algorithm (see Figures 1 and 2) makes at most one call to the  $\text{REBUILD}(\leq i)$  subroutine, over all  $i \in [0, L]$ . Hence, we can safely ignore the term  $O(\log(Cn)/\epsilon)$  in  $T$  above, as this term gets subsumed within our desired update time bound of  $O(f \log(Cn)/\epsilon^2)$ . We *split-up* the remaining chunk of  $T$  into two parts:  $T_1 = O\left(f \cdot \left| A_{\leq k}^{(t')} \right|\right)$  and  $T_2 = O\left(f \cdot \left| \mathcal{I}_{\leq k}^{(t' \rightarrow t)} \right|\right)$ . We *charge* the cost  $T_1$  (resp.  $T_2$ ) by distributing it evenly among the elements that get deleted from (resp. inserted into) level  $\leq k$  during the time-interval  $[t', t]$ . We now bound the total charge accumulated by an element in this fashion.

First, note that as per (4.3),  $\epsilon \cdot |A_{\leq k}^{(t')}|$  elements get deleted from level  $\leq k$  during the time-interval  $[t', t]$ . When we distribute the cost  $T_1$  evenly among them, each of these elements accumulate a charge of  $O(f/\epsilon)$ . Now, consider any element  $e$  that accumulates some charge in this fashion due to the call to REBUILD( $\leq k$ ) just after time-step  $t$ . By definition, this element  $e$  gets deleted during the time-interval  $[t', t]$ . Accordingly, it is not possible for the same element  $e$  to accumulate a similar charge from the same level  $k$  at some future time-step  $t'' > t$ .<sup>9</sup> To summarize, an element  $e$  gets charged at most once from a given level in this manner.

Next, note that when we distribute the cost  $T_2$  evenly among the elements in  $\mathcal{I}_{\leq k}^{(t' \rightarrow t)}$ , each such element accumulates a charge of  $O(f)$ . Consider any element  $e$  that accumulates some charge from level  $k$  just after some time-step  $t$  in this manner. By definition, this element got inserted during the time-interval  $[t', t]$ , and thus it will never get charged due to a call to REBUILD( $\leq k$ ) at some future time-step  $t'' > t$ . To summarize, here again we derive that an element  $e$  gets charged at most once from a given level in this fashion.

From the discussion in the preceding two paragraphs, we conclude that any element  $e$  accumulates a charge of at most  $O(f/\epsilon + f) = O(f/\epsilon)$  from each level. Thus, the total charge accumulated by any element is at most  $O((f/\epsilon)L) = O((f/\epsilon) \log_{(1+\epsilon)}(Cn)) = O(f \log(Cn)/\epsilon^2)$ . This means that the amortized update time of our dynamic algorithm is also  $O(f \log(Cn)/\epsilon^2)$ .  $\square$

## 4.2 Bounding the approximation ratio

Our main result is summarized in the theorem below.

**Theorem 4.7.** *In the hierarchical partition maintained by our dynamic algorithm, the tight sets  $\mathcal{S}_{tight}^* = \{s \in \mathcal{S} : (1 + \epsilon)^{-1}c_s \leq W^*(s) \leq c_s\}$  form a  $(1 + 5\epsilon)f$ -approximate minimum set cover in  $(\mathcal{S}, \mathcal{E})$ .*

We now give a high-level overview of the proof of the above theorem. First, recall that  $\mathcal{E}^* = \mathcal{E} \cup D$ . Hence, the element-weights  $\{w(e)\}$  define a valid fractional packing in the shadow-input  $(\mathcal{S}, \mathcal{E}^*)$  as per Invariant 3.2, and the sets in  $\mathcal{S}_{tight}^*$  form a valid set cover in  $(\mathcal{S}, \mathcal{E}^*)$  as per Invariant 3.3. As per Invariant 3.3, every element  $e \in \mathcal{E} \cup D$  is contained in at least one tight set. In other words, the fractional packing  $\{w(e)\}$  is *approximately maximal*, in the sense that every element belongs to at least one set whose weight cannot be increased by more than  $(1 + \epsilon)$ -factor. Armed with this observation, it is not too difficult to show that the total cost of the dual set cover (defined by the tight sets) is within a multiplicative factor  $(1 + \epsilon)f$  of the total weight of the fractional packing  $\{w(e)\}$  in  $(\mathcal{S}, \mathcal{E}^*)$ . This already implies that the collection of tight sets  $\mathcal{S}_{tight}^*$  forms a  $(1 + \epsilon)f$ -approximate minimum set cover in the shadow input (see Lemma 4.9). The key challenge now is to show that the sets in  $\mathcal{S}_{tight}^*$  also constitute an approximately minimum set cover in the actual input  $(\mathcal{S}, \mathcal{E})$ .

To address this challenge, we exploit the fact that the number of elements in  $D$  is relatively small compared to the number of elements in  $A$  (see Corollary 4.5). This implies that the total weight of the elements in  $D$  is also small compared to the total weight of the elements in  $A$  (see Lemma 4.8). Hence, even if we delete all the elements in  $D$  from the fractional packing  $\{w(e)\}$ , the objective value of the resulting solution will remain close to the objective value of the original fractional packing, which in turn was within a factor  $(1 + \epsilon)f$  of the total cost of the sets in  $\mathcal{S}_{tight}^*$ . So the total weight of the new fractional packing (after deleting the elements in  $D$ ) will be very close to  $(1 + \epsilon)f \cdot c(\mathcal{S}_{tight}^*)$ , where  $c(\mathcal{S}_{tight}^*)$  is the total cost of the sets in  $\mathcal{S}_{tight}^*$  (see Corollary 4.10). Now, this also happens to be a valid fractional packing in the actual input  $(\mathcal{S}, \mathcal{E})$ , because we already had  $W^*(s) \leq c_s$  for all sets  $s \in \mathcal{S}$  and removing the elements in  $D$  will *not* increase the weights of the sets any further. On the other hand, the sets in  $\mathcal{S}_{tight}^*$  form a *valid* set cover in the actual input  $(\mathcal{S}, \mathcal{E})$  as well, as every  $e \in \mathcal{E}$  belongs to a set in  $\mathcal{S}_{tight}^*$ , see Invariant 3.3. In other words, we have identified a valid fractional packing and a valid set cover in  $(\mathcal{S}, \mathcal{E})$  whose objective values are within

<sup>9</sup>If  $e$  is inserted and deleted again later, we consider this to be a different (instance of the) element.

a  $(1 + O(\epsilon))f$ -factor of each other. So the corresponding set cover must be an approximately minimum set cover in  $(\mathcal{S}, \mathcal{E})$ .

**Lemma 4.8.** *We always have  $\sum_{e \in D} w(e) \leq 2\epsilon \cdot \sum_{e \in A} w(e)$ .*

*Proof.* We first express the weight of an element  $e$  as a sum of *increments*, where each increment corresponds to a specific level  $k \geq \ell(e)$ . To be more precise, we define:

$$\Delta_k = \begin{cases} (1 + \epsilon)^{-k} & \text{for } k = L; \\ (1 + \epsilon)^{-k} - (1 + \epsilon)^{-(k+1)} & \text{for } 0 \leq k < L. \end{cases}$$

From Invariant 3.1, we conclude that:

$$w(e) = (1 + \epsilon)^{-\ell(e)} = \sum_{k=\ell(e)}^L \Delta_k \text{ for all } e \in A. \quad (4.6)$$

$$w(e) \leq (1 + \epsilon)^{-\ell(e)} = \sum_{k=\ell(e)}^L \Delta_k \text{ for all } e \in D. \quad (4.7)$$

Now, we derive that:

$$\begin{aligned} \sum_{e \in D} w(e) &\leq \sum_{e \in D} \sum_{k=\ell(e)}^L \Delta_k \\ &= \sum_{k=0}^L \sum_{e \in D: \ell(e) \leq k} \Delta_k \\ &= \sum_{k=0}^L \Delta_k \cdot |D_{\leq k}| \\ &\leq \sum_{k=0}^L \Delta_k \cdot 2\epsilon |A_{\leq k}| \\ &= 2\epsilon \cdot \sum_{k=0}^L \sum_{e \in A: \ell(e) \leq k} \Delta_k \\ &= 2\epsilon \cdot \sum_{e \in A} \sum_{k=\ell(e)}^L \Delta_k \\ &= 2\epsilon \cdot \sum_{e \in A} w(e). \end{aligned}$$

In the above derivation, the first inequality follows from (4.7). The second inequality follows from Corollary 4.5. Finally, the last equality follows from (4.6). This concludes the proof of the lemma.  $\square$

**Lemma 4.9.** *We have  $\sum_{e \in \mathcal{E}^*} w(e) \geq ((1 + \epsilon)f)^{-1} \cdot c(\mathcal{S}_{tight}^*)$ .*

*Proof.* By Invariant 3.3, every element  $e \in \mathcal{E} \cup D = \mathcal{E}^*$  belongs to at least one set in  $\mathcal{S}_{tight}^*$ . We sum over the weights of these elements. Since  $f$  is an upper bound on the maximum frequency of an element, we get:

$$\begin{aligned} \sum_{e \in \mathcal{E}^*} w(e) &\geq f^{-1} \sum_{s \in \mathcal{S}} W^*(s) \geq f^{-1} \sum_{s \in \mathcal{S}_{tight}^*} W^*(s) \\ &\geq f^{-1} \sum_{s \in \mathcal{S}_{tight}^*} (1 + \epsilon)^{-1} c_s \\ &= ((1 + \epsilon)f)^{-1} \cdot c(\mathcal{S}_{tight}^*). \end{aligned}$$

□

**Corollary 4.10.** We have  $\sum_{e \in \mathcal{E}} w(e) \geq ((1 + \epsilon)(1 + 2\epsilon)f)^{-1} \cdot c(\mathcal{S}_{tight}^*)$ .

*Proof.* Using Lemma 4.8, we derive that:

$$(1 + (2\epsilon)^{-1}) \cdot \sum_{e \in A} w(e) \geq \frac{1}{(2\epsilon)} \cdot \left( \sum_{e \in D} w(e) + \sum_{e \in A} w(e) \right)$$

Multiplying both sides in the above inequality by  $2\epsilon(1 + 2\epsilon)^{-1} = (1 + (2\epsilon)^{-1})^{-1}$ , we get:

$$\sum_{e \in A} w(e) \geq (1 + 2\epsilon)^{-1} \cdot \sum_{e \in A \cup D} w(e). \quad (4.8)$$

Now, adding the weights of the passive elements – given by  $\sum_{e \in P} w(e)$  – on both sides of (4.8), we get:

$$\sum_{e \in A \cup P} w(e) \geq (1 + 2\epsilon)^{-1} \cdot \sum_{e \in A \cup P \cup D} w(e). \quad (4.9)$$

Since  $\mathcal{E} = A \cup P$  and  $\mathcal{E}^* = A \cup P \cup D$ , the corollary follows from (4.9) and Lemma 4.9. □

*Proof of Theorem 4.7.* By Invariant 3.3, every element  $e \in \mathcal{E}$  belongs to at least one set in  $\mathcal{S}_{tight}^*$ . In other words, the collection of sets  $\mathcal{S}_{tight}^*$  forms a valid set cover in the input  $(\mathcal{S}, \mathcal{E})$ . Next, from Invariant 3.2 we infer the following bound on the weight  $W(s)$  of any set  $s \in \mathcal{S}$ :

$$W(s) \leq W^*(s) \leq c_s \text{ for all sets } s \in \mathcal{S}.$$

The first inequality holds since we consider the weights of the elements  $e \in \mathcal{E}^* = \mathcal{E} \cup D$  while calculating  $W^*(s)$ , whereas we only consider the weights of the elements  $e \in \mathcal{E}$  while calculating  $W(s)$ . Thus, the element-weights  $\{w(e)\}_{e \in \mathcal{E}}$  form a valid *fractional packing* in  $(\mathcal{S}, \mathcal{E})$ . Finally, Corollary 4.10 implies that the size of this fractional packing is at least  $\alpha$  times of the total cost of the set cover  $\mathcal{S}_{tight}^*$  in  $(\mathcal{S}, \mathcal{E})$ , where  $\alpha = ((1 + \epsilon)(1 + 2\epsilon)f)^{-1} \geq ((1 + 5\epsilon)f)^{-1}$  when  $0 < \epsilon < 1/2$ . Theorem 4.7 now follows from Lemma 2.1.

## 5 Describing the REBUILD( $\leq k$ ) subroutine

The subroutine works in 8 steps. Throughout this section, we use the symbols  $\mathcal{E}_{\leq k}^*$ ,  $\mathcal{A}_{\leq k}^*$ ,  $\mathcal{P}_{\leq k}^*$  and  $\mathcal{D}_{\leq k}^*$  respectively to denote the status of the sets  $\mathcal{E}_{\leq k}$ ,  $\mathcal{A}_{\leq k}$ ,  $\mathcal{P}_{\leq k}$  and  $\mathcal{D}_{\leq k}$  just before the call to the subroutine.

**A note on the invariants:** While going through the description of the subroutine below, it will be helpful to remember that Invariant 3.1 will continue to hold all the time. In contrast, Invariants 3.2 and 3.3 will continue to hold only for those elements and sets that remain *unaffected* (do not change their levels/weights)

during the call to  $\text{REBUILD}(\leq k)$ . These two invariants will be satisfied by the affected elements and sets only at the end of the subroutine.

**Step 1:** Scan through all the elements in  $\mathcal{E}_{\leq k}^* \cup \mathcal{D}_{\leq k}^*$  and identify the collection of sets  $\mathcal{S}' = \{s \in \mathcal{S} : \ell(s) \leq k, s \cap (\mathcal{E}_{\leq k}^* \cup \mathcal{D}_{\leq k}^*) \neq \emptyset\}$ . A set belongs to  $\mathcal{S}'$  iff it is at level  $\leq k$  at this point in time and contains at least one element from  $\mathcal{E}_{\leq k}^* \cup \mathcal{D}_{\leq k}^*$ . These are the sets whose levels and weights will be affected due to the call to  $\text{REBUILD}(\leq k)$ .

**Remark:** Consider any element  $e \in \mathcal{E}_{\leq k}^* \cup \mathcal{D}_{\leq k}^*$ . These are the elements that get affected due to the call to  $\text{REBUILD}(\leq k)$ . Since the level of an element is defined to be the maximum level among all the sets it belongs to, we make the following important observation that will be used throughout the rest of this section.

**Observation 5.1.** *Every set  $s \in \mathcal{S}$  that contains some element  $e \in \mathcal{E}_{\leq k}^* \cup \mathcal{D}_{\leq k}^*$  belongs to the collection  $\mathcal{S}'$ . Furthermore, for every element  $e' \in \mathcal{E} \cup \mathcal{D}$  such that every set containing  $e'$  belongs to  $\mathcal{S}'$ , we must have  $e' \in \mathcal{E}_{\leq k}^* \cup \mathcal{D}_{\leq k}^*$ .*

**Step 2:** Remove all the elements from  $D_{\leq k}$ , and accordingly modify the weights  $W^*(s)$  of the sets in  $\mathcal{S}'$ . Thus, we get  $D_{\leq k} = D_{\leq k} \setminus \mathcal{D}_{\leq k}^* = \emptyset$ . (See part (1) of Property 3.6.)

**Step 3:** For every element  $e \in \mathcal{P}_{\leq k}^*$ , set  $w(e) \leftarrow 0$  and accordingly modify the weight  $W^*(s)$  of each set  $s \in \mathcal{S}'$  that contains  $e$ . Now, move every set  $s \in \mathcal{S}'$  to level  $(k+1)$ , by setting  $\ell(e) \leftarrow (k+1)$  for all  $s \in \mathcal{S}'$ . Since the level of an element is defined to be the maximum level among all the sets it belongs to, this implies that all the elements  $e \in \mathcal{P}_{\leq k}^* \cup \mathcal{A}_{\leq k}^*$  also move up to level  $k+1$  (see Observation 5.1). Each element  $e \in \mathcal{P}_{\leq k}^*$  becomes part of  $P_{k+1}$ , whereas each element  $e \in \mathcal{A}_{\leq k}^*$  becomes part of  $A_{k+1}$ . Thus, at this point in time we get  $P_{\leq k} = A_{\leq k} = D_{\leq k} = \emptyset$ . We continue to have  $w(e) = 0$  for all  $e \in \mathcal{P}_{\leq k}^*$  (this does not violate Invariant 3.1 since we now have  $\mathcal{P}_{\leq k}^* \subseteq P_{k+1}$ ). However, to ensure that Invariant 3.1 holds for the active elements, we now set  $w(e) = (1+\epsilon)^{-(k+1)}$  for all  $e \in \mathcal{A}_{\leq k}^*$ , and we accordingly modify the weights  $W^*(s)$  of the sets in  $\mathcal{S}'$ .

**Remark:** Steps 2 and 3 can only decrease the weights  $W^*(s)$  of the sets  $s \in \mathcal{S}'$ . This is because just before step 2 we had  $w(e) \geq (1+\epsilon)^{-k}$  for all elements  $e \in \mathcal{A}_{\leq k}^*$ , and  $w(e) \geq 0$  for all elements  $e \in \mathcal{P}_{\leq k}^*$ . We also had  $W^*(s) \leq c_s$  for all sets  $s \in \mathcal{S}'$ , as per Invariant 3.2. Now, Step 2 removed the elements from  $D_{\leq k}$  and Step 3 decreased the weights of the elements in  $\mathcal{A}_{\leq k}^* \cup \mathcal{P}_{\leq k}^*$ . Thus, we continue to have  $W^*(s) \leq c_s$  for all sets  $s \in \mathcal{S}'$  even after Step 3.

**Step 4:** FOR every element  $e \in \mathcal{P}_{\leq k}^*$ :

- Let  $\mathcal{S}'_e \subseteq \mathcal{S}'$  denote the collection of all sets that contain  $e$ . Note that  $|\mathcal{S}'_e| \leq f$ .
- (a) IF  $W^*(s) \leq c_s - (1+\epsilon)^{-(k+1)}$  for all sets  $s \in \mathcal{S}'_e$ , THEN
  - Increase the weight of element  $e$  from 0 (see Step 3 above) to  $w(e) \leftarrow (1+\epsilon)^{-(k+1)}$ , move the element  $e$  from  $P_{k+1}$  to  $A_{k+1}$ , and accordingly modify the weight  $W^*(s)$  of every set  $s \in \mathcal{S}'_e$ .
- (b) ELSE
  - Let  $\lambda = \min\{x : W^*(s) + x = c_s \text{ for some } s \in \mathcal{S}'_e\}$ . Note that in this case  $\lambda < (1+\epsilon)^{-(k+1)}$ .
  - Increase the weight of element  $e$  from 0 (see Step 3 above) to  $w(e) \leftarrow \lambda$ , and accordingly modify the weight  $W^*(s)$  of every set  $s \in \mathcal{S}'_e$ .

**Remark:** In Step 4 above, we set a weight  $w(e) = (1+\epsilon)^{-(k+1)}$  to an element  $e \in \mathcal{P}_{\leq k}^*$  (thereby making it active) only if  $W^*(s) \leq c_s - (1+\epsilon)^{-(k+1)}$  for all sets  $s \in \mathcal{S}'$  containing  $e$ . Thus, even after Step 4, we continue to have  $W^*(s) \leq c_s$  for all sets  $s \in \mathcal{S}'$ .

**Defining the collection of tight sets  $\mathcal{S}'' \subseteq \mathcal{S}'$ :** At this point in time, let  $\mathcal{S}'' = \{s \in \mathcal{S}' : (1 + \epsilon)^{-1}c_s \leq W^*(s) \leq c_s\}$  denote the collection of sets in  $\mathcal{S}'$  whose weights lie between  $(1 + \epsilon)^{-1}c_s$  and  $c_s$ . The remark above implies that every remaining set  $s \in \mathcal{S}' \setminus \mathcal{S}''$  has weight  $W^*(s) < (1 + \epsilon)^{-1}c_s$  at this point in time. We now prove two important claims.

**Claim 5.1.** *Just after the end of Step 4, each element  $e \in \mathcal{P}_{\leq k}^* \cap P_{k+1}$  belongs to at least one set  $s \in \mathcal{S}''$  and has weight  $w(e) \leq (1 + \epsilon)^{-(k+1)}$  (so that it continues to satisfy Invariant 3.1).*

*Proof.* Consider any element  $e \in \mathcal{P}_{\leq k}^* \cap P_{k+1}$ . This means that the element  $e$  was processed under case (b) in Step 4, because if it were processed under case (a) then it would no longer be part of  $P_{k+1}$  at the end of Step 4. Accordingly, recall what we do with such an element  $e$  under case (b) in Step 4. When we assign a weight  $\lambda$  to the element  $e$ , at least one set  $s' \in \mathcal{S}'_e$  gets a weight  $W^*(s') = c_{s'}$ , and that set  $s'$  becomes part of  $\mathcal{S}''$ .

Next, note that if an element  $e \in \mathcal{P}_{\leq k}^*$  was processed under case (b) in Step 4, then it receives weight  $w(e) \leq \lambda \leq (1 + \epsilon)^{-(k+1)}$ . Since such an element gets added to the set  $P_{k+1}$  in Step 5, it continues to satisfy Invariant 3.1.  $\square$

**Claim 5.2.** *Just after the end of Step 4, each element  $e \in \mathcal{P}_{\leq k}^* \setminus P_{k+1}$  belongs to  $A_{k+1}$  and has weight  $(1 + \epsilon)^{-(k+1)}$  (so that it continues to satisfy Invariant 3.1).*

*Proof.* Consider any element  $e \in \mathcal{P}_{\leq k}^* \cap P_{k+1}$ . Such an element  $e$  was processed under case (a) in Step 4. The claim follows from the description of that case.  $\square$

**Step 5:** Move each set  $s \in \mathcal{S}' \setminus \mathcal{S}''$  down to level  $k$ , by setting  $\ell(s) \leftarrow k$  for all  $s \in \mathcal{S}' \setminus \mathcal{S}''$ . Since the level of an element is defined to be the maximum level among all the sets it belongs to, some elements also move down to level  $k$  along with the sets in  $\mathcal{S}' \setminus \mathcal{S}''$ . Let  $X \subseteq E \cup D$  denote the subset of precisely those elements that move down to level  $k$ . Observation 5.1, Claim 5.1 and Claim 5.2 imply that  $X \subseteq \mathcal{A}_{\leq k}^* \cup (\mathcal{P}_{\leq k}^* \setminus P_{k+1}) = X \subseteq \mathcal{A}_{\leq k}^* \cup (\mathcal{P}_{\leq k}^* \cap A_{k+1})$ . In other words, if an element  $e$  moves down to level  $k$  during this step, then it must be the case that: (a)  $e$  is active right now, and (b)  $e \in \mathcal{A}_{\leq k}^* \cup \mathcal{P}_{\leq k}^*$ . Each element  $e \in X$  becomes part of  $A_k$ . To ensure that Invariant 3.1 holds, we set  $w(e) \leftarrow (1 + \epsilon)^{-k}$  for all  $e \in X$ , and accordingly modify the weights  $W^*(s)$  of the concerned sets in  $\mathcal{S}' \setminus \mathcal{S}''$ .

**Remark:** Consider a set  $s \in \mathcal{S}' \setminus \mathcal{S}''$  that moved down to level  $k$  in Step 5. It follows that at the end of Step 4, we had  $W^*(s) < (1 + \epsilon)^{-1}c_s$ . During Step 5, we change the weights of some elements contained in  $s$  from  $(1 + \epsilon)^{-(k+1)}$  to  $(1 + \epsilon)^{-k}$ . This increases its weight  $W^*(s)$  by at most a multiplicative factor of  $(1 + \epsilon)$ . Thus, we continue to have  $0 \leq W^*(s) \leq c_s$  at the end of Step 5 for all sets  $s \in \mathcal{S}' \setminus \mathcal{S}''$ .

**Summary of the situation after Step 5:** To summarize, at the end of Step 5 we end up with the following situation. (a) Invariants 3.1, 3.2, 3.3 hold for every element and set at level  $\geq k + 1$ . (b) All the remaining, relevant sets  $s \in \mathcal{S}' \setminus \mathcal{S}''$  are at level  $k$  with weight  $0 \leq W^*(s) \leq c_s$ . All the elements  $e \in \mathcal{E}_{\leq k}$  are at level  $k$ , with weight  $w(e) = (1 + \epsilon)^{-k}$ . (c) There is no passive or dead element at level  $\leq k$ , that is, we have  $P_{\leq k} = D_{\leq k} = \emptyset$ . (d) Finally, until this point the total time spent in the call to REBUILD( $\leq k$ ) is proportional to  $f$  times the number of elements in  $\mathcal{E}_{\leq k} \cup D_{\leq k}$  in the beginning of the call (see Property 3.5). It now remains to fix the hierarchical partition at levels  $\leq k$ , by calling a subroutine that is very similar to the static algorithm from Appendix A.

**Step 6:** Call the subroutine FIX-LEVEL( $k, \mathcal{S}' \setminus \mathcal{S}''$ ,  $\mathcal{E}_{\leq k}$ ). See Section 5.2 for the details.

**Step 7:** For all levels  $j \leq k$ , set  $C_{\leq j} = \epsilon \cdot |\mathcal{E}_{\leq j}|$  (see Property 3.5).

**Step 8:** RETURN.

## 5.1 Justifying Properties 3.4, 3.5 and 3.6

*Proof sketch for Property 3.4:* Claim 5.1, Claim 5.2 and the descriptions of Step 3 and Step 5 demonstrate that Invariant 3.1 holds at the end of Step 5. Further, it is easy to check that Invariant 3.2 and Invariant 3.3 hold for all the elements and sets at level  $\geq k + 1$  at the end of Step 5. Now, as explained in Section 5.2, the subroutine called in Step 6 simply gives a fast implementation of the static algorithm (see the Appendix) from level  $k$  downward. This static algorithm moves all the slack sets down to level 0 (see Claim A.3 in the Appendix). This ensures that at the end of Step 6 all three invariants are satisfied.

*Proof sketch for Property 3.5:* The time taken to implement Steps 1 – 5 is clearly proportional to  $f$  times the size of the set  $\mathcal{E}_{\leq k} \cup D_{\leq k}$  in the beginning of the call to  $\text{REBUILD}(\leq k)$ . The additive  $O(\log(Cn)/\epsilon)$  term comes from the runtime analysis of the subroutine called in Step 6 (see Section 5.2). Finally, Step 7 ensures that  $C_{\leq j} = \epsilon \cdot |\mathcal{E}_{\leq j}|$  for all  $j \in [0, k]$  at the end of the call to  $\text{REBUILD}(\leq k)$ .

*Proof sketch for Property 3.6-(1):* Follows from Step 2.

*Proof sketch for Property 3.6-(2):* Claim 5.1 and Claim 5.2 imply the following: Consider any element  $e$  that belonged to  $P_{\leq k}$  just before the call to  $\text{REBUILD}(\leq k)$ . At the end of Step 4, either (a) the element  $e$  is active or (b) the element  $e$  is passive and it belongs to at least one set in  $\mathcal{S}''$ . Under case (a), it is easy to check that the element  $e$  continues to remain active throughout the remainder of the call to  $\text{REBUILD}(\leq k)$ . Under case (b), Step 5 ensures that the element  $e$  (along with the set in  $\mathcal{S}''$  it belongs to) remains at level  $(k + 1)$  throughout the remainder of the call to  $\text{REBUILD}(\leq k)$ .

*Proof sketch for Property 3.6-(3):* Note that every element  $e \in \mathcal{A}_{\leq k}^*$  continues to remain in  $A_{\leq k+1}$  at the end of Step 5. This is because Steps 1 – 5 do *not* ever change the state of an element from active to passive, Step 3 only moves some elements from level  $\leq k$  to level  $k + 1$ , and Step 5 again moves some of these elements back to level  $k$ . Finally, the subroutine in Step 6 never changes the state of an element from active to passive and never changes the level of any element at level  $\geq k + 1$  (see Section 5.2).

*Proof sketch for Property 3.6-(4):* None of the Steps 1 – 5 affects any element that was at level  $\geq k + 1$  just before the call to  $\text{REBUILD}(\leq k)$ . The same holds true for the subroutine in Step 6 (see Section 5.2).

## 5.2 The subroutine $\text{FIX-LEVEL}(k, \mathcal{S}', \mathcal{E}')$

The most natural way to think about this subroutine is as follows. Suppose that we are executing the algorithm described in Appendix A, and *so far we have fixed everything above level  $k$* . Let  $\mathcal{S}'$  and  $\mathcal{E}'$  be the remaining sets and elements whose levels are still undecided, meaning that every set  $s \in \mathcal{S} \setminus \mathcal{S}'$  and every element  $e \in \mathcal{E} \setminus \mathcal{E}'$  have already been assigned to some levels  $> k$ . **Throughout the rest of this section, we assume that  $|\mathcal{S}'| = m'$  and  $|\mathcal{E}'| = n'$ .** At the present moment, all the elements in  $\mathcal{E}'$  have weight  $(1 + \epsilon)^{-k}$ , because we know for sure that these elements will eventually get assigned to some level  $\leq k$ . Furthermore, at the present moment every set  $s \in \mathcal{S}'$  has weight  $W^*(s) \leq c_s$ .<sup>10</sup> In order to ensure Property 3.5, our task now is to construct the levels  $\leq k$  of the hierarchical partition in  $O(fn' + \log(Cn)/\epsilon)$  time. If we follow the exact procedure in Appendix A, then we will need  $O(k \cdot fn')$  time for performing this task, because in that procedure we need to pay  $O(fn')$  time per level and we have to construct  $k$  levels overall. Unfortunately, the resulting running time can be as large as  $\Omega(fn' \cdot \log(Cn)/\epsilon)$  because  $k$  can be as large as  $\Omega(L) = \Omega(\log(Cn)/\epsilon)$ . The main challenge, therefore, is to come up with a procedure for our task that is much faster than the one described in Appendix A.

Our new algorithm will maintain a partition of the collection of elements  $\mathcal{E}'$  into two subsets:  $\mathcal{E}'_{\text{frozen}} \subseteq \mathcal{E}'$  and  $\mathcal{E}'_{\text{alive}} = \mathcal{E}' \setminus \mathcal{E}'_{\text{frozen}}$ . Similarly, it will maintain a partition of the collection of sets  $\mathcal{S}'$  into two

<sup>10</sup>Note that in Appendix A we denoted the weight of a set  $s$  by  $W(s)$ , but here we are denoting its weight by  $W^*(s)$ . This is because here a part of the weight  $W^*(s)$  might be coming from the weight of dead elements at level  $> k$ .

subsets:  $\mathcal{S}'_{frozen} \subseteq \mathcal{S}'$  and  $\mathcal{S}'_{alive} = \mathcal{S}' \setminus \mathcal{S}'_{frozen}$ . In the very beginning, it will start by setting  $\mathcal{E}' = \mathcal{E}'_{alive}$ ,  $\mathcal{S}' = \mathcal{S}'_{alive}$  and  $\mathcal{E}'_{frozen} = \mathcal{S}'_{frozen} = \emptyset$ . Intuitively, the levels of all the sets in  $\mathcal{S}'$  and all the elements in  $\mathcal{E}'$  are undecided in the beginning. Whenever the algorithm decides the final level of an alive set (resp. element), it will make the set (resp. element) frozen from that point onward.

We next introduce the crucial notion of the *target level*  $\ell_T(s)$  of a set  $s \in \mathcal{S}'$ . It is defined as follows.

If  $s \cap \mathcal{E}'_{alive} = \emptyset$ , then  $\ell_T(s) = 0$ . Otherwise, we define  $\ell_T(s)$  to be the maximum level  $i \in [1, k]$  such that

$$W^*(s) + \left( (1 + \epsilon)^{-i} - (1 + \epsilon)^{-k} \right) \cdot |s \cap \mathcal{E}'_{alive}| \geq \frac{c_s}{(1 + \epsilon)}.$$

We now explain the intuition behind this definition. Fix any set  $s \in \mathcal{S}'_{alive}$ . Suppose that there is no other alive set that shares a common alive element with  $s$ , i.e.,  $s \cap s' \cap \mathcal{E}'_{alive} = \emptyset$  for all  $s' \in \mathcal{S}'_{alive} \setminus \{s\}$ . If this is the case, then a moment's thought will reveal that the set  $s$  will get assigned to the level  $\ell_T(s)$  at the end of the algorithm in Appendix A. This holds for the following reason: the set  $s$  will keep decreasing its level and the alive elements in  $s$  will keep increasing their weights in powers of  $(1 + \epsilon)$  until the weight of  $s$  (given by  $W^*(s)$  here) becomes larger than or equal to  $(1 + \epsilon)^{-1}c_s$ . Also, note that if we move an alive element down from level  $k$  to some level  $i \leq k$ , then its weight increases by  $(1 + \epsilon)^{-i} - (1 + \epsilon)^{-k}$ . Hence, by definition,  $\ell_T(s)$  is the maximum level  $i$  with the following property: If we move down the set  $s$  (along with all the alive elements contained in  $s$ ) to level  $i$ , then  $W^*(s)$  becomes  $\geq (1 + \epsilon)^{-1}c_s$ . So the set  $s$  will get assigned to level  $\ell_T(s)$  at the end of the algorithm in Appendix A.

Unfortunately, the above argument does not hold if the set  $s$  has some alive element  $e$  in common with some other alive set  $s'$ . This is because in the algorithm described in Appendix A, the set  $s'$  can get assigned to a level  $i' > \ell_T(s)$ . This creates some problem in the argument above, because there we assumed that we can move *all* the alive elements in  $s$  down to level  $\ell_T(s)$ . But now, when the set  $s'$  gets stuck at level  $i' > \ell_T(s)$ , it *enforces* that the element  $e \in s'$  also gets stuck at level  $i'$ . In other words, we cannot move the element  $e$  down all the way to level  $\ell_T(s)$ . So even after the set  $s$  moves down to level  $\ell_T(s)$ , its weight  $W^*(s)$  might still remain below the threshold  $(1 + \epsilon)^{-1}c_s$ .

Nevertheless, we can still salvage the situation because of the following fact. Suppose that  $s$  is an alive set with the maximum possible value of  $\ell_T(s)$ . Then the objection in the preceding paragraph does not apply to the set  $s$ .<sup>11</sup> Accordingly, we pick the alive set  $s$  with maximum possible value of  $\ell_T(s)$ , and then we move that set  $s$  down to level  $\ell_T(s)$ , along with all the alive elements contained in  $s$ . Then we move the set  $s$  from  $\mathcal{S}'_{alive}$  to  $\mathcal{S}'_{frozen}$  and also move all the “relevant” elements  $e \in s \cap \mathcal{E}'_{alive}$  from  $\mathcal{E}'_{alive}$  to  $\mathcal{E}'_{frozen}$ . For every relevant element  $e$ , we next visit all the alive sets  $s''$  that contain  $e$ , and accordingly update their target levels  $\ell_T(s'')$  in light of the facts that (a) the weight  $W^*(s'')$  has increased by  $(1 + \epsilon)^{-i} - (1 + \epsilon)^{-k}$  as the element  $e$  has moved down from level  $k$  to level  $i$ , and (b) the element  $e$  is no longer alive.

Now comes a very crucial observation. Suppose that we pick an alive set  $s$  that maximizes  $\ell_T(s)$  and assign the set  $s$  (and all the alive elements contained in  $s$ ) to level  $\ell_T(s)$ , as described in the previous paragraph. *This can only decrease the value of  $\ell_T(s'')$  for the other alive sets  $s''$ .* To see why this is the case, fix any other alive element  $s'' \neq s'$  and suppose that we had  $\ell_T(s'') = i''$  just before we decided to move the set  $s$ . At that point, we had  $i'' \leq \ell_T(s)$  by definition of  $s$ . Hence, only the following situation can occur as the set  $s$  moves down to level  $\ell_T(s)$ . Prior to this event, the set  $s''$  felt that its weight  $W^*(s'')$  will exceed the threshold  $(1 + \epsilon)^{-1}c_{s''}$  if it can move down to level  $i''$  (along with all the alive elements in  $s''$ ). But now, after the set  $s$  settles at level  $\ell_T(s)$ , the set  $s''$  realizes that it can no longer take some element  $e'' \in s''$  (that was alive just before we moved  $s$ ) all the way down to level  $i''$ , because  $e''$  also belonged to  $s$  and now it has become frozen at a higher level  $\ell_T(s) > i''$  with a smaller weight  $w(e'') < (1 + \epsilon)^{-i''}$ . Thus, at this point (after we have moved  $s$ ), if the set  $s''$  decides to go down to level  $i''$ , then its weight  $W^*(s)$  can only be less than what it would have been prior to the instant we moved  $s$ . In other words, while moving  $s$  down to level

<sup>11</sup>Because in the paragraph we crucially relied on the fact that  $i' > \ell_T(s)$ , which will not be the case if  $\ell_T(s) \geq \ell_T(s')$ .

$\ell_T(s)$ , we can only decrease the value of  $\ell_T(s'')$ , for the set  $s''$  will now need to go down to an even lower level in order to ensure that  $W^*(s)$  exceeds the threshold  $(1 + \epsilon)^{-1}c_{s''}$ .

This leads us to the following natural algorithm. **Consider an array**  $\Gamma[0, \dots, k]$ , **where**  $\Gamma[i] = \{s \in \mathcal{S}'_{alive} : \ell_T(s) = i\}$  **for each**  $i \in [0, k]$ . As far as data structures are concerned, we can store each entry  $\Gamma[i]$  of this array as a doubly linked list. Using appropriate pointers, we can insert/delete a given set  $s'$  in such a doubly linked list in  $O(1)$  time. Now, the algorithm proceeds in rounds  $i = k, \dots, 0$ . In the beginning of round  $i$ , we have  $\Gamma[j] = \emptyset$  for all  $j > i$ . During round  $i$ , we repeatedly keep pulling out a set  $s$  from  $\Gamma[i]$  (this is a set which currently maximizes  $\ell_T(s)$ ), move the set  $s$  – along with all the elements in  $\mathcal{E}'_{alive} \cap s$  – down to level  $\ell_T(s)$ , transfer the set  $s$  from  $\mathcal{S}'_{alive}$  to  $\mathcal{S}'_{frozen}$ , and also transfer all the elements  $e \in s \cap \mathcal{E}'_{alive}$  from  $\mathcal{E}'_{alive}$  to  $\mathcal{E}'_{frozen}$ . By the discussion above, these steps can only decrease the values of  $\ell_T(s'')$  of the other alive sets  $s''$ . Accordingly, we continue to satisfy the invariant that  $\Gamma[j] = \emptyset$  for all  $j > i$ . The current round ends when we have  $\Gamma[i] = \emptyset$ . At that point, we proceed with round  $(i - 1)$ .

From the above discussion, it becomes clear that this algorithm produces exactly the same output as the algorithm described in Appendix A. It now remains to bound the total runtime of this algorithm. The pseudocode of the algorithm appears in Figure 3.

```

01. INITIALIZE:  $\mathcal{S}'_{alive} \leftarrow \mathcal{S}'$ ,  $\mathcal{S}'_{frozen} \leftarrow \emptyset$ ,  $\mathcal{E}'_{alive} \leftarrow \mathcal{E}'$ ,  $\mathcal{E}'_{frozen} \leftarrow \emptyset$ .
02. Compute  $\ell_T(s)$  for every set  $s \in \mathcal{S}'_{alive}$ , and set up the array  $\Gamma[0, \dots, k]$  accordingly.
05. FOR  $i = k$  down to 0:
06.   WHILE  $\Gamma[i] \neq \emptyset$ :
07.     Pick any  $s \in \Gamma[i]$ .
08.      $\Gamma[i] \leftarrow \Gamma[i] \setminus \{s\}$ .
09.      $\ell(s) \leftarrow \ell_T(s)$ .
10.     Move the set  $s$  from  $\mathcal{S}'_{alive}$  to  $\mathcal{S}'_{frozen}$ .
11.     FOR every element  $e \in \mathcal{E}'_{alive} \cap s$ :
12.        $\ell(e) \leftarrow \ell_T(s)$ .
13.        $w(e) \leftarrow w(e) + (1 + \epsilon)^{-\ell_T(s)} - (1 + \epsilon)^{-k}$ . // We had  $w(e) = (1 + \epsilon)^{-k}$  before this step.
14.        $W^*(s) \leftarrow W^*(s) + (1 + \epsilon)^{-\ell_T(s)} - (1 + \epsilon)^{-k}$ .
15.       Move the element  $e$  from  $\mathcal{E}'_{alive}$  to  $\mathcal{E}'_{frozen}$ .
16.       FOR every set  $s' \in \mathcal{S}'_{alive}$  that contains  $e$ :
17.          $W^*(s') \leftarrow W^*(s') + (1 + \epsilon)^{-\ell_T(s)} - (1 + \epsilon)^{-k}$ .
18.         Update the value of  $\ell_T(s')$  and the array  $\Gamma[0, \dots, k]$  accordingly.

```

Figure 3: The subroutine  $\text{FIX-LEVEL}(k, \mathcal{S}', \mathcal{E}')$ .

**Runtime analysis:** Since each element is contained in at most  $f$  sets, steps 12 – 18 in Figure 3 can be implemented in  $O(f)$  time (note that step 18 can be implemented in  $O(1)$  time). Furthermore, while executing steps 12 – 18 on a given element  $e$ , we move the element from  $\mathcal{E}'_{alive}$  to  $\mathcal{E}'_{frozen}$ . Hence, steps 12 – 18 get executed on a given element  $e$  at most once. Summing over all the  $n'$  elements in  $\mathcal{E}'$ , the subroutine in Figure 3 spends at most  $O(f \cdot n')$  time on steps 11 – 18.

Next, note that while executing steps 07 – 10 on a given set  $s$ , the subroutine in Figure 3 moves the set from  $\mathcal{S}'_{alive}$  to  $\mathcal{S}'_{frozen}$ . Accordingly, steps 07 – 10 get executed on a given set  $s$  at most once. Since every such execution of steps 07 – 10 takes  $O(1)$  time and since there are  $m'$  sets in  $\mathcal{S}'$ , we conclude that the subroutine overall spends  $O(m')$  time on steps 06 – 10.

Finally, the subroutine spends  $O(k) \leq O(L) = O(\log(Cn)/\epsilon)$  time on step 05. Since  $m' \leq fn'$ , the total time required by the subroutine in Figure 3 is at most  $O(fn' + m' + \log(Cn)/\epsilon) = O(fn' + \log(Cn)/\epsilon)$ .

## Acknowledgement

We thank Xiaowei Wu for spotting an error in an earlier version of the paper.

The research leading to these results has received funding from the European Research Council under the European Union’s Seventh Framework Programme (FP/2007-2013) / ERC Grant Agreement no. 340506.

This project has received funding from the European Research Council (ERC) under the European Union’s Horizon 2020 research and innovation programme under grant agreement No 715672. Nanongkai was also supported by the Swedish Research Council (Reg. No. 2015-04659).

## References

- [1] Amir Abboud, Raghavendra Addanki, Fabrizio Grandoni, Debmalya Panigrahi, and Barna Saha. Dynamic set cover: Improved algorithms & lower bounds. In *STOC*. ACM, 2019. (cit. on p. 1, 2, 3, 5)
- [2] Amir Abboud and Virginia Vassilevska Williams. Popular conjectures imply strong lower bounds for dynamic problems. In *FOCS*, pages 434–443. IEEE Computer Society, 2014. (cit. on p. 1)
- [3] Surender Baswana, Manoj Gupta, and Sandeep Sen. Fully dynamic maximal matching in  $o(\log n)$  update time (corrected version). *SIAM J. Comput.*, 47(3):617–650, 2018. announced at FOCS’ 11. (cit. on p. 1, 3)
- [4] Aaron Bernstein and Shiri Chechik. Deterministic decremental single source shortest paths: beyond the  $o(mn)$  bound. In *STOC*, pages 389–397. ACM, 2016. (cit. on p. 2, 5)
- [5] Aaron Bernstein and Shiri Chechik. Deterministic partially dynamic single source shortest paths for sparse graphs. In *SODA*, pages 453–469. SIAM, 2017. (cit. on p. 2, 5)
- [6] Aaron Bernstein, Sebastian Forster, and Monika Henzinger. A deamortization approach for dynamic spanner and dynamic maximal matching. In *SODA*, pages 1899–1918. SIAM, 2019. (cit. on p. 1)
- [7] Aaron Bernstein and Cliff Stein. Fully dynamic matching in bipartite graphs. In *ICALP (1)*, volume 9134 of *Lecture Notes in Computer Science*, pages 167–179. Springer, 2015. (cit. on p. 1)
- [8] Aaron Bernstein and Cliff Stein. Faster fully dynamic matchings with small approximation ratios. In *SODA*, pages 692–711. SIAM, 2016. (cit. on p. 1)
- [9] Sayan Bhattacharya, Deeparnab Chakrabarty, and Monika Henzinger. Deterministic fully dynamic approximate vertex cover and fractional matching in  $O(1)$  amortized update time. In *IPCO*, volume 10328 of *Lecture Notes in Computer Science*, pages 86–98. Springer, 2017. (cit. on p. 1, 2, 5)
- [10] Sayan Bhattacharya, Monika Henzinger, and Giuseppe F. Italiano. Deterministic fully dynamic data structures for vertex cover and matching. *SIAM J. Comput.*, 47(3):859–887, 2018. announced at SODA’ 15. (cit. on p. 1, 2)
- [11] Sayan Bhattacharya, Monika Henzinger, and Giuseppe F. Italiano. Dynamic algorithms via the primal-dual method. *Inf. Comput.*, 261(Part):219–239, 2018. announced at ICALP’ 15. (cit. on p. 2, 3, 5)
- [12] Sayan Bhattacharya, Monika Henzinger, and Danupon Nanongkai. New deterministic approximation algorithms for fully dynamic matching. In *STOC*, pages 398–411. ACM, 2016. (cit. on p. 1, 2)

- [13] Sayan Bhattacharya, Monika Henzinger, and Danupon Nanongkai. Fully dynamic approximate maximum matching and minimum vertex cover in  $O(\log^3 n)$  worst case update time. In *SODA*, pages 470–489. SIAM, 2017. (cit. on p. 1)
- [14] Julia Chuzhoy and Sanjeev Khanna. A new algorithm for decremental single-source shortest paths with applications to vertex-capacitated flow and cut problems. In *STOC*. ACM, 2019. (cit. on p. 2, 5)
- [15] Vasek Chvátal. A greedy heuristic for the set-covering problem. *Math. Oper. Res.*, 4(3):233–235, 1979. (cit. on p. 1)
- [16] Irit Dinur, Venkatesan Guruswami, Subhash Khot, and Oded Regev. A new multilayered PCP and the hardness of hypergraph vertex cover. *SIAM J. Comput.*, 34(5):1129–1146, 2005. announced at STOC’03. (cit. on p. 1)
- [17] Irit Dinur and David Steurer. Analytical approach to parallel repetition. In *STOC*, pages 624–633. ACM, 2014. (cit. on p. 1)
- [18] Fabrizio Grandoni, Stefano Leonardi, Piotr Sankowski, Chris Schwiegelshohn, and Shay Solomon.  $(1 + \epsilon)$ -approximate incremental matching in constant deterministic amortized time. In *SODA*, pages 1886–1898. SIAM, 2019. (cit. on p. 1)
- [19] Anupam Gupta, Ravishankar Krishnaswamy, Amit Kumar, and Debmalya Panigrahi. Online and dynamic algorithms for set cover. In *STOC*, pages 537–550. ACM, 2017. (cit. on p. 1, 2, 5)
- [20] Manoj Gupta and Richard Peng. Fully dynamic  $(1 + \epsilon)$ -approximate matchings. In *FOCS*, pages 548–557. IEEE Computer Society, 2013. (cit. on p. 1)
- [21] Monika Henzinger, Sebastian Krinninger, and Danupon Nanongkai. Decremental single-source shortest paths on undirected graphs in near-linear total update time. In *FOCS*, pages 146–155. IEEE Computer Society, 2014. (cit. on p. 5)
- [22] Monika Henzinger, Sebastian Krinninger, and Danupon Nanongkai. Sublinear-time decremental algorithms for single-source reachability and shortest paths on directed graphs. In *STOC*, pages 674–683. ACM, 2014. (cit. on p. 5)
- [23] Monika Henzinger, Sebastian Krinninger, and Danupon Nanongkai. Dynamic approximate all-pairs shortest paths: Breaking the  $o(mn)$  barrier and derandomization. *SIAM J. Comput.*, 45(3):947–1006, 2016. announced at FOCS’13. (cit. on p. 2)
- [24] Monika Henzinger, Sebastian Krinninger, and Danupon Nanongkai. Decremental single-source shortest paths on undirected graphs in near-linear total update time. *J. ACM*, 65(6):36:1–36:40, 2018. (cit. on p. 5)
- [25] Monika Henzinger, Sebastian Krinninger, Danupon Nanongkai, and Thatchaphol Saranurak. Unifying and strengthening hardness for dynamic problems via the online matrix-vector multiplication conjecture. In *STOC*, pages 21–30. ACM, 2015. (cit. on p. 1)
- [26] Subhash Khot and Oded Regev. Vertex cover might be hard to approximate to within  $2 - \epsilon$ . *J. Comput. Syst. Sci.*, 74(3):335–349, 2008. announced at CCC’03. (cit. on p. 1)
- [27] Danupon Nanongkai and Thatchaphol Saranurak. Dynamic spanning forest with worst-case update time: adaptive, las vegas, and  $o(n^{1/2 - \epsilon})$ -time. In *STOC*, pages 1122–1129. ACM, 2017. (cit. on p. 2)

- [28] Danupon Nanongkai, Thatchaphol Saranurak, and Christian Wulff-Nilsen. Dynamic minimum spanning forest with subpolynomial worst-case update time. In *FOCS*, pages 950–961. IEEE Computer Society, 2017. (cit. on p. 2)
- [29] Danupon Nanongkai, Thatchaphol Saranurak, and Sorrachai Yingchareonthawornchai. Breaking quadratic time for small vertex connectivity and an approximation scheme. In *STOC*. ACM, 2019. (cit. on p. 2)
- [30] Ofer Neiman and Shay Solomon. Simple deterministic algorithms for fully dynamic maximal matching. *ACM Trans. Algorithms*, 12(1):7:1–7:15, 2016. Announced at STOC’13. (cit. on p. 1)
- [31] Krzysztof Onak and Ronitt Rubinfeld. Maintaining a large matching and a small vertex cover. In *STOC*, pages 457–464. ACM, 2010. (cit. on p. 1)
- [32] David Peleg and Shay Solomon. Dynamic  $(1 + \epsilon)$ -approximate matchings: A density-sensitive approach. In *SODA*, pages 712–729. SIAM, 2016. (cit. on p. 1)
- [33] Piotr Sankowski. Faster dynamic matchings and vertex connectivity. In *SODA*, pages 118–126. SIAM, 2007. (cit. on p. 1)
- [34] Peter Slavík. A tight analysis of the greedy algorithm for set cover. *J. Algorithms*, 25(2):237–254, 1997. (cit. on p. 1)
- [35] Shay Solomon. Fully dynamic maximal matching in constant update time. In *IEEE 57th Annual Symposium on Foundations of Computer Science, FOCS*, pages 325–334, 2016. (cit. on p. 1, 3)
- [36] Jan van den Brand, Danupon Nanongkai, and Thatchaphol Saranurak. Dynamic matrix inverse: Improved algorithms and matching conditional lower bounds. In *FOCS*. IEEE Computer Society, 2019. (cit. on p. 1)

## A A Static Primal-Dual Algorithm for Minimum Set Cover

1. INITIALIZE:  $L = \lceil \log_{(1+\epsilon)}(C \cdot n) \rceil + 1$ ,  
 $w(e) \leftarrow (1 + \epsilon)^{-L}$  for all elements  $e \in \mathcal{E}$ .  
 $\ell(s) \leftarrow L$  for every set  $s \in \mathcal{S}$ .  
 $\ell(e) \leftarrow L$  for every element  $e \in \mathcal{E}$ .
2. FOR rounds  $t = L$  to 1:
3. Let  $\mathcal{S}_{slack}^{(t)} = \{s \in \mathcal{S} : W(s) < (1 + \epsilon)^{-1}c_s\}$  be the collection of sets that are *slack* in the beginning of round  $t$ , and let  $\mathcal{E}_{slack}^{(t)} = \{e \in \mathcal{E} : e \notin s \text{ for all } s \in \mathcal{S} \setminus \mathcal{S}_{slack}^{(t)}\}$  be the collection of elements that are *exclusively* covered by the sets in  $\mathcal{S}_{slack}^{(t)}$ .
4. FOR ALL sets  $s \in \mathcal{S}_{slack}^{(t)}$ :
5.  $\ell(s) = \ell(s) - 1$ .
6. FOR ALL elements  $e \in \mathcal{E}_{slack}^{(t)}$ :
7.  $w(e) = (1 + \epsilon) \cdot w(e)$ .
8.  $\ell(e) = \ell(e) - 1$ . // This ensures that  $\ell(e) = \max\{\ell(s) : s \in \mathcal{S} \text{ and } e \in s\}$ .

Figure 4: A static  $(1 + \epsilon)f$ -approximation algorithm for minimum set cover.

We use the same notations as in Section 2. The discretized primal-dual algorithm proceeds in  $L$  rounds (see Figure 4). In the beginning, we start by assigning a weight  $w(e) = (1 + \epsilon)^{-L} \leq (Cn)^{-1}$  to every element  $e \in \mathcal{E}$ , and a level  $\ell(s) = \ell(e) = L$  to every set  $s \in \mathcal{S}$  and every element  $e \in \mathcal{E}$ . Since each set in  $\mathcal{S}$  contains at most  $n$  elements, at this point in time we have  $W(s) = \sum_{e \in s} w(e) \leq n \cdot (Cn)^{-1} = 1/C$  for all sets  $s \in \mathcal{S}$ . Hence, from (1.1) we infer that  $0 \leq W(s) \leq c_s$  for all sets  $s \in \mathcal{S}$ . In other words, we have a valid fractional packing at this point in time. Throughout the rest of this section, we say that a set  $s \in \mathcal{S}$  is *tight* if  $(1 + \epsilon)^{-1}c_s \leq W(s) \leq c_s$ , and *slack* if  $0 \leq W(s) < (1 + \epsilon)^{-1}c_s$ .

In the beginning, we start with a counter  $t = L$ . The value of  $t$  keeps decreasing by one until we reach  $t = 0$ . Each value of  $t$  corresponds to a distinct *round* in the algorithm. In each round  $t$ , we increase (by a  $(1 + \epsilon)$  factor) the weights of the elements that are *exclusively* covered by the slack sets, and we decrease by one the levels of the concerned sets (that were slack in the beginning of the current round) and elements (whose weights got increased in the current round). Thus, there are  $L$  rounds overall, one for each  $t \in \{L, \dots, 1\}$ . At the end of round 1, we have a *hierarchical partition* of  $\mathcal{S}$  into  $L + 1$  levels  $\{L, \dots, 0\}$ . We now describe a few key properties that are satisfied at the end of the algorithm in Figure 4.

**Claim A.1.** *For every element  $e \in \mathcal{E}$ , we have  $\ell(e) = \max\{\ell(s) : s \in \mathcal{S}, e \in s\}$  and  $w(e) = (1 + \epsilon)^{-\ell(e)}$ .*

*Proof.* (Sketch) Fix any element  $e \in \mathcal{E}$ . Initially, every set and every element is assigned to level  $L$ . See step 1 in Figure 4. At this point, we clearly have  $\ell(e) = \max\{\ell(s) : s \in \mathcal{S}, e \in s\}$ . Subsequently, during a given round  $t$ , we decrement the level of  $e$  only if  $e \in \mathcal{E}_{slack}^{(t)}$ . See step 8 in Figure 4. Now, note that if  $e \in \mathcal{E}_{slack}^{(t)}$ , then by definition every set  $s \in \mathcal{S}$  containing  $e$  belongs to  $\mathcal{S}_{slack}^{(t)}$ . Furthermore, as is evident from step 5 in Figure 4, during round  $t$  we also decrement the level of every set  $s \in \mathcal{S}_{slack}^{(t)}$ . Thus, we always have:  $\ell(e) = \max\{\ell(s) : s \in \mathcal{S}, e \in s\}$ .

Next, note that as per step 1 in Figure 4, we initially have  $\ell(e) = L$  and  $w(e) = (1 + \epsilon)^{-L} = (1 + \epsilon)^{-\ell(e)}$ . Subsequently, whenever we increase  $w(e)$  by a factor of  $(1 + \epsilon)$ , we also decrement the level  $\ell(e)$  by one (see steps 7, 8 in Figure 4). Hence, it follows that  $w(e) = (1 + \epsilon)^{-\ell(e)}$  at the end of the algorithm.  $\square$

**Claim A.2.** *For every element  $e \in \mathcal{E}$ , at least one of the sets containing  $e$  lies at level  $\geq 1$  (i.e.,  $\ell(e) \geq 1$ ).*

*Proof.* (Sketch) Fix any element  $e \in \mathcal{E}$ . For the sake of contradiction, suppose that  $\ell(e) = 0$  at the end of the algorithm. This implies that  $e \in \mathcal{E}_{slack}^{(1)}$  in the beginning of round 1. This is because only the elements in  $\mathcal{E}_{slack}^{(1)}$  get moved down to level 0. Hence, as per the proof of Claim A.1, we have  $w(e) = (1 + \epsilon)^{-1}$  in the beginning of round 1. This means that every set  $s \in \mathcal{S}$  containing the element  $e$  has  $W(s) \geq w(e) \geq (1 + \epsilon)^{-1} \geq (1 + \epsilon)^{-1}c_s$  in the beginning of round 1. The last inequality holds since  $c_s \leq 1$  for all sets  $s \in \mathcal{S}$ , as per (1.1). Accordingly, no set containing  $e$  can be part of  $\mathcal{S}_{slack}^{(1)}$ . This leads to a contradiction, for we assumed that  $e \in \mathcal{E}_{slack}^{(1)}$ , which in turn means that  $e$  is exclusively covered by the sets from  $\mathcal{S}_{slack}^{(1)}$ .  $\square$

**Corollary A.1.** *We have  $\mathcal{E}_{slack}^{(1)} = \emptyset$ .*

*Proof.* Follows from the proof of Claim A.2.  $\square$

**Claim A.3.** *For every set  $s \in \mathcal{S}$  we have:*

$$W(s) \in \begin{cases} [(1 + \epsilon)^{-1}c_s, c_s] & \text{if } \ell(s) > 0; \\ [0, (1 + \epsilon)^{-1}c_s] & \text{else if } \ell(s) = 0. \end{cases}$$

*Proof.* (Sketch) Fix any set  $s \in \mathcal{S}$ . Initially, in the beginning of round  $L$ , we have  $W(s) = \sum_{e \in s} w(e) \leq |\mathcal{E}| \cdot (Cn)^{-1} = 1/C$ . From (1.1), we conclude that  $0 \leq W(s) \leq c_s$  at this point in time. Subsequently, in each round  $t \in \{L, \dots, 1\}$ , we keep moving down the set  $s$  to level  $t - 1$  iff we have  $W(s) < (1 + \epsilon)^{-1}c_s$  in

the beginning of the current round. Consider such a round  $t$  where the set  $s$  gets moved down to level  $t - 1$ . In the beginning of round  $t$ , we had  $W(s) < (1 + \epsilon)^{-1}c_s$ . During round  $t$ , we only increase (by a  $(1 + \epsilon)$  factor) the weights  $w(e)$  of some of the elements  $e$  contained in  $s$ . Thus, even at the end of round  $t$ , we have  $W(s) \leq c_s$ . This is sufficient for us to conclude that at the end of the algorithm, we have:

$$(a) W_s \leq c_s \text{ and } (b) W_s \in [(1 + \epsilon)^{-1}c_s, c_s] \text{ if } \ell(s) > 0.$$

It remains now to consider the case where  $\ell(s) = 0$  at the end of the algorithm. If this is the case, then we must have had  $s \in \mathcal{S}_{slack}^{(1)}$ , for only the sets in  $\mathcal{S}_{slack}^{(1)}$  gets demoted to level 0 during round 1. Accordingly, we infer that  $W_s < (1 + \epsilon)^{-1}c_s$  in the beginning of round 1. Now, by Corollary A.1, we have  $\mathcal{E}_{slack}^{(1)} = \emptyset$ . In other words, no element changes its weight during round 1. So the weight  $W(s)$  of the set  $s$  also remains unchanged during round 1. We accordingly infer that  $W(s) < (1 + \epsilon)^{-1}c_s$  at the end of the algorithm.  $\square$

Property 2.2 follows from Claim A.1. Property 2.3 follows from Claim A.3. Finally, Property 2.4 follows from Claim A.2 and Claim A.3.