Dynamic Set Cover: Improved Amortized and Worst-Case Update Time*

Sayan Bhattacharya† Monika Henzinger‡ Danupon Nanongkai§ Xiaowei Wu¶

Abstract

In the dynamic minimum set cover problem, a challenge is to minimize the update time while guaranteeing close to the optimal $\min(O(\log n), f)$ approximation factor. (Throughout, m, n, f, and C are parameters denoting the maximum number of sets, number of elements, frequency, and the cost range.) In the *high-frequency* range, when $f = \Omega(\log n)$, this was achieved by a deterministic $O(\log n)$-approximation algorithm with $O(f \log n)$ amortized update time [Gupta et al. STOC'17]. In the *low-frequency* range, the line of work by Gupta et al. [STOC'17], Abboud et al. [STOC'19], and Bhattacharya et al. [ICALP'15, IPCO'17, FOCS'19] led to a deterministic $(1+\epsilon)f$-approximation algorithm with $O(f \log(Cn)/\epsilon^2)$ amortized update time. In this paper we improve the latter update time and provide the first bounds that subsume (and sometimes improve) the state-of-the-art dynamic vertex cover algorithms. We obtain: (1) $(1+\epsilon)f$**-approximation ratio in** $O(f \log^2(Cn)/\epsilon^3)$ **worst-case update time:** No non-trivial worst-case update time was previously known for dynamic set cover. Our bound subsumes and improves by a logarithmic factor the $O(\log^3 n/\mathsf{poly}(\epsilon))$ worst-case update time for unweighted dynamic vertex cover (i.e., when $f = 2$ and $C = 1$) by Bhattacharya et al. [SODA'17]. (2) $(1+\epsilon)f$**-approximation ratio in** $O\left((f^2/\epsilon^3) + (f/\epsilon^2)\log C\right)$ **amortized update time:** This result improves the previous $O(f \log(Cn)/\epsilon^2)$ update time bound for most values of f in the low-frequency range, i.e. whenever $f = o(\log n)$. It is the first that is independent of m and n. It subsumes the constant amortized update time of Bhattacharya and Kulkarni [SODA'19] for unweighted dynamic vertex cover (i.e., when $f = 2$ and $C = 1$). These results are achieved by leveraging the approximate complementary slackness and background schedulers techniques. These techniques were used in the *local* update scheme for dynamic vertex cover. Our main technical contribution is to adapt these techniques within the *global* update scheme of Bhattacharya et al. [FOCS'19] for the dynamic set cover problem.

1 Introduction

In the *minimum set cover* problem, we get a universe of elements $\mathcal{U}$ and a collection of sets $\mathcal{S} \subseteq 2^{\mathcal{U}}$ as input, where $\bigcup_{s\in\mathcal{S}} s = \mathcal{U}$ and each set $s \in \mathcal{S}$ has a *cost* $c_s > 0$ associated with it. A collection of sets $\mathcal{S}' \subseteq \mathcal{S}$ forms a *set-cover* of $\mathcal{U}$ iff $\bigcup_{s\in\mathcal{S}'} s = \mathcal{U}$. The goal is to compute a set cover $\mathcal{S}'$ of $\mathcal{U}$ with minimum total cost $c(\mathcal{S}') = \sum_{s\in\mathcal{S}'} c_s$. This is one of the most fundamental problems in approximation algorithms. In recent years, this problem has also received significant attention in the *dynamic setting*, where the input keeps changing over time. Specifically, here we want to design a *dynamic algorithm* for minimum set cover that can handle the following operations:

Preprocessing: Initially, the algorithm receives as input a universe of elements $\mathcal{U}$, a collection of sets $\mathcal{S} \subseteq 2^{\mathcal{U}}$ with $\bigcup_{s\in\mathcal{S}} s = \mathcal{U}$, and a cost $c_s \geq 0$ for each set $s \in \mathcal{S}$.

Updates: Subsequently, the input keeps changing via a sequence of updates, where each update either (1) deletes an element e from the universe $\mathcal{U}$ and from every set $s \in \mathcal{S}$ that contains e, or (2) inserts an element e into the universe $\mathcal{U}$ and specifies the sets in $\mathcal{S}$ that the element e belongs to.

After each update, we would like to maintain an approximate cost of the optimal set cover of the updated set system. (Some algorithms also allow accessing a solution with such cost. See the remark after Theorem 1.1.) A dynamic algorithm has an *amortized update time* of $O(t)$ iff it takes $O((\alpha+\beta)\cdot t)$ total time (including the time spent on preprocessing) to handle any sequence of $\beta \geq 1$ updates, where α is the number of elements being preprocessed. We want to design a dynamic algorithm with small approximation ratio and update time. We get two main results:

THEOREM 1.1. *There are deterministic dynamic algorithms for the minimum set cover problem with $(1+\epsilon)f$-approximation ratio and*

1. *a worst-case update time of $O(f \log^2(Cn)/\epsilon^3)$, and*
2. *an amortized update time of $O(\frac{f^2}{\epsilon^3} + \frac{f}{\epsilon^2}\log C)$.*

Here, the symbol f denotes an upper bound on the maximum frequency of any element across all the updates[1], $C \geq 1$ is a parameter such that $1/C \leq c_s \leq 1$ for all sets $s \in \mathcal{S}$, m is the number of sets in $\mathcal{S}$, and n is the maximum number of elements in the universe $\mathcal{U}$ across all the updates.

*The full version of the paper can be found at https://arxiv.org/abs/2002.11171.

†Department of Computer Science, University of Warwick, UK. Email: S.Bhattacharya@warwick.ac.uk

‡Faculty of Computer Science, University of Vienna, Austria. Email: monika.henzinger@univie.ac.at

§KTH, Stockholm, Sweden. Email: danupon@kth.se

¶IOTSC, University of Macau, China. Part of the work was done while the author was a postdoc in the University of Vienna. Email: xiaoweiwu@um.edu.mo

[1]Frequency of an element $e \in \mathcal{U}$ is defined as the number of sets in $\mathcal{S}$ that contain e.

Remark: Both our algorithms maintain an $(1+\epsilon)f$-approximation to the *cost* of the minimum set cover after every update and can return this value in constant time. In addition, the algorithm for amortized update time maintains a *solution* of such value throughout the updates (i.e. it outputs the change of the maintained solution after every update), while the algorithm for worst-case update time does not and instead outputs the whole solution in time linear to the solution size whenever the solution is asked for (similar to the dynamic matching algorithm in [4]).

Perspective: The minimum set cover problem is very well understood in the static setting. There is a simple primal-dual algorithm that gives an f-approximation in $\Theta(fn)$ time, whereas a simple greedy algorithm gives a $\Theta(\log n)$-approximation in $\Theta(fn)$ time. Furthermore, there are strong inapproximability results which imply that the approximation guarantees achieved by these simple primal-dual and greedy algorithms are essentially the best possible [18, 17, 21].

Reference	Approximation	Update Time
[19]	$O(\log n)$	$O(f \log n)$
[19, 6]	$O(f^3)$	$O(f^2)$
[7]	$O(f^2)$	$O(f \log(m+n))$
[1]	$(1+\epsilon)f$	$O\left(\frac{f^2}{\epsilon} \log n\right)$
[11]	$(1+\epsilon)f$	$O\left(\frac{f}{\epsilon^2} \log(Cn)\right)$
Amortized	$(1+\epsilon)f$	$O\left(\frac{f^2}{\epsilon^3} + \frac{f}{\epsilon^2} \log C\right)$
Worst case	$(1+\epsilon)f$	$O(f \log^2(Cn)/\epsilon^3)$

Table 1: Summary of previous results (first five rows) and our results (last two rows) on dynamic set cover. All the previous update times are amortized. Except for the result of [1] (which is randomized and unweighted), all other results are deterministic and weighted (different sets have different costs).

In the dynamic setting, an important challenge is to match the approximation ratio of the (static) greedy and primal-dual algorithms, while minimizing the update time. In recent years, a series of papers on dynamic algorithms have been devoted to this topic. See Table 1 for a concise description of the results obtained in these papers. To summarize, we currently know how to get a $\Theta(\log n)$-approximation in $O(f \log n)$ update time, and a $(1+\epsilon)f$-approximation in $O(f \log(Cn)/\epsilon^2)$ update time. In addition, there is a strong conditional lower bound [1] which states that any dynamic set cover algorithm with nontrivial approximation ratio must have an update time of $\Omega(f^{1-\delta})$, for any constant $\delta > 0$. This explains the $O(\mathsf{poly}(f))$ factor inherent in all the update time bounds of Table 1, but leaves open the following question.

(Question 1) Must we necessarily incur a $\mathsf{polylog}(m,n)$ factor in the update time if we want to aim for near-optimal approximation ratio?

The above question falls within the study of *constant update time* (see below). Besides helping us understand the best possible update time for a dynamic problem to its limit, this study is useful in ruling out non-trivial cell-probe lower bounds [28, 22, 23]. Another important line of work in dynamic graph algorithms is achieving *worst-case* update time. All previous dynamic set cover algorithms can guarantee only *amortized* update time, leaving it widely open the following.

(Question 2): Is there a dynamic algorithm with non-trivial worst-case update time?

When $f = 2$ and $C = 1$, the above questions are equivalent to asking whether there are 2-approximation algorithms for *dynamic (unweighted) vertex cover* with (i) constant update time and (ii) non-trivial worst-case update time. There exists a long line of work on this dynamic (unweighted) vertex cover problem [27, 3, 20, 26, 8, 29, 30, 9, 13]. Currently, the state of the art results on this problem are as follows.

- The deterministic algorithm of [13] achieves $(2+\epsilon)$-approximation in $O(1/\epsilon^2)$ amortized update time for unweighted vertex cover, and the randomized algorithm of [30] achieves 2-approximation in $O(1)$ amortized update time for unweighted vertex cover.
- The deterministic algorithm of [10] achieves $(2+\epsilon)$-approximation in $O(\log^3 n/\mathsf{poly}(\epsilon))$ worst-case update time for unweighted vertex cover (also see [15, 2, 5]).

Our $O(f \log^2(Cn))$ worst-case bound in Theorem 1.1, when restricted to unweighted vertex cover, improves the $O(\log^3 n)$ bound of [10] by a logarithmic factor. Moreover, ours is the first non-trivial worst-case update time that holds for $f > 2$. On the other hand, our amortized bound in Theorem 1.1 is the first generalization of the vertex cover results from [30, 13]: When $f > 2$ and $C > 1$, a possible generalization of the constant amortized update time obtained in [13, 30] is the one guaranteeing $(1+\epsilon)f$-approximation ratio and $O(\mathsf{poly}(f, C))$ update time. The only previous result of this kind is the $O(f^2)$ update time achieved by [19, 6]; however this comes with a higher approximation ratio of $O(f^3)$. Our amortized update time is the first to achieve the target $O(\mathsf{poly}(f, C))$ bound *simultaneously* with a $(1+\epsilon)f$-approximation ratio.

Finally, note that our amortized update time improves the previous one in [11] in almost the whole range of parameters that we should be interested in: For a fixed $\epsilon > 0$, we get an update time of $O(f^2 + f \log C)$, whereas [11] obtained an update time of $O(f \log(Cn))$.

Note that in the *high-frequency range*, when $f = \omega(\log n)$, the $\Theta(\log n)$-approximation ratio obtained by [19] is already better than a $(1+\epsilon)f$-approximation. In other words, we are typically interested in getting an $(1+\epsilon)f$-approximation only in the *low-frequency range*, when $f = O(\log n)$. In this regime, our $O(f^2 + f \log C)$ update time strictly improves upon the update time of [11] for most values of f, i.e. whenever $f = o(\log n)$.

1.1 Techniques Both our results build on the recent algorithm of Bhattacharya, Henzinger and Nanongkai [11]. This algorithm and most previous deterministic algorithms for dynamic set cover and vertex cover (e.g. [13, 8, 9, 10]) are based on the following static primal-dual algorithm. (For ease of exposition, in this section we assume that $C = 1$; i.e., every set has the same cost.)

The static primal-dual algorithm assigns a fractional *weight* $w_e \geq 0$ to every element $e \in \mathcal{U}$, as follows. Initially, we set $w_e \leftarrow 0$ for all elements $e \in \mathcal{U}$ and $F \leftarrow \mathcal{U}$. Subsequently, the algorithm proceeds in rounds. In each round, we continuously raise the weights of all the elements in F until some set $s \in \mathcal{S}$ becomes *tight* (a set s becomes tight when its total weight $w_s = \sum_{e \in s} w_e$ becomes equal to 1). At this point, we delete the elements contained in the newly tight sets from F, and after that we proceed to the next round. The process stops when F becomes empty. At that point, we return the collection of tight sets as a valid set cover and the weights $\{w_e\}$ as the dual certificate. Specifically, it turns out that the weights $\{w_e\}$ returned at the end of the algorithm form a valid solution to the dual *fractional packing problem*, which asks us to assign a weight $w_e \geq 0$ to each element in $\mathcal{U}$ so as to maximize the objective $\sum_{e \in \mathcal{U}} w_e$, subject to the constraint that $\sum_{e \in s} w_e \leq 1$ for all sets $s \in \mathcal{S}$. We can also show that the collection of tight sets returned at the end of the static algorithm forms a valid set cover, whose cost is at most f times the cost of the dual objective $\sum_{e \in \mathcal{U}} w_e$. This leads to an approximation guarantee of f. In the dynamic setting, the main challenge now is to maintain the (approximate) output of the static algorithm described above in small update time. This is where [11] and previous deterministic algorithms use radically different approaches.

More specifically, previous deterministic algorithms (e.g. [13, 8, 9, 10]) follow some *local update rules* and maintain the approximate complementary slackness conditions. Thus, whenever the weight w_s of a tight set $s \in \mathcal{S}$ becomes too large (resp. too small) compared to 1, these algorithm decrease (resp. increase) the weights of some of the elements contained in s. This step affects the weights of some other sets that share these elements with s, and hence it might lead to a chain of cascading effects. Using very carefully chosen potential functions, these algorithms are able to bound these cascading effects over any sufficiently long sequence of updates. For technical reasons, however, this approach seems to work only when $f = 2$. Thus, although previous works could get constant and worst-case update time for maintaining a $(2+\epsilon)$-approximate vertex cover, it seems very difficult to extend their potential function analysis to the more general minimum set cover problem (or, equivalently, to minimum vertex cover on hypergraphs).

In contrast, [11] makes no attempt at maintaining the approximate complementary slackness conditions. It simply waits until the overall cost of the dual solution changes by a significant amount (compared to the cost of the set cover maintained by the algorithm). (This approach shares some similarities with the earlier randomized algorithm by [1], although [1] is not based on the static algorithm described above.) At that point, the algorithm identifies a critical collection of affected elements and recomputes their weights from scratch using a *global rebuilding* subroutine. The time taken for this recomputation step is, roughly speaking, proportional to the number of critically affected elements, which leads to a bound on the amortized update time. The strength of this framework is that this global rebuilding strategy extends seamlessly to the general set cover problem (where $f > 2$). Unfortunately this strategy incurs an additional $\Theta(\log n)$ factor in the update time that seems impossible to overcome.

Our algorithm with amortized update time results from carefully combining the these two sharply different approaches, namely the algorithm of [13] that uses some local update rules to obtain an $O(1/\epsilon^2)$ amortized update time, and the new approach of [11]. In our *hybrid* approach, whenever the weight w_s of a tight set s becomes too large compared to 1, we decrease the weights of some of the elements contained in s using the same local rule as in [13]. In contrast, whenever the weight w_s of a tight set s becomes too small compared to 1, we follow a lazy strategy and try to wait it out. After some period of time, when the total cost of the dual solution becomes significantly small compared to the size of the set cover maintained by the algorithm, we apply a *global rebuilding* subroutine as in [11] to fix the weights of some critical elements. This hybrid approach allows us to combine the best of both worlds, leading to a dynamic algorithm with $(1+\epsilon)f$-approximation ratio and an amortized $O(f^2/\epsilon^3)$ update time for any $f \geq 2$.

Our algorithm with worst-case update time extends the approach of [11] by having many *schedulers* working in parallel. This general idea has been used in many dynamic algorithms with worst-case update time (e.g.

[14, 20, 16, 25, 24, 31]). The main challenge is typically how to make the schedulers *consistent* in what they maintain, especially if they maintain an overlapping part of the solution. More specifically, we have $k = O(\log n)$ schedulers, where the i^{th} scheduler is associated with an integer r_i such that $r_1 \geq r_2 \geq \ldots \geq r_k$. The i^{th} scheduler is responsible for running the global rebuilding subroutine of [11] on sets that get tight *at and after round* r_i in the static algorithm described above. Thus, the sets that the i^{th} scheduler is responsible for are also under the responsibilities of the j^{th} schedulers for all $j > i$. A complication arises when these schedulers want to rebuild these sets at the same time, since it is not clear which solution of which scheduler we should use as a final solution. Typically, this can be resolved by forcing all schedulers to be *consistent*; i.e. the i^{th} and j^{th} schedulers agree on what happens to each set that they are both responsible for. This seems very hard to achieve in our case. At a high level, we get around this issue by requiring the schedulers to be only *loosely consistent*: Schedulers may maintain drastically different *local views* on the sets they are responsible for, except that there are some mild consistency conditions that tie their behaviors together. This way, each scheduler can work independently while our conditions guarantee that we can still combine results from the schedulers when needed. More specifically, we use the solution that the i^{th} scheduler maintains for level r_i. Due to the consistency conditions this results in an approximately minimum set cover.

2 A Static Primal-Dual Algorithm

Uniform-cost case: Recall the notations defined in the beginning of Section 1. In order to highlight the main ideas behind our algorithms, in this extended abstract we only consider the special case where every set has the same cost ($c_s = c_{s'}$ for all $s, s' \in \mathcal{S}$) and our goal is to compute a set cover in $(\mathcal{U}, \mathcal{S})$ of minimum size. For the full version of the paper, please refer to [12].

The dual: In the *maximum fractional packing* problem, we get a set system $(\mathcal{U}, \mathcal{S})$ as input. We have to assign a weight $w_e \in [0, 1]$ to every element $e \in \mathcal{U}$, subject to the constraint that $\sum_{e \in s} w_e \leq 1$ for all sets $s \in \mathcal{S}$. The goal is to maximize $\sum_{e \in \mathcal{U}} w_e$. We let $w_s = \sum_{e \in s} w_e$ denote the total weight received by a set $s \in \mathcal{S}$. LP-duality implies the following lemma.

LEMMA 2.1. *Consider any instance $(\mathcal{U}, \mathcal{S})$ of the set cover problem. Let* OPT *denote the size of the minimum set cover on this instance, and let $\{w_e\}_{e \in \mathcal{U}}$ denote any feasible fractional packing solution on the same input instance. Then we have $\sum_{e \in \mathcal{U}} w_e \leq$* OPT.

We now describe an f-approximation algorithm for minimum set cover in the static setting. The algorithm works as follows. There is a *time-variable* t that is initially set to $-\infty$, and it keeps increasing continuously throughout the duration of the algorithm. At every time t, the algorithm maintains a partition of the universe of elements $\mathcal{U}$ into two subsets: $\mathcal{A}(t) \subseteq \mathcal{U}$ and $\mathcal{F}(t) = \mathcal{U} \setminus \mathcal{A}(t)$. The elements in $\mathcal{A}(t)$ and $\mathcal{F}(t)$ are respectively called *alive* and *frozen* at time t. In the beginning, we have $\mathcal{A}(-\infty) = \mathcal{U}$ and $\mathcal{F}(-\infty) = \emptyset$. As t increases starting from $-\infty$, alive elements become frozen one after the other. Specifically, we have $\mathcal{A}(t) \supseteq \mathcal{A}(t')$ and $\mathcal{F}(t) \subseteq \mathcal{F}(t')$ for all $t \leq t'$. Let t_e be the time at which an element $e \in \mathcal{U}$ becomes frozen (i.e., moves from $\mathcal{A}$ to $\mathcal{F}$). We refer to t_e as the *freezing time* of e. Note that $t_e \leq t$ for all $e \in \mathcal{F}(t)$. At any time t, the weight of an element $e \in \mathcal{U}$ is determined as follows: If $e \in \mathcal{A}(t)$, then $w_e(t) = (1 + \epsilon)^t$. Otherwise, $e \in \mathcal{F}(t)$ and $w_e(t) = (1 + \epsilon)^{t_e}$.

Let $w_s(t) = \sum_{e \in s} w_e(t)$ denote the weight of a set $s \in \mathcal{S}$ at time t. We say that the set s is *tight* (resp. *slack*) at time t if $w_s(t) = 1$ (resp. $w_s(t) < 1$). Let $\mathcal{T}(t) \subseteq \mathcal{S}$ denote the collection of tight sets at time t. When $t = -\infty$, we have $w_e(t) = (1 + \epsilon)^t = 0$ for all elements $e \in \mathcal{A}(t) = \mathcal{U}$, and hence $w_s(t) = 0$ for all sets $s \in \mathcal{S}$. This implies that $\mathcal{T}(-\infty) = \emptyset$. Now, the following invariant completes the description of the algorithm: At any time t, we have $\mathcal{F}(t) = \bigcup_{s \in \mathcal{T}(t)} s$.

To summarize, the algorithm starts at time $t = -\infty$. At that point every element is alive and has weight 0, and all the sets are slack with weight 0. As t starts increasing continuously, the weights of the alive elements keep increasing according to the equation $w_e(t) = (1+\epsilon)^t$. Whenever a set s becomes tight during this process, every alive element $e \in s$ becomes frozen at the same time-instant, which ensures that the weights of all the elements $e \in s$ (and that of the set s) do not increase any further as the value of t keeps increasing. The process stops at time $t = 0$. Note that at time $t = 0$, if an element e is alive, then all sets containing e must be tight. This means that any element e has freezing time $t_e \leq 0$, which leads to the following claim.

CLAIM 2.1. *We have $\mathcal{A}(0) = \emptyset$ and $\mathcal{F}(0) = \mathcal{U}$.*

Levels of elements and sets: Claim 2.1 implies that the continuous process describing the static algorithm ends at time $t = 0$. At that point, every element $e \in \mathcal{U} = \mathcal{F}(0)$ has a well-defined freezing time $t_e \leq 0$. We define the *level* of an element $e \in \mathcal{U}$ to be $\ell(e) = -t_e$. The *level* of a set $s \in \mathcal{T}(0)$ is defined as $\ell(s) = -t_s$, where t_s is the time at which the set s became tight. The levels of the remaining sets $s \in \mathcal{S} \setminus \mathcal{T}(0)$ are defined to be $\ell(s) = 0$.

Henceforth, we use the symbol w_e to denote the

weight of an element e at time $t = 0$ (i.e., $w_e = w_e(0)$). Similarly, we use the symbol w_s to denote $w_s(0)$, and the symbol $\mathcal{T}$ to denote $\mathcal{T}(0)$. Finally, when we say that a set s is tight, we mean that it is tight at time $t = 0$.

PROPERTY 2.1. *We have* $\ell(e) = \max_{s \in \mathcal{S}: e \in s} \ell(s)$ *and* $w_e = (1+\epsilon)^{-\ell(e)}$ *for all elements* $e \in \mathcal{U}$.

PROPERTY 2.2. *We have* $w_s \leq 1$ *for all* $s \in \mathcal{S}$. *Further, every set* $s \in \mathcal{S}$ *at level* $\ell(s) > 0$ *is tight.*

LEMMA 2.2. *The weights* $\{w_e\}_{e \in \mathcal{U}}$ *form a fractional packing and the collection of sets* $\mathcal{T}$ *forms an* f*-approximate minimum set cover in* $(\mathcal{U}, \mathcal{S})$.

Proof. Since $w_s \leq 1$ for all $s \in \mathcal{S}$, the weights $\{w_e\}_{e \in \mathcal{U}}$ form a fractional packing. Consider any element $e \in \mathcal{U}$. If at least one set $s \in \mathcal{S}$ containing e lies at level $\ell(s) > 0$, then $s \in \mathcal{T}$ and hence the element e is covered by $\mathcal{T}$. Otherwise, every set $s \in \mathcal{S}$ containing the element e lies at level $\ell(s) = 0$. So Property 2.1 implies that $\ell(e) = 0$, and hence $w_e = (1+\epsilon)^{-0} = 1$. Thus, every set s containing e has weight $w_s \geq w_e = 1$. In other words, every set s containing e is tight, and so the element e is again covered by $\mathcal{T}$. This implies that $\mathcal{T}$ forms a valid set cover. Since each element e contributes to the weight w_s of only the (at most f) sets that contain it, we have $\sum_{e \in \mathcal{U}} f \cdot w_e \geq \sum_{s \in \mathcal{S}} w_s \geq \sum_{s \in \mathcal{T}} w_s = |\mathcal{T}|$. The last equality holds since $w_s = 1$ for all sets $s \in \mathcal{T}$. Now, Lemma 2.1 implies $f \cdot \mathsf{OPT} \geq f \cdot \sum_{e \in \mathcal{U}} w_e \geq |\mathcal{T}|$. ∎

3 Overview for the Amortized Update time

Preprocessing: We start by computing the solution returned by the static algorithm from Section 2. Let $\ell(s)$ be the level of a set $s \in \mathcal{S}$ in the output of this static algorithm. The level $\ell(e)$ and the weight w_e of every element $e \in \mathcal{U}$ are determined by the levels of the sets containing it, in accordance with Property 2.1. The weight of a set $s \in \mathcal{S}$ is defined as $w_s = \sum_{e \in s} w_e$. In addition, we associate a variable ϕ_s with every set $s \in \mathcal{S}$. The value of ϕ_s is called the *dead-weight* of s. In contrast, the value of w_s denotes the *real-weight* of s. The *total-weight* of a set $s \in \mathcal{S}$ is given by $w^*_s = w_s + \phi_s$. Just after preprocessing, we have $\phi_s = 0$ for all $s \in \mathcal{S}$, so that the total-weight of every set is equal to its dead-weight. Throughout Section 3, we will say that a set $s \in \mathcal{S}$ is *tight* if $w^*_s = 1$ and *slack* if $w^*_s < 1$. Accordingly, the total-weights $\{w^*_s\}_{s \in \mathcal{S}}$ satisfy Property 2.2 just after preprocessing. The significance of the notion of dead-weights will become clear shortly.

3.1 Handling deletions of elements When an element e gets deleted, the real-weight w_s of every set s containing e decreases by w_e. To compensate for this loss, we set $\phi_s \leftarrow \phi_s + w_e$ for all sets $s \in \mathcal{S}$ that contained e. Thus, the total-weight of every set remains unchanged due to an element-deletion. It should now be apparent that our algorithm satisfies Property 2.2 if we replace the real-weights w_s by the total-weights w^*_s.

INVARIANT 3.1. *We have* $w^*_s \leq 1$ *for all* $s \in \mathcal{S}$. *Further, every set* $s \in \mathcal{S}$ *at level* $\ell(s) > 0$ *is tight.*

As the elements keep getting deleted the size of minimum set cover keeps decreasing. But the set cover maintained by the algorithm we have described so far remains unchanged. Hence, after sufficiently many deletions, the approximation ratio of our algorithm will degrade by a significant amount. To address this concern, our algorithm *rebuilds* part of the solution once the sum of the dead-weights of the sets becomes too large. Specifically, we maintain the following invariant.

INVARIANT 3.2. *We have* $\sum_{s \in \mathcal{S}} \phi_s \leq \epsilon \cdot f \cdot \sum_{e \in \mathcal{U}} w_e$.

After preprocessing, the above invariant holds since $\phi_s = 0$ for all $s \in \mathcal{S}$. Subsequently, after handling each element-deletion in the manner described above, we perform the following operations.

- WHILE Invariant 3.2 is violated:
  - Identify the smallest level $k \geq 0$ such that $\sum_{s \in \mathcal{S}: \ell(s) \leq k} \phi_s > \epsilon \cdot f \cdot \sum_{e \in \mathcal{U}: \ell(e) \leq k} w_e$.
  - Call REBUILD($\leq k$) as described below.

The subroutine REBUILD($\leq k$): Let $\mathcal{S}'_k$ (resp. $\mathcal{U}'_k$) be the collection of sets $s \in \mathcal{S}$ at levels $\ell(s) \leq k$ (resp. the collection of elements $e \in \mathcal{U}$ at levels $\ell(e) \leq k$) just before the call to REBUILD($\leq k$). The subroutine works in two steps: Step I (clean-up) and Step II (rebuild). To simplify the analysis, we make the following crucial assumption in this extended abstract.

ASSUMPTION 3.1. *Every set* $s \in \mathcal{S}'_k$ *contains at least one element from* $\mathcal{U}'_k$.

Step I (clean-up): We set $\phi_s \leftarrow 0$, $\ell(s) \leftarrow k$ for all $s \in \mathcal{S}'_k$. This resets $\ell(e) \leftarrow k$ and $w_e \leftarrow (1+\epsilon)^{-k}$ for all $e \in \mathcal{U}'_k$, as per Property 2.1. The real-weights $\{w_s\}_{s \in \mathcal{S}'_k}$ get updated accordingly.

The clean-up step as described above can only decrease the weight w_e of an element $e \in \mathcal{U}'_k$, since it moves up from its earlier level (which was $\leq k$) to level k. Hence, the real-weights w_s of the sets $s \in \mathcal{S}'_k$ can also only decrease due to this step. Furthermore, since $\ell(e) = \max_{s \in \mathcal{S}: e \in s} \ell(s)$ for every element e (see Property 2.1), all the sets containing an element $e \in \mathcal{U}'_k$ belong to $\mathcal{S}'_k$. Accordingly, we do not change the real-weight w_s of any set $s \in \mathcal{S}$ at level $\ell(s) > k$ during the

clean-up step. Neither do we change the dead-weight ϕ_s of any set s or the level/weight of any element e at level $> k$. Since Invariant 3.1 was satisfied just before the clean-up step, we get:

OBSERVATION 3.1. *Just after the clean-up step, every set $s \in \mathcal{S}$ at level $\ell(s) > k$ is tight. All the remaining sets $s \in \mathcal{S}'_k$ are at level $\ell(s) = k$ with real-weights $w_s \leq 1$ and dead-weights $\phi_s = 0$.*

Step II (rebuild): Recall that the static algorithm from Section 2 starts at time $t = -\infty$ and stops when t becomes equal to 0. Observation 3.1 implies that after the clean-up step, we are in exactly the same state as the static algorithm from Section 2 at time $t = -k$ (provided we replace the real-weights w_s by the total-weights w^*_s for all sets s at level $\ell(s) > k$). At this point, we perform the remaining steps prescribed by the static algorithm from Section 2 as its time-variable moves from $t = -k$ to $t = 0$. We emphasize that while executing these remaining steps, we do not change the dead-weights of the sets in $\mathcal{S}'_k$ (these dead-weights were reset to zero during the clean-up step, and they continue to remain zero). This leads us to the following observation.

OBSERVATION 3.2. *At the end of the call to the subroutine* REBUILD($\leq k$)*, Invariant 3.1 is satisfied. Furthermore, we have $\phi_s = 0$ for all $s \in \mathcal{S}$ at levels $\ell(s) \leq k$.*

We note that using appropriate data structures this subroutine can be implemented efficiently.

LEMMA 3.1. *Under Assumption 3.1, the subroutine* REBUILD($\leq k$) *runs in $O(f \cdot |\mathcal{U}'_k|)$ time.*

3.2 Handling insertions of elements We handle the insertion of an element e' by calling the procedure in Figure 1, where $\mathcal{S}_{e'} = \{s \in \mathcal{S} : e' \in s\}$ denotes the collection of sets containing e'. From the outset, we often do not explicitly specify how the real-weight and dead-weight of a set s changes due to the execution of the procedure in Figure 1. Instead, they will be implicitly determined as: $w_s = \sum_{e \in s} w_e$ and $w^*_s = w_s + \phi_s$. Step (01) assigns the element e' a level and a weight in accordance with Property 2.1. This increases the real-weight and the total-weight of every set $s \in \mathcal{S}_{e'}$ by $w_{e'}$. So the sets in $\mathcal{S}_{e'}$ can now potentially violate Invariant 3.1. We say that a set s is *dirty* if it violates Invariant 3.1, and *clean* otherwise. Note that Observation 3.3 is satisfied at this juncture. The FOR loop in Step (02) takes care of these dirty sets. Before proceeding further, we need to define a few important notations.

01. Assign element e' a level $\ell(e') \leftarrow \max_{s \in \mathcal{S}_{e'}} \ell(s)$ and weight $w_{e'} \leftarrow (1+\epsilon)^{-\ell(e')}$.
02. FOR every $s \in \mathcal{S}_{e'}$: Call the subroutine FIX(s).
03. WHILE Invariant 3.2 is violated:
04. Identify the smallest level $k \geq 0$ such that $\sum_{s \in \mathcal{S}: \ell(s) \leq k} \phi_s > \epsilon \cdot f \cdot \sum_{e \in \mathcal{U}: \ell(e) \leq k} w_e$.
05. Call the subroutine REBUILD($\leq k$) as described in Section 3.1.

Figure 1: Handling the insertion of an element e'.

Notations: For any set $s \in \mathcal{S}$, we let $\mathcal{N}(s) = \{s' \in \mathcal{S} \setminus \{s\} : s \cap s' \neq \emptyset\}$ denote the *neighbors* of s. Next, fix any set $s \in \mathcal{S}$ and consider the following thought experiment. Suppose that we move the set s from its current level to some other level j, while keeping the levels of all the remaining sets $s' \in \mathcal{S} \setminus \{s\}$ unchanged. This potentially changes the levels and weights of some of the elements $e \in s$ in accordance with Property 2.1, and hence the real-weights $w_{s'}$ of some of the sets $s' \in \mathcal{N}(s)$ also get changed. Let $w_{s'}(s \to j)$ denote the resulting real-weight of a set s' after s has moved to level j. It is easy to check that $w_{s'}(s \to j)$ is a continuous non-increasing function of j for all $s', s \in \mathcal{S}$, and that $w_s(s \to \infty) = 0$ for all $s \in \mathcal{S}$. This leads us to the concept of the *target level* $\ell^*(s)$ of a set $s \in \mathcal{S}$ with real-weight $w_s > 1$: If a set $s \in \mathcal{S}$ has real-weight $w_s > 1$, then $\ell^*(s) = \min\{j : w_s(s \to j) = 1\}$. Note that $\ell^*(s) > \ell(s)$.

OBSERVATION 3.3. *A set s' is dirty only if $s' \in \mathcal{S}_{e'}$ and $w^*_{s'} > 1$.*

The subroutine FIX(s): By induction, suppose that Observation 3.3 holds at the start of a given call to FIX(s). The subroutine first checks if $w^*_s > 1$. If not, then Observation 3.3 implies that the set s is already clean and hence the subroutine finishes execution and returns the call. From now onward, we assume that $w^*_s = 1 + \mu_s$ for some $\mu_s > 0$ at the beginning of the call. If $\phi_s \geq \mu_s$, then we set $\phi_s \leftarrow \phi_s - \mu_s$. This makes the set s clean, and again the subroutine finishes execution. Hence, from now onward, we assume that $\mu_s > \phi_s$ at the beginning of the call. We first set $\phi_s \leftarrow 0$, in order to reduce the total-weight of s as much as possible. At this stage, we have $w^*_s = w_s > 1$ and $\phi_s = 0$. The subroutine now moves the set s up to its target-level $\ell^*(s) = j$ (say). This upward movement is achieved via a continuous process. Informally, as the set s keeps moving up, some of its neighbors $s' \in \mathcal{N}(s)$ keep losing their real-weights (because the weights of the some of the elements $e \in s' \cap s$ keep decreasing). In order to compensate for this loss, the affected neighbors

$s' \in \mathcal{N}(s)$ keep increasing their dead-weights $\phi_{s'}$ in a continuous manner, whenever possible.

To be more specific, consider an infinitesimal time-interval during this continuous process when the set s moves up from level λ to $\lambda + d\lambda$. As a result, some of the elements $e \in s$ have their weights decreased. This in turn change the real-weights $w_{s'}$ of some of the neighbors $s' \in \mathcal{N}(s)$ by $dw_{s'}(s \to \lambda)$. Note that $dw_{s'}(s \to \lambda) \leq 0$. If $w_{s'}(s \to \lambda) \leq 1$, then we set $\phi_{s'} \leftarrow \phi_{s'} - dw_{s'}(s \to \lambda)$, in order to compensate for the loss of real-weight of s' during this infinitesimally small time-interval.

Thus, from the perspective of a neighbor $s' \in \mathcal{N}(s)$, the process looks like this: As the set s keeps moving up, the real-weight of s' keep decreasing in a continuous manner, until $w_{s'}$ becomes ≤ 1. From this point onward, the dead-weight $\phi_{s'}$ keeps increasing at the same rate at which the real-weight $w_{s'}$ decreases (thereby keeping the total-weight $w^*_{s'}$ constant).

OBSERVATION 3.4. *A call to* FIX(s) *never leads to an already clean set becoming dirty. Furthermore, if Observation 3.3 holds in the beginning of the call, then it continues to hold at the end of the call.*

At the end of the FOR loop in Figure 1, every set is clean and hence Invariant 3.1 is satisfied. However, the dead-weights of some of the sets are increased due to the calls to FIX(s). This might lead to a violation of Invariant 3.2. This is addressed by the WHILE loop in steps (03)-(05). Observation 3.2 implies that both the invariants hold at the end of procedure in Figure 1.

3.3 Bounding the approximation ratio and amortized update time The following theorem upper bounds the approximation ratio of our dynamic algorithm.

THEOREM 3.1. *The collection of tight sets* $\mathcal{T}^* = \{s \in \mathcal{S} : w^*_s = 1\}$ *forms a* $(1+\epsilon)f$*-approximate minimum set cover in* $(\mathcal{U}, \mathcal{S})$.

Proof. Following the argument in the proof of Lemma 2.2, Invariant 3.1 implies that the collection of tight sets $\mathcal{T}^*$ forms a set cover in $(\mathcal{U}, \mathcal{S})$, and the element-weights $\{w_e\}_{e \in \mathcal{U}}$ form a fractional packing in $(\mathcal{U}, \mathcal{S})$. Next, as in the proof of Lemma 2.2, we first derive that $\sum_{e \in \mathcal{U}} f \cdot w_e \geq \sum_{s \in \mathcal{T}^*} w_s$. Adding the term $\sum_{s \in \mathcal{T}^*} \phi_s$ to both sides of this inequality, we get: $\sum_{e \in \mathcal{U}} f \cdot w_e + \sum_{s \in \mathcal{T}^*} \phi_s \geq \sum_{s \in \mathcal{T}^*} (w_s + \phi_s) = \sum_{s \in \mathcal{T}^*} w^*_s = |\mathcal{T}^*|$. Next, from Invariant 3.2 we derive that: $(1+\epsilon)f \cdot \sum_{e \in \mathcal{U}} w_e \geq \sum_{e \in \mathcal{U}} f \cdot w_e + \sum_{s \in \mathcal{T}^*} \phi_s \geq |\mathcal{T}^*|$. In other words, there is a fractional packing $\{w_e\}_{e \in \mathcal{U}}$ whose value is within a multiplicative $(1+\epsilon)f$ factor of the size of a valid set cover $\mathcal{T}^*$. Hence, $\mathcal{T}^*$ is a $(1+\epsilon)f$-approximate minimum set cover in $(\mathcal{U}, \mathcal{S})$ according to Lemma 2.1. ∎

We spend the rest of this section explaining the main ideas behind the analysis of the amortized update time of our algorithm. We start with an assumption that helps simplify this analysis.

ASSUMPTION 3.2. *Suppose that an element* e' *getting inserted is assigned to a level* $\ell(e') = j'$ *in step (01) of Figure 1. After that, the level of* e' *does not change during the* FOR *loop in step (02).*

The update time of our algorithm is dominated by two main types of operations: (1) an iteration of the FOR loop in Figure 1 where a set s potentially moves up to its target-level, and (2) a call to the subroutine REBUILD($\leq k$). For an operation of type (1), in this section we bound the *fractional work* done by our algorithm instead of the actual time taken to implement it. We give an intuitive justification as to why fractional work is a useful proxy for the actual running time that is analyzed in the full version. In order to bound the time spent on operations of type (2), we introduce the notion of *down-tokens*. We now explain each of these concepts in more details.

Fractional work: Consider an event where a set $s \in \mathcal{S}$ moves up from level j_0 to level j_1, and the level of every other set remains unchanged. This event can change the level of an element $e \in \mathcal{U}$ only if $e \in s$. For all $e \in s$, let $\ell_0(e)$ and $\ell_1(e)$ respectively denote the level of e just before and just after the event. Then the total *fractional work* done during this event $= \sum_{e \in s} f \cdot (\ell_1(e) - \ell_0(e))$.

Justification for fractional work: In the full version our starting point will be a *discretized variant* of the static algorithm from Section 2, where in each round the weights of of the alive elements increase by a multiplicative factor of $(1+\epsilon)$ and the level of every set and element is an integer in the range $\{0, 1, \ldots, \lceil \log_{(1+\epsilon)} n \rceil\}$. Using appropriate data structures, we can ensure that our algorithm spends $O(f)$ time for each element increasing its level by one unit. This precisely corresponds to the notion of fractional work defined above (when the levels are integers).

Down-tokens: We associate $(1+\epsilon)^{\ell(s)} \cdot \phi_s$ amount of *down-tokens* with each set $s \in \mathcal{S}$. The *total volume* of down-tokens is given by $\sum_{s \in \mathcal{S}} (1+\epsilon)^{\ell(s)} \cdot \phi_s$.

The parameter γ_ϵ: In the rest of this section, to ease notations we define $\gamma_\epsilon = (\ln(1+\epsilon))^{-1} = O(\frac{1}{\epsilon})$.

Overview of our analysis: By Lemma 3.3, the total fractional work done per update due to operations of type (1) is at most $O(f^2 \gamma_\epsilon) = O(f^2/\epsilon)$. It now remains

to bound the total time spent on operations of type (2). Towards this end, we make the following important observations: (a) Excluding the calls to REBUILD($\leq k$), the procedure for handling the insertion of an element increases the total volume of down-tokens by at most $O(f^2)$ (see Corollary 3.1). (b) Excluding the calls to REBUILD($\leq k$), the procedure for handling the deletion of an element increases the total-volume of down-tokens by at most $O(f)$. This holds because when an element e gets deleted, the dead-weight ϕ_s associated with each set s containing it increases by $w_e = (1+\epsilon)^{-\ell(e)}$ (see the first paragraph in Section 3.1). Hence, the total volume of down-tokens increases by $\sum_{s\in\mathcal{S}:e\in s}(1+\epsilon)^{\ell(s)} \cdot (1+\epsilon)^{-\ell(e)} \leq \sum_{s\in\mathcal{S}:e\in s}(1+\epsilon)^{\ell(e)} \cdot (1+\epsilon)^{-\ell(e)} \leq f$. (c) The total time spent on all the calls to REBUILD($\leq k$) is at most $O(1/\epsilon)$ times the decrease in the total volume of down-tokens (see Lemma 3.4). Since the total volume of down-tokens is always nonnegative, all these observations taken together imply an amortized update time of $O(f^2/\epsilon)$. This is slightly better than the bound in Theorem 1.1, because in the full version we do not have the luxury of analyzing *fractional work*.

LEMMA 3.2. *Consider a call to the subroutine* FIX(s) *in Figure 1, which moves the set s up to its target-level. Consider an infinitesimally small time-interval during this iteration when the set s moves up from level λ to level $\lambda + d\lambda$. During this infinitesimally small time-interval, (a) the fractional work done $= -f\gamma_\epsilon(1+\epsilon)^\lambda \cdot dw_s(s \to \lambda)$, and (b) the total volume of down-tokens increases by $\leq -f(1+\epsilon)^\lambda \cdot dw_s(s \to \lambda)$. Here, we have $dw_s(s \to \lambda) = w_s(s \to \lambda + d\lambda) - w_s(s \to \lambda)$.*

Proof. (Sketch) Consider the collection of elements $X_s(\lambda) = \{e \in \mathcal{U} : e \in s$, and $\ell(s') \leq \lambda$ for all $s' \in \mathcal{S}\setminus\{s\}$ with $e \in s'\}$. As the set s moves up from level λ to level $\lambda + d\lambda$, each element $e \in X_s(\lambda)$ changes its level by $d\lambda$ and no other element changes its level.[2] Hence the fractional work done during this interval $= f|X_s(\lambda)| \cdot d\lambda$.

When the set s is at level λ, each element $e \in X_s(\lambda)$ has weight $(1+\epsilon)^{-\lambda}$. Starting from λ, if we increase the level of s by an infinitesimal amount, then only the elements $e \in X_s(\lambda)$ change their weights, while the weights of every other element in s remains unchanged. Hence, we derive that:

$$\begin{aligned}\frac{dw_s(s\to\lambda)}{d\lambda} &= \frac{d}{d\lambda}\left(|X_s(\lambda)| \cdot (1+\epsilon)^{-\lambda}\right)\\ &= -\gamma_\epsilon^{-1} \cdot (1+\epsilon)^{-\lambda} \cdot |X_s(\lambda)|.\end{aligned}$$

Thus, we get: $|X_s(\lambda)| \cdot d\lambda = -\gamma_\epsilon \cdot (1+\epsilon)^\lambda \cdot dw_s(s \to \lambda)$.

[2]Since we consider an infinitesimally small interval and the collection $\{\ell(e) : e \in \mathcal{U}\}$ is finite, $X_s(\lambda) = X_s(\lambda + d\lambda)$.

Let dw be the change in the weight of an element $e \in X_s(\lambda)$ as the set s moves up from level λ to level $\lambda + d\lambda$. Note that $dw < 0$. From the preceding discussion, it follows that:

$$dw_s(s \to \lambda) = |X_s(\lambda)| \cdot dw. \tag{3.1}$$

Consider the collection of sets $\mathcal{S}^*(\lambda) = \{s' \in \mathcal{S}\setminus\{s\} : s' \cap X_s(\lambda) \neq \emptyset$ and $\ell(s') \leq \lambda\}$. As the set s moves up from level λ to level $\lambda + d\lambda$, each set $s' \in \mathcal{S}^*(\lambda)$ decreases its real-weight $w_{s'}$ by $|s' \cap X_s(\lambda)| \cdot (-dw) \geq 0$, and the real-weight of every other set $s' \notin \mathcal{S}^*(\lambda) \cup \{s\}$ remains unchanged. For each of these sets $s' \in \mathcal{S}^*(\lambda)$, the increase in its dead-weight $\phi_{s'}$ is upper bounded by the decrease in its real-weight (see the description of FIX(s) in Section 3.2). Hence, for each set $s' \in \mathcal{S}^*(\lambda)$, we get $0 \leq d\phi_{s'} \leq |s' \cap X_s(\lambda)| \cdot (-dw)$. None of the other sets change their dead-weights as s moves up from level λ to level $\lambda + d\lambda$. Thus, total volume of the down-tokens increases by:

$$\begin{aligned}&\textstyle\sum_{s'\in\mathcal{S}^*(\lambda)}(1+\epsilon)^{\ell(s')} \cdot d\phi_{s'} \leq (1+\epsilon)^\lambda \cdot \sum_{s'\in\mathcal{S}^*(\lambda)} d\phi_{s'}\\ \leq&(1+\epsilon)^\lambda \cdot (-dw) \cdot \textstyle\sum_{s'\in\mathcal{S}^*(\lambda)} |s' \cap X_s(\lambda)|\\ \leq&(1+\epsilon)^\lambda \cdot (-dw) \cdot f \cdot |X_s(\lambda)|.\end{aligned}$$

The last inequality holds since each element in $X_s(\lambda)$ is contained in at most f sets from $\mathcal{S}^*(\lambda)$. By (3.1), the increase in the total volume of down-tokens is $\leq -f(1+\epsilon)^\lambda \cdot dw_s(s \to \lambda)$. ∎

LEMMA 3.3. *During steps (01)-(02) in Figure 1, the total fractional work done is $O(f^2\gamma_\epsilon)$.*

Proof. Suppose that the FOR loop in step (02) runs for r iterations, where in each iteration $i \in \{1,\ldots,r\}$ it deals with a distinct set $s_i \in \mathcal{S}_{e'}$. We will show that the fractional work done during each iteration is $O(f\gamma_\epsilon)$. Since $r = |\mathcal{S}_{e'}| \leq f$, this will imply the lemma.

For the rest of the proof, focus on any iteration $i \in \{1,\ldots,r\}$, and the call to FIX(s_i). At the start of this call, we have $w_s^* = w_s = 1+\delta_i$, for some $\delta_i > 0$, even after resetting the dead-weight $\phi_{s_i} \leftarrow 0$. (Otherwise, the set s_i does not change its level and the fractional work done $= 0$). At the end of this iteration, the set s_i has moved up to its target-level $\ell^*(s) \leq \ell(e')$ (this inequality follows from Assumption 3.2), and it is clean with weights $w_s^* = w_s = 1$. Part (a) of Lemma 3.2 now implies that the fractional work done during this iteration is:

$$\begin{aligned}&\textstyle\int_{1+\delta_i}^{1} -f\gamma_\epsilon(1+\epsilon)^\lambda dw_{s_i}(s_i \to \lambda)\\ \leq&f\gamma_\epsilon(1+\epsilon)^{\ell^*(s)} \textstyle\int_{1+\delta_i}^{1} -dw_{s_i}(s_i \to \lambda)\\ =&\delta_i f\gamma_\epsilon(1+\epsilon)^{\ell^*(s)} \leq \delta_i f\gamma_\epsilon(1+\epsilon)^{\ell(e')}.\end{aligned} \tag{3.2}$$

Let $w'_{s_i}, w''_{s_i}, w'''_{s_i}, w''''_{s_i}$ respectively denote the real-weight of the set s_i just before the insertion of the element e', just after step (01), just before it starts moving up towards its target-level during the call to FIX(s_i), and just after the call to FIX(s_i). Thus, we have $w'''_{s_i} = 1 + \delta_i$ and $w''''_{s_i} = 1$. Since a call to FIX(.) never increases the real-weight of any set, it follows that $w'''_{s_i} \leq w''_{s_i}$. Since the set s_i was clean just before the insertion of the element e', we get $w'_{s_i} \leq 1$. Finally, step (01) in Figure 1 implies that $w''_{s_i} = w'_{s_i} + (1+\epsilon)^{-\ell(e')}$. To summarize, we have:

$$w'_{s_i} \leq 1 = w''''_{s_i} < w'''_{s_i} = 1+\delta_i \leq w''_{s_i} = w'_{s_i}+(1+\epsilon)^{-\ell(e')}.$$

Then we derive that $1 + \delta_i \leq w'_{s_i} + (1+\epsilon)^{-\ell(e')} \leq 1 + (1+\epsilon)^{-\ell(e')}$, which gives us: $\delta_i \leq (1+\epsilon)^{-\ell(e')}$. This observation, along with (3.2), concludes the proof of the lemma. ∎

COROLLARY 3.1. *Steps (01)-(02) in Figure 1 increase the total volume of down-tokens by $O(f^2)$.*

Proof. By Lemma 3.2, the increase in the total volume of down-tokens during steps (01)-(02) is at most γ_ϵ^{-1} times the total fractional work done. The corollary now follows from Lemma 3.3. ∎

CLAIM 3.1. *Let $\alpha_1 \ldots \alpha_j$ and $\beta_1 \ldots \beta_j$ be nonnegative real numbers satisfying the following property: j is the smallest index $j' \in \{1, \ldots, j\}$ such that $\sum_{i=1}^{j'} \alpha_i > \sum_{i=1}^{j'} \beta_i$. Then for all $0 \leq \lambda_1 \leq \cdots \leq \lambda_j$, we have $\sum_{i=1}^{j} (1+\epsilon)^{\lambda_i} \cdot \alpha_i \geq \sum_{i=1}^{j} (1+\epsilon)^{\lambda_i} \cdot \beta_i$.*

LEMMA 3.4. *The time spent to implement a call to the subroutine* REBUILD($\leq k$) *is at most $O(1/\epsilon)$ times the decrease in the total volume of down-tokens during the same call.*

Proof. Unless specified otherwise, throughout this proof we focus on the time-instant just before the call to REBUILD($\leq k$). At that time, from Section 3.1 and Figure 1 we infer that k is the smallest level such that $\sum_{s \in \mathcal{S}: \ell(s) \leq k} \phi_s > \sum_{e \in \mathcal{U}: \ell(e) \leq k} \epsilon f \cdot w_e$. Define $\mathcal{L}_k = \{\lambda : \lambda \leq k \text{ and } \lambda = \ell(s) \text{ for some set } s \in \mathcal{S}\}$. Since each set gets assigned to exactly one level, we have $|\mathcal{L}_k| \leq |\mathcal{S}|$. In particular, the collection $\mathcal{L}_k$ is finite. Let $\mathcal{L}_k = \{\lambda_1, \ldots, \lambda_j\}$ where $\lambda_1 < \lambda_2 < \cdots < \lambda_j = k$. For any $\lambda_i \in \mathcal{L}_k$, let $\phi_i = \sum_{s \in \mathcal{S}: \ell(s) = \lambda_i} \phi_s$ and $w_i = \sum_{e \in \mathcal{U}: \ell(e) = \lambda_i} w_e$ respectively denote the total dead-weight and element-weight stored at level λ_i. Note that $\sum_{s \in \mathcal{S}: \ell(s) \leq k} \phi_s = \sum_{i=1}^{j} \phi_i$ and $\sum_{e \in \mathcal{U}: \ell(e) \leq k} w_e = \sum_{i=1}^{j} w_i$. Since k is the smallest level such that $\sum_{s \in \mathcal{S}: \ell(s) \leq k} \phi_s > \sum_{e \in \mathcal{U}: \ell(e) \leq k} \epsilon f \cdot w_e$, we get j is the smallest index $j' \in \{1, \ldots, j\}$ s.t. $\sum_{i=1}^{j'} \phi_i > \sum_{i=1}^{j'} \epsilon f \cdot w_i$. Now, Claim 3.1 gives us:

$$\sum_{i=1}^{j} (1+\epsilon)^{\lambda_i} \cdot \phi_i \geq \sum_{i=1}^{j'} (1+\epsilon)^{\lambda_i} \cdot \epsilon f \cdot w_i \tag{3.3}$$

Each element $e \in \mathcal{U}$ at level $\ell(e) = \lambda_i$ has weight $w_e = (1+\epsilon)^{-\lambda_i}$. Hence, the quantity $(1+\epsilon)^{\lambda_i} w_i$ equals the number of elements at levels λ_i. Summing over all the levels in $\mathcal{L}_k$, we infer that the right hand side (RHS) of (3.3) equals $\epsilon f \cdot \mathcal{U}'_{\leq k}$, where $\mathcal{U}'_{\leq k}$ is the collection of elements at levels $\leq k$ just before the call to REBUILD($\leq k$). In contrast, the left hand side (LHS) of (3.3) equals the total volume of down-tokens at level $\leq k$ just before the call to REBUILD($\leq k$). The call to REBUILD($\leq k$) does not change the dead-weight of any set $s \in \mathcal{S}$ at level $\ell(s) > k$, and Observation 3.2 states that at the end of the call to this subroutine every set at level $\leq k$ has zero dead-weight. To summarize, the LHS of (3.3) equals the amount by which the total volume of down-tokens decreases during the call to REBUILD($\leq k$), whereas the RHS of (3.3) equals ϵ times the total time spent by our algorithm to implement this call (see Lemma 3.1). This completes the proof of the lemma. ∎

4 Overview for the Worst Case Update Time

Our complete algorithm needs to deal with a lot of subtle issues, and is presented in the full version. Here, to highlight the main ideas, we only focus on the decremental (deletions only) setting.

OBSERVATION 4.1. *The static algorithm in Section 2 never assigns a set $s \in \mathcal{S}$ to a level $\ell(s) \geq L = 1 + \lceil \log_{(1+\epsilon)} n \rceil$, where n is the total number of elements.*

Proof. Recall the continuous process from Section 2 that starts at time $t = -\infty$. When $t = L$, every element $e \in s$ has weight $w_e \leq (1+\epsilon)^{-L} < 1/n$, and hence $w_s = \sum_{e \in s} w_e < |s| \cdot (1/n) \leq 1$. Since s is not yet tight at time $t = L$, it gets assigned to a level $< L$ at the end of the algorithm. ∎

Schedulers: Our dynamic algorithm uses L subroutines called SCHEDULERS – one for each level in $[L] = \{1, \ldots, L\}$. Informally, for each $k \in [L]$ the subroutine SCHEDULER(k) is responsible for all the sets and elements at levels $\leq k$. Each scheduler works on its own local memory that is disjoint from the memory locations used by the other schedulers. Another key feature of our algorithm is that we allow different schedulers to hold mutually inconsistent views regarding the level of an individual set or element. Before proceeding any further, we introduce some important concepts and notations.

Most of the concepts defined below – such as the notions of real-weights, dead-weights and total-weights – closely mirror their counterparts from Section 3.

Let $\mathcal{S}^{(k)} \subseteq \mathcal{S}$ and $\mathcal{U}^{(k)} \subseteq \mathcal{U}$ respectively denote the collection of sets and elements SCHEDULER(k) is responsible for. Let $\ell^{(k)}(e)$ and $\ell^{(k)}(s)$ respectively denote the level of an element $e \in \mathcal{U}^{(k)}$ and a set $s \in \mathcal{S}^{(k)}$ from the perspective of SCHEDULER(k). As usual, the level of an element $e \in \mathcal{U}^{(k)}$ according to SCHEDULER(k) is completely determined by the levels of the sets in $\mathcal{S}^{(k)}$ that contain it: We have $\ell^{(k)}(e) = \max_{s \in \mathcal{S}^{(k)}: e \in s} \ell^{(k)}(s) \leq k$ for all $e \in \mathcal{U}^{(k)}$. Let $w_e^{(k)} = (1+\epsilon)^{-\ell^{(k)}(e)}$ be the weight of an element $e \in \mathcal{U}^{(k)}$ according to SCHEDULER(k). The *real-weight* of a set $s \in \mathcal{S}^{(k)}$ according to SCHEDULER(k) equals $w_s^{(k)} = \sum_{e \in s \cap \mathcal{U}^{(k)}} w_e^{(k)} + \delta_s^{(k)}$, where $\delta_s^{(k)}$ is the *extra-weight* of s. We will shortly see that the concept of extra-weight has a natural explanation. Intuitively, the quantity $\delta_s^{(k)}$ measures the weight received by a set $s \in \mathcal{S}^{(k)}$ from elements that are at level $> k$. Each set $s \in \mathcal{S}^{(k)}$ has a *dead-weight* $\phi_s^{(k)}$, and its *total-weight* is given by $w_s^{*(k)} = w_s^{(k)} + \phi_s^{(k)}$. Finally, SCHEDULER($k$) maintains a collection $D^{(k)}$ of some elements that got deleted from $\mathcal{U}^{(k)}$ in the past due to an external update operation. The elements in $D = \bigcup_{k=1}^{L} D^{(k)}$ are called *dead elements*.

Preprocessing: We first run the static algorithm from Section 2. Let $\ell(s), \ell(e), w_s, w_e$ denote the levels and weights of elements and sets returned by this static algorithm. At this stage, all the different schedulers completely agree with each other regarding the level of each element and set. Specifically, consider any level $k \in [L]$. At this point in time, we have $\mathcal{S}^{(k)} = \{s \in \mathcal{S} : \ell(s) \leq k\}$, $\mathcal{U}^{(k)} = \{e \in \mathcal{U} : \ell(e) \leq k\}$ and $\mathsf{D}^{(k)} = \emptyset$. For all elements $e \in \mathcal{U}^{(k)}$ and sets $s \in \mathcal{S}^{(k)}$, we have $\ell^{(k)}(e) = \ell(e)$, $w_e^{(k)} = w_e$ and $\ell^{(k)}(s) = \ell(s)$. For all $s \in \mathcal{S}^{(k)}$, we set $\delta_s^{(k)} = \sum_{e \in s: \ell(e) > k} w_e$ and $\phi_s^{(k)} = 0$. Hence, we have $w_s^{*(k)} = w_s^{(k)} = w_s$ for all $s \in \mathcal{S}^{(k)}$ just after the end of preprocessing.

Invariants: We now describe three important invariants that are satisfied by our dynamic algorithm. Invariant 4.1 closely mirrors Invariant 3.1 from Section 3.1, and it clearly holds at the end of preprocessing. Invariant 4.2 dictates that the number of dead elements in the control of a SCHEDULER(k) is really small compared to the number of elements in $\mathcal{U}^{(k)}$. At the end of preprocessing, this invariant trivially holds because $D^{(k)} = \emptyset$. Invariant 4.3 says that the sets $\mathcal{S}^{(k)}, \mathcal{U}^{(k)}$ and $D^{(k)}$ form a very nice laminar structure as k ranges from L to 1. Specifically, the sets/elements a SCHEDULER($k-1$) is responsible for are precisely the ones that lie at level $\leq k-1$ according to the next SCHEDULER(k). Furthermore, SCHEDULER(L) is responsible for all the elements and sets. Again, it is easy to check that this invariant holds at the end of preprocessing.

INVARIANT 4.1. *Consider any $k \in [L]$. Every set $s \in \mathcal{S}^{(k)}$ at level $\ell^{(k)}(s) > 0$ has total-weight $w^{*(k)} = 1$. Furthermore, every set $s \in \mathcal{S}^{(k)}$ at level $\ell^{(k)}(s) = 0$ has total-weight $w^{*(k)} \leq 1$.*

INVARIANT 4.2. *For all $k \in [L]$, $\left|D^{(k)}\right| \leq 2\epsilon \cdot \left|\mathcal{U}^{(k)}\right|$.*

INVARIANT 4.3. *For every $k \in [2, L]$, we have $\mathcal{S}^{(k-1)} = \{s \in \mathcal{S}^{(k)} : \ell^{(k)}(s) \leq k-1\}$, $\mathcal{U}^{(k-1)} = \{e \in \mathcal{U}^{(k)} : \ell^{(k)}(e) \leq k-1\}$ and $\mathsf{D}^{(k-1)} = \{e \in \mathsf{D}^{(k)} : \ell^{(k)}(e) \leq k-1\}$. Furthermore, we have $\mathcal{S}^{(L)} = \mathcal{S}$ and $\mathcal{U}^{(L)} = \mathcal{U}$.*

Ownerships: We say that an element $e \in \mathcal{U}$ (resp. $e \in D$) is *owned* by a level $k \in [L]$ iff $e \in \mathcal{U}^{(k)}$ (resp. $e \in D^{(k)}$) and $e \notin \mathcal{U}^{(j)}$ (resp. $e \in D^{(j)}$) for all $j \in [k-1]$. A set $s \in \mathcal{S}$ is *owned* by a level $k \in [L]$ iff $s \in \mathcal{S}^{(k)}$ and $s \notin \mathcal{S}^{(j)}$ for all $j \in [k-1]$. Note that each element/set is owned by a unique level. We now describe how to handle a sequence of element deletions after preprocessing.

Handling the deletion of an element: Suppose that an element e, which was owned by level j, gets deleted. Accordingly, we *feed* this deletion to SCHEDULER(j), ..., SCHEDULER(L). Note that these are precisely the schedulers that are responsible for this element and are affected by the concerned deletion. Each of these affected schedulers works within its own local memory, independently of others. We describe the actions taken by SCHEDULER(k), for any $k \in \{j, \ldots, L\}$.

Suppose that the element e was at level $\ell^{(k)}(e) = i$ just before its deletion. SCHEDULER(k) moves the element e from $\mathcal{U}^{(k)}$ to $D^{(k)}$, without changing the its level $\ell^{(k)}(e) = i$ or weight $w_e^{(k)} = (1+\epsilon)^{-i}$. For each set $s \in \mathcal{S}^{(k)}$ containing the element e, this reduces its real-weight $w_s^{(k)}$ by $(1+\epsilon)^{-i}$. To compensate for this loss, we increase its dead-weight $\phi_s^{(k)}$ by the same amount $(1+\epsilon)^{-i}$. We do not change the extra-weights of the sets. Thus, the total-weight of every set also remains unchanged.

Triggering a rebuild: It is easy to check that the above actions do not violate Invariant 4.1 and Invariant 4.3. However, if we keep acting in this lazy manner, then the number of dead elements keep growing with time, and so after some number of updates Invariant 4.2 will get violated. Furthermore, unlike in Section 3 here we cannot even afford to wait until the moment Invariant 4.2 is violated before triggering a rebuild, because we are shooting for worst-case update time and the rebuild needs to occur *in the background* – a few

steps at a time. Accordingly, for each $k \in [L]$, whenever SCHEDULER(k) finds that $|D^{(k)}| \geq \epsilon \cdot |\mathcal{U}^{(k)}|$ (note that this is still ϵ-*far* from violating Invariant 4.2), it starts *rebuilding* its part of the input $(\mathcal{U}^{(k)}, \mathcal{S}^{(k)}, D^{(k)})$ *in the background* in a separate memory location that is not affected by the happenings elsewhere. We now explain this rebuilding procedure in a bit more details.

Rebuilding done by SCHEDULER(k): Suppose that SCHEDULER(k) has triggered a rebuild at the present moment because $|D^{(k)}| = \epsilon \cdot |\mathcal{U}^{(k)}|$. In order to see the high level idea, for now assume that SCHEDULER(k) does not need to handle an element deletion due to any external update operation while it is performing the rebuild.[3] Then the rebuild subroutine will *clean-up* all the dead elements in $D^{(k)}$, by resetting $D^{(k)} = \emptyset$ and $\phi_s^{(k)} = 0$ for all $s \in \mathcal{S}^{(k)}$, and then run the static algorithm (see Section 3) on input $(\mathcal{S}^{(k)}, \mathcal{U}^{(k)})$ starting from time $t = -k$ onward. During this run of the static algorithm, the extra-weights $\delta_s^{(k)}$ of the sets $s \in \mathcal{S}^{(k)}$ will not change, because these extra-weights are coming from elements that are at levels $> k$. Note that this is exactly the same principle that underpins the rebuild subroutine described in Section 3.1. At the end of this static algorithm, we will get the following guarantees: Invariant 4.1 is satisfied by SCHEDULER(k), and $D^{(k)} = \emptyset$. At this point, SCHEDULER(k) will *order* all the schedulers corresponding to levels $j \in [k-1] = \{1, \ldots, k-1\}$ to abort whatever they are doing and synchronize their perspectives with the perspective of SCHEDULER(k). We refer to this as the *synchronization* event. At the end of this, each SCHEDULER(j), $j \in [k-1]$, will satisfy: $\mathcal{S}^{(j)} = \{s \in \mathcal{S}^{(k)} : \ell^{(k)}(s) \leq j\}$, $\mathcal{U}^{(j)} = \{e \in \mathcal{U}^{(k)} : \ell^{(k)}(s) \leq j\}$, $D^{(j)} = \emptyset$, $\ell^{(j)}(e) = \ell^{(k)}(e)$ for all $e \in \mathcal{U}^{(j)}$, and $w_s^{*(j)} = w_s^{*(k)}$ for all $s \in \mathcal{S}^{(j)}$. Our algorithm ensures that this synchronization is achieved in $O(k)$ time *on the fly*, as follows. The rebuild subroutine of SCHEDULER(k) will run k different *threads* – one for each level $j \in \{1, \ldots, k\}$. Each of these threads will run in a designated separate region of memory, independent of others. It will be the responsibility of thread j to prepare everything in its allocated memory region, which can then be handed over to SCHEDULER(j) at the time of synchronization by simply passing a pointer to the beginning of the concerned memory region. Thus, synchronization involves the passing of k pointers, and hence takes $O(k) = O(L) = O(\log n/\epsilon)$ time. It is easy to check that Invariants 4.1 and 4.3 continue to remain satisfied at the end of synchronization. Furthermore, the remaining Invariant 4.2 is trivially satisfied under our working assumption that no element deletion occurs in $\mathcal{U}^{(k)}$ as SCHEDULER(k) is rebuilding in the background. Next, we analyze the worst-case update time of this algorithm, and outline what happens when this working assumption does not hold.

Worst-case update time: We measure the time taken to implement a procedure in terms of *units of work*. The update time of our algorithm is dominated by the time spent on the rebuild subroutines of the individual schedulers. There are L schedulers running in the background, and we ensure that each of these individual schedulers perform $O(fL/\epsilon)$ units of work after every external update (element deletion). This implies a worst-case update time of $O(L \cdot fL/\epsilon) = O(f \log^2 n/\epsilon^3)$.

We now explain why each SCHEDULER(k) needs to perform $O(fL/\epsilon)$ units of work per update. The factor $O(L)$ comes from the fact that the rebuild subroutine of SCHEDULER(k) needs to run $k = O(L)$ threads in the background so as to ensure that it can synchronize on the fly when it finishes execution. It now remains to explain the rationale behind the remaining $O(f/\epsilon)$ factor. Suppose that $d = |D^{(k)}|$ and $u = |\mathcal{U}^{(k)}|$ when the rebuild subroutine of SCHEDULER(k) gets triggered. We have $d = \epsilon u$, and hence the subroutine needs to perform $O(f(u+d)) = O(fu)$ units of work overall (see Lemma 3.1). According to our scheme, the subroutine splits this work across a sequence of external updates (element deletions), performing cf/ϵ units of work per update for some large constant $c > 1$. This ensures that in the general setting (when the working assumption in the previous paragraph does not hold), at most $fu/(cf/\epsilon) = \epsilon u/c$ many elements get deleted from $\mathcal{U}^{(k)}$ when the rebuild subroutine is in progress. In the full version, we show that there is a way to handle these incoming deletions *on the fly* during the rebuild subroutine. However, this comes at a cost: When an element e gets deleted that has already been processed by the rebuild subroutine, it gets classified as a dead element. But since the subroutine is working at a sufficient fast rate, at most $\epsilon u/c$ many new dead elements might get created in this manner at the time of synchronization, whereas the old dead elements that were present at the time the rebuild subroutine was triggered are removed anyway. Taking everything together, just after synchronization we end up having $|D^{(j)}| \leq (\epsilon/c) \cdot |\mathcal{U}^{(j)}|$ for all $j \in [k]$ and hence Invariant 4.2 continues to remain satisfied.

Approximation ratio: The main challenge here is to show that all the same element/set might get assigned to different levels by different schedulers, there is a way to come up with a *consistent* assignment of levels to all the elements in $\mathcal{U} \cup D$ and all the sets in $\mathcal{S}$. Furthermore, the

[3] According to this assumption, during the same time-interval when SCHEDULER(k) is performing a rebuild, some other SCHEDULER(j) with $j > k$ might still have to handle element deletions due to external updates.

sets with weight = 1 in this consistent assignment form a $(1+\epsilon)f$-approximate minimum set cover in $(\mathcal{U}, \mathcal{S})$.

The consistent assignment is as follows: Each set $s \in \mathcal{S}$ gets assigned to the level that owns it. Let $\ell(s)$ denote the level of a set $s \in \mathcal{S}$ in the consistent assignment. This automatically defines the level $\ell(e)$ and weight w_e of every element $e \in \mathcal{U} \cup D$, since $\ell(e) = \max_{s \in \mathcal{S}: e \in s} \ell(s)$ and $w_e = (1+\epsilon)^{-\ell(s)}$. For all $s \in \mathcal{S}$, we define its weight to be $w_s^{(\mathcal{U} \cup \mathsf{D})} = \sum_{e \in \mathcal{U} \cup D: e \in s} w_e$.

For each $k \in [L]$, define $\mathcal{S}(\leq k) = \{s \in \mathcal{S} : \ell(s) \leq k\}$, $D(\leq k) = \{e \in D : \ell(e) \leq k\}$ and $\mathcal{U}(\leq k) = \{e \in \mathcal{U} : \ell(e) \leq k\}$. The lemma below shows that the levels of elements/sets in the consistent assignment are nicely aligned with the levels of the same elements/sets according to the individual schedulers. Lemma 4.1 and Invariant 4.2 together imply Corollary 4.1.

LEMMA 4.1. *For each $k \in [L]$, we have $\mathcal{S}(\leq k) = \mathcal{S}^{(k)}$, $\mathcal{U}(\leq k) = \mathcal{U}^{(k)}$, and $D(\leq k) = D^{(k)}$.*

Proof. (Sketch) The lemma follows from induction: It is easy to check that the lemma holds just after preprocessing. When an element e gets deleted, each SCHEDULER(k) responsible for e simply moves it from $\mathcal{U}^{(k)}$ to $D^{(k)}$ without changing its weight or level. Hence, the lemma continues to remain satisfied. Finally, a moment's thought reveals that the lemma continues to hold after a rebuild subroutine of some SCHEDULER(k) executes its synchronization step. ∎

COROLLARY 4.1. $\forall k \in [L]$, $|D(\leq k)| \leq 2\epsilon \cdot |\mathcal{U}(\leq k)|$.

COROLLARY 4.2. $\sum_{e \in D} w_e \leq 2\epsilon(1+\epsilon) \cdot \sum_{e \in \mathcal{U}} w_e$.

Proof. (Sketch) The proof is almost the same as the proof of Lemma 4.8 in the Arxiv version of [11]. Basically, consider any dead-element $e' \in D$ and any actual element $e \in \mathcal{U}$ such that both their levels lie within the interval $[k-1, k]$, i.e., $k-1 \leq \ell(e), \ell(e') \leq k$. Then it follows that their weights $w_{e'}$ and w_e are within a $(1+\epsilon)$ multiplicative factor of each other. This observation, along with Corollary 4.1, is sufficient to ensure that $\sum_{e \in D} w_e \leq 2\epsilon(1+\epsilon) \cdot \sum_{e \in \mathcal{U}} w_e$. ∎

LEMMA 4.2. *For every set $s \in \mathcal{S}$ at level $\ell(s) > 0$, we have $w_s^{(\mathcal{U} \cup \mathsf{D})} = 1$. Furthermore, for every set $s \in \mathcal{S}$ at level $\ell(s) = 0$, we have $w_s^{(\mathcal{U} \cup \mathsf{D})} \leq 1$.*

Lemma 4.2 (whose proof follows from induction) closely mirrors Lemma 2.2 from Section 2. Consider the collection of sets $\mathcal{T} = \{s \in \mathcal{S} : w_s^{(\mathcal{U} \cup D)} = 1\}$. Lemma 4.2 implies that (see the proof of Lemma 2.2) the element-weights $\{w_e\}$ form a valid fractional packing in $(\mathcal{U} \cup D, \mathcal{S})$ and $\mathcal{T}$ forms a valid set cover in $(\mathcal{U} \cup D, \mathcal{S})$. Furthermore, following the proof of Lemma 2.2, we get: $f \cdot \sum_{e \in \mathcal{U} \cup D} w_e \geq |\mathcal{T}|$. Applying Corollary 4.2, we now derive that:

$$\begin{aligned} f \cdot \textstyle\sum_{e \in \mathcal{U}} w_e &\geq (1 + 2\epsilon(1+\epsilon))^{-1} \cdot f \cdot \textstyle\sum_{e \in \mathcal{U} \cup D} w_e \\ &\geq (1 + 2\epsilon(1+\epsilon))^{-1} \cdot |\mathcal{T}|. \end{aligned} \tag{4.4}$$

Since $\mathcal{T}$ is a set cover in $(\mathcal{U} \cup D, \mathcal{S})$, it also forms a set cover in $(\mathcal{U}, \mathcal{S})$. Similarly, since the element-weights $\{w_e\}_{e \in \mathcal{U} \cup D}$ form a fractional packing in $(\mathcal{U} \cup D, \mathcal{S})$, the weights $\{w_e\}_{e \in \mathcal{U}}$ form a fractional packing in $(\mathcal{U}, \mathcal{S})$. Thus, we have a set cover $\mathcal{T}$ and a fractional packing $\{w_e\}$ in $(\mathcal{U}, \mathcal{S})$ whose sizes are within a $(1 + 2\epsilon(1+\epsilon))f$ multiplicative factor of each other, according to (4.4). Hence, Lemma 2.1 implies that $\mathcal{T}$ forms a $(1 + 2\epsilon(1+\epsilon))f$-approximate minimum set cover in $(\mathcal{U}, \mathcal{S})$.

Acknowledgments

The project has received funding from the Engineering and Physical Sciences Research Council, UK (EPSRC) under Grant Ref: EP/S03353X/1.

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. Henzinger was also supported by the Vienna Science and Technology Fund (ICT 15-003).

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).

Xiaowei Wu is funded by the Science and Technology Development Fund, Macau SAR (File no. SKL-IOTSC-2018-2020), the Start-up Research Grant of University of Macau (File no. SRG2020-00020-IOTSC)

References

[1] Amir Abboud, Raghavendra Addanki, Fabrizio Grandoni, Debmalya Panigrahi, and Barna Saha. Dynamic set cover: improved algorithms and lower bounds. In *STOC*, 2019.

[2] Moab Arar, Shiri Chechik, Sarel Cohen, Cliff Stein, and David Wajc. Dynamic matching: Reducing integral algorithms to approximately-maximal fractional algorithms. In *ICALP*, volume 107 of *LIPIcs*, pages 7:1–7:16. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2018.

[3] Surender Baswana, Manoj Gupta, and Sandeep Sen. Fully dynamic maximal matching in $o(\log n)$ update time. In *FOCS*, 2011.

[4] Aaron Bernstein, Sebastian Forster, and Monika Henzinger. A deamortization approach for dynamic spanner and dynamic maximal matching. In *SODA*, pages 1899–1918. SIAM, 2019.

[5] Aaron Bernstein, Sebastian Forster, and Monika Henzinger. A deamortization approach for dynamic spanner and dynamic maximal matching. In *SODA*, pages 1899–1918. SIAM, 2019.

[6] Sayan Bhattacharya, Deeparnab Chakrabarty, and Monika Henzinger. Deterministic fully dynamic approximate vertex cover and fractional matching in O(1) amortized update time. In *IPCO*, 2017.

[7] Sayan Bhattacharya, Monika Henzinger, and Giuseppe F. Italiano. Design of dynamic algorithms via primal-dual method. In *ICALP*, 2015.

[8] Sayan Bhattacharya, Monika Henzinger, and Giuseppe F. Italiano. Deterministic fully dynamic data structures for vertex cover and matching. In *SODA*, 2015.

[9] Sayan Bhattacharya, Monika Henzinger, and Danupon Nanongkai. New deterministic approximation algorithms for fully dynamic matching. In *STOC*, pages 398–411. ACM, 2016.

[10] 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.

[11] Sayan Bhattacharya, Monika Henzinger, and Danupon Nanongkai. A new deterministic algorithm for dynamic set cover. In *FOCS*, 2019.

[12] Sayan Bhattacharya, Monika Henzinger, Danupon Nanongkai, and Xiaowei Wu. Dynamic set cover: Improved amortized and worst-case update time. *CoRR*, abs/2002.11171, 2020.

[13] Sayan Bhattacharya and Janardhan Kulkarni. Deterministically maintaining a $(2 + \epsilon)$-approximate minimum vertex cover in $o(1/\epsilon^2)$ amortized update time. In *SODA*, 2019.

[14] Timothy M. Chan, Mihai Patrascu, and Liam Roditty. Dynamic connectivity: Connecting to networks and geometry. *SIAM J. Comput.*, 40(2):333–349, 2011. Announced at FOCS'08.

[15] Moses Charikar and Shay Solomon. Fully dynamic almost-maximal matching: Breaking the polynomial worst-case time barrier. In *ICALP*, volume 107 of *LIPIcs*, pages 33:1–33:14. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2018.

[16] Moses Charikar and Shay Solomon. Fully dynamic almost-maximal matching: Breaking the polynomial worst-case time barrier. In *ICALP*, volume 107 of *LIPIcs*, pages 33:1–33:14. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2018.

[17] 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.

[18] Irit Dinur and David Steurer. Analytical approach to parallel repetition. In *STOC*, 2014.

[19] Anupam Gupta, Ravishankar Krishnaswamy, Amit Kumar, and Debmalya Panigrahi. Online and dynamic algorithms for set cover. In *STOC*, 2017.

[20] Manoj Gupta and Richard Peng. Fully dynamic $(1+\epsilon)$-approximate matchings. In *FOCS*, 2013.

[21] Subhash Khot and Oded Regev. Vertex cover might be hard to approximate to within 2-\varepsilon. In *CCC*, 2003.

[22] Kasper Green Larsen. The cell probe complexity of dynamic range counting. In *STOC*, pages 85–94. ACM, 2012.

[23] Kasper Green Larsen, Omri Weinstein, and Huacheng Yu. Crossing the logarithmic barrier for dynamic boolean data structure lower bounds. In *STOC*, pages 978–989. ACM, 2018.

[24] 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.

[25] 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.

[26] Ofer Neiman and Shay Solomon. Simple deterministic algorithms for fully dynamic maximal matching. In *STOC*, 2013.

[27] Krzysztof Onak and Ronitt Rubinfeld. Maintaining a large matching and a small vertex cover. In *STOC*, 2010.

[28] Mihai Patrascu and Erik D. Demaine. Logarithmic lower bounds in the cell-probe model. *SIAM J. Comput.*, 35(4):932–963, 2006.

[29] David Peleg and Shay Solomon. Dynamic $(1 + \epsilon)$-approximate matchings: A density-sensitive approach. In *SODA*, 2016.

[30] Shay Solomon. Fully dynamic maximal matching in constant update time. In *FOCS*, 2016.

[31] Christian Wulff-Nilsen. Fully-dynamic minimum spanning forest with improved worst-case update time. In *STOC*, pages 1130–1143. ACM, 2017.