Fully Dynamic Set Cover: Worst-Case Recourse and Update Time

Sayan Bhattacharya* Ruoxu Cen† Debmalya Panigrahi‡

November 2025

Abstract

In (fully) dynamic set cover, the goal is to maintain an approximately optimal solution to a dynamically evolving instance of set cover, where in each step either an element is added to or removed from the instance. The two main desiderata of a dynamic set cover algorithm are to minimize at each time-step,

- the *recourse*, which is the number of sets removed from or added to the solution, and
- the *update time* to compute the updated solution.

This problem has been extensively studied over the last decade leading to many results that achieve ever-improving bounds on the recourse and update time, while maintaining a solution whose cost is comparable to that of offline approximation algorithms.

In this paper, we give the **first** algorithms to simultaneously achieve non-trivial **worst-case bounds** for recourse and update time. Specifically, we give fully-dynamic set cover algorithms that simultaneously achieve $O(\log n)$ recourse and $f \cdot \text{poly}\log(n)$ update time in the worst-case, for both approximation regimes: $O(\log n)$ and $O(f)$ approximation. (Here, n, f respectively denote the maximum number of elements and maximum frequency of an element across all instances.) Prior to our work, all results for this problem either settled for amortized bounds on recourse and update time, or obtained $f \cdot \text{poly}\log(n)$ update time in the worst-case but at the cost of $\Omega(m)$ worst-case recourse. (Here, m denotes the number of sets. Note that *any* algorithm has recourse at most m.)

*Department of Computer Science, University of Warwick, Coventry, UK. Email: S.Bhattacharya@warwick.ac.uk.
†Department of Computer Science, Duke University, Durham, NC. Email: ruoxu.cen@duke.edu.
‡Department of Computer Science, Duke University, Durham, NC. Email: debmalya@cs.duke.edu.

Contents

Part I

Extended Abstract

1 Introduction

Consider the fundamental problem of computing a minimum *set cover*. As input, we receive a universe U of n elements and a collection $\mathcal{S} \subseteq 2^U$ of m sets defined over U, such that $\bigcup_{s \in \mathcal{S}} s = U$. A *set cover* is a collection of sets $\mathcal{S}^\star \subseteq \mathcal{S}$ that covers every element, i.e., $\bigcup_{s \in \mathcal{S}^\star} s = U$. The goal is to return a set cover of minimum size. Let f denote an upper bound on the maximum *frequency* of any element in U, where the frequency of a given element is the number of sets in $\mathcal{S}$ it belongs to. There are two textbook algorithms for this problem: one greedy and the other primal-dual. In $O(fn)$ time, they respectively achieve an approximation ratio of $\ln(n)$ and f (see e.g., [WS11]). Both the runtime and approximation guarantees of these two algorithms are tight, under standard complexity theoretic assumptions.[1]

We focus on the minimum set cover problem in a *dynamic* setting, where the input keeps changing via a sequence of *updates*. Each update corresponds to the insertion/deletion of an element. Throughout these updates, we have to maintain an approximately minimum set cover of the current input. It is common to quantify the performance of such a *dynamic algorithm* according to three parameters.

- *Approximation ratio*: It indicates the quality of the solution $\mathcal{S}^\star \subseteq \mathcal{S}$ maintained by the algorithm.
- *Recourse*: It equals the number of changes (i.e., insertions/deletions of sets) in the maintained solution $\mathcal{S}^\star$, per update. It measures the *stability* of the solution maintained by the algorithm.
- *Update time*: It is the time taken to process an update. It measures the *efficiency* of the algorithm.

The ultimate goal in this setting is to design a dynamic algorithm that has **worst-case** guarantees, and matches the performance of the best known static algorithm.[2] This leads us to the following two natural questions.

- **Q1:** Is there a dynamic set cover algorithm with $O(f)$ approximation ratio, $\tilde{O}(1)$ **worst-case** recourse and $\tilde{O}(f)$ **worst-case** update time? (Throughout the paper, the $\tilde{O}(.)$ notation hides polylog(n) factors.)
- **Q2:** Is there a dynamic set cover algorithm with $O(\log n)$ approximation ratio, $\tilde{O}(1)$ **worst-case** recourse and $\tilde{O}(f)$ **worst-case** update time?

The study of a fundamental, textbook problem like set cover occupies a central place in the dynamic algorithms literature. Indeed, as we outline in Section 1.1, over the past decade an extensive and influential line of work has been devoted towards designing dynamic set cover algorithms, exploring various trade-offs between the three performance measures (approximation ratio, recourse and update time). Surprisingly, however, **all previous dynamic set cover algorithms with non-trivial update times had $\Omega(m)$ worst-case recourse**. Note that this worst-case recourse bound is trivial since *any* algorithm has recourse at most m. So, while there was substantial progress in dynamic set cover, particularly for amortized bounds, in prior work, it did not offer an answer to **Q1** or **Q2**.

[1] The size of the input can be $\Omega(fn)$, and hence this $O(fn)$ runtime is asymptotically optimal. Further, it is highly unlikely that we can beat either the f or the $\ln(n)$ approximation guarantee, even if we allow for any arbitrary polynomial runtime [DS14, KR03].

[2] In other words, we want to recover a near-optimal static algorithm by feeding the universe U as a sequence of n element-insertions, to be processed by the concerned dynamic algorithm. Thus, if the dynamic algorithm has update time $O(\tau)$, then the resulting static algorithm would have a runtime of $O(\tau \cdot n)$.

We answer *both* **Q1** and **Q2** in the affirmative, by obtaining the first algorithms to *simultaneously* achieve near-optimal worst-case recourse and worst-case update time bounds for dynamic set cover.

Our results are summarized in the following two theorems.

Theorem 1.1. *There is a deterministic $O(f)$-approximation dynamic set cover algorithm with $O(\log n)$* **worst-case** *recourse and $O(f \log^3(n))$* **worst-case** *update time.*

Theorem 1.2. *There is a deterministic $O(\log n)$-approximation dynamic set cover algorithm with $O(\log n)$* **worst-case** *recourse and $O(f \log^3(n))$* **worst-case** *update time.*

1.1 Prior Work

Low Frequency Regime ($O(f)$-approximation). First, note that if $f = 2$, then the dynamic set cover problem is equivalent to *dynamic vertex cover*. Here, we have an input graph $G = (V, E)$, with n nodes and m edges, that is undergoing a sequence of edge insertions/deletions, and we have to maintain an approximately minimum vertex cover at all times. Accordingly, **Q1** can be recast as follows, *in the special case when $f = 2$.*

- **Q1*:** Can we maintain a $O(1)$-approximate minimum vertex cover in a dynamic graph with $\tilde{O}(1)$ **worst-case** recourse and $\tilde{O}(1)$ **worst-case** update time?

Approx Ratio	Amortized Update Time	**Worst Case** Update Time	Amortized Recourse	**Worst Case** Recourse	Paper
$O\left(f^2\right)$	$O(f \log(m+n))$	-	$O(\log(m+n))$	$\Omega(m)$	[BHI15a]
$O\left(f^3\right)$	$O\left(f^2\right)$	-	$O(f)$	$\Omega(m)$	[BCH17]
$O\left(f^3\right)$	$O\left(f^2\right)$	-	$O(f)$	$\Omega(m)$	[GKKP17]
$(1+\epsilon)f$	$O\left(f^2 \log(n)/\epsilon^5\right)$	-	$O\left(f \log(n)/\epsilon^5\right)$	$\Omega(m)$	[AAG+19] **
$(1+\epsilon)f$	$O\left(f \log(n)/\epsilon^2\right)$	-	$O\left(\log(n)/\epsilon^2\right)$	$\Omega(m)$	[BHN19]
f	$O\left(f^2\right)$	-	$O(f)$	$\Omega(m)$	[AS21]**
$(1+\epsilon)f$	$O\left(f^2/\epsilon^2\right)$	-	$O\left(f/\epsilon^2\right)$	$\Omega(m)$	[BHNW21]
$(1+\epsilon)f$	$O\left(f \log^2(n)/\epsilon^3\right)$	$O\left(f \log^2(n)/\epsilon^3\right)$	$O\left(\log^2(n)/\epsilon^3\right)$	$\Omega(m)$	[BHNW21]
$(1+\epsilon)f$	$O\left(\frac{f \log f}{\epsilon} + \frac{f}{\epsilon^3}\right)$	-	$O\left(\frac{\log f}{\epsilon} + \frac{1}{\epsilon^3}\right)$	$\Omega(m)$	[BSZ25]
$(1+\epsilon)f$	$O\left(\frac{f \log^\star f}{\epsilon} + \frac{f}{\epsilon^3}\right)$	-	$O\left(\frac{\log^\star f}{\epsilon} + \frac{1}{\epsilon^3}\right)$	$\Omega(m)$	[BSZ25]*
$(1+\epsilon)f$	$O\left(f \log(n)/\epsilon^2\right)$	$O\left(f \log(n)/\epsilon^2\right)$	$O\left(\log(n)/\epsilon^2\right)$	$\Omega(m)$	[SUZ24]
$O(f)$	$O\left(f \log^3(n)\right)$	$O\left(f \log^3(n)\right)$	$O\left(\log n\right)$	$O\left(\log n\right)$	**our result**

*Table 1: Prior results for dynamic set cover in the low frequency regime. The entries marked with ** (resp. *) correspond to randomized algorithms that work only against an* oblivious adversary *(resp.* adaptive adversary*). All the other entries correspond to deterministic algorithms. Whenever the guarantee specified by* **Q1** *is attained for a certain parameter (i.e., approximation ratio, worst-case update time or worst-case recourse), the concerned entry is marked in blue. Otherwise, the concerned entry is marked in red. If an entry for worst-case update time is left blank, then it implies a trivial worst-case update time of $\tilde{O}(fn)$.*

The first nontrivial progress towards answering this question was made by [OR10] in 2010. They designed a randomized dynamic algorithm that works against an oblivious adversary, with an approximation ratio of

$O(1)$, an **amortized** recourse of $\tilde{O}(1)$ and an **amortized** update time of $\tilde{O}(1)$.[3] This was followed by a long sequence of results on dynamic vertex cover and its dual problem of dynamic matching [BGS11, NS13, BHI15b, BS16, BHN17, ACC+18, BK19, Waj20, BDL21, BKSW23, Beh23, BG24, AKK25]. Already in 2017, [BHN17] answered **Q1*** in the affirmative, by presenting a deterministic dynamic vertex cover algorithm with $(2+\epsilon)$ approximation ratio, $O(\log n)$ worst-case recourse and $O(\log^3(n)/\text{poly}(\epsilon))$ worst-case update time.

Beyond the special case of $f = 2$, however, progress towards resolving **Q1** has been rather limited. Table 1 summarizes the prior results on dynamic set cover in the low frequency regime.[4] This line of work was initiated by [BHI15a] in 2015. By 2019, there was a deterministic dynamic set cover algorithm with $(1+\epsilon)f$ approximation ratio, $\tilde{O}(1)$ amortized recourse and $\tilde{O}(1)$ amortized update time [BHN19]. Following up on this, [BHNW21] and [SUZ24] showed how to deamortize the result of [BHN19], which led to new algorithms with $(1+\epsilon)f$ approximation ratio and $\tilde{O}(f)$ worst-case update time. Unfortunately, both these latter algorithms [BHNW21, SUZ24] have a worst-case recourse of $\Omega(m)$. This is because each of these algorithms maintains $O(\log n)$ different solutions (i.e., collection of sets) *in the background*, while spending $\tilde{O}(f)$ worst-case time per update on each of these background solutions. It is possible, however, that none of these background solutions constitute a valid set cover. Instead, the actual output of the algorithm is a *foreground* solution, which is guaranteed to be a valid $(1+\epsilon)f$ approximate set cover of the current input. This foreground solution is obtained via carefully combining all the background solutions, by means of manipulating some pointers. Thus, although each individual background solution incurs $\tilde{O}(1)$ worst-case recourse, because of the pointer switches the foreground solution might incur a recourse as large as $\Omega(m)$ during a single update. In sharp contrast, as summarized in Theorem 1.1, we *simultaneously* achieve $O(f)$ approximation ratio, $\tilde{O}(1)$ worst-case recourse and $\tilde{O}(f)$ worst-case update time, thereby resolving **Q1**.

Approximation Ratio	Amortized Update Time	**Worst Case** Update Time	Amortized Recourse	**Worst Case** Recourse	Paper
$O(\log n)$	exponential	exponential	$O(1)$	$O(1)$	[GKKP17]
$O(\log n)$	$O(f \log n)$	-	$O(1)$	$\Omega(m)$	[GKKP17]
$(1+\epsilon)\ln n$	$O\left(f \log(n)/\epsilon^5\right)$	-	$O\left(1/\epsilon^4\right)$	$\Omega(m)$	[SU23]
$(1+\epsilon)\ln n$	$O\left(f \log(n)/\epsilon^2\right)$	$O\left(f \log(n)/\epsilon^2\right)$	$O\left(\log(n)/\epsilon^2\right)$	$\Omega(m)$	[SUZ24]
$O(\log n)$	$O(f \log^3(n))$	$O(f \log^3(n))$	$O(\log n)$	$O(\log n)$	**our result**

Table 2: Prior results for dynamic set cover in the high frequency regime. All the algorithms in this table are deterministic. Whenever the guarantee specified by **Q2** *is attained for a certain parameter (i.e., approximation ratio, worst-case update time or worst-case recourse), the concerned entry is marked in blue. Otherwise, the concerned entry is marked in red. If an entry for worst-case update time is left blank, then it implies a trivial worst-case update time of $\tilde{O}(fn)$.*

High Frequency Regime ($O(\log n)$-approximation). Table 2 summarizes the prior results on $O(\log n)$ approximation algorithms for dynamic set cover. This line of work was initiated by [GKKP17], who designed a deterministic dynamic algorithm with $O(\log n)$ approximation ratio, $O(1)$ **amortized** recourse and $O(f \log n)$ **amortized** update time. They also developed a dynamic algorithm with $O(\log n)$ approximation ratio and $O(1)$ worst-case recourse, albeit with an exponential update time. In 2024, [SUZ24] presented the first dy-

[3] We say that a dynamic algorithm has amortized recourse (resp. amortized update time) of $O(\tau)$ iff it incurs a total recourse (resp. time) of $O(t \cdot \tau)$ to process any sequence of t updates starting from an empty input instance. Note that a worst-case recourse (resp. update time) of $O(\tau)$ implies an amortized recourse (resp. update time) of $O(\tau)$, but not vice versa.

[4] Several of the results in Tables 1 and 2 do not explicitly state an amortized recourse bound. The bounds stated in the tables are the best ones we could infer from the algorithms, but even if better amortized bounds were achievable, it would not affect the contributions of the current article.

namic set cover algorithm that achieves $\tilde{O}(f)$ worst-case update time with $O(\log n)$, or even a $(1+\epsilon)\ln(n)$, approximation ratio. Just as in the low frequency regime (and for exactly the same reason), however, the worst-case recourse of the [SUZ24] algorithm is $\Omega(m)$. In sharp contrast, as stated in Theorem 1.2, we *simultaneously* achieve good worst-case recourse and update time, thereby resolving **Q2**.

1.2 Organization of the Rest of the Paper

Both our algorithms for Theorem 1.1 and Theorem 1.2 utilize a common set of high-level ideas. However, because they maintain different approximation ratios, the technical details are quite different. To highlight the key ideas without going deep into the technical details, we organize the paper in three parts: Part I is an extended abstract of our paper that describes the high-level ideas, using our $O(\log n)$-approximation algorithm to instantiate these ideas, Part II gives the proof of Theorem 1.2 with all technical details, and Part III gives the proof of Theorem 1.1 with all technical details.

Within the extended abstract, we define the problem formally and introduce some preliminary definitions in Section 2. Next, in Section 3, we highlight the key ideas behind our algorithm, and how they overcome the main obstacles to obtaining worst-case recourse. We conclude with some future directions in Section 4.

2 Preliminaries

The Fully Dynamic Set Cover Problem. In this problem, we are given a (fixed) universe U of elements and a collection of sets $\mathcal{S}$ that cover the elements in the universe, i.e., $\cup_{s\in\mathcal{S}} s = U$. At any *time-step* $t \geq 0$, there is a set of *live* elements $L(t)$ that have to be covered by the algorithm. Elements that are not live are called *dormant* and denoted by $D(t) := U \setminus L(t)$. Initially, all elements are dormant, i.e. $L(0) = \emptyset$. The set $L(t)$ evolves over time via the following operations:

- if $L(t) = L(t-1) \cup \{e\}$ for some $e \notin L(t-1)$, we say that the element e has been *inserted* at time-step t, and
- if $L(t) = L(t-1) \setminus \{e\}$ for some $e \in L(t-1)$, we say that the element e has been *deleted* at time-step t.

We denote the number of elements to be covered at time-step t by $n(t) := |L(t)|$, and its maximum value over time by $n := \max_t n(t)$. The *frequency* of an element $e \in U$ in $\mathcal{S}$ is the number of sets containing it $f(e) := |\{s \in \mathcal{S} : e \in s\}|$, and the maximum frequency over all elements is denoted $f := \max_{e\in U} f(e)$.

Any solution is a collection of sets in $\mathcal{S}$. At time-step t, a solution $S(t) \subseteq \mathcal{S}$ is said to be *feasible* if it covers all the live elements at that time-step, i.e. if $\cup_{s\in S(t)} s \supseteq L(t)$. The *cost* of a solution is the number of sets in the solution, $c(S(t)) = |S(t)|$. The goal of the algorithm is to maintain a solution of approximately optimal cost. Let $S^*(t)$ denote an optimal solution at time-step t. Then, the algorithm is said to be an α-approximation if for all time-steps t, we have $c(S(t)) \leq \alpha \cdot c(S^*(t))$.

The two important attributes of dynamic set cover algorithms that we study in this paper are *recourse* and *update time*. Recourse refers to the change in the solution from one time-step to the next. Formally, the recourse at time-step t is defined as $\eta_t := |S(t) \setminus S(t-1)| + |S(t-1) \setminus S(t)|$. We say that an algorithm has β-recourse if the recourse at all time-steps is at most β. The update time at time-step t is the running time of the algorithm in this time-step. Similar to recourse, we say that an algorithm has update time T if for all time-steps t, the update time at time-step t is at most T.

Throughout the rest of the paper, we use the symbol $X(t)$ to denote the status of any object X at time-step t.

Hierarchical Solution. Throughout the paper, we consider certain structured set cover solutions that we call *hierarchical* solutions.

Definition 1. A *hierarchical solution* $S \subseteq \mathcal{S}$ is a collection of sets such that each set $s \in S$ is associated with a non-empty *coverage* set $\text{cov}(s) \subseteq s$. We denote $\text{cov}(S) := \cup_{s \in S} \text{cov}(s)$ to be the coverage set of the entire solution S. The coverage sets in the solution are required to be disjoint, i.e. $\text{cov}(s) \cap \text{cov}(s') = \emptyset$ for any distinct $s, s' \in S$. In particular, the coverage set of a feasible solution $S(t)$ at time-step t contains all live elements at that time-step, i.e., $\text{cov}(S(t)) = \cup_{s \in S(t)} \text{cov}(s) \supseteq L(t)$. In addition to the live elements, the coverage sets can contain dormant elements as well.

Each set s in a hierarchical solution S is assigned a non-negative integer *level* denoted $\text{lev}(s)$. The level of a set is inherited by all elements in its coverage set, i.e., $\text{lev}(e) := \text{lev}(\text{cov}^{-1}(e))$. (Note that the level of an element is unambiguous because the coverage sets are disjoint for different sets.) If e is not in the coverage set of a solution S, then its level is undefined.

We require the levels of elements in a hierarchical solution satisfy the following invariant. Intuitively, we would like $\text{lev}(s)$ to be approximately $\log |\text{cov}(s)|$, but for some technical reasons, we relax this to only hold in one direction.

- (Level Invariant) $\forall s \in S$, we have $\text{lev}(s) \leq \log_{1+\varepsilon} |\text{cov}(s)|$.

A hierarchical solution S also assigns a *passive level* $\text{plev}(e)$ for elements. We require the passive levels to be defined for all elements in $\text{cov}(S)$, but it can also be defined on other elements. The passive level $\text{plev}(e)$ of an element e is a dynamic parameter maintained by the algorithm that we will define later. We require the passive levels to satisfy the following invariant that it will always be at least $\text{lev}(e)$ for elements $e \in \text{cov}(S)$.

- (Passive Level Invariant) $\forall e \in \text{cov}(S)$, we have $\text{lev}(e) \leq \text{plev}(e)$. In addition, if $e \in \text{cov}(S)$ is dormant, then we require $\text{plev}(e) = \text{lev}(e)$.

This completes the definition of a hierarchical solution. The algorithm ensures that the maximum level $\ell_{\max}$ is at most $O(\log n)$.

To disambiguate between different hierarchical solutions, we use the superscript S to indicate that a parameter refers to a specific solution S. In particular, $\text{cov}^S(s)$ denotes the set of elements in the coverage set of s in solution S, while $\text{lev}^S(e), \text{plev}^S(e)$ respectively denote the level and passive level of an element $e \in \text{cov}(S)$.

We next introduce some new notation for a hierarchical solution S.

- (Sub-universe) For a level k, the set of live elements at time-step t whose level is at most k is denoted by $L^S_k(t) := \{e \in L(t) : \text{lev}^S(e) \leq k\}$. In particular, we define $L^S(t) := L^S_{\ell_{\max}}(t) = L(t) \cap \text{cov}(S)$.
- (Coverage) The collection of elements in the coverage sets at levels at most k is denoted by $C^S_k := \{e \in \text{cov}(S) : \text{lev}^S(e) \leq k\}$.
- (Active coverage) Of the elements covered at levels at most k, the ones that have passive level larger than k are denoted by $A^S_k := \{e \in C^S_k : \text{plev}^S(e) > k\}$.
- (Passive coverage) The remaining elements covered at levels at most k are denoted $P^S_k := C^S_k \setminus A^S_k$.

Elements in $\text{cov}(S)$ with $\text{plev}(e) = \text{lev}(e)$ cannot appear in A^S_k for any k. We say such elements are *universally passive* in S.

Approximation Bound. To show that the algorithm has a bounded approximation factor, we define the notions of ε-stable and ε-tidy solutions. We define these next:

Definition 2. A solution S is ε-stable at level k iff for every set $s \in \mathcal{S}$, it satisfies $|A_k^S \cap s| < (1+\varepsilon)^{k+1}$. Furthermore, we say that the solution S is ε-stable iff it is ε-stable at every level $k \in [0, \ell_{\max}]$. (Note that the inequality needs to hold for every set in $\mathcal{S}$ and not just those in the solution S.)

Intuitively, the above condition is a form of dual feasibility that we use in our analysis.

Definition 3. A solution S is ε-tidy at a level k if $|P_k^S| \leq \varepsilon \cdot |A_k^S|$, and ε-dirty at level k otherwise. Furthermore, we say that the solution S is ε-tidy iff it is ε-tidy at every level $k \in [0, \ell_{\max}]$.

Intuitively, the above condition is a form of approximation of the dual objective that we use in our analysis.

Throughout the rest of the paper, we use the symbol $\mathrm{OPT}(X)$ to denote the size of minimum set cover for the universe of elements X. The next lemma from [SUZ24] asserts that it suffices to show that a solution satisfies the above two properties in order to establish its approximation factor:

Lemma 2.1 (Lemma 3.1 of [SUZ24])**.** *Suppose S is a hierarchical solution that is ε-tidy and ε-stable at time-step t. Then, $|S| \leq (1+O(\varepsilon)) \ln n \cdot OPT(L(t))$.*

We remark that the constant ε is merely for analysis. We want to call Lemma 2.1 directly, which depends on the parameter ε for measuring tidy and stable properties, as well as defining the level invariant of hierarchical solutions. We simply set $\varepsilon = 0.5$ to define the level invariant, and we will establish the properties for some constant $\leq \varepsilon$ to obtain $O(\log n)$ approximation.

In our analysis of the approximation bound, we will establish the ε-stable and ε-tidy properties of our algorithm and then invoke this lemma. Overall, our goal is an algorithm that achieves $O(\log n)$ approximation ratio, $O(\log n)$ worst case recourse and $O(f \log^3 n)$ worst case update time.

3 Technical Overview

To highlight the main conceptual ideas behind our approach, throughout this section we don't try to optimize the precise polylogarithmic factors in our recourse and update time bounds. Instead, we focus on achieving $O(\log n)$-approximation, $\widetilde{O}(1)$ worst-case recourse and $\widetilde{O}(f)$ worst-case update time. Furthermore, for ease of exposition, we ignore all the low-level details about the data structures maintained by our algorithm.

Roadmap. In Section 3.1, we present a basic algorithmic template, which we refer to as "catch-up greedy", that serves as a very useful subroutine for designing dynamic set cover algorithms with worst-case update time [SUZ24]. Subsequently, in Section 3.2, we present (our interpretation of) the dynamic set cover algorithm of [SUZ24], and explain why it achieves $O(\log n)$-approximation ratio and $\widetilde{O}(f)$ worst-case update time, **but incurs large worst-case recourse**. We omit all the technical proofs (of lemmas, corollaries etc.) in Section 3.1 and Section 3.2, because they follow, either explicitly or implicitly, from the work of [SUZ24].

In Section 3.3, we present our first original insight in this paper, which shows that for our purpose it suffices to design a dynamic set cover algorithm with $O(\log n)$-approximation ratio, $\widetilde{O}(f)$ worst-case update time and $\widetilde{O}(1)$ worst-case **insertion recourse** (which measures how many sets get inserted into the maintained solution per time-step). Motivated by this insight, we develop our overall dynamic algorithms in three stages; they are presented in Section 3.4, Section 3.5 and Section 3.6, respectively, culminating in Theorem 3.11.

3.1 A "Catch-Up Greedy" Algorithm

Imagine a scenario where we wish to run a *static* algorithm for set cover on an input instance. During the execution of the algorithm, however, the input keeps getting updated (via insertions/deletions of elements) on the fly. We want to ensure the property that when the algorithm terminates, it still returns an $O(\log n)$-approximate set cover solution on the current input (which is different from the original input at the time the algorithm started). Our goal in this section is to explain that the standard greedy algorithm for set cover indeed satisfies this property, provided the *speed* at which it runs is sufficiently large compared to the speed at which the input keeps getting updated. We will refer to the greedy algorithm in this setting as **catch-up greedy**. Before elaborating further, we need to set up the stage by introducing the following key notion.

Lazily Updated Hierarchical Solution. We say that a subset $X \subseteq L$ of live elements is **slowly evolving** at time-step t iff $|X(t) \oplus X(t-1)| \leq 1$,[5] and a hierarchical solution $\mathbf{H}$ is **contained within** X iff $L^{\mathbf{H}} \subseteq X$.

Now, consider a hierarchical solution $\mathbf{H}$ and a subset of live elements X such that: (i) X is slowly evolving at time-step t and (ii) $\mathbf{H}$ is contained within X just before time-step t. Then, we say that $\mathbf{H}$ is **lazily updated** at time-step t w.r.t. X iff it undergoes the changes described in Algorithm 1 below.

Algorithm 1: Lazily updating hierarchical solution $\mathbf{H}$ w.r.t. X at time-step t

```
1.1  if X(t) = X(t − 1) then
1.2      H remains the same, i.e., H(t) = H(t − 1)
1.3  else
1.4      Let e := X(t) ⊕ X(t − 1), where ⊕ denotes the symmetric difference between two sets
1.5      if e = X(t − 1) \ X(t) then
1.6          plev^H(e) ← lev^H(e)
1.7      else
1.8          e = X(t) \ X(t − 1)
1.9          if there is at least one set in H that contains e then
1.10             Let s ∈ H be the set containing e with largest lev^H(s)
1.11             Assign e to s, i.e., e is added to cov^H(s) (but we don't change lev^H(s))
1.12             Set plev^H(e) := lev^H(e) := lev^H(s)
1.13         else
1.14             Let s be any set that contains e
1.15             Add s to H at level 0
1.16             Set cov^H(s) := {e}
1.17             Set plev^H(e) := lev^H(e) := lev^H(s) = 0
```

Catch Up Greedy. Suppose that we are given a **starting time-step** t, a level $k \in [0, \ell_{\max}]$, and a subset $X \subseteq L$ of live elements that is slowly evolving from time-step t onward. For our purpose, the set X will always correspond to a subset of live elements in some hierarchical solution. Accordingly, we let $\text{plev}^X(e)$ denote the passive level of an element $e \in X$ in the concerned hierarchical solution that generates X. In addition, we are given access to another hierarchical solution $\mathbf{H}$ that is initially (i.e., in the beginning of time-step t) empty. We refer to (t, k, X) as the **source**, and $\mathbf{H}$ as the **target** of the following computational task:

We have to ensure that at a not-too-distant future time-step t^{end} of our own choosing, the hierarchical solution $\mathbf{H}(t^{\text{end}})$ is a feasible and $O(\log n)$-approximate set cover on $X(t^{\text{end}})$, and additionally that $\text{lev}^{\mathbf{H}}(e) \leq k + 1$ for

[5] The symbol $\oplus$ denotes the symmetric difference between two sets.

all elements $e \in \mathrm{cov}(\mathbf{H})$. Moreover, we can only perform $\widetilde{O}(f)$ units of computation per time-step, i.e., the worst-case update time of the procedure is $\widetilde{O}(f)$.

Remark. For technical reasons, we need to consider the parameter k as part of our target. In the ensuing discussions, the reader might find it more intuitive to think of the scenario where $k = \ell_{\max}$, in which case we trivially have $\mathrm{lev}^{\mathbf{H}}(e) \leq k+1$ for all elements $e \in \mathrm{cov}(\mathbf{H})$.

Algorithm 2: A catch up greedy algorithm with source (t, k, X) and target $\mathbf{H}$

2.1 $p_k \leftarrow k+1$
2.2 **while** *the uncovered sub-universe* $X \setminus \mathrm{cov}(\mathbf{H})$ *is nonempty* **do**
2.3 Find the set s that covers the maximum number of uncovered elements in the sub-universe, i.e.

$$s = \arg\max_{s \in \mathcal{S}} |s \cap (X \setminus \mathrm{cov}(\mathbf{H}))|.$$

2.4 Add s to $\mathbf{H}$, with $\mathrm{cov}^{\mathbf{H}}(s) \leftarrow s \cap (X \setminus \mathrm{cov}(\mathbf{H}))$.
2.5 $\mathrm{lev}^{\mathbf{H}}(s) \leftarrow \min\left\{p_k, \lfloor \log_{1+\varepsilon} |\mathrm{cov}(s)| \rfloor\right\}$.
2.6 **for** *For each* $e \in \mathrm{cov}^{\mathbf{H}}(s)$ **do**
2.7 $\mathrm{lev}^{\mathbf{H}}(e) \leftarrow \mathrm{lev}^{\mathbf{H}}(s)$.
2.8 **if** $\mathrm{plev}^{\mathbf{H}}(e)$ *is not already defined* **then**
2.9 $\mathrm{plev}^{\mathbf{H}}(e) \leftarrow \max\left\{k+1, \mathrm{plev}^{X}(e)\right\}$
2.10 Update $p_k \leftarrow \min\left\{p_k, \mathrm{lev}^{\mathbf{H}}(s)\right\}$.

We wish to run the greedy algorithm for set cover on the input X, as specified in Algorithm 2, but the issue is that the input X itself is evolving, albeit slowly, over time. To cope with this challenge, we take a very natural approach: We ensure that $\mathbf{H}$ is contained within X from time-step t onward (this is trivially true at time-step t, when $\mathbf{H}$ is empty). Moreover, in the beginning of time-step t, we initialize a **copy pointer** $p_k \leftarrow k+1$, which would henceforth keep monotonically decreasing over time. Subsequently, during each time-step $t' \geq t$, we perform the following operations.

- First, we update $\mathbf{H}$ as per the procedure in Algorithm 3.
- Then, we execute the next $\log n$ steps[6] of computation as specified by Algorithm 2. This steps only measures the operations to process elements in the inner for loop, which is the bottleneck of Algorithm 2.

The algorithm terminates when $\mathbf{H}$ becomes a feasible hierarchical solution w.r.t. the input X, i.e., when $L^{\mathbf{H}} = X$.

We now observe a few basic properties.

- The algorithm covers $\log n$ elements from X per time-step. We refer to this as the *speed* of the catch up greedy algorithm. (In contrast, the set X itself changes by at most one element per time-step.)
- It assigns sets in $\mathbf{H}$ to levels that decrease monotonically over time. Specifically, suppose that the sets s_1 and s_2 get added to $\mathbf{H}$ at time-steps t_1 and t_2, respectively. If $t_1 < t_2$, then $\mathrm{lev}^{\mathbf{H}}(s_1) \geq \mathrm{lev}^{\mathbf{H}}(s_2)$.
- It essentially implements the static greedy algorithm on X, which picks the sets in decreasing order of their marginal coverages, with the computation being spread out across a sequence of time-steps.

[6] All logarithms are base 2 in default.

Algorithm 3: Updating $\mathbf{H}$ at time-step t', while running catch-up greedy with source (t, k, X)

3.1 **if** $X(t') = X(t'-1)$ **then**
3.2 $\mathbf{H}$ remains the same, i.e., $\mathbf{H}(t') = \mathbf{H}(t'-1)$
3.3 **else**
3.4 Let $e := X(t') \oplus X(t'-1)$, where $\oplus$ denotes the symmetric difference between two sets
3.5 **if** $e = X(t'-1) \setminus X(t')$ **then**
3.6 **if** $e \in \mathrm{cov}(\mathbf{H})$ **then**
3.7 $\mathrm{plev}^{\mathbf{H}}(e) \leftarrow \mathrm{lev}^{\mathbf{H}}(e)$
3.8 **else**
3.9 $e = X(t') \setminus X(t'-1)$
3.10 **if** *there is at least one set in* $\mathbf{H}$ *that contains* e **then**
3.11 Let $s \in \mathbf{H}$ be the set containing e with largest $\mathrm{lev}^{\mathbf{H}}(s)$
3.12 Assign e to s, i.e., e is added to $\mathrm{cov}^{\mathbf{H}}(s)$ (but we don't change $\mathrm{lev}^{\mathbf{H}}(s)$)
3.13 Set $\mathrm{plev}^{\mathbf{H}}(e) := \mathrm{lev}^{\mathbf{H}}(e) := \mathrm{lev}^{\mathbf{H}}(s)$
3.14 **else**
3.15 Set $\mathrm{plev}^{\mathbf{H}}(e) \leftarrow p_k$

- The algorithm can be implemented in $\widetilde{O}(f)$ worst-case update time. The bottleneck is due to maintaining some data structure for number of uncovered elements in each set. In each time-step, the algorithm processes $\widetilde{O}(1)$ elements, and each element is incident to at most f sets.
- $\mathrm{lev}^{\mathbf{H}}(e) \leq k+1$ for all elements $e \in \mathrm{cov}(\mathbf{H})$.

Extension of the Target Hierarchical Solution H. Consider any time-step t' during the lifetime of a catch-up greedy algorithm, with source (t, k, X) and target $\mathbf{H}$, as described above. We use the symbol $\widetilde{\mathbf{H}}(t')$ to define another hierarchical solution, which we refer to as the **extension of** $\mathbf{H}$ at time-step t', as follows: $\widetilde{\mathbf{H}}$ is the hierarchical solution obtained by continuing to run Algorithm 2 at the current time-step t' (i.e., in a static setting, without any element insertion or deletion) until Algorithm 2 completes execution. It is easy to verify that $\mathbf{H}(t') \subseteq \widetilde{\mathbf{H}}(t')$ at every time-step t', and that $\mathbf{H} = \widetilde{\mathbf{H}}$ when the catch-up greedy algorithm terminates. We will crucially use the notion of such an **extended hierarchical solution** $\widetilde{\mathbf{H}}$ in our analysis.

The lemma below summarizes the key guarantees of the catch up greedy algorithm, and it follows as a direct consequence of the properties summarized above. **Throughout this technical overview section, we use ϵ as an arbitrarily small constant.**

Lemma 3.1. *The catch up greedy algorithm described above terminates at time-step* $t^{\mathrm{end}} \leq t + \frac{2}{\log n} \cdot |X(t)|$. *Furthermore, at time-step* $t \in \left[t, t^{\mathrm{end}}\right]$, *the following conditions hold.*

1. $\widetilde{\mathbf{H}}$ *is* ϵ*-tidy at every level* $k' \in [0, k]$.
2. $\widetilde{\mathbf{H}}$ *is* ϵ*-stable.*
3. $|\mathbf{H}| \leq |\widetilde{\mathbf{H}}| = O(\log n) \cdot \mathrm{OPT}(X)$, *where* $\mathrm{OPT}(X)$ *denotes the size of the minimum set cover on input* X.

3.2 Achieving Good Worst-Case Update Time: An Overview of [SUZ24] Algorithm

We now summarize the main insight behind how [SUZ24] achieves $\widetilde{O}(f)$ worst case update time for maintaining a $O(\log n)$-approximate minimum set cover, albeit with a trivial worst case recourse bound of $O(m)$

where m is the number of sets. In a bit more detail, the algorithm maintains two different sets of hierarchical solutions, some of which correspond to different levels. The solutions are:

- a **foreground** solution $\mathbf{F}$, and
- a set of $\ell_{\max} + 1$ **background** solutions $\mathbf{B}_k$ for levels $k \in [0, \ell_{\max}]$.

The output of the dynamic algorithm comprises only of the foreground solution, which is feasible for the set of live elements $L(t)$ at each time-step t (i.e., $L^{\mathbf{F}}(t) = L(t)$). Moreover, the foreground solution $\mathbf{F}$ gets lazily updated w.r.t. L in the beginning of every time-step. Accordingly, the subset of live elements $L^{\mathbf{F}}_k$ is also slowly evolving, for all $k \in [0, \ell_{\max}]$.

We next focus on the background solution $\mathbf{B}_k$ for a given level $k \in [0, \ell_{\max}]$. This background solution evolves in **phases**, and is maintained by a subroutine that we refer to as **background thread** T_k. Each phase lasts for a sequence of consecutive time-steps. Let us consider a specific phase that starts at (say) time-step t^{start}. In the beginning of time-step t^{start}, the hierarchical solution $\mathbf{B}_k$ is empty. Throughout the duration of the phase, we run a catch up greedy algorithm with source $(t, k, L^{\mathbf{F}}_k)$ and target $\mathbf{B}_k$. The phase ends (say) at time-step $t^{\text{end}} \geq t^{\text{start}}$, because of one of the following two reasons.

1. The catch up greedy algorithm for $\mathbf{B}_k$ completes execution at time-step t^{end}. We refer to this event as a **normal termination** of the background thread T_k. In this event, we first **switch** the background solution $\mathbf{B}_k$ to the foreground, by removing all sets in $\mathbf{F}$ at levels $\leq k$ and replacing them with all the sets in the corresponding levels in $\mathbf{B}_k$. Additionally, $\mathbf{B}_k$ might have sets at level $k+1$, which are now merged into level $k+1$ in the foreground solution $\mathbf{F}$. Next, for each level $k' \in [0, k-1]$, we reset $\mathbf{B}_{k'}$ to be empty and terminate the background thread $T_{k'}$. We say that the thread T_k **naturally terminates**, whereas for each $k' \in [0, k-1]$ the thread $T_{k'}$ **abnormally terminates**. From the next time-step $t^{\text{end}}+1$ onward, each of these threads $T_0, \cdots, T_k$ **restarts** a new phase.
2. A higher background thread $T_{k'}$, with $k' \in [k+1, \ell_{\max}]$, normally terminates at time-step t^{end}, and this leads to an abnormal termination of the thread T_k.

The above discussion makes it clear that we need a tie-breaking rule, for the scenario when we have a collection $\mathcal{T}$ of multiple threads that complete execution at the same time-step. In such a scenario, we select the thread $T_k \in \mathcal{T}$ with the largest index k for normal termination, which in turn leads to abnormal terminations of all threads in $\{T_{k'} : k' < k\} \supseteq \mathcal{T} \setminus \{T_k\}$.

Using appropriate pointers and data structures that support merging, switching a background thread to the foreground can be handled in $\widetilde{O}(1)$ time. **Such a switch, however, might potentially lead to $\Omega(m)$ worst-case recourse**, m being the total number of sets in the input. Furthermore, since each background thread runs in $\widetilde{O}(f)$ worst-case update time, and there are $\ell_{\max} + 1 = O(\log n)$ background threads, we conclude that:

Corollary 3.2. *The algorithm described above has an overall worst-case update time of $\widetilde{O}(f)$.*

Finally, using Lemma 3.1, we can argue that if a background thread T_k starts a new phase at (say) time-step t, then that phase ends (due to either a normal or an abnormal termination) by time-step $t^\star \leq t + \frac{2}{\log n} \cdot \left|L^{\mathbf{F}}_k(t)\right|$. This observation, along with Lemma 3.1, in turn gives us the following corollary.

Corollary 3.3. *Consider any background thread T_k.*

1. *If T_k enters a new phase at time-step t, then that phase lasts for at most $\frac{2}{\log n} \cdot \left|L^{\mathbf{F}}_k(t)\right|$ time-steps.*

2. *The extended hierarchical solution* $\widetilde{\mathbf{B}}_k$ *is always* ϵ*-tidy at every level* $k' \in [0, k]$*, and* ϵ*-stable.*

3. *We always have* $|\mathbf{B}_k| = O(\log n) \cdot \mathrm{OPT}(L_k^{\mathbf{F}})$*, where* $\mathrm{OPT}(L_k^{\mathbf{F}})$ *is the minimum set-cover size on input* $L_k^{\mathbf{F}}$*.*

Now, Item 2 of Corollary 3.3 guarantees that immediately after switching the background solution $\mathbf{B}_k$ to the foreground, the foreground solution $\mathbf{F}$ is ϵ-tidy at every level $k' \in [0, k]$ and ϵ-stable at every level $k' \in [0, k]$. In contrast, Item 1 of Corollary 3.3 guarantees that within every $\frac{2}{\log n} \cdot \left|L_k^{\mathbf{F}}(t)\right| \leq \epsilon \cdot \left|L_k^{\mathbf{F}}(t)\right|$ time-steps (the inequality holds because ϵ is an absolute constant), some background solution $\mathbf{B}_{k'}$ with $k' \geq k$ is switched to the foreground. Since the interval between any two consecutive switches is so short, the foreground solution $\mathbf{F}$ only accumulates a small number of passive elements within that interval. In particular, we can show that $\mathbf{F}$ always remain 2ϵ-tidy for every level $k \in [0, \ell_{\max}]$, as it keeps getting lazily updated w.r.t. the set of live elements L. Furthermore, the foreground solutions $\mathbf{F}$ can *not* cease to be ϵ-stable at some level k because of a lazy update. The preceding discussion leads to the corollary below.

Corollary 3.4. *The foreground solution* $\mathbf{F}$ *is* 2ϵ*-tidy and* ϵ*-stable at every time-step.*

The main result in [SUZ24] now follows from Corollary 3.2, Lemma 2.1 and Corollary 3.4.

Theorem 3.5. *There is a dynamic* $O(\log n)$*-approximation for set cover with* $\widetilde{O}(f)$ *worst case update time.*

3.3 A New Reduction from Worst-Case Insertion Recourse

Say that a dynamic set cover algorithm incurs one unit of **insertion recourse** (resp. **deletion recourse**) whenever a set gets inserted into (resp. deleted from) its maintained solution. The overall recourse of the algorithm is the sum of its insertion recourse and deletion recourse. It is easy to verify that a worst-case bound on insertion recourse immediately implies the same bound on deletion recourse, but only in an amortized sense. Our first insight in this paper is the observation that in a black-box manner, we can transform any $O(\log n)$-approximate dynamic (unweighted) set cover algorithm with worst-case $O(\log n)$ insertion recourse into another dynamic algorithm with $O(\log n)$-approximation ratio and $O(\log n)$ worst-case overall recourse.

This insight is summarized in Lemma 3.6 below. Instead of arguing specifically about our algorithm, the lemma prove a more general property about de-amortizing deletion recourse. Consider any online optimization problem where a solution comprises a set of objects, and the goal is to minimize the number of objects subject to a feasibility constraint that changes online. Let us call such a problem an online unweighted minimization problem. We prove the following for any such problem (for our purpose, set $\alpha = O(\log n)$, $\beta = \widetilde{O}(1)$ and $\delta = 1$ in the lemma statement).

Lemma 3.6. *Consider an online unweighted minimization problem, where the size of the optimal solution changes by at most* δ *in each time-step. Suppose there exists an algorithm* $\mathcal{A}$ *that maintains an* α*-approximate solution and has worst-case insertion recourse at most* β*, for some parameters* $\alpha, \beta \geq 1$*. (The worst-case deletion recourse of algorithm* $\mathcal{A}$ *can be arbitrarily large.) Then, there exists an alternative algorithm* $\mathcal{A}'$ *that also maintains an* α*-approximate solution and has worst-case recourse (both insertion and deletion) at most* $2\beta + \delta\alpha$*.*

Proof. We define algorithm $\mathcal{A}'$ to simulate $\mathcal{A}$, but with the following modification: Whenever $\mathcal{A}$ deletes objects from its solution, move those objects to a garbage set instead of deleting them immediately in $\mathcal{A}'$. The garbage set is added to the output of $\mathcal{A}$ to form the output of $\mathcal{A}'$. Moreover, $\mathcal{A}'$ performs an additional garbage removal operation at the end of the processing in each time-step: delete any $\beta + \delta\alpha$ objects (or all objects if fewer) from the garbage set. The recourse bound for $\mathcal{A}'$ is straightforward from this description.

We bound the approximation factor of $\mathcal{A}'$ by induction over time. For the base case, note that whenever the garbage set is empty (which is the case at initiation), the solutions of $\mathcal{A}$ and $\mathcal{A}'$ are identical. Hence, at

these time-steps, $\mathcal{A}'$ is an α-approximation. The remaining case is when the garbage set is non-empty at the end of a time-step t. In this case, $\mathcal{A}'$ must have deleted $\beta + \delta\alpha$ objects from the garbage set in time-step t. Since $\mathcal{A}'$ has insertion recourse at most β, the solution size at the end of time-step t decreases by at least $\delta\alpha$ compared to that at the end of time-step $t-1$. Combined with the induction hypothesis that the solution was an α-approximation at the end of $t-1$, and the assumption that optimal solution size can only change by at most δ in a time-step, we conclude that the solution is still an α-approximation at the end of time-step t. □

In the full version (see Lemma 6.30), we show that for our specific dynamic algorithm, the reduction guaranteed by Lemma 3.6 can be implemented while incurring only a $\widetilde{O}(1)$ factor overhead in the worst-case update time. Accordingly, henceforth we focus on designing a dynamic set cover algorithm with $O(\log n)$ approximation ratio, $\widetilde{O}(f)$ worst-case update time and $\widetilde{O}(1)$ worst-case **insertion recourse**.

3.4 Our Dynamic Algorithm: Take I

Armed with Lemma 3.6, we propose a very natural modification of the algorithm from Section 3.2, as described below. As we will soon see, the issue with this approach is that it leads to an approximation ratio of $O(\log^2 n)$.

The algorithm maintains three different sets of solutions, some of which correspond to different levels:

- a **foreground** solution $\mathbf{F}$, and
- a set of $\ell_{\max} + 1$ **background** solutions $\mathbf{B}_k$ for levels $k \in [0, \ell_{\max}]$.
- a set of $\ell_{\max} + 1$ **buffer** solutions $\mathbf{R}_k$ for levels $k \in [0, \ell_{\max}]$.

The foreground and the background solutions evolve in exactly the same manner as in Section 3.2. As opposed to the foreground and background solutions (which are hierarchical solutions), each buffer solution $\mathbf{R}_k$ is simply a collection of sets. The buffer solutions are used as intermediaries between the background and foreground solutions, and they help ensure good worst-case insertion recourse. The output maintained by the algorithm, against which we measure its approximation ratio and recourse, is the union of the foreground solution and all the buffer solutions.

Specifically, a background thread T_k ensures that $\mathbf{B}_k = \mathbf{R}_k$ at every time-step. In other words, within a given phase, whenever the thread T_k adds a set s to $\mathbf{B}_k$, it copies the same set s to $\mathbf{R}_k$. Since the thread T_k works at a speed of $\log n$, at most $\log n$ sets get added to $\mathbf{R}_k$ during any given time-step. Since $\ell_{\max} = \widetilde{O}(1)$, the worst-case insertion recourse of the maintained solution $\mathbf{F} \bigcup_{k \in [0, \ell_{\max}]} \mathbf{R}_k$ is also $\widetilde{O}(1)$. Crucially, note that whenever a background solution $\mathbf{B}_k$ is switched into the foreground or a background thread T_k abnormally terminates, this leads to *zero* insertion recourse; although the deletion recourse due to such an event can be $\Omega(m)$, where m is the total number of sets. (Whenever a thread T_k terminates, normally or abnormally, we reset *both* its background solution $\mathbf{B}_k$ and its buffer solution $\mathbf{R}_k$ to being empty.)

Clearly, the worst-case update time of this algorithm is $\widetilde{O}(f)$.

As we alluded to it before, the issue with this approach is that the approximation ratio of the resulting algorithm is $O(\log^2 n)$, instead of $O(\log n)$. This is because $|\mathbf{R}_k| = |\mathbf{B}_k| = O(\log n) \cdot \text{OPT}(L_k^{\mathbf{F}}) = O(\log n) \cdot \text{OPT}(L)$ for each buffer solution $\mathbf{R}_k$ (see Item 3 of Corollary 3.3), and $|\mathbf{F}| = O(\log n) \cdot \text{OPT}(L)$ (see Theorem 3.5). Thus, we get $|\mathbf{F}| + \sum_{k=0}^{\ell_{\max}} |\mathbf{R}_k| = O(\log^2 n) \cdot \text{OPT}(L)$. This is a fundamental challenge we need to overcome.

To address this challenge, our first attempt would be to modify the algorithm so as to ensure that at each time-step, only a subset of buffer solutions are nonempty, and the sizes (i.e., the number of sets) of the nonempty buffer solutions form a decreasing geometric series. This, in turn, necessitates splitting up the lifetime of a background thread into three different phases. In Section 3.5, we explain our new framework in more detail.

3.5 Our Dynamic Algorithm: Take II

As before, we maintain three different sets of solutions, some of which correspond to different levels:

- a **foreground** solution $\mathbf{F}$, and
- a set of $\ell_{\max} + 1$ **background** solutions $\mathbf{B}_k$ for levels $k \in [0, \ell_{\max}]$.
- a set of $\ell_{\max} + 1$ **buffer** solutions $\mathbf{R}_k$ for levels $k \in [0, \ell_{\max}]$.

The foreground and background solutions are hierarchical solutions, whereas each buffer solution $\mathbf{R}_k$ is simply a collection of sets. The output maintained by the algorithm is given by $\mathbf{F} \bigcup_{k \in [0, \ell_{\max}]} \mathbf{R}_k$.

The foreground solution is always feasible for the live elements, i.e., $L^{\mathbf{F}}(t) = L(t)$ at each time-step t. Furthermore, in the beginning of each time-step, we lazily update $\mathbf{F}$ w.r.t. L. Accordingly, the subset of live elements $L^{\mathbf{F}}_k$ is also slowly evolving, for all $k \in [0, \ell_{\max}]$.

Next, consider any level $k \in [0, \ell_{\max}]$, and focus on the background solution $\mathbf{B}_k$ maintained by the background thread T_k. The lifetime of this thread consists of three phases – a **computation phase**, a **suspension phase** and a **copy phase** – one after another. Each phase lasts for a sequence of consecutive time-steps.

I: Computation Phase. Suppose that the thread T_k enters computation phase at (say) time-step t^{comp}. At the start of time-step t^{comp}, both the background solution $\mathbf{B}_k$ and the buffer solution $\mathbf{R}_k$ are empty. In fact, the buffer solution $\mathbf{R}_k$ will remain empty throughout the duration of the computation phase.

Now, the thread T_k runs a catch-up greedy algorithm with source $(t^{\text{comp}}, k, L^{\mathbf{F}}_k)$ and target $\mathbf{B}_k$. Let $t^{\text{sus}} - 1$ be the time-step at which the catch-up greedy algorithm completes execution. Then, the computation phase also ends at time-step $t^{\text{sus}} - 1$, and the thread T_k enters suspension phase from the next time-step t^{sus}. Before entering the suspension phase, the thread T_k takes a snapshot of the solution size $|\mathbf{B}_k|$ and denotes it by τ^{sus}_k.

II: Suspension Phase. Throughout the duration of the suspension phase, the thread T_k lazily updates $\mathbf{B}_k$ w.r.t. $L^{\mathbf{F}}_k$, and the buffer solution $\mathbf{R}_k$ continues to remain empty.

During each time-step, the algorithm runs a *scheduler* to determine which (if any) of the threads currently in suspension phase will enter copy phase. The scheduler for time-step t implements Algorithm 4 below.

Algorithm 4: Transitioning background threads from suspension to copy phase

```
4.1 τ ← (1/2) · min_j τ^sus_j, where the minimum is taken over all the threads currently in copy phase
4.2 for k = ℓ_max down to 0 do
4.3     if τ^sus_k ≤ τ then
4.4         The thread T_k will enter copy phase from the beginning of the next time-step t + 1
4.5         Update τ ← (1/2) · τ^sus_k
```

III: Copy Phase. Throughout the duration of the copy phase, the thread T_k lazily updates $\mathbf{B}_k$ w.r.t. $L^{\mathbf{F}}_k$. Furthermore, at each time-step during its copy phase, the thread T_k copies $\log n$ sets from $\mathbf{B}_k$ into the buffer solution $\mathbf{R}_k$. When all the sets from $\mathbf{B}_k$ have been copied into $\mathbf{R}_k$, the thread T_k completes execution.

The algorithm now performs the following procedure at the end of each time-step t. Let $Q(t) \subseteq [0, \ell_{\max}]$ denote the collection of indices of all those background threads that have just completed execution, and let $j^\star \leftarrow \max\{Q(t)\}$. We **switch** the background solution $\mathbf{B}_{j^\star}$ to the foreground. Moreover, for each $k \in [0, j^\star]$, the background solution $\mathbf{B}_k$ and the buffer solution $\mathbf{R}_k$ are both reset to being empty at the end of time-step

t, and the thread T_k restarts its computation phase from the next time-step $t+1$. We say that the thread $T_{j^\star}$ **normally terminates**, and every thread $k \in [0, j^\star - 1]$ **abnormally terminates** at time-step t.

A key conceptual contribution in this paper to prove the lemma below, which bounds for how long can a background thread remain waiting in the suspension phase.

Lemma 3.7. *Suppose that a background thread T_k enters suspension phase from the beginning of time-step t_k^{sus}. Then, the thread T_k terminates (either naturally or abnormally) at or before time-step $t_k^{\text{sus}} + o(1) \cdot \tau_k^{\text{sus}}$.*

3.5.1 Proof (Sketch) of Lemma 3.7

For the sake of analysis, we consider a modified version of the algorithm, where a background thread T_k **aborts** after waiting in suspension phase for $\frac{8}{\log n} \cdot \tau_k^{\text{sus}}$ time-steps. Specifically, suppose that the thread T_k enters suspension phase at time-step t_k^{sus}. If T_k is still at suspension phase at (say) time-step $t_k^{\text{end}} = t_k^{\text{sus}} + \frac{8}{\log n} \cdot \tau_k^{\text{sus}}$, then from the beginning of the next time-step $t_k^{\text{end}} + 1$, the thread T_k restarts a computation phase (after resetting $\mathbf{B}_k$ to being empty). We will show that a background thread actually never gets aborted in this manner (see Claim 3.10), and so this modified version of the algorithm is equivalent to the original description. The only reason for sticking with this version is that it makes the proof of Lemma 3.7 simpler and more intuitive.

We start by upper bounding the duration of a copy phase in the claim below.

Claim 3.8. *A background thread T_k remains in copy phase for less than $\frac{2}{\log n} \cdot \tau_k^{\text{sus}}$ time-steps.*

Proof. (Sketch) Suppose that the thread enters suspension phase and copy phase at the start of time-steps t_k^{sus} and $t_k^{\text{copy}} > t_k^{\text{sus}}$, respectively. Since T_k was *not* aborted while waiting in the suspension phase, we have

$$t_k^{\text{sus}} < t_k^{\text{copy}} < t_k^{\text{sus}} + \frac{8}{\log n} \cdot \tau_k^{\text{sus}}. \tag{1}$$

Throughout the suspension phase, the background solution $\mathbf{B}_k$ gets lazily updated w.r.t. L_k^{F}, and so the solution size $|\mathbf{B}_k|$ grows at most by an additive one per time-step. Thus, at the start of the copy phase, we have

$$\left|\mathbf{B}_k(t_k^{\text{copy}})\right| \leq \left|\mathbf{B}_k(t_k^{\text{sus}})\right| + (t_k^{\text{copy}} - t_k^{\text{sus}}) < \left(1 + \frac{8}{\log n}\right) \cdot \tau_k^{\text{sus}}. \tag{2}$$

Next, throughout the copy phase, the background solution $\mathbf{B}_k$ continues to get lazily updated w.r.t. L_k^{F}, and so the solution size $|\mathbf{B}_k|$ continues to grow at most by an additive one per time-step. Simultaneously, the thread T_k keeps copying $\log n$ sets from $\mathbf{B}_k$ into the buffer solution $\mathbf{R}_k$ per time-step, and the copy phase ends when all the sets from $\mathbf{B}_k$ have been copied into $\mathbf{R}_k$ (or, if the thread T_k gets abnormally terminated before that). Since there are $< \left(1 + \frac{8}{\log n}\right) \cdot \tau_k^{\text{sus}}$ sets in $\mathbf{B}_k$ at the start of the copy phase (see Equation (2)), the copy phase lasts for at most $\frac{\left(1+\frac{8}{\log n}\right) \cdot \tau_k^{\text{sus}}}{(\log n) - 1} < \frac{2}{\log n} \cdot \tau_k^{\text{sus}}$ time-steps. This concludes the proof. □

To highlight the main idea behind the proof of Lemma 3.7, we now make a simplifying assumption.

Assumption 3.9. *Let $Q(t)$ be the collection of the indices of the threads that are in either suspension or copy phase at the start of time-step t. Then, at every time-step t, we have $\tau_j^{\text{sus}} \geq \tau_k^{\text{sus}}$ for all $j, k \in Q(t)$ with $j > k$.*

In other words, Assumption 3.9 enforces a certain *monotonicity* on the τ_k^{sus} values of the relevant threads, guaranteeing that threads corresponding to lower levels have smaller τ_k^{sus} values.

Remark. A priori, Assumption 3.9 might intuitively seem to be true. After all, a thread T_k is responsible for the elements in the foreground solution that are at levels $\leq k$, which is a subset of the elements under the purview of the next thread T_{k+1}. Accordingly, we expect that the sizes of the background solutions $\mathbf{B}_k$ would be monotonically non-decreasing with the index k. Since τ_k^{sus} is a snapshot of the size of $\mathbf{B}_k$ at the moment the thread T_k enters suspension phase, this provides an intuitive justification for Assumption 3.9.

Unfortunately, however, Assumption 3.9 is not necessarily true. This is because:

1. For two distinct threads T_j and T_k, the values τ_j^{sus} and τ_k^{sus} refer to two different snapshots in time (corresponding to the possibly different time-steps at which T_j and T_k enter suspension phase).
2. The catch-up greedy algorithms run by T_j and T_k, which eventually determine the values τ_j^{sus} and τ_k^{sus}, are *not* synchronized with each other.

We provide a complete, formal proof of Lemma 3.7 in Section 6.2 (see Lemma 6.12). In this technical overview section, however, we rely on Assumption 3.9 while presenting a proof-sketch of Lemma 3.7; since our goal here is only to showcase the high-level intuitions behind our algorithm and analysis.

Claim 3.10. *A background thread T_k never gets aborted for waiting too long in the suspension phase.*

Proof. (Sketch) Suppose that the thread T_k enters suspension phase at time-step t_k^{sus}. Consider any time-step $t \geq t_k^{\text{sus}}$ in the suspension phase, and suppose that the thread T_k fails to enter copy phase during time-step t. This happens iff one of the following two (mutually exclusive) scenarios hold.

1. There is another background thread T_j, with $j > k$ and $\tau_j^{\text{sus}} < 2 \cdot \tau_k^{\text{sus}}$, that is either in copy phase at the start of time-step t, or Algorithm 4 decides that T_j will enter copy phase from the next time-step $t+1$. We say that the thread T_k is **blocked-from-above** by the thread T_j at time-step t.
2. The thread T_k is *not* blocked-from-above at time-step t. However, there is another background thread T_j, with $j < k$ and $\tau_j^{\text{sus}} \leq \tau_k^{\text{sus}}$ (as per Assumption 3.9), that is in copy phase at the start of time-step t. Furthermore, at the start of time-step t, no other thread $T_{j'}$ with $j' \in [j+1, k-1]$ is in copy phase. We say that the thread T_k is **blocked-from-below** by the thread T_j at time-step t.

Now, we consider two possible cases that might occur if T_k continues to remain in the suspension phase.

Case 1. The thread T_k is blocked-from-above by some other thread T_j at some time-step $t \in \left[t_k^{\text{sus}}, \frac{4}{\log n} \cdot t_k^{\text{sus}}\right]$. This means that $j > k$, $\tau_j^{\text{sus}} < 2 \cdot \tau_k^{\text{sus}}$, and either T_j is already in copy phase at the start of time-step t, or T_j enters copy phase at the start of the next time-step $t+1$. Accordingly, by Claim 3.8, the thread T_j terminates by time-step $t + \frac{2}{\log n} \cdot \tau_j^{\text{sus}} < t + \frac{4}{\log n} \cdot \tau_k^{\text{sus}} \leq t_k^{\text{sus}} + \frac{8}{\log n} \cdot t_k^{\text{sus}}$. Since $j > k$, whenever T_j terminates, this leads to an abnormal termination of the thread T_k. Thus, we conclude that in this case, the thread T_k is terminated before waiting for $\frac{8}{\log n} \cdot \tau_k^{\text{sus}}$ time-steps in the suspension phase. In other words, the thread T_k does *not* get aborted while waiting in the suspension phase.

Case 2. The thread T_k is not *blocked-from-above at any time-step $t \in \left[t_k^{\text{sus}}, \frac{4}{\log n} \cdot t_k^{\text{sus}}\right]$.* Here, the thread T_k must be blocked-from-below by some thread T_j, with $j < k$ and $\tau_j^{\text{sus}} \leq \tau_k^{\text{sus}}$ (see Assumption 3.9), at time-step t_k^{sus}. Now, by Claim 3.8, the thread T_j terminates at some time-step $t_j < t_k^{\text{sus}} + \frac{2}{\log n} \cdot \tau_j^{\text{sus}} \leq t_k^{\text{sus}} + \frac{2}{\log n} \cdot \tau_k^{\text{sus}}$. This, in turn, leads to all threads $T_{j'}$ with $j' \in [0, j-1]$ getting abnormally terminated at the same time-step t_j.

Note that due to Assumption 3.9, no thread $T_{j''}$ with $j'' > j$ enters the copy phase during the time-interval $[t_k^{\text{sus}}, t_j]$. Moreover, since $t_j + 1 \in \left[t_k^{\text{sus}}, \frac{2}{\log n} \cdot t_k^{\text{sus}}\right]$, the thread T_k is neither blocked-from-above nor blocked-from-below at time-step $t_j + 1$. Accordingly, the thread T_k must enter copy phase at time-step $t_j + 1$. In other words, the thread T_k does *not* get aborted while waiting in the suspension phase. □

Proof of Lemma 3.7. Consider any background thread T_k that enters suspension phase at time-step t_k^{sus}. By Claim 3.10, the thread T_k never gets aborted while waiting in the suspension phase. Thus, from the description of our modified version of the algorithm, it follows that T_k either gets abnormally terminated or enters copy phase within the next $\frac{8}{\log n} \cdot \tau_k^{\text{sus}}$ time-steps. Further, if T_k enters copy phase, then by Claim 3.8 it remains in copy phase for less than $\frac{2}{\log n} \cdot \tau_k^{\text{sus}}$ time-steps. Overall, from the preceding discussion we infer that T_k terminates (either naturally or abnormally) at or before time-step $t_k^{\text{sus}} + \frac{8}{\log n} \cdot \tau_k^{\text{sus}} + \frac{2}{\log n} \cdot \tau_k^{\text{sus}} = t_k^{\text{sus}} + o(1) \cdot \tau_k^{\text{sus}}$.

3.6 Our Dynamic Algorithm: Take III

In this section, we outline the final version of our algorithm, and sketch the proof of the following theorem.

Theorem 3.11. *There is a dynamic set cover algorithm with $O(\log n)$-approximation ratio, $\widetilde{O}(f)$ worst-case update time and $\widetilde{O}(1)$ worst-case recourse.*

We first explain that Lemma 3.7 is not sufficient to guarantee an $O(\log n)$-approximation ratio of the algorithm presented in Section 3.5. To see why this is the case, define a **critical level** ℓ_k for the background thread T_k, as follows: It is the maximum level $\ell \in [0, k]$ such that $\left|C_\ell^{\mathbf{B}_k}(t_k^{\text{sus}})\right| \leq \frac{1}{2} \cdot \tau_k^{\text{sus}}$, where $\tau_k^{\text{sus}} := \left|\mathbf{B}_k(t_k^{\text{sus}})\right|$. Next, consider some level $k' \ll \ell_k$, where $\left|C_{k'}^{\mathbf{B}_k}(t_k^{\text{sus}})\right| \ll \tau_k^{\text{sus}}$. During each time-step when the thread T_k is in suspension or copy phase, the background solution $\mathbf{B}_k$ gets lazily updated w.r.t. $L_k^{\mathbf{F}}$. Such a lazy update might potentially involve the creation of a universally passive element (because of an external deletion) at level $\leq k'$ in $\mathbf{B}_k$. If this keeps happening repeatedly, then after very few time-steps an overwhelming majority of the elements at level $\leq k'$ in $\mathbf{B}_k$ would become universally passive. This would imply that $\mathbf{B}_k$ is no longer ϵ-tidy at level k' when T_k terminates, which, in turn, would lead to a violation of Corollary 3.4.

To address this issue, we create a new **tail phase** for background threads, which occurs after the copy phase. As in Section 3.5, we maintain a foreground solution $\mathbf{F}$, a set of background solutions $\mathbf{B}_k$ and buffer solutions $\mathbf{R}_k$ for each level $k \in [0, \ell_{\max}]$. The foreground solution $\mathbf{F}$ keeps getting lazily updated w.r.t. L.

A background thread T_k, which is responsible for maintaining $\mathbf{B}_k$, now runs in four phases: (i) **computation phase**, (ii) **suspension phase**, (iii) **copy phase** and (iv) **tail phase**. The computation and suspension phases work in exactly the same manner as in Section 3.5, whereas the copy and tail phases are different. Accordingly, below we only explain these last two phases for a given background thread T_k.

III: Copy Phase. At each time-step in the copy phase, the thread T_k copies $\log n$ sets $s \in \mathbf{B}_k$ with $\text{lev}^{\mathbf{B}_k}(s) > \ell_k$ into the buffer solution $\mathbf{R}_k$. In addition, throughout the duration of the copy phase, the thread T_k lazily updates $\mathbf{B}_k$ w.r.t. $L_k^{\mathbf{F}}$. The phase ends when all the sets in $\mathbf{B}_k$ at levels $> \ell_k$ have been copied into $\mathbf{R}_k$. Suppose that this happens during time-step $t_k^{\text{tail}} - 1$. Then, the tail phase begins from time-step t_k^{tail}.

IV: Tail Phase. Throughout the tail phase, the thread T_k lazily updates $\mathbf{B}_k$ w.r.t. $L_k^{\mathbf{F}}$. Furthermore, in the beginning of time-step t_k^{tail}, it initializes an empty hierarchical solution $\mathbf{B}_k^{\text{tail}}$. Next, the thread T_k runs a catch-up greedy algorithm with source $\left(t_k^{\text{tail}}, \ell_k, L_{\ell_k}^{\mathbf{B}_k}\right)$ and target $\mathbf{B}_k^{\text{tail}}$. In addition, whenever the catch-up greedy algorithm adds a new set s into $\mathbf{B}_k^{\text{tail}}$, the thread T_k immediately copies the set s into $\mathbf{R}_k$ on the fly. The tail phase ends at some time-step t_k^{end} (say), when the catch-up greedy algorithm completes execution.

We introduce the notation $\mathbf{B}_k^\star$ to denote a new hierarchical solution, which is obtained by removing from $\mathbf{B}_k$ all the sets that are at level $\leq \ell_k$, and adding back to it the hierarchical solution $\widetilde{\mathbf{B}}_k^{\text{tail}}$ (the extension of $\mathbf{B}_k^{\text{tail}}$; see the discussion just before the statement of Lemma 3.1). It is easy to verify that during the tail phase we always have $\mathbf{R}_k \subseteq \mathbf{B}_k^\star$. Furthermore, using appropriate pointer switches, *at the termination of the tail phase*

we can construct $\mathbf{B}_k^\star$ from $\mathbf{B}_k$ and $\mathbf{B}_k^{\text{tail}}$ in $\widetilde{O}(1)$ time.[7]

The algorithm performs the following procedure at the end of each time-step t. Let $Q(t) \subseteq [0, \ell_{\max}]$ denote the collection of indices of all those background threads that have just completed execution in their tail phases, and let $j^\star \leftarrow \max\{Q(t)\}$ be the highest such thread. We **switch** the solution $\mathbf{B}_{j^\star}^\star$ to the foreground. After that, for each $k \in [0, j^\star]$, the background solution $\mathbf{B}_k$ and the buffer solution $\mathbf{R}_k$ are both reset to being empty at the end of time-step t, the solution $\mathbf{B}_k^{\text{tail}}$ is thrown away, and T_k restarts computation phase from the next time-step $t+1$. We say that the thread $T_{j^\star}$ **normally terminates**, and every thread $k \in [0, j^\star - 1]$ **abnormally terminates** at time-step t.

We now summarize a few basic properties of the dynamic algorithm presented above.

Lemma 3.12. *Throughout the tail phase, the hierarchical solution $\mathbf{B}_k^\star$ always satisfies the following properties.*

1. *It is 2ϵ-tidy at every level $k' \in [0, k]$, and also ϵ-stable.*
2. $\left|\mathbf{B}_k^\star\right| = O(\log n) \cdot \text{OPT}$, *where* OPT *is the minimum set-cover size on input L.*

Proof. (Sketch) Suppose that the thread T_k enters suspension phase in the beginning of time-step t_k^{sus}.

Consider any level $k' \in [\ell_k + 1, k]$. By definition, we have $\left|C_{k'}^{\mathbf{B}_k}(t_k^{\text{sus}})\right| > \frac{1}{2} \cdot \tau_k^{\text{sus}}$. Using the same argument as in the proof of Lemma 3.7, we conclude that the thread T_k remains in suspension, copy or tail phase for at most $o(1) \cdot \tau_k^{\text{sus}}$ time-steps. During each such time-step, at most one element in $\mathbf{B}_k$ can become passive at level $\le k'$ (as $\mathbf{B}_k$ gets lazily updated). Thus, at every time-step t in the copy or tail phase, we have

$$\left|P_{k'}^{\mathbf{B}_k}(t)\right| \le \left|P_{k'}^{\mathbf{B}_k}(t_k^{\text{sus}})\right| + o(1) \cdot \tau_k^{\text{sus}} < \left|P_{k'}^{\mathbf{B}_k}(t_k^{\text{sus}})\right| + o(1) \cdot \left|C_{k'}^{\mathbf{B}_k}(t_k^{\text{sus}})\right| \tag{3}$$

Next, by the same argument as in Item 2 of Corollary 3.3, the background solution $\mathbf{B}_k$ is ϵ-tidy at level k' at time-step t_k^{sus}. Combining this observation with Equation (3), we get that

$$\mathbf{B}_k \text{ is } 2\epsilon\text{-tidy at level } k' \text{ throughout the copy and tail phases.} \tag{4}$$

We claim without proof that replacing the sets at level $\le \ell_k$ by $\widetilde{\mathbf{B}}_k^{\text{tail}}$ does not increase dirty for level $k' > \ell_k$. Then, we also have $\mathbf{B}_k^\star$ is 2ϵ-tidy at level k' throughout the tail phase.

Next, consider any level $k' \in [0, \ell_k]$. Recall that during the tail phase the thread T_k runs catch-up greedy with source $\left(t_k^{\text{tail}}, \ell_k, L_{\ell_k}^{\mathbf{B}_k}\right)$ and target $\mathbf{B}_k^{\text{tail}}$. Thus, by Item 1 of Lemma 3.1, we get that throughout the tail phase $\widetilde{\mathbf{B}}_k^{\text{tail}}$ is ϵ-tidy at level k' . Since $k' \le \ell_k$, this also implies that $\mathbf{B}_k^\star$ is 2ϵ-tidy at level k' throughout the tail phase.

Next, using similar ideas as in Item 2 of Corollary 3.3 (resp. Item 2 of Lemma 3.1), we infer that $\mathbf{B}_k^\star$ is ϵ-stable at every level $k' \in [\ell_k + 1, k]$ (resp. $k' \in [0, \ell_k]$) throughout the tail phase.

Finally, using similar ideas as in Item 3 of Corollary 3.3, we can show that throughout the tail phase, the number of sets at levels $> \ell_k$ in $\mathbf{B}_k$ is at most $O(\log n) \cdot \text{OPT}$. In contrast, using similar ideas as in Item 3 of Lemma 3.1, we can show that throughout the tail phase we have $\left|\widetilde{\mathbf{B}}_k^{\text{tail}}\right| = O(\log n) \cdot \text{OPT}$. Taken together, these two observations imply that $\left|\mathbf{B}_k^\star\right| = O(\log n) \cdot \text{OPT}$ throughout the tail phase. □

Corollary 3.13. *At every time-step, the foreground solution* $\mathbf{F}$ *is 3ϵ-tidy and ϵ-stable.*

Proof. (Sketch) When a thread T_k normally terminates, the way we switch the hierarchical solution $\mathbf{B}_k^\star$ to the foreground and abnormally terminate all the threads T_j with $j < k$ is analogous to the corresponding

[7] This is because when the tail phase terminates, we have $\widetilde{\mathbf{B}}_k^{\text{tail}} = \mathbf{B}_k^{\text{tail}}$.

procedure in Section 3.2 (where we switch $\mathbf{B}_k$ to the foreground, as opposed to $\mathbf{B}_k^\star$). Accordingly, following the same line of reasoning that led us to Corollary 3.4 starting from Corollary 3.3, we can derive Corollary 3.13 starting from Lemma 3.12. □

Remark. In the lemma below, we need to refer to the hierarchical solution $\mathbf{B}_k^\star$ in the *copy phase* of the thread T_k. Towards this end, we follow the convention that throughout the copy phase, the hierarchical solution $\mathbf{B}_k^{\text{tail}}$ is empty, and the hierarchical solution $\mathbf{B}_k^\star$ is constructed in the same manner as in the tail phase. In other words, during the copy phase, the hierarchical solution $\mathbf{B}_k^\star$ is obtained by removing from $\mathbf{B}_k$ all the sets that are at level $\leq \ell_k$, and adding back a static greedy solution for $L_{\ell_k}^{\mathbf{B}_k}$. We claim that with this convention, Item 2 of Lemma 3.12 continues to hold even during the copy phase.

Lemma 3.14. *At any time-step t during the copy or tail phase for thread T_k, we have $\tau_k^{\text{sus}} = O\left(\left|\mathbf{B}_k^\star(t)\right|\right)$.*

Proof. Suppose that T_k enters suspension phase at time-step t^{sus}.

We partition $\mathbf{B}_k(t^{\text{sus}})$ into sets at levels higher than ℓ_k, and sets at levels at most ℓ_k. Note that the sets in the former part are included in $\mathbf{B}_k^\star$, and cannot be modified throughout the copy and tail phase. So, its size is at most $\mathbf{B}_k^\star(t)$. Also, the size of the latter part can be bounded by its coverage size, since every set has nonempty coverage. So, we have

$$\tau_k^{\text{sus}} = |\mathbf{B}_k(t^{\text{sus}})| \leq |\mathbf{B}_k^\star(t)| + \left|C_{\ell_k}^{\mathbf{B}_k}(t^{\text{sus}})\right| \leq |\mathbf{B}_k^\star(t)| + 0.5\tau_k^{\text{sus}}.$$

The proof follows by rearranging the terms in the above inequality. □

Lemma 3.15. *Throughout the duration of copy and tail phases for thread T_k, we have $|\mathbf{R}_k| = O(\tau_k^{\text{sus}})$.*

Proof. (Sketch) Suppose that the thread T_k enters suspension phase at time-step t_k^{sus}. By definition, at the start of time-step t_k^{sus}, we have $|\mathbf{B}_k| = \tau_k^{\text{sus}}$. Using the same argument as in the proof of Lemma 3.7, we conclude that the thread T_k remains in suspension, copy or tail phase for at most $o(1) \cdot \tau_k^{\text{sus}}$ time-steps. During each such time-step, the size of $\mathbf{B}_k$ increases by at most one (due to a potential lazy update w.r.t. $L_k^{\mathbf{F}}$). Thus, we infer that $|\mathbf{B}_k(t)| \leq \left|\mathbf{B}_k(t_k^{\text{sus}})\right| + o(1) \cdot \tau_k^{\text{sus}} = O(\tau_k^{\text{sus}})$ at every time-step t during the copy and tail phases of T_k.

Next, observe that during each time-step the thread T_k remains in suspension, copy or tail phase, the size of $C_{\ell_k}^{\mathbf{B}_k}$ increases by at most one (due to a potential lazy update of $\mathbf{B}_k$ w.r.t. $L_k^{\mathbf{F}}$). Recall that $\left|C_{\ell_k}^{\mathbf{B}_k}(t_k^{\text{sus}})\right| \leq \tau_k^{\text{sus}}$ by definition. Furthermore, the thread T_k remains in suspension, copy or tail phase for at most $o(1) \cdot \tau_k^{\text{sus}}$ time-steps. Thus, at every time-step t in the copy or tail phase of thread T_k, we have

$$\left|\widetilde{\mathbf{B}}_k^{\text{tail}}(t)\right| \leq |L_{\ell_k}^{\mathbf{B}_k}(t_k^{\text{sus}})| + (t - t_k^{\text{sus}}) \leq \left|C_{\ell_k}^{\mathbf{B}_k}(t_k^{\text{sus}})\right| + o(1) \cdot \tau_k^{\text{sus}} = O(\tau_k^{\text{sus}}),$$

where the first inequality holds because the number of sets in $\mathbf{B}_k^{\text{tail}}(t)$ cannot larger than the number of elements they are meant to cover.

Since $\mathbf{R}_k \subseteq \mathbf{B}_k \cup \widetilde{\mathbf{B}}_k^{\text{tail}}$, we derive that $|\mathbf{R}_k| \leq |\mathbf{B}_k| + \left|\widetilde{\mathbf{B}}_k^{\text{tail}}\right| = O(\tau_k^{\text{sus}})$ throughout copy and tail phases of T_k. □

Corollary 3.16. *At every time-step t, we have $\sum_{k \in [0, \ell_{\max}]} |\mathbf{R}_k| = O(\log n) \cdot \text{OPT}(L)$.*

Proof. (Sketch) Note that every background thread T_k has $\mathbf{R}_k = \varnothing$ until the start of its copy phase.

At the start of a time-step t, let $Z(t) \subseteq [0, \ell_{\max}]$ denote the collection of indices of those background threads that are in copy or tail phases. Now, the scheduler in Algorithm 4 ensures that at every time-step t, the values in the collection $\{\tau_k^{\text{sus}} : k \in Z(t)\}$ are geometrically decreasing. Let $k(t) := \arg\max_{k \in Z(t)} \{\tau_k^{\text{sus}}\}$. Applying

Lemma 3.15, Lemma 3.14 and Item 2 of Lemma 3.12 for both copy and tail phases (see the remark just before Lemma 3.14), we now infer that

$$\sum_{k\in[0,\ell_{\max}]} |\mathbf{R}_k(t)| = \sum_{k\in Z(t)} |\mathbf{R}_k(t)| = \sum_{k\in[0,\ell_{\max}]} O(\tau_k^{\text{sus}}) = O\left(\tau_{k(t)}^{\text{sus}}\right) = O\left(\left|\mathbf{B}_{k(t)}^{\star}(t)\right|\right) = O(\log n)\cdot \text{OPT}(L(t)).$$

This concludes the proof of the corollary. □

Proof of Theorem 3.11. Our dynamic algorithm ensures the invariant that $\mathbf{F}$ is a feasible set cover for the collection of all live elements. Hence, the maintained solution $\mathbf{F}\bigcup_{k\in[0,\ell_{\max}]}\mathbf{R}_k$ is also a feasible set cover for the collection of all live elements. The size of the maintained solution is $|\mathbf{F}| + \sum_{k\in[0,\ell_{\max}]}|\mathbf{R}_k|$, and so the approximation guarantee of our algorithm follows from Corollary 3.13, Lemma 2.1 and Corollary 3.16.

Using standard data structures, the foreground solution, the background solutions and the buffer solutions can all be maintained in $\widetilde{O}(f)$ worst-case update time. Furthermore, we have already emphasized that using appropriate pointer switches, we can construct the hierarchical solution $\mathbf{B}_k^{\star}$ from $\mathbf{B}_k$ and $\mathbf{B}_k^{\text{tail}}$ in $\widetilde{O}(1)$ time, and we can also switch $\mathbf{B}_k^{\star}$ into the foreground in $\widetilde{O}(1)$ time, whenever necessary. Thus, the overall worst-case update time of our algorithm is also $\widetilde{O}(f)$.

Finally, we observe that during each time-step within the copy and tail phases of the thread T_k, at most $\widetilde{O}(1)$ sets get added to the buffer solution $\mathbf{R}_k$. In contrast, the buffer solution $\mathbf{R}_k$ remains empty throughout the computation and suspension phases of the thread T_k. Furthermore, whenever a thread T_k terminates, the buffer solution $\mathbf{R}_k$ is reset to being empty and (provided T_k normally terminates) we switch $\mathbf{B}_k^{\star}$ to the foreground; but this event does not cause any insertion recourse (as opposed to causing potentially massive amount of deletion recourse). Finally, the foreground solution $\mathbf{F}$ keeps getting lazily updated w.r.t. L at every time-step, and this also leads to a worst-case insertion recourse of at most 1. Thus, the total worst-case insertion recourse of the maintained solution $\mathbf{F}\bigcup_{k\in[0,\ell_{\max}]}\mathbf{R}_k$ is still $\widetilde{O}(1)$. Now, applying the reduction outlined in Section 3.3, we obtain a dynamic set cover algorithm with $O(\log n)$-approximation, $\widetilde{O}(f)$ worst-case update time and $\widetilde{O}(1)$ worst-case recourse. This concludes the proof of Theorem 3.11.

4 Closing Remarks

In this paper, we presented the first fully dynamic set cover algorithms to achieve non-trivial worst-case guarantees on both recourse and update time. In particular, we *simultaneously* obtained $\text{poly}\log(n)$ worst-case recourse and $O(f\cdot\text{poly}\log(n))$ worst-case update time, while guaranteeing $O(\log n)$ or $O(f)$ approximations.

A natural next goal is to refine these bounds further to match the best results known in the amortized setting, e.g., $(1+\varepsilon)\ln n$ and $(1+\epsilon)f$ approximations with $\widetilde{O}(1)$ worst-case recourse and $\widetilde{O}(f\cdot\text{poly}(1/\varepsilon))$ worst-case update time. It seems plausible that our framework can be used to reduce the approximation factors to $(2+\varepsilon)\ln n$ and $(2+\varepsilon)f$ respectively. However, the gap of 2 is an inherent barrier. Recall that to bound recourse, we used a combination of two solutions, the foreground solution and the buffer solutions, in the output. Each solution, at best, is a tight approximation that adds a factor of $\ln n$ or f to the approximation ratio. Overcoming this barrier of 2 will require more new ideas that go beyond the algorithmic framework that we introduced in this paper.

We also remark that our results only hold in the unweighted setting, i.e., when the sets have uniform cost. It is natural to ask whether our algorithmic framework extends to the weighted case. At a conceptual level, certain ideas that we introduced in this paper seem agnostic to sets having weights, while other ideas seem to fail. For instance, the idea of using a buffer solution to limit worst-case recourse is independent of whether the sets are weighted or unweighted. On the other hand, consider the idea of suspending and later restarting (or aborting)

background processes to only allow a set of well-separated solutions to proceed to the buffer, that we used to bound our approximation ratio. This mechanism does not extend to the weighted case, because we can no longer trade off computation speed (which depends on the number of sets) with solution cost (which depends on the costs of sets) in the way that we did in our current algorithm. Nevertheless, the history of dynamic set cover algorithms makes us hopeful that our results will eventually be extended to the weighted setting. A case in point is the first $(1+\varepsilon)f$-approximation result obtained by [AAG+19]. The algorithm presented in this paper also applied only to the unweighted setting, but was later extended (with a mild dependence on the range of weights) to the weighted case by [BHN19]. In contrast to this pair of results, the f-approximation algorithm obtained by [AS21] also applies to the unweighted setting only, but an extension to the weighted case has remained elusive.

Finally, we remark on the logarithmic bounds in update time and recourse that we obtain in this paper. In the low frequency regime (i.e., $O(f)$-approximation), there are a series of results in prior work that removed the $O(\log n)$ dependence in the *amortized* update time bounds [AS21, BHNW21, BSZ25]. So, it is natural to ask whether we can obtain a dynamic set cover algorithm with $O(f)$ worst-case update time and $O(1)$ worst-case recourse. We note that neither of these results is known even in isolation (the only exception is [GKKP17] which obtains $O(1)$ worst-case recourse but at the cost of *exponential* update time), and designing algorithms that achieve either bound represents an interesting direction for future research.

Part II

$O(\log n)$-Competitive Algorithm

Remark. Throughout Part II, we use the notations and terminologies introduced in Section 2.

5 Fully Dynamic $O(\log n)$-Competitive Set Cover Algorithm

The algorithm maintains a set of different set cover solutions, some of which correspond to different levels. The solutions are:

- a **foreground** solution $\mathbf{F}$,
- a set of $\ell_{\max} + 1$ **background** solutions $\mathbf{B}_k$ for levels $k \in [0, \ell_{\max}]$, and
- a set of $\ell_{\max} + 1$ **buffer** solutions $\mathbf{R}_k$ for levels $k \in [0, \ell_{\max}]$.

We maintain the invariant that the foreground solution is feasible for the set of live elements at any time. The background solutions are not part of the output, but are used to update the foreground solution to maintain the desired approximation bounds. However, doing this directly incurs large recourse on the foreground solution. Instead, we use the buffer solution as an intermediary between the background and foreground solutions. We slowly update a buffer solution using the corresponding background solution, and once this process completes, switch the background solution to the foreground solution. The output comprises the foreground and the buffer solutions; so, switching the background solution (which is identical to the buffer solution) to the foreground does not cost us in recourse.

The foreground solution and each of the background solutions are hierarchical solutions (with levels and passive levels), while the buffer solutions are merely collections of sets. The algorithm creates $\ell_{\max}+1$ background threads T_k, one for each level $k \in [0, \ell_{\max}]$, where each thread runs a variant of the greedy set cover algorithm that we will describe later. These background threads run in a sequential order from $T_{\ell_{\max}}$ to T_0, and update their respective background solutions independently. The background threads run after the foreground thread. The algorithm periodically copies the background solutions into the buffer solutions. When copying is complete for a background solution, the algorithm switches the background solution to the foreground to update the foreground solution.

Initialization. The algorithm initializes the foreground, background, and buffer solutions to be empty since there is no live element at this stage, i.e. $L(0) = \emptyset$.

5.1 Foreground Thread

We describe the algorithm to maintain the foreground solution on the deletion or insertion of an element.

On deletion of element e, we set $\text{plev}^{\mathbf{F}}(e) \leftarrow \text{lev}^{\mathbf{F}}(e)$. Note that $e \in \text{cov}(\mathbf{F})$ as $\mathbf{F}$ is a feasible solution for all live elements.

On insertion of element e, let s be the set containing e with largest $\text{lev}^{\mathbf{F}}(s)$. If such a set doesn't exist, let s be any set that contains e – in this case, we add s to $\mathbf{F}$ at level 0. In both cases, we assign e to s, i.e. e is added to $\text{cov}(s)$ (but we don't change $\text{lev}(s)$). We also set $\text{plev}(e) = \text{lev}(e) = \text{lev}(s)$.

Besides the above changes triggered by the deletion or insertion of an element, the foreground solution can also be modified by switching to a background solution, which happens at the termination of a background

thread. We describe background threads next and then discuss how the background, buffer, and foreground solutions interact.

5.2 Background Threads

At initialization, the algorithm starts a background thread T_k for every level $k \in [0, \ell_{\max}]$. Whenever a background thread terminates, the algorithm re-starts a thread at the same level. So, for every level k, there is always a background thread T_k running in the algorithm.

Conceptually, each background thread aims to run a greedy algorithm and copy the solution to the buffer. Indeed, if we allow an $O(\log^2 n)$ approximation factor, we can simply let all threads copy their solutions, as described in Section 3.4. In this case, all threads can run independently in parallel. However, to obtain an approximation factor of $O(\log n)$, we have to restrict the threads that are allowed to copy to the buffer at any time. This causes some additional complexity to coordinate the behavior of different background threads.

We partition the algorithm run by a background thread into five phases. For a background thread T_k, we first give a short overview of the purpose of each of these five phases:

- **Preparation:** The background thread initializes a greedy set cover algorithm with the set of elements $L_k^{\mathbf{F}}$, i.e. the sub-universe of live elements at levels $\leq k$ in the foreground solution $\mathbf{F}$.
- **Computation:** The background thread runs a greedy set cover algorithm on the sub-universe above to generate a hierarchical background solution $\mathbf{B}_k$. This background solution covers most elements in the sub-universe, but might leave a few elements uncovered. These latter elements are said to form the tail-universe of this background thread.
- **Suspension:** At the end of the computation phase, the background thread might enter a suspension phase where it suspends operation.
- **Copy:** Once out of the suspension phase, the background thread copies the background solution it computed to a corresponding buffer solution.
- **Tail:** In this final phase, the background thread resumes the greedy algorithm on the (small) tail-universe and immediately copies each set in the solution to the corresponding buffer solution.

Next, we describe the algorithm for each phase in detail. In terms of data structures, the background thread T_k maintains a priority queue Q_k over all sets $\mathcal{S}$, where the key of a set s is the size of its intersection with the uncovered sub-universe $L_k^{\mathbf{F}} \setminus \mathrm{cov}(\mathbf{B}_k)$. We defer the details of the data structure to Section 6.5.

We also note that the sub-universe $L_k^{\mathbf{F}}$ can change during the execution of a background thread T_k. This happens when an element e in the sub-universe is deleted, or an element e is inserted and the foreground thread sets $\mathrm{lev}^{\mathbf{F}}(e) \leq k$. We describe how to handle these changes separately in each phase.

Let c_{spd} be a large constant that we determine later in Section 6.2. It measures the number of elements processed in each time-step.

Specially, if $|L_k^{\mathbf{F}}| \leq c_{\mathrm{spd}}$ at the beginning of T_k, all work of T_k can be finished in one time-step. In this case, we execute T_k to termination in the time-step. We call this the rule of base threads.

Preparation Phase. During the preparation phase, T_k builds the priority queue Q_k. This requires scanning the sub-universe $L_k^{\mathbf{F}}$, which is done at the speed of processing c_{spd} elements in each time-step.

During the scanning, T_k also sets $\mathrm{plev}^{\mathbf{B}_k}(e) \leftarrow \max\{k+1, \mathrm{plev}^{\mathbf{F}}(e)\}$ for each element in the sub-universe. (The levels $\mathrm{lev}^{\mathbf{B}_k}(e)$ of elements $e \in L_k^{\mathbf{F}}$ will be decided later.)

If the sub-universe $L_k^{\mathbf{F}}$ changes during the preparation phase, we modify Q_k to reflect this change in the sub-universe $L_k^{\mathbf{F}}$ and reset the passive level of the element if required.

Computation Phase. The background thread T_k executes the greedy algorithm given in Algorithm 5 in its computation phase, which is an adaptaion of the standard offline greedy algorithm for set cover. A crucial property of the offline algorithm is that the sets are added to the solution in order of monotonically decreasing coverage. But, this property might be violated in our setting because of element insertions. To assert this property artificially, the algorithm uses an index p_k to indicate the level of the last set computed in the greedy algorithm. The algorithm ensures that p_k monotonically decreases over time. We call p_k the *copy pointer* of the background thread T_k.

In terms of the speed of processing, the greedy algorithm adds c_{spd} elements to the coverage of the solution per time-step. Since the algorithm is processing elements much faster than insertions and deletions, the elements that change status (from live to dormant or vice-versa) during the execution of the greedy algorithm are a small fraction of all the elements in the greedy solution. This latter property is crucial in ensuring that the algorithm produces a competitive solution.

Algorithm 5: Greedy algorithm run by background thread T_k

5.1 $p_k \leftarrow k+1$

5.2 **while** *the uncovered sub-universe* $L_k^{\mathbf{F}} \setminus \text{cov}(\mathbf{B}_k)$ *is nonempty* **do**

5.3 Find the set s that covers the maximum number of uncovered elements in the sub-universe (using Q_k.FindMax), i.e.
$$s = \arg\max_{s \in \mathcal{S}} |s \cap (L_k^{\mathbf{F}} \setminus \text{cov}(\mathbf{B}_k))|.$$

5.4 Add s to $\mathbf{B}_k$. Set $\text{cov}(s) \leftarrow s \cap (L_k^{\mathbf{F}} \setminus \text{cov}(\mathbf{B}_k))$.

5.5 Set $\text{lev}(s) \leftarrow \min\{\lfloor \log_{1+\varepsilon} |\text{cov}(s)| \rfloor, p_k\}$.

5.6 For each $e \in \text{cov}(s)$, set $\text{lev}^{\mathbf{B}_k}(e) \leftarrow \text{lev}(s)$, and update Q_k to reflect that e is removed from the uncovered sub-universe.

5.7 Update $p_k \leftarrow \min\{p_k, \text{lev}^{\mathbf{B}_k}(s)\}$.

We pause the greedy solution once the number of uncovered elements in the sub-universe becomes small. Intuitively, this is because at this stage, the changes in the sub-universe can significantly affect this small uncovered set. These uncovered elements form the tail-universe which is handled later in the tail phase of the background thread. In particular,

- After processing a set s (i.e. when Algorithm 5 reaches the end of the while loop), if $|L_k^{\mathbf{F}} \setminus \text{cov}(\mathbf{B}_k)| \leq |\mathbf{B}_k|$, then pause the greedy algorithm and enter the suspension phase.

Specially, if $|L_k^{\mathbf{F}} \setminus \text{cov}(\mathbf{B}_k)| \leq |\mathbf{B}_k| \leq c_{\text{spd}}$ when we pause the greedy algorithm, then all remaining work of T_k can be finished in one time-step. In this case, we continue to execute the algorithm for copy and tail phases (to be defined later) until the termination of T_k. In this case, the thread is viewed to terminate in computation phase, and not viewed to enter later phases. We call this the rule of shortcut threads. This implies that if T_k enters later phases, then $|\mathbf{B}_k(t^{\text{sus}})| > c_{\text{spd}}$ where t^{sus} is the time-step when T_k enters suspension phase.

The sub-universe $L_k^{\mathbf{F}}$ can change due to insertions and deletions of elements during the computation phase. We give the algorithm to handle these updates below in Algorithms 6 and 7.

Suspension Phase. When $|L_k^{\mathbf{F}} \setminus \text{cov}(\mathbf{B}_k)| \leq |\mathbf{B}_k|$, the background thread T_k enters the suspension phase. In the suspension phase, the thread does not run the greedy algorithm, and does not copy any sets to the buffer

Algorithm 6: Updates by background thread T_k on deletion of element e in $L_k^{\mathbf{F}}$

if $e \in \mathrm{cov}(s)$ *for some set* $s \in \mathbf{B}_k$ **then**
  $\mathrm{plev}^{\mathbf{B}_k}(e) \leftarrow \mathrm{lev}^{\mathbf{B}_k}(e)$
else
  Update Q_k to reflect that e is removed from the uncovered sub-universe.

Algorithm 7: Updates by background thread T_k on insertion of element e in $L_k^{\mathbf{F}}$ (i.e. $\mathrm{lev}^{\mathbf{F}}(e) \leq k$)

if e *is contained in some set in* $\mathbf{B}_k$ **then**
  Cover e using the set s with the maximum level in $\mathbf{B}_k$ among all those that contain e.
  $\mathrm{plev}^{\mathbf{B}_k}(e) \leftarrow \mathrm{lev}(s)$
  $\mathrm{lev}^{\mathbf{B}_k}(e) \leftarrow \mathrm{lev}(s)$
else
  Update Q_k to reflect that e is inserted into the uncovered sub-universe.
  $\mathrm{plev}^{\mathbf{B}_k}(e) \leftarrow p_k$

solution. When T_k enters suspension phase, we take a snapshot of its solution size and denote it by τ_k^{sus}. In other words, $\tau_k^{\mathrm{sus}} := |\mathbf{B}_k(t^{\mathrm{sus}})|$ where t^{sus} is the time-step when T_k enters suspension phase.

If the sub-universe $L_k^{\mathbf{F}}$ changes during the suspension phase, we update $\mathbf{B}_k$ in an identical way to that in the computation phase (Algorithms 6 and 7).

We now describe the criteria for suspended threads to enter the copy phase. The main desiderata is that the set of threads simultaneously in copy and tail phases form a geometric series in terms of their solution sizes. We give the algorithm for ensuring this below:

Algorithm 8: Transitioning background threads from suspension to copy phase

$\tau \leftarrow \frac{1}{2} \cdot \min_j \tau_j^{\mathrm{sus}}$, where the minimum is taken over for all threads T_j that are in copy and tail phases.
while *there exists some* $T_{k'}$ *currently in suspension phase such that* $\tau_{k'}^{\mathrm{sus}} \leq \tau$ **do**
  $k \leftarrow \max\{k' : T_{k'} \text{ in suspension phase and } \tau_{k'}^{\mathrm{sus}} \leq \tau\}$
  Move T_k from suspension to copy phase.
  Update $\tau \leftarrow \frac{1}{2} \cdot \tau_k^{\mathrm{sus}}$.

Let t^{sus} denote the time-step when a background thread T_k enters the suspension phase. If T_k remains too long in the suspension phase, then the previously computed solution $\mathbf{B}_k(t^{\mathrm{sus}})$ can become uncompetitive due to many insertions and deletions. So, we add a thresholding condition: if T_k is in the suspension until time-step $t^{\mathrm{sus}} + 0.1 \cdot \tau_k^{\mathrm{sus}}$, then we abort the thread and restart the preparation phase.

Copy Phase. During the copy phase, T_k copies sets in $\mathbf{B}_k$ to the corresponding buffer solution $\mathbf{R}_k$ at the speed of c_{spd} sets per time-step.

If the sub-universe $L_k^{\mathbf{F}}$ changes during the copy phase, we update $\mathbf{B}_k$ identically to the computation phase (Algorithms 6 and 7). In addition, any sets added to $\mathbf{B}_k$ (because of element insertions) are immediately reflected in the buffer solution $\mathbf{R}_k$.

Tail Phase. When entering the suspension phase, the elements in the tail-universe are still to be covered. These elements remain uncovered during the suspension and copy phases. After the solution $\mathbf{B}_k$ has been copied to $\mathbf{R}_k$, the background thread T_k enters the tail phase. In this phase, the algorithm resumes the greedy algorithm on the tail-universe. The speed to run the greedy algorithm is c_{spd} elements per each time-step, same as the computation phase. In the tail phase, T_k copies each new set added to $\mathbf{B}_k$ immediately to $\mathbf{R}_k$.

If the sub-universe $L_k^{\mathbf{F}}$ changes during the tail phase, we handle this identically to the copy phase. I.e., we run Algorithms 6 and 7 and any sets added to $\mathbf{B}_k$ (because of element insertions) are immediately reflected in the buffer solution $\mathbf{R}_k$.

Termination. The tail phase ends when $\mathbf{B}_k$ (and therefore $\mathbf{R}_k$) covers every element in the sub-universe $L_k^{\mathbf{F}}$. Now, the algorithm switches the background solution $\mathbf{B}_k$ to the foreground by removing all sets in $\mathbf{F}$ at levels $\leq k$, and replacing them with the sets in the corresponding levels in $\mathbf{B}_k$. Additionally, $\mathbf{B}_k$ might have sets at level $k+1$, which are now merged into level $k+1$ in the foreground solution $\mathbf{F}$.

We will show later that these transformations can be done efficiently within the update time afforded by a single time-step of the dynamic instance. Note that the foreground solution $\mathbf{F}$ and the buffer solutions $\mathbf{R}_k$ collectively constitute the output. Since the buffer solution $\mathbf{R}_k$ is a copy of the background solution $\mathbf{B}_k$ at this stage, the switch does not change the collection of sets in the output. Therefore, recourse is unaffected. But, this step is important in correctly maintaining our invariants on the data structures of the foreground solution. As part of the switch, the levels and passive levels of the relevant elements in $\mathbf{F}$ are updated to those in $\mathbf{B}_k$.

After this switch for a background thread T_k, we abort all background threads $T_{k'}$, $k' < k$. When a thread gets aborted, it discards all sets copied to $\mathbf{R}_{k'}$ and discards all data structures. Then, the threads $T_{k'}$, $k' \leq k$ return to the preparation phase (on the new sub-universe $L_{k'}^{\mathbf{F}}$). To distinguish between the two situations causing a background thread to end – that it completed its tail phase and it got aborted by another background thread – we call the former *normal termination* and the latter an *abnormal termination*. Termination by the rule of base threads or shortcut threads is also viewed as normal termination.

5.3 Sequential Behavior of Threads

In the above algorithm description, we consider each thread separately. Since our overall algorithm is a sequential algorithm, the threads need to coordinate in a global scheduling. We use the following simple scheduling: The foreground thread is executed first. Then, the background threads are executed in the order of decreasing levels, from $T_{\ell_{\max}}$ to T_0. Each thread runs in a consecutive time block.

This order is related to the following features of the algorithm.

- The foreground thread handles the insertion or deletion in the foreground solution, which determines the change of uncovered sub-universes of background threads.
- Although Algorithm 8 is described as a global loop, the condition to enter copy phase from suspension phase is actually examined by each thread separately. Before executing any background threads, we initialize τ according to Line 8.1. When executing background threads T_k in suspension phase, we compare its snapshot size τ_k^{sus} with the thread τ. If $\tau_k^{\text{sus}} \leq \tau$, we let T_k enter copy phase and update $\tau \leftarrow \frac{1}{2} \cdot \tau_k^{\text{sus}}$. The decreasing order of executing background threads ensures an equivalent behavior to Algorithm 8.

We explain some related details below.

- When a background thread is executed, it first handles the insertion or deletion, then continue the algorithm for its current phase.

- (Rule of one phase in a time-step.) A background thread is only allowed to undergo one phase in a time-step, with the special case of a phase change in the end. So, once a background thread exists one phase and enters the next phase, it will stop execution for the current time-step, and start the next phase at the subsequent time-step.

- Recall the normal termination of a background thread will abort all lower-level background threads. Since the background threads are executed in decreasing order, whenever we encounter a normal termination of a background level T_k, the lower-level background threads are not executed yet, and they can be directly aborted. Also, according to the rule of one phase in a time-step, these lower-level background threads will start preparation phase in the subsequent time-step.

6 Analysis of the Fully Dynamic $O(\log n)$-Competitive Set Cover Algorithm

6.1 Properties of the solution

Recall that $\text{cov}(\mathbf{F})$ and $C_k^{\mathbf{F}}$ respectively denote the set of elements (live and dormant) in the coverage of all levels and levels $\leq k$ of the foreground solution $\mathbf{F}$. Thus, $\text{cov}(\mathbf{F}) \setminus C_k^{\mathbf{F}}$ represents the set of elements in the coverage of levels $> k$.

Feasibility. We start by showing that the sets of live elements in $\text{cov}(\mathbf{B}_k)$ and $L_k^{\mathbf{F}}$ are identical at normal termination of a background thread T_k (i.e. if it terminates at the end of its tail phase and not because it got aborted by another background thread):

Claim 6.1. *We have the following properties:*

(i) For any time t and any background thread T_k, we have $\text{cov}(\mathbf{B}_k(t)) \cap L(t) \subseteq L_k^{\mathbf{F}}(t)$.

(ii) When a background thread T_k terminates normally, we have $\text{cov}(\mathbf{B}_k) \cap L = L_k^{\mathbf{F}}$.

(iii) When a background thread T_k terminates normally, the update to the foreground solution does not change $L_j^{\mathbf{F}}$ at any level $j > k$.

Proof. Fix any k. It is immediate that property (i) implies property (ii) since at normal termination of T_k, all live elements in $L_k^{\mathbf{F}}$ are in $\text{cov}(\mathbf{B}_k)$. Moreover, when a background thread T_k terminates normally, the foreground solution $\mathbf{F}$ is updated only at levels $\leq k+1$. This means that property (iii) is an immediate consequence of property (ii).

We are left to prove property (i), which we do by induction on level k and time t. As the base case, when any thread T_k is created, $\text{cov}(\mathbf{B}_k)$ is empty, and the statement is trivial. We now consider the inductive steps:

- The greedy algorithm run by T_k adds a new set s to $\mathbf{B}_k$. Since Line 5.4 assigns $\text{cov}(s)$ to be a subset of $L_k^{\mathbf{F}}$, the property is preserved.

- Consider insertion of an element e. If $\text{lev}^{\mathbf{F}}(e) > k$, then it affects neither $\text{cov}(\mathbf{B}_k)$ nor $L_k^{\mathbf{F}}$. Otherwise, e is added to $L_k^{\mathbf{F}}$, and may or may not be added to $\text{cov}(\mathbf{B}_k)$ depending on whether it is already covered by a set in $\mathbf{B}_k$. So, the property is preserved.

- Consider deletion of an element e. In this case, e is removed from L, and $\mathrm{cov}(\mathbf{B}_k)$ remains unchanged. So, the property is preserved.

- Finally, consider the case that $\mathbf{F}$ is updated at normal termination of another background thread $T_{k'}$. We have $k' < k$, otherwise T_k would be aborted. By the inductive hypothesis, (iii) holds for k', which means that $L_k^{\mathbf{F}}$ is not changed by the update. So, the property is preserved. □

Using the claim above, we now show that the foreground solution is always feasible, i.e. all elements in L are always covered in $\mathbf{F}$:

Lemma 6.2. $\mathbf{F}$ *is always a feasible set cover.*

Proof. We prove this lemma by induction over time. Initially, $L = \emptyset$, so the lemma trivially holds. We now consider the inductive steps:

- On insertion of an element e, the algorithm adds e to $\mathrm{cov}(s)$ for some existing set $s \in \mathbf{F}$ or adds a new set s to F and sets $\mathrm{cov}(s) = \{e\}$.

- On deletion of an element e, the algorithm does not change $\mathrm{cov}(\mathbf{F})$.

- Finally, when a thread T_k terminates normally, the set of live elements covered in $\mathrm{cov}(\mathbf{F})$ does not change, by property (ii) of Claim 6.1. □

Hierarchical solution. Next, we establish that the foreground and background solutions are hierarchical solutions, and do not have any duplicate elements. First, we show that the set of elements in the coverage of a background thread T_k is disjoint from $\mathrm{cov}(\mathbf{F}) \setminus C_k^{\mathbf{F}}$. This ensures that when we switch the background solution to the foreground, we do not create duplicate elements.

Claim 6.3. *For any background thread* T_k*, we have* $\mathrm{cov}(\mathbf{B}_k) \cap (\mathrm{cov}(\mathbf{F}) \setminus C_k^{\mathbf{F}}) = \emptyset$.

Proof. We prove by induction on time. The base case is trivial since a background thread T_k is initialized in the preparation phase with an empty solution, i.e., $\mathrm{cov}(\mathbf{B}_k)$ is empty. For the inductive step, we have the following cases:

1. The greedy algorithm run by T_k adds a new set s to $\mathbf{B}_k$. Since Line 5.4 assigns $\mathrm{cov}(s)$ to be a subset of $L_k^{\mathbf{F}} \subseteq C_k^{\mathbf{F}}$, the property is preserved.

2. A new element e is inserted in $L_k^{\mathbf{F}}$, and therefore also in $C_k^{\mathbf{F}}$. Algorithm 7 adds e to $\mathrm{cov}(\mathbf{B}_k)$ if an existing set in $\mathbf{B}_k$ contains e, else e is added to the uncovered set of elements. In the latter case, $\mathrm{cov}(\mathbf{B}_k)$ remains unchanged. Therefore, the property is preserved.

3. An element e is deleted from $L_k^{\mathbf{F}}$. The foreground thread only modifies $\mathrm{plev}^{\mathbf{F}}(e)$, so e is retained in $C_k^{\mathbf{F}}$ as a dormant element. Therefore, the property is trivially preserved.

4. A background thread at some level $k' < k$ normally terminates and updates the foreground solution $\mathbf{F}$ at levels $\leq k' + 1$. Since levels $> k$ are not changed in $\mathbf{F}$, the set $\mathrm{cov}(\mathbf{F}) \setminus C_k^{\mathbf{F}}$ cannot grow because of this update to $\mathbf{F}$. Therefore, the property is preserved. □

Next, we show a property of the copy pointer that is crucial for ensuring that the foreground and background solutions are hierarchical solution:

Claim 6.4. *During the lifetime of a background thread* T_k*, the following hold for its copy pointer* p_k*:*

- *p_k is monotone decreasing, starting from $k+1$.*
- *for every element $e \in L_k^{\mathbf{F}}$, we have* $\text{plev}^{\mathbf{B}_k}(e) \geq p_k$.

Proof. Since p_k is initialized to $k+1$, and only modified by Line 5.7 in Algorithm 5, the first part of the claim follows.

For the second part, note that in the preparation phase of T_k, passive levels of all elements $e \in L_k^{\mathbf{F}}$ are set to $k+1$. For an element inserted into $L_k^{\mathbf{F}}$, its passive level is set to p_k, which is monotone decreasing. The passive levels in a background solution can only be modified due to deletion, which moves e out of $L_k^{\mathbf{F}}$. □

We are now ready to show that the foreground and background solutions are hierarchical solutions:

Lemma 6.5. *The foreground solution* $\mathbf{F}$ *and the background solutions* $\mathbf{B}_k$ *for every* k *are hierarchical solutions as defined in Definition 1. In particular, the coverages are non-empty and disjoint, and the levels and passive levels satisfy the level and passive level invariants (as given in Section 2).*

Proof. We verify the properties of a hierarchical solution:

- Coverages are non-empty: When a set is added to a background solution (in the computation or tail phase) by the greedy algorithm, its coverage set is non-empty. Furthermore, coverage sets do not shrink over time because elements that become dormant due to deletion are not deleted from their respective coverage sets.

  In the foreground solution, sets are added in one of two ways: either they are obtained from a background solution after a normal termination, or they are added due to insertion of an element. In either case, the coverage sets are non-empty.

- Coverages are disjoint: First, we consider background solutions. When a set is added by the greedy algorithm, its coverage is disjoint from all sets that are already in the background solution. Moreover, if a set gains coverage due to an element insertion, the newly inserted element is added to the coverage of only one set in the solution. Hence, the coverages of all sets in the background solution remain disjoint throughout.

  The coverages of sets in the foreground solution change over time in two ways. First, this may be due to an element insertion. In this case, the new element is added to the coverage of exactly one set in the foreground solution. Second, if a background solution is switched to the foreground after normal termination, then Claim 6.3 ensures that the set of elements in the coverage of the background solution is disjoint from the coverages of the sets that remain in the foreground solution after the switch.

- Level Invariant: When a set is added to a background solution by the greedy algorithm, its level is set to satisfy the level invariant. When a set is added to the foreground solution to handle element insertion, its level is set to 0, which satisfies the level invariant. After a set is added, the level invariant is preserved because the coverage of a set cannot shrink, and the level of a set cannot change.

- Passive Level Invariant: For a set added to a background solution by the greedy algorithm, Line 5.5 sets the level of the set to at most p_k, while all elements in its coverage have passive level at least p_k by Claim 6.4. After an element is inserted or deleted, the algorithm sets its passive level to be equal to its level. The switch of a background solution to the foreground after normal termination of a background thread preserves levels and passive levels. So, the passive level invariant is maintained throughout the algorithm. □

No Duplicate Sets. In the previous lemma, we ruled out the possibility of the same element appearing in the coverages of two sets in a hierarchical solution. But, this does not rule out the same set appearing multiple times in the same solution with disjoint coverages. We rule out this latter possibility in the next lemma.

Lemma 6.6. *In any hierarchical solution S (foreground or background solution), a set s appears at most once.*

Proof. First, we consider a background solution $\mathbf{B}_k$. We show the following property (we call this *maximal coverage*):

(Maximal Coverage.) For any element e in the uncovered sub-universe $L_k^{\mathbf{F}} \setminus \text{cov}(\mathbf{B}_k)$ and for any set $s \in \mathbf{B}_k$, we have $e \notin s$.

Before proving this property, we observe that it implies the statement of the lemma. By maximal coverage, for any set s in $\mathbf{B}_k$, none of its elements can be in the uncovered sub-universe $L_k^{\mathbf{F}} \setminus \text{cov}(\mathbf{B}_k)$. This implies that a duplicate copy of s cannot be added to $\mathbf{B}_k$ by the greedy algorithm.

we now show maximal coverage holds inductively over time. For the base case, in the preparation phase of T_k, we have that $\mathbf{B}_k = \emptyset$. So, the property is trivial. For the inductive step, there are three cases:

- The greedy algorithm run by T_k adds a new set s to $\mathbf{B}_k$: Since Line 5.4 assigns $\text{cov}(s)$ to be $s \cap (L_k^{\mathbf{F}} \setminus \text{cov}(\mathbf{B}_k))$, it follows that no element in the remaining uncovered sub-universe $L_k^{\mathbf{F}} \setminus \text{cov}(\mathbf{B}_k \cup \{s\})$ can belong to s.
- A new element e is added to $L_k^{\mathbf{F}}$ due to element insertion: In Algorithm 7, e is added to the uncovered sub-universe if and only if e does not belong to any set in $\mathbf{B}_k$. So, maximal coverage holds for e.
- An element $e \in L_k^{\mathbf{F}}$ is deleted: Since Algorithm 6 does not change the coverages of sets, maximal coverage is unaffected.

This completes the proof of the lemma for background solutions.

We now consider the foreground solution $\mathbf{F}$. Note that by Lemma 6.2, the foreground solution is feasible and hence does not have any uncovered element. So, maximal coverage does not apply to the foreground solution. To establish the lemma for the foreground solution, we use the following property:

(Highest Coverage.) Suppose element e belongs to $\text{cov}^S(s)$ for some set s in a hierarchical solution S. (S can be either the foreground solution $\mathbf{F}$ or any background solution $\mathbf{B}_k$.) It holds that e does not belong to any set s' at a higher level than s in S. I.e. if $\text{lev}^S(s') > \text{lev}^S(s)$, then $e \notin s'$.

Before proving this property, we first show that it implies the statement of the lemma for the foreground solution $\mathbf{F}$. The foreground solution can add a set s in two ways. First, if s is added to handle insertion of element e, then e cannot belong to any existing set in $\mathbf{F}$. So, s was not in $\mathbf{F}$.

The remaining case is when the foreground solution is updated by switching a background solution $\mathbf{B}_k$ after its normal termination. Assume for contradiction that some set s is in $\mathbf{F}$ at some level $> k$ and also in $\mathbf{B}_k$. Let t, t' denote the time-steps when $\mathbf{B}_k$ was initiated and when s was added to $\mathbf{B}_k$ respectively. Note that foreground sets at levels $> k$ can only be modified by the normal termination of a background thread at some level $\geq k$. But, such a normal termination of $\mathbf{B}_{k'}$ for some $k' > k$ would abort T_k. Since T_k had a normal termination, it must be that the sets in levels $> k$ in $\mathbf{F}$ are identical at time-steps t, t' (and also at termination of T_k). This means that s was already in $\mathbf{F}$ at a level $> k$ at time-step t. Now, consider any element $e \in s$. We will show that e cannot be in the uncovered sub-universe of T_k at any stage. Since this holds for all elements of s, it follows that s cannot be added by the greedy algorithm to $\mathbf{B}_k$. First, $e \notin L_k^{\mathbf{F}}(t)$ because this would violate the highest coverage property of $\mathbf{F}$ at time-step t. Thus, e must have been inserted between time-steps

t and t'. But, in this case, the foreground thread will add e to the set containing e at the highest level in $\mathbf{F}$. Since s is such a candidate set at level $> k$, the element e cannot be added to $L_k^{\mathbf{F}}$, and therefore, cannot appear in the uncovered sub-universe of T_k. This establishes that s cannot be in $\mathbf{B}_k$, and therefore, we do not create duplicate sets in the foreground solution $\mathbf{F}$.

We are left to prove the highest coverage property. Although we only need it for the foreground solution to establish the lemma, we will first show it for background solutions and then use this to show it for the foreground solution. Our proof is by induction over time. For the base case, $\mathbf{B}_k$ is empty in the preparation phase, and the property is trivial. For the inductive step, there are three cases:

- The greedy algorithm run by T_k adds a new set s to $\mathbf{B}_k$: For newly covered elements, the highest coverage property follows from the maximal coverage property of those elements before the new set is added. Previously covered elements continue to satisfy the highest coverage property, since the new set has lowest level in $\mathbf{B}_k$ by Claim 6.4.

- A new element e is added to $L_k^{\mathbf{F}}$ due to element insertion: If e is in some set in $\mathbf{B}_k$, then Algorithm 7 will add it to the coverage of the set s with the highest level (among sets in $\mathbf{B}_k$ containing e). So, the highest coverage property holds for e.

- An element $e \in L_k^{\mathbf{F}}$ is deleted: Since Algorithm 6 does not change the levels and coverages of sets, highest coverage is unaffected.

We now proceed to establishing the highest coverage property for the foreground solution. As before, our proof is by induction over time. For the base case, initially $\mathbf{F}$ is empty, and the property is trivial. For the inductive step, there are three cases:

- The foreground solution is updated by switching a background solution $\mathbf{B}_k$ after its normal termination: This adds sets at levels $\leq k+1$ to $\mathbf{F}$; thus, the highest coverage property in $\mathbf{F}$ for elements e with $\text{lev}^{\mathbf{F}}(e) \geq k+1$ are not affected. The remaining elements are now covered according to $\text{cov}(\mathbf{B}_k)$. Let e be such an element that is now covered at level $\text{lev}^{\mathbf{B}_k}(e) \leq k+1$. We note that e cannot belong to a set that was already in $\mathbf{F}$ before the switch at level $\geq k+1$ by the inductive hypothesis for $\mathbf{F}$. Furthermore, e cannot belong to a set in $\mathbf{B}_k$ at a level higher than $\text{lev}^{\mathbf{B}_k}(e)$ by the highest coverage property of $\mathbf{B}_k$. Since these are the only new sets added to $\mathbf{F}$ and they inherit their levels from those in $\mathbf{B}_k$, the property continues to hold for these sets in $\mathbf{F}$ after the switch. Combined, this establishes the highest coverage property for e in $\mathbf{F}$ after the switch.

- A new element e is inserted: If e belongs to any set in $\mathbf{F}$, then the foreground thread adds e to the coverage of the set at the highest level among all those containing e. Otherwise, it adds a new level-0 set if e is not in any set in $\mathbf{F}$. In both cases, the highest coverage property holds for e.

- An element e is deleted: The foreground thread does not change the levels and coverages, and hence, the highest coverage property is unaffected.

This concludes the proof of the highest coverage property for the foreground solution. □

6.2 Lifetime Bounds

We set $c_{\text{spd}} = 400$. Intuitively, we view c_{spd} as a sufficiently large constant. So, we will keep some dependence on c_{spd} in the analysis.

From the rule of one phase in a time-set, we will say a time-step t is both the last time-step of the previous phase and the first time-step of the next phase. In this case, the next phase does not perform work at time-step t. So, in the following analysis, we will count the work starting from time-step $t+1$.

Fact 6.7. *Suppose that a background thread T_k is in the preparation phase at time-step t. Then, T_k will finish preparation phase and enter computation phase (or get aborted) by time-step $t + \left\lceil \frac{1.1}{c_{\text{spd}}} \cdot |L_k^{\mathbf{F}}(t)| \right\rceil$.*

Proof. Suppose that the preparation phase starts at time-step t^{prep}. During the preparation phase, T_k scans elements in the sub-universe $L_k^{\mathbf{F}}$, at a speed of c_{spd} elements per time-step, except for the last time-step when T_k finishes the preparation phase. On the other hand, the sub-universe $L_k^{\mathbf{F}}$ itself may be modified due to insertion/deletion of elements, but at the speed of at most one element per time-step. So, the scan finishes in Δ time-steps after t, where

$$\Delta \leq \left\lceil \frac{1}{c_{\text{spd}} - 1} \cdot |L_k^{\mathbf{F}}(t^{\text{prep}})| \right\rceil \leq \left\lceil 0.01 \cdot |L_k^{\mathbf{F}}(t^{\text{prep}})| \right\rceil.$$

Since t is a time-step within the preparation phase, we have $t \in [t^{\text{prep}}, t^{\text{prep}} + \Delta - 1]$, and hence $|L_k^{\mathbf{F}}(t)| \geq |L_k^{\mathbf{F}}(t^{\text{prep}})| - (t - t^{\text{prep}}) \geq |L_k^{\mathbf{F}}(t^{\text{prep}})| - (\Delta - 1) \geq 0.99 \cdot |L_k^{\mathbf{F}}(t^{\text{prep}})|$. Thus, we get

$$\Delta \leq \left\lceil \frac{1}{c_{\text{spd}} - 1} \cdot |L_k^{\mathbf{F}}(t^{\text{prep}})| \right\rceil \leq \left\lceil \frac{1}{0.99 \cdot (c_{\text{spd}} - 1)} \cdot |L_k^{\mathbf{F}}(t)| \right\rceil \leq \left\lceil \frac{1.1}{c_{\text{spd}}} \cdot |L_k^{\mathbf{F}}(t)| \right\rceil.$$

This concludes the proof. □

Fact 6.8. *Suppose that a background thread T_k is in the computation phase at time-step t. Then, T_k will finish computation phase by time-step $t + \left\lceil \frac{1.1}{c_{\text{spd}}} \cdot |L_k^{\mathbf{F}}(t) \setminus \text{cov}(\mathbf{B}_k(t))| \right\rceil$.*

Proof. During the computation phase, T_k keeps covering elements at a speed of c_{spd} elements per time-step. These elements are from the uncovered universe $L_k^{\mathbf{F}} \setminus \text{cov}(\mathbf{B}_k)$. The uncovered universe may be modified due to insertion/deletion of elements, but at the speed of at most one element per time-step. In contrast, the uncovered universe cannot be modified by termination of lower levels, as per Claim 6.1 (iii). This implies that the computation phase finishes in $\left\lceil \frac{1}{c_{\text{spd}}-1} \cdot |L_k^{\mathbf{F}}(t) \setminus \text{cov}(\mathbf{B}_k(t))| \right\rceil \leq \left\lceil \frac{1.1}{c_{\text{spd}}} \cdot |L_k^{\mathbf{F}}(t) \setminus \text{cov}(\mathbf{B}_k(t))| \right\rceil$ time-steps after t. □

Lemma 6.9. *Suppose that a background thread T_k is in the copy phase at time-step t. Then, T_k will finish copy phase and enter tail phase (or get aborted) by time-step $t + \left\lceil \frac{1}{c_{\text{spd}}} \cdot |\mathbf{B}_k(t)| \right\rceil$.*

Proof. During each time-step within the copy phase, the thread T_k copies c_{spd} sets from the background solution $\mathbf{B}_k$ into the buffer solution $\mathbf{R}_k$. Further, the collection of sets that define the background solution $\mathbf{B}_k$ does *not* change during the copy phase. Accordingly, the copy phase finishes in $\left\lceil \frac{1}{c_{\text{spd}}} \cdot |\mathbf{B}_k(t)| \right\rceil$ time-steps after t. □

Lemma 6.10. *Suppose that a background thread T_k is in the tail phase at time-step t. Then, T_k will terminate by time-step $t + \min\left\{ \left\lceil \frac{1.1}{c_{\text{spd}}} \cdot |L_k^{\mathbf{F}}(t) \setminus \text{cov}(\mathbf{B}_k(t))| \right\rceil, \left\lceil \frac{1.4}{c_{\text{spd}}} \cdot |\mathbf{B}_k(t)| \right\rceil \right\}$.*

Proof. Using the same argument as in the proof of Fact 6.8, we infer that T_k will terminate by time-step $t + \left\lceil \frac{1.1}{c_{\text{spd}}} \cdot |L_k^{\mathbf{F}}(t) \setminus \text{cov}(\mathbf{B}_k(t))| \right\rceil$. It now remains to show that T_k will terminate by time-step $t + \left\lceil \frac{1.2}{c_{\text{spd}}} \cdot |\mathbf{B}_k(t)| \right\rceil$. Suppose that T_k enters suspension phase, copy phase and tail phase respectively at time-steps $t^{\text{sus}}, t^{\text{copy}}$ and t^{tail}. This means that

$$t^{\text{sus}} \leq t^{\text{copy}} \leq t^{\text{tail}} \leq t, \text{ and} \tag{5}$$

$$t - t^{\text{tail}} \leq \left\lceil \frac{1.1}{c_{\text{spd}}} \cdot |L_k^{\text{F}}(t^{\text{tail}}) \setminus \text{cov}(\mathbf{B}_k(t^{\text{tail}}))| \right\rceil. \tag{6}$$

As the uncovered universe $L_k^{\text{F}} \setminus \text{cov}(\mathbf{B}_k)$ grows by at most one element per time-step during $[t^{\text{sus}}, t]$, we get

$$|L_k^{\text{F}}(t_2) \setminus \text{cov}(\mathbf{B}_k(t_2))| \leq |L_k^{\text{F}}(t_1) \setminus \text{cov}(\mathbf{B}_k(t_1))| + (t_2 - t_1), \text{ for all } t^{\text{sus}} \leq t_1 \leq t_2 \leq t. \tag{7}$$

Furthermore, by the criterion to enter suspension phase, we have

$$|L_k^{\text{F}}(t^{\text{sus}}) \setminus \text{cov}(\mathbf{B}_k(t^{\text{sus}}))| \leq |\mathbf{B}_k(t^{\text{sus}})|. \tag{8}$$

Since T_k is in the suspension phase for at most $0.1 \cdot |\mathbf{B}_k(t^{\text{sus}})|$ time-steps, it follows that

$$t^{\text{copy}} - t^{\text{sus}} \leq 0.1 \cdot |\mathbf{B}_k(t^{\text{sus}})|.$$

During the interval $[t^{\text{sus}}, t^{\text{tail}}]$, the collection of sets that define $\mathbf{B}_k$ remain unchanged. This implies that

$$|\mathbf{B}_k(t^{\text{sus}})| = |\mathbf{B}_k(t^{\text{copy}})| = |\mathbf{B}_k(t^{\text{tail}})| \tag{9}$$

Next, by Lemma 6.9, we have $t^{\text{tail}} - t^{\text{copy}} \leq \left\lceil \frac{1}{c_{\text{spd}}} \cdot |\mathbf{B}_k(t^{\text{copy}})| \right\rceil \leq \lceil 0.1 \cdot |\mathbf{B}_k(t^{\text{copy}})| \rceil$. This gives us

$$t^{\text{tail}} - t^{\text{sus}} = (t^{\text{copy}} - t^{\text{sus}}) + (t^{\text{tail}} - t^{\text{copy}}) \leq 0.11 \left(|\mathbf{B}_k(t^{\text{sus}})| + |\mathbf{B}_k(t^{\text{copy}})|\right) = 0.22|\mathbf{B}_k(t^{\text{sus}})|. \tag{10}$$

From the preceding discussion, we now derive that

$$\begin{aligned}
|L_k^{\text{F}}(t^{\text{tail}})| \setminus \text{cov}(\mathbf{B}_k(t^{\text{tail}}))| &\leq |L_k^{\text{F}}(t^{\text{sus}}) \setminus \text{cov}(\mathbf{B}_k(t^{\text{sus}}))| + (t^{\text{tail}} - t^{\text{sus}}) && \text{(Equation (7))} \\
&\leq |\mathbf{B}_k(t^{\text{sus}})| + 0.22|\mathbf{B}_k(t^{\text{sus}})| && \text{(Equations (8) and (10))} \\
&\leq 1.22 \cdot |\mathbf{B}_k(t^{\text{tail}})| && \text{(Equation (9))}.
\end{aligned} \tag{11}$$

Next, we observe that no set gets deleted from $\mathbf{B}_k$ during the tail phase. This gives us

$$|\mathbf{B}_k(t^{\text{tail}})| \leq |\mathbf{B}_k(t)|. \tag{12}$$

Finally, we conclude that

$$\begin{aligned}
|L_k^{\text{F}}(t) \setminus \text{cov}(\mathbf{B}_k(t))| &\leq |L_k^{\text{F}}(t^{\text{tail}}) \setminus \text{cov}(\mathbf{B}_k(t^{\text{tail}}))| + (t - t^{\text{tail}}) && \text{(Equation (7))} \\
&\leq 1.01 \cdot |L_k^{\text{F}}(t^{\text{tail}}) \setminus \text{cov}(\mathbf{B}_k(t^{\text{tail}}))| + 1 && \text{(Equation (6))} \\
&\leq 1.24 \cdot |\mathbf{B}_k(t^{\text{tail}})| + 0.01 \cdot |\mathbf{B}_k(t^{\text{sus}})| && \text{(Equation (11), rule of shortcut threads)} \\
&\leq 1.25 \cdot |\mathbf{B}_k(t)| && \text{(Equations (9) and (12))}
\end{aligned} \tag{13}$$

From Equation (13), we infer that T_k will terminate by time-step $t + \left\lceil \frac{1.1}{c_{\text{spd}}} \cdot |L_k^{\text{F}}(t) \setminus \text{cov}(\mathbf{B}_k(t))| \right\rceil \leq t + \left\lceil \frac{1.4}{c_{\text{spd}}} \cdot |\mathbf{B}_k(t)| \right\rceil$. This concludes the proof of the lemma. □

Corollary 6.11. *Suppose that a background thread T_k is in copy or tail phase at time-step t. Then, the thread T_k will terminate by time-step $t + \frac{5}{c_{\text{spd}}} \cdot \min\left\{\tau_k^{\text{sus}}, |\mathbf{B}_k(t)|\right\} = t + \frac{5}{c_{\text{spd}}} \cdot \tau_k^{\text{sus}}$.*

Proof. Suppose that T_k enters suspension, copy and tail phases respectively at time-steps $t^{\text{sus}}, t^{\text{copy}}$ and t^{tail}, and terminates at time-step t^{end}. This means that $t^{\text{sus}} \leq t^{\text{copy}} \leq t^{\text{tail}} \leq t^{\text{end}}$, and $t \in [t^{\text{copy}}, t^{\text{end}}]$. Now, from

the description of our algorithm, we recall that the collection of sets that define $\mathbf{B}_k$ does *not* change during the interval $[t^{\text{sus}}, t^{\text{tail}}]$. Further, no set gets deleted from $\mathbf{B}_k$ during the interval $[t^{\text{tail}}, t^{\text{end}}]$. Thus, we get

$$\tau_k^{\text{sus}} = |\mathbf{B}_k(t^{\text{sus}})| = |\mathbf{B}_k(t^{\text{copy}})| = |\mathbf{B}_k(t^{\text{tail}})| \leq |\mathbf{B}_k(t)|. \tag{14}$$

This implies that

$$\min\left\{\tau_k^{\text{sus}}, |\mathbf{B}_k(t)|\right\} = \tau_k^{\text{sus}}. \tag{15}$$

We now consider two possible cases.

Case 1: $t \in [t^{\text{tail}}, t^{\text{end}}]$. In this case, the corollary immediately follow from Equation (15) and Lemma 6.10.

Case 2: $t \in [t^{\text{copy}}, t^{\text{tail}}]$. In this case, we derive that

$$\begin{aligned}
t^{\text{end}} - t &= (t^{\text{end}} - t^{\text{tail}}) + (t^{\text{tail}} - t) \\
&\leq \frac{1.4}{c_{\text{spd}}} \cdot |\mathbf{B}_k(t^{\text{tail}})| + \frac{1}{c_{\text{spd}}} \cdot |\mathbf{B}_k(t)| + 2 && \text{(Lemmas 6.9 and 6.10)} \\
&\leq \frac{1.4}{c_{\text{spd}}} \cdot |\mathbf{B}_k(t^{\text{tail}})| + \frac{1}{c_{\text{spd}}} \cdot |\mathbf{B}_k(t)| + \frac{2}{c_{\text{spd}}} \cdot |\mathbf{B}_k(t^{\text{sus}})| && \text{(rule of shortcut threads)} \\
&\leq \frac{5}{c_{\text{spd}}} \cdot |\mathbf{B}_k(t)| && \text{(Equation (14))}
\end{aligned}$$

This concludes the proof of the corollary. □

Lemma 6.12. *Every background thread T_k remains in the suspension phase for less than $0.1 \cdot \tau_k^{\text{sus}}$ time-steps.*

The above lemma shows that removing the time limit of suspension phase results in an equivalent algorithm. We add the limit in algorithm description to simplify the proofs and eliminate circular argument.

We combine the results in this section as follows.

Lemma 6.13. *Suppose that a background thread T_k is running at time-step t. Then, the lifetime of T_k is at most $0.2 \cdot |L_k^{\text{F}}(t)|$ time-steps.*

Proof. Suppose T_k starts at t^{prep} and terminates at t^{end}. If $|L_k^{\text{F}}(t^{\text{prep}})| \leq c_{\text{spd}} - 1$, then T_k terminate in one time-step by the rule of base threads. In this case, the statement holds. In the rest of the proof, we may assume $|L_k^{\text{F}}(t^{\text{prep}})| \geq c_{\text{spd}}$, i.e. $1 \leq \frac{|L_k^{\text{F}}(t^{\text{prep}})|}{c_{\text{spd}}}$.

T_k may terminate in one of the three cases: (1) normal termination at tail phase, (2) normal termination by rule of shortcut threads at computation phase, and (3) aborted by a higher thread. We may assume case (1) w.l.o.g., because its upper bound subsumes the other cases.

Suppose that T_k enters computation phases, suspension phase, and copy phase respectively at time-steps t^{comp}, t^{sus} and t^{copy}. We have the following time bounds:

$$\begin{aligned}
t^{\text{comp}} - t^{\text{prep}} &\leq \left\lceil \frac{1.1}{c_{\text{spd}}} \cdot |L_k^{\text{F}}(t^{\text{prep}})| \right\rceil && \text{(Fact 6.7)} \\
t^{\text{sus}} - t^{\text{comp}} &\leq \left\lceil \frac{1.1}{c_{\text{spd}}} \cdot |L_k^{\text{F}}(t^{\text{comp}})| \right\rceil && \text{(Fact 6.8)} \\
t^{\text{copy}} - t^{\text{sus}} &\leq \left\lceil 0.1 \cdot \tau_k^{\text{sus}} \right\rceil && \text{(Lemma 6.12)} \\
t^{\text{end}} - t^{\text{copy}} &\leq \frac{5}{c_{\text{spd}}} \cdot \tau_k^{\text{sus}} && \text{(Corollary 6.11)}
\end{aligned}$$

We will show that $t^{\text{end}} - t^{\text{prep}} \leq \Delta := 0.2 \cdot |L_k^{\mathbf{F}}(t^{\text{prep}})|$. During $t \in [t^{\text{prep}}, t^{\text{prep}} + \Delta]$, $|L_k^{\mathbf{F}}(t)|$ can change by at most 1 per time-step, so we always have $0.8|L_k^{\mathbf{F}}(t^{\text{prep}})| \leq |L_k^{\mathbf{F}}(t)| \leq 1.2|L_k^{\mathbf{F}}(t^{\text{prep}})|$. To bound $\tau_k^{\text{sus}} = |\mathbf{B}_k(t^{\text{sus}})|$, we consider a bijection from each set in $\mathbf{B}_k(t^{\text{sus}})$ to an arbitrary element in its coverage. These elements are distinct and belong to $L_k^{\mathbf{F}}$ when the set is added by the greedy algorithm. So, $\tau_k^{\text{sus}} \leq 1.2|L_k^{\mathbf{F}}(t^{\text{prep}})|$. In conclusion,

$$t^{\text{end}} - t^{\text{prep}} \leq (1.1 + 1.1 \times 1.2 + 0.1c_{\text{spd}} + 5 + 3)\frac{1}{c_{\text{spd}}}|L_k^{\mathbf{F}}(t^{\text{prep}})| \leq 0.15|L_k^{\mathbf{F}}(t^{\text{prep}})| \leq 0.2|L_k^{\mathbf{F}}(t)|$$

□

6.2.1 Proof of Lemma 6.12

Suppose that the thread T_k enters the suspension phase at time-step t^{sus} and ends at time-step $t^{\text{end}} > t^{\text{sus}}$. From the algorithm description, this can potentially happen because of one of the following three reasons.

- (i) T_k normally terminates at time-step t^{end}.
- (ii) T_k gets aborted because another thread T_j, with $j > k$, normally terminates at time-step t^{end}.
- (iii) T_k spends $0.1 \cdot \tau_k^{\text{sus}}$ time-steps in the suspension phase, and $t^{\text{end}} = t^{\text{sus}} + \lfloor 0.1 \cdot \tau_k^{\text{sus}} \rfloor$.

We will show that T_k ends because of reason (i) or reason (ii) (and *never* because of reason (iii)). Specifically, suppose that T_k is in suspension phase during the interval $[t^{\text{sus}}, t^{\text{sus}} + \Delta_{\text{sus}}]$, for some integer $\Delta_{\text{sus}} \geq 0$. We will show:

$$\Delta_{\text{sus}} \leq 0.1 \cdot \tau_k^{\text{sus}}. \tag{16}$$

Consider any $t \in [t^{\text{sus}}, t^{\text{sus}} + \Delta_{\text{sus}} - 1]$. We say that T_k is *blocked-from-above at t* iff there is some other thread T_j, with $j > k$ and $\frac{1}{2} \cdot \tau_j^{\text{sus}} < \tau_k^{\text{sus}}$, that is in either copy or tail phase at the end of time-step t. In contrast, we say that T_k is *blocked-from-below at t* iff (a) it is *not* blocked-from-above at t, and (b) there is some other thread T_j, with $j < k$ and $\frac{1}{2} \cdot \tau_j^{\text{sus}} < \tau_k^{\text{sus}}$, that is in either copy or tail phase at the end of time-step t. The observation below follows from the description of our algorithm (see Algorithm 8).

Observation 6.14. *T_k is either blocked-from-above or blocked-from-below at every $t \in [t^{\text{sus}}, t^{\text{sus}} + \Delta_{\text{sus}} - 1]$.*

Claim 6.15. *Suppose that the thread T_k is blocked-from-above at some time-step $t \in [t^{\text{sus}}, t^{\text{sus}} + \Delta_{\text{sus}} - 1]$. Then, the thread T_k must terminate before time-step $t + \frac{10}{c_{\text{spd}}} \cdot \tau_k^{\text{sus}}$.*

Proof. Since T_k is blocked-from-above at some time-step $t \in [t^{\text{sus}}, t^{\text{sus}} + \Delta - 1]$, there exists another thread T_j, with $j > k$ and $\frac{1}{2} \cdot \tau_j^{\text{sus}} < \tau_k^{\text{sus}}$, that is in either copy or tail phase at time-step t. Then, Corollary 6.11 guarantees that T_j will terminate by time-step $t + \frac{5}{c_{\text{spd}}} \cdot \tau_j^{\text{sus}} < t + \frac{10}{c_{\text{spd}}} \cdot \tau_k^{\text{sus}}$. Finally, note that since $j > k$, the thread T_k gets aborted when T_j terminates. □

Let $\mathcal{T}^{\text{down}}(t)$ denote the collection of threads T_j, with $j < k$ and $\frac{1}{2} \cdot \tau_j^{\text{sus}} < \tau_k^{\text{sus}}$, that are in either copy or tail phase at the end of time-step t. By Claim 6.15, if T_k is blocked-from-above at time-step t^{sus}, then $\Delta_{\text{sus}} < \frac{10}{c_{\text{spd}}} \cdot \tau_k^{\text{sus}} < 0.1 \cdot \tau_k^{\text{sus}}$. This immediately implies Lemma 6.12. Accordingly, throughout the rest of the proof, we make the following assumption.

Assumption 6.16. *The thread T_k is* not *blocked-from-above at time-step t^{sus}, and $\Delta_{\text{sus}} \geq 1$.*

Corollary 6.17. *We have $\mathcal{T}^{\text{down}}(t^{\text{sus}}) \neq \emptyset$.*

Proof. Follows from Observation 6.14 and Assumption 6.16. □

Let $t^\star(T_j)$ denote the last time-step during which a given thread T_j is in either copy or tail phase. For some integer $r \geq 0$, we now define a sequence of r consecutive intervals $[t_1, t_2 - 1], [t_2, t_3 - 1], \ldots, [t_r, t_{r+1} - 1]$, and a sequence of indices $j_1, j_2, \ldots, j_r$, according to the procedure described in Algorithm 9.

Algorithm 9: Constructing the intervals $[t_1, t_2 - 1], \ldots, [t_r, t_{r+1} - 1]$ and the indices $j_1, \ldots, j_r$.

9.1 $t_1 \leftarrow t^{\text{sus}}$
9.2 $T_{j_1} \leftarrow \arg\max_{T_j \in \mathcal{T}^{\text{down}}(t_1)} \{t^\star(T_j)\}$, breaking ties arbitrarily (see Corollary 6.17).
9.3 $t_2 \leftarrow t^\star(T_{j_1}) + 1$
9.4 $i \leftarrow 1$.
9.5 **while** $\mathcal{T}^{\text{down}}(t_{i+1} - 1) \setminus \{T_{j_i}\} \neq \emptyset$ **do**
9.6 $\quad T_{j_{i+1}} \leftarrow \arg\max_{T_j \in \mathcal{T}^{\text{down}}(t_{i+1}-1)} \{t^\star(T_j)\}$, breaking ties arbitrarily.
9.7 $\quad t_{i+2} \leftarrow t^\star(T_{j_{i+1}}) + 1$
9.8 $\quad i \leftarrow i + 1$
9.9 $r \leftarrow i$

Observation 6.18. *For all $i \in [1, r-1]$, we have $\tau^{\text{sus}}_{j_{i+1}} \leq \frac{1}{2} \cdot \tau^{\text{sus}}_{j_i}$.*

Proof. By definition, we have $T_{j_{i+1}} \in \mathcal{T}^{\text{down}}(t_{i+1} - 1)$, and hence $t^\star(T_{j_{i+1}}) \geq t_{i+1} > t^\star(T_{j_i})$. It follows that $T_{j_{i+1}} \notin \mathcal{T}^{\text{down}}(t_i - 1)$, for otherwise we would have defined $j_i \leftarrow j_{i+1}$ at time-step $t_i - 1$. Accordingly, the thread $T_{j_{i+1}}$ enters copy phase during some time-step $t \in [t_i, t_{i+1} - 1]$, when thread T_{j_i} is in either copy or tail phase. Thus, from the description of Algorithm 8, we have $\tau^{\text{sus}}_{j_{i+1}} \leq \frac{1}{2} \cdot \tau^{\text{sus}}_{j_i}$. □

In particular, Observation 6.18 implies that $r = O(\log n)$, because $\tau^{\text{sus}}_j \in [1, n]$ for all background threads T_j.

Observation 6.19. *Suppose that $t_{r+1} \in [t^{\text{sus}}, t^{\text{sus}} + \Delta_{\text{sus}} - 1]$. Then, the thread T_k is blocked-from-above at t_{r+1}.*

Proof. From the description of Algorithm 9, we have $\mathcal{T}^{\text{down}}(t_{r+1} - 1) \setminus \{T_{j_r}\} = \emptyset$. Furthermore, since $t_{r+1} - 1 = t^\star(T_{j_r})$, the thread T_{j_r} itself is no longer in copy or tail phase after the end of time-step $t_{r+1} - 1$. This means that at the start of time-step t_{r+1}, every thread T_j with $j < k$ that is in copy or tail phase has $\frac{1}{2} \cdot \tau^{\text{sus}}_j \geq \tau^{\text{sus}}_k$. Nevertheless, since we have assumed that $t_{r+1} \in [t^{\text{sus}}, t^{\text{sus}} + \Delta_{\text{sus}} - 1]$, the thread T_k does *not* enter copy or tail phase during time-step t_{r+1}. This can happen only if there exists some other thread T_j, with $j > k$ and $\frac{1}{2} \cdot \tau^{\text{sus}}_j < \tau^{\text{sus}}_k$, that is in copy or tail phase at the end of step t_{r+1}. By definition, this means that the thread T_k is blocked-from-above at t_{r+1}. □

Observation 6.20. *We have $t_{r+1} - t_1 \leq \frac{20}{c_{\text{spd}}} \cdot \tau^{\text{sus}}_k$.*

Proof. Consider any $i \in [1, r]$. From the description in Algorithm 9, it follows that the thread T_{j_i} is in copy or tail phase at every time-step $t \in [t_i, t_{i+1} - 1]$. Accordingly, Corollary 6.11 guarantees that

$$t_{i+1} - t_i \leq \frac{5}{c_{\text{spd}}} \cdot \tau^{\text{sus}}_{j_i}.$$

Summing the above inequality over all $i \in [1, r]$, we get:

$$t_{r+1} - t_1 \leq \frac{5}{c_{\text{spd}}} \cdot \sum_{i=1}^{r} \tau^{\text{sus}}_{j_i} \leq \frac{10}{c_{\text{spd}}} \cdot \tau^{\text{sus}}_{j_1} < \frac{20}{c_{\text{spd}}} \cdot \tau^{\text{sus}}_k.$$

The penultimate step in the above derivation follows from Observation 6.18, whereas the last step holds because $T_{j_1} \in \mathcal{T}^{\text{down}}(t_1)$. □

We are now ready to complete the proof of Lemma 6.12. There are two cases to consider.

Case 1: $t_{r+1} \in [t^{\text{sus}}, t^{\text{sus}} + \Delta_{\text{sus}} - 1]$. In this case, Claim 6.15, Observation 6.19 and Observation 6.20 together imply that T_k terminates before time-step $t_{r+1} + \frac{10}{c_{\text{spd}}} \cdot \tau_k^{\text{sus}} \leq t_1 + \frac{30}{c_{\text{spd}}} \cdot \tau_k^{\text{sus}} = t^{\text{sus}} + \frac{30}{c_{\text{spd}}} \cdot \tau_k^{\text{sus}} \leq t^{\text{sus}} + 0.1 \cdot \tau_k^{\text{sus}}$. This concludes the proof of Lemma 6.12.

Case 2: $t_{r+1} \notin [t^{\text{sus}}, t^{\text{sus}} + \Delta_{\text{sus}} - 1]$. In this case, we clearly have $\Delta_{\text{sus}} \leq t_{r+1} - t^{\text{sus}} = t_{r+1} - t_1 \leq \frac{20}{c_{\text{spd}}} \cdot \tau_k^{\text{sus}} < 0.1 \cdot \tau_k^{\text{sus}}$. Here, the penultimate inequality follows from Observation 6.20. This concludes the proof of Lemma 6.12.

6.3 Analyzing Recourse

In this section, we establish a bound of $O(\log n)$ on the worst-case recourse of our dynamic set cover algorithm. We separate the total recourse into *insertion recourse* and *deletion recourse*. Insertion recourse measures the maximum number of sets that are added to the output of the algorithm in any time-step. Similarly, deletion recourse refers to the maximum number of sets that are removed from the output in any time-step. First, we bound the insertion recourse of the algorithm in the next lemma.

Lemma 6.21. *The insertion recourse is* $O(\log n)$.

Proof. Note that the foreground solution $\mathbf{F}$ and buffer solutions $\mathbf{R}_k$ are part of the output of the algorithm. A background thread T_k copies sets from the background solution $\mathbf{B}_k$ to the corresponding buffer solution $\mathbf{R}_k$ at the rate of $c_{\text{spd}} = O(1)$ sets per time-step in the copy and tail phases. Since there are at most $\ell_{\max} = O(\log n)$ background threads in copy or tail phases in any time-step, the buffer solutions cumulatively add at most $O(\log n)$ sets in a time-step. Now, consider the foreground solution $\mathbf{F}$. Sets are added to $\mathbf{F}$ in two ways. First, a single set may be added to $\mathbf{F}$ on the insertion of an element that does not belong to any set in $\mathbf{F}$. Second, $\mathbf{F}$ is updated after normal termination of a background thread T_k. But, this process simply adds to $\mathbf{F}$ the sets in $\mathbf{B}_k$ that are already in the buffer solution $\mathbf{R}_k$. Therefore, this latter step does not add to insertion recourse of the algorithm. Overall, this means that the insertion recourse is $O(\log n)$. □

We are left to bound the deletion recourse of the algorithm. First, note that the worst-case bound on insertion recourse immediately implies the same bound on deletion recourse, but only in an amortized sense. So, we need to de-amortize the deletion recourse of the algorithm. Instead of arguing specifically about our algorithm, we prove a more general property about de-amortizing deletion recourse. Consider any online optimization problem where a solution comprises a set of objects, and the goal is to minimize the number of objects subject to a feasibility constraint that changes online. Let us call such a problem an online unweighted minimization problem. We proved a general reduction for de-amortizing deletion recourse in Lemma 3.6, which we restate below.

Lemma 3.6. *Consider an online unweighted minimization problem, where the size of the optimal solution changes by at most* δ *in each time-step. Suppose there exists an algorithm* $\mathcal{A}$ *that maintains an* α*-approximate solution and has worst-case insertion recourse at most* β*, for some parameters* $\alpha, \beta \geq 1$*. (The worst-case deletion recourse of algorithm* $\mathcal{A}$ *can be arbitrarily large.) Then, there exists an alternative algorithm* $\mathcal{A}'$ *that also maintains an* α*-approximate solution and has worst-case recourse (both insertion and deletion) at most* $2\beta + \delta\alpha$*.*

6.4 Approximation Factor

We start by introducing the notion of an *extended background solution*, which will be used in our analysis throughout the rest of this section.

Extended background solution. Let T_k be a background thread at some time-step t. Consider a thought experiment where we complete the execution of the remaining operations specified by Algorithm 5 at the

present time-step t, and let $\widetilde{\mathbf{B}}_k(t)$ denote the solution returned by this algorithm. Note that $\widetilde{\mathbf{B}}_k(t) \supseteq \mathbf{B}_k(t)$ and $\text{cov}(\widetilde{\mathbf{B}}_k(t)) \supseteq L_k^{\mathrm{F}}(t)$. In other words, $\widetilde{\mathbf{B}}_k(t)$ is a feasible hierarchical solution for the input defined by $L_k^{\mathrm{F}}(t)$. We define $\widetilde{\mathbf{B}}_k(t)$ to be the *extended background solution* of the thread T_k at time-step t. Recall that we say $\widetilde{\mathbf{B}}_k(t)$ is ϵ-dirty at some level j if $\left|P_j^{\widetilde{\mathbf{B}}_k(t)}\right| > \epsilon \cdot \left|A_j^{\widetilde{\mathbf{B}}_k(t)}\right|$, and ϵ-tidy at level j otherwise.

Lemma 6.22. *Consider any background thread T_k at any time-step t. Then, $\widetilde{\mathbf{B}}_k(t)$ is 0.2-tidy at every level $j \leq k$.*

Proof. Fix any level $j \leq k$. Let t^{comp} denote the time-step at which the background thread T_k enters the computation phase, and let $t^{\text{end}} > t^{\text{comp}}$ denote the time-step at which T_k terminates. Recall that at time-step t^{comp}, the copy pointer p_k of the background thread T_k is initialized to $k+1 > j$. Furthermore, the value of p_k can only decrease over time (Claim 6.4). Based on this observation, we next define a time-step t_j, as follows.

- If $p_k > j$ at time-step t^{end}, then we define $t_j := t^{\text{end}}$.
- Otherwise, we define t_j to be the first (i.e., smallest) time-step at which p_k becomes equal to j.

At time-step t^{comp}, we have $\text{plev}^{\mathbf{B}_k}(e) \geq k+1$ for all elements $e \in L_k^{\mathrm{F}}$. Subsequently, the passive levels in $\mathbf{B}_k$ can only be modified due to insertions/deletions of elements. To be more specific, while handling the insertion of an element e, the thread T_k ensures that $\text{plev}^{\mathbf{B}_k}(e) \geq p_k$ (see Algorithm 7). Similarly, while handling the deletion of an element e, if $e \in \text{cov}(\mathbf{B}_k)$ then the thread T_k ensures that $\text{plev}^{\mathbf{B}_k}(e) \geq p_k$; otherwise T_k simply removes e from its uncovered sub-universe (see Algorithm 6). This implies that up until the start of time-step t_j, no element e has $\text{plev}^{\mathbf{B}_k}(e) \leq j$ (including t_j, since at time-step t_j, T_k handles insertion/deletion before the greedy algorithm drops p_k to $\leq j$). In other words, we have

$$P_j^{\widetilde{\mathbf{B}}_k}(t') = \emptyset \text{ for all time-steps } t' \in [t^{\text{comp}}, t_j]. \tag{17}$$

If $t \leq t_j$, then Equation (17) immediately implies the lemma. Accordingly, for the rest of the proof, we assume:

$$t_j < t \leq t^{\text{end}}. \tag{18}$$

During any given time-step within the interval $[t_j + 1, t]$, only the element e being inserted/deleted can potentially have its passive level in $\widetilde{\mathbf{B}}_k$ set to $\leq j$, whereas the passive level of every other element in $\widetilde{\mathbf{B}}_k$ remains unchanged. Thus, from Equation (18), we infer that

$$\left|P_j^{\widetilde{\mathbf{B}}_k}(t)\right| \leq t - t_j. \tag{19}$$

We now fork into each of two possible cases, one after another.

Case 1: During the interval $[t_j+1, t]$, the thread T_k does not exit computation phase. Here, we observe that from time-step $t_j + 1$ onward, the thread T_k only adds sets at levels $\leq j$ to $\mathbf{B}_k$. During the computation phase, T_k covers c_{spd} elements per time-step, except for the last time-step of computation phase. Since we assumed T_k does not exit computation phase, the latter case cannot happen. So, during $[t_j + 1, t]$, T_k covers c_{spd} elements at levels $\leq j$ per time-step. It follows that

$$\left|C_j^{\widetilde{\mathbf{B}}_k}(t)\right| \geq \left|C_j^{\mathbf{B}_k}(t)\right| = c_{\text{spd}} \cdot (t - t_j). \tag{20}$$

From Equation (19) and Equation (20), we get $\left|C_j^{\widetilde{\mathbf{B}}_k}(t)\right| \geq c_{\text{spd}} \cdot \left|P_j^{\widetilde{\mathbf{B}}_k}(t)\right|$. Since $A_j^{\widetilde{\mathbf{B}}_k}(t) = C_j^{\widetilde{\mathbf{B}}_k}(t) \setminus P_j^{\widetilde{\mathbf{B}}_k}(t)$, this implies that $\widetilde{\mathbf{B}}_k$ is ϵ-tidy at level j at time-step t, and concludes the proof of the lemma.

Case 2: During the interval $[t_j + 1, t]$, the thread T_k is in suspension or copy phase for Δ time-steps, for some $\Delta > 0$. Here, let $t^{\text{sus}} \in [t_j, t]$ denote the time-step at which T_k enters the suspension phase. Since the sets that define the background solution $\mathbf{B}_k$ do *not* change during suspension or copy phase, throughout the duration of the suspension or copy phase, we continue to have $|\mathbf{B}_k| = \tau_k^{\text{sus}} = |\mathbf{B}_k(t^{\text{sus}})|$. By the rule of shortcut threads, $|\mathbf{B}_k(t^{\text{sus}})| > c_{\text{spd}}$. Accordingly, Lemma 6.9 implies that

$$\Delta < 0.1 \cdot |\mathbf{B}_k(t^{\text{sus}})| + \frac{1}{c_{\text{spd}}} \cdot |\mathbf{B}_k(t^{\text{sus}})| + 2 \leq 0.11 \cdot |\mathbf{B}_k(t^{\text{sus}})|. \tag{21}$$

Next, observe that during the interval $[t_j + 1, t]$, the thread T_k covers c_{spd} elements at levels $\leq j$ per time-step, except when T_k is in suspension or copy phase. Thus, we have

$$\left|C_j^{\widetilde{\mathbf{B}}_k}(t)\right| \geq \left|C_j^{\mathbf{B}_k}(t)\right| = c_{\text{spd}} \cdot (t - t_j - \Delta). \tag{22}$$

Since the thread T_k enters the suspension phase at time-step t^{sus}, we must necessarily have $|\mathbf{B}_k(t^{\text{sus}})| \leq |L_k^{\mathbf{F}}(t^{\text{sus}}) \setminus \text{cov}(\mathbf{B}_k(t^{\text{sus}}))|$. Further, since $t_j \leq t^{\text{sus}}$, all the elements in $L_k^{\mathbf{F}}(t^{\text{sus}}) \setminus \text{cov}(\mathbf{B}_k(t^{\text{sus}}))$ are covered in $\widetilde{\mathbf{B}}_k(t^{\text{sus}})$ at levels $\leq j$. This implies that

$$|\mathbf{B}_k(t^{\text{sus}})| \leq \left|L_k^{\mathbf{F}}(t^{\text{sus}}) \setminus \text{cov}(\mathbf{B}_k(t^{\text{sus}}))\right| \leq \left|C_j^{\widetilde{\mathbf{B}}_k}(t^{\text{sus}})\right|. \tag{23}$$

Next, we compare the two sets $C_j^{\widetilde{\mathbf{B}}_k}(t)$ and $C_j^{\widetilde{\mathbf{B}}_k}(t^{\text{sus}})$. Specifically, we note if an element appears in one of these sets but not in the other, then that element must have been inserted or deleted during the interval $[t^{\text{sus}} + 1, t]$. This gives us

$$\left|C_j^{\widetilde{\mathbf{B}}_k}(t^{\text{sus}})\right| \leq \left|C_j^{\widetilde{\mathbf{B}}_k}(t)\right| + (t - t^{\text{sus}}) \leq \left|C_j^{\widetilde{\mathbf{B}}_k}(t)\right| + (t - t_j). \tag{24}$$

We now put together all these observations, and derive that

$$\begin{aligned}
\left|C_j^{\widetilde{\mathbf{B}}_k}(t)\right| &\geq c_{\text{spd}} \cdot (t - t_j - \Delta) && \text{(Equation (22))} \\
&\geq c_{\text{spd}} \cdot \left((t - t_j) - 0.11 \cdot |\mathbf{B}_k(t^{\text{sus}})|\right) && \text{(Equation (21))} \\
&\geq c_{\text{spd}} \cdot \left((t - t_j) - 0.11 \cdot \left(\left|C_j^{\widetilde{\mathbf{B}}_k}(t)\right| + (t - t_j)\right)\right) && \text{(Equations (23) and (24))} \\
(1 + 0.11 c_{\text{spd}}) \left|C_j^{\widetilde{\mathbf{B}}_k}(t)\right| &\geq 0.89 c_{\text{spd}} \cdot (t - t_j) \geq 0.89 c_{\text{spd}} \cdot \left|P_j^{\widetilde{\mathbf{B}}_k}(t)\right| && \text{(Equation (19))} \\
\left|P_j^{\widetilde{\mathbf{B}}_k}(t)\right| &\leq \frac{1 + 0.11 c_{\text{spd}}}{0.89 c_{\text{spd}}} \cdot \left|C_j^{\widetilde{\mathbf{B}}_k}(t)\right| \leq 0.2 \cdot \left|C_j^{\widetilde{\mathbf{B}}_k}(t)\right|
\end{aligned}$$

Since $A_j^{\widetilde{\mathbf{B}}_k}(t) = C_j^{\widetilde{\mathbf{B}}_k}(t) \setminus P_j^{\widetilde{\mathbf{B}}_k}(t)$, rearranging the terms in the last inequality, we infer that $\widetilde{\mathbf{B}}_k$ is 0.2-tidy at level j at time-step t. This concludes the proof of the lemma. □

Lemma 6.23. *The foreground solution $\mathbf{F}$ is 0.5-tidy at the end of every time-step.*

Proof. We prove by induction on time-step t. As the base case, the initial greedy solution is 0-tidy. For the inductive case, it suffices to prove the following claim for every level $k \in [0, \ell_{\max}]$: Suppose $\mathbf{F}$ is 0.2-tidy at level k just before T_k starts. Then, $\mathbf{F}$ is 0.2-tidy at level k when T_k terminates, and $\mathbf{F}$ is 0.5-tidy at level k throughout the lifetime of T_k.

We now prove the claim. Let t_1 be the last time-step before T_k starts, and t_2 be the time-step that T_k terminates. At t_2, T_k may normally terminate or aborted by a higher thread. In both cases, the normal termination of T_k

or a higher thread will replace levels $\leq k$ of $\mathbf{F}$ by the corresponding background solution. By Lemma 6.22, the background solution (which is identical to the extended background solution at normal termination) is 0.1-tidy at level k. This establishes the first part of the claim.

For the second part of the claim, we have $t - t_1 \leq 0.2 \cdot |L_k(t)|$ for every $t \in [t_1 + 1, t_2]$ by Lemma 6.13. So for every $t \in [t_1 + 1, t_2]$,

$$\begin{aligned}\frac{|P_k(t)|}{|A_k(t)|} &\leq \frac{|P_k(t_1)| + (t - t_1)}{|A_k(t)|} \leq \frac{0.2|A_k(t_1)| + (t - t_1)}{|A_k(t)|} \\ &\leq \frac{0.2(|A_k(t)| + (t - t_1)) + (t - t_1)}{|A_k(t)|} \leq 0.2 + 1.2 \cdot \frac{0.2|L_k(t)|}{|A_k(t)|} \leq 0.44\end{aligned}$$

Here, the last inequality uses $|L_k^{\mathbf{F}}| \subseteq |A_k^{\mathbf{F}}|$, which holds because all dormant elements in $\mathrm{cov}(\mathbf{F})$ are universally passive. This establishes the second part of the claim. □

Lemma 6.24. *Consider any background thread T_k that starts (i.e., enters preparation phase) at time-step t^{prep} and terminates at time-step $t^{\mathrm{end}} > t^{\mathrm{prep}}$. Suppose that the foreground solution $\mathbf{F}$ is ϵ-stable at time-step t^{prep}. Then, the background solution $\mathbf{B}_k$ is also ϵ-stable at every time-step $t \in [t^{\mathrm{prep}}, t^{\mathrm{end}}]$.*

Proof. To prove the lemma, we need to show that

$$\left|A_j^{\mathbf{B}_k}(t) \cap s\right| < (1+\epsilon)^{j+1} \text{ for all sets } s \in \mathcal{S}, \text{ time-steps } t \in [t^{\mathrm{prep}}, t^{\mathrm{end}}] \text{ and levels } j. \tag{25}$$

As the thread T_k never assigns an element to a level $> (k+1)$, we always have $A_j^{\mathbf{B}_k} = A_{k+1}^{\mathbf{B}_k}$ for all $j > (k+1)$. Accordingly, we only need to prove Equation (25) for levels $j \leq k+1$.

We first focus on the case where $j = k+1$. We start by observing that $A_{k+1}^{\widetilde{\mathbf{B}}_k}(t^{\mathrm{prep}}) \subseteq A_k^{\mathbf{F}}(t^{\mathrm{prep}})$. Since $\mathbf{F}$ is ϵ-stable (see Definition 2) at time-step t^{prep}, we get

$$\left|A_{k+1}^{\widetilde{\mathbf{B}}_k}(t^{\mathrm{prep}}) \cap s\right| \leq \left|A_k^{\mathbf{F}}(t^{\mathrm{prep}}) \cap s\right| < (1+\epsilon)^{k+1} \text{ for all sets } s \in \mathcal{S}. \tag{26}$$

Subsequently, the thread T_k changes (resp. create) the passive level of an element only when it gets deleted (resp. inserted). Furthermore, Algorithm 6 and Algorithm 7 ensure that T_k sets the passive levels of the elements being inserted/deleted to $\leq (k+1)$. Thus, we have $A_{k+1}^{\widetilde{\mathbf{B}}_k}(t^{\mathrm{prep}}) \supseteq A_{k+1}^{\widetilde{\mathbf{B}}_k}(t)$ at each time-step $t \in [t^{\mathrm{prep}}, t^{\mathrm{end}}]$, and so Equation (26) implies that

$$\left|A_{k+1}^{\widetilde{\mathbf{B}}_k}(t) \cap s\right| < (1+\epsilon)^{k+1} \text{ for all sets } s \in \mathcal{S} \text{ and all } t \in [t^{\mathrm{prep}}, t^{\mathrm{end}}]. \tag{27}$$

Since we always have $A_j^{\mathbf{B}_k} \subseteq A_j^{\widetilde{\mathbf{B}}_k}$, Equation (27) implies that $\mathbf{B}_k$ satisfies Equation (25) for level $j = k+1$.

For the rest of the proof, we focus on a level $j \leq k$. Let $t_j \in [t^{\mathrm{prep}}, t^{\mathrm{end}}]$ be the first (i.e., smallest) time-step at which the copy pointer p_k of the thread T_k becomes equal to j (see Claim 6.4). Otherwise, if $p_k(t) > j$ for all $t \in [t^{\mathrm{prep}}, t^{\mathrm{end}}]$, then we define $t_j := t^{\mathrm{end}}$. Now, we consider two possible cases, depending on the value of t.

Case 1: $t < t_j$. In this case, we have $A_j^{\mathbf{B}_k}(t) = \emptyset$, and so Equation (25) trivially holds.

Case 2: $t \geq t_j$. Here, we consider two sub-cases, depending on the status of the set s. First, suppose that $s \in \mathbf{B}_k$ and $\mathrm{lev}^{\mathbf{B}_k}(s) \geq j+1$. In this sub-case, every element $e \in s$ is assigned a level $\mathrm{lev}^{\mathbf{B}_k}(e) \geq j+1$, and this level cannot change in future. Furthermore, whenever an element e that belongs to s gets inserted at some future time-step $t' \geq t_j$, the thread T_k would assign e to a level $\geq j+1$ (see Algorithm 7). Thus, we have $A_j^{\mathbf{B}_k}(t) \cap s = \emptyset$, and so Equation (25) trivially holds.

Finally, consider the remaining sub-case that $s \notin \mathbf{B}_k$ or $\text{lev}^{\mathbf{B}_k}(s) \leq j$. Let s_j be the set that causes p_k to drop to $\leq j$ in the greedy algorithm at time-step t_j. In this case, s is not added to $\mathbf{B}_k$ before s_j is selected by the greedy algorithm. According to Algorithm 5, when the greedy algorithm selects s_j with $\text{lev}(s_j) \leq j$, it cannot find any set with marginal coverage $\geq (1+\epsilon)^{j+1}$. This implies that

$$\left| s \cap \left(L_k^{\mathbf{F}}(t_j) \setminus \text{cov}(\mathbf{B}_k(t_j)) \right) \right| < (1+\epsilon)^{j+1}. \tag{28}$$

Now, consider any element $e \in A_j^{\mathbf{B}_k}(t)$. Clearly, the element e either belongs to $L_k^{\mathbf{F}}(t_j) \setminus \text{cov}(\mathbf{B}_k(t_j))$, or it gets inserted at some time-step $t' \in [t_j, t]$. In the latter scenario, however, the thread T_k would either make the element e universally passive upon insertion, or set $\text{plev}^{\mathbf{B}_k}(e)$ to $p_k(t') \leq j$. This implies that

$$A_j^{\mathbf{B}_k}(t) \subseteq L_k^{\mathbf{F}}(t_j) \setminus \text{cov}(\mathbf{B}_k(t_j)). \tag{29}$$

From the preceding discussion, we derive that

$$\begin{aligned} \left| A_j^{\mathbf{B}_k}(t) \cap s \right| &\leq \left| s \cap \left(L_k^{\mathbf{F}}(t_j) \setminus \text{cov}(\mathbf{B}_k(t_j)) \right) \right| && \text{(Equation (29))} \\ &< (1+\epsilon)^{j+1} && \text{(Equation (28))} \end{aligned}$$

In other words, Equation (25) holds. This concludes the proof of the lemma. □

Lemma 6.25. *The foreground solution* $\mathbf{F}$ *is always* ϵ*-stable.*

Proof. The foreground solution $\mathbf{F}$ can change only because of one of the following three events.

- (i) Insertion of an element.
- (ii) Deletion of an element.
- (iii) A background thread T_k normally terminates, and accordingly we remove all the sets in $\mathbf{F}$ at levels $\leq k$, and replace them with the sets in $\mathbf{B}_k$. We refer to this event as a *background-switch by thread* T_k.

During event (i), the element being inserted is made universally passive in $\mathbf{F}$. This does not affect the active coverages. On the other hand, during event (ii), the element being deleted becomes universally passive. This may remove the element from some active coverages, which can only make the stable property weaker. So, the foreground solution $\mathbf{F}$ can never cease to be ϵ-stable due to the insertion or deletion of an element.

For the rest of the proof, we focus on the background-switch by a thread T_k at (say) time-step t. Let $t^{\text{prep}} < t$ be the time-step at which the thread T_k gets created (i.e., enters the preparation phase). (By the rule of one phase per time-step, T_k starts to work at the subsequent time-step after it gets created at t^{prep}. So, $t^{\text{prep}} < t$ even if T_k terminates in one time-step by the rule of base threads.) We perform an induction over time-steps. By our induction hypothesis, the foreground solution $\mathbf{F}$ is ϵ-stable at all time-steps $< t$. Since $t^{\text{prep}} < t$, by Lemma 6.24 the background solution $\mathbf{B}_k$ is ϵ-stable at time-step t. Furthermore, Claim 6.1 guarantees that $\text{cov}(\mathbf{B}_k) \cap L = L_k^{\mathbf{F}}$ just before the background-switch by thread T_k at time-step t. So, immediately after the background-switch by thread T_k at time-step t, we continue to have

$$\left| A_j^{\mathbf{F}} \cap s \right| = \left| A_j^{\mathbf{B}_k} \cap s \right| < (1+\epsilon)^{j+1} \text{ for all sets } s \in \mathcal{S} \text{ and levels } j \leq k. \tag{30}$$

The background thread T_k sets the levels of elements in $L_k^{\mathbf{F}}$ to be $\leq k+1$. For the passive levels, we have the following claim before the switch: T_k sets $\text{plev}^{\mathbf{B}_k}(e) > k+1$ if and only if $\text{plev}^{\mathbf{F}}(e) > k+1$ and $e \in L_k^{\mathbf{F}}$. This is because every element e in the initial sub-universe is assigned $\text{plev}^{\mathbf{B}_k}(e) = \max\{\text{plev}^{\mathbf{F}}(e), k+1\}$, and

every new elements e inserted to $L_k^{\mathbf{F}}$ is assigned $\text{plev}^{\mathbf{B}_k}(e) \le k+1$. From the claim, we have that for all levels $j > k+1$, the background-switch by thread T_k does *not* change $A_j^{\mathbf{F}}$. Thus, applying our induction hypothesis, immediately after the background-switch by thread T_k at time-step t we continue to have

$$\left|A_j^{\mathbf{F}} \cap s\right| < (1+\epsilon)^{j+1} \text{ for all sets } s \in \mathcal{S} \text{ and levels } j > k+1. \tag{31}$$

Finally, we focus on level $j = k+1$. The background-switch by thread T_k inserts some sets into level $k+1$, thereby bringing up the levels of some elements from $\le k$ to $k+1$. For every element e that belongs to one of these sets, however, whether $\text{plev}^{\mathbf{F}}(e) > k+1$ is not changed by the switch according to the claim. This ensures that $A_{k+1}^{\mathbf{F}}$ also does *not* change due to the background-switch by thread T_k at time-step t. Thus, by our induction hypothesis, immediately after the switch we continue to have

$$\left|A_{k+1}^{\mathbf{F}} \cap s\right| < (1+\epsilon)^{k+2} \text{ for all sets } s \in \mathcal{S}. \tag{32}$$

The lemma now follows from Equations (30) to (32). □

Lemma 6.26. *The foreground solution is an $O(\log n)$-approximate minimum set cover.*

Proof. Combine Lemmas 6.23 and 6.25 and apply Lemma 2.1. □

Lemma 6.27. *For any running background thread T_k, the size of extended background solution solution $|\widetilde{\mathbf{B}}_k| \le O(\log n) \cdot OPT(L)$.*

Proof. From Lemma 6.22, $\widetilde{\mathbf{B}}_k$ is ε-tidy at levels $\le k$, but might be ε-dirty at levels $> k$. Consider modifying the greedy algorithm of T_k by initializing p_k to $\ell_{\max}+1$ instead of $k+1$. This may change the definition of levels and passive levels in $\mathbf{B}_k$. Nevertheless, notice that the modification does not affect the choice of sets in the greedy algorithm. So, we can obtain the same sets as $\widetilde{\mathbf{B}}_k$. Now, we can apply the same argument as in Lemmas 6.22 and 6.24 to show that $\widetilde{\mathbf{B}}_k$ is ε-tidy and ε-stable under modified levels and passive levels. By Lemma 2.1, this implies $|\widetilde{\mathbf{B}}_k| \le O(\log n) \cdot \text{OPT}(L)$. □

Lemma 6.28. *The output solution is an $O(\log n)$-approximate minimum set cover.*

Proof. The output solution is union of foreground solution and background solutions of threads in copy or tail phase. By combining Lemmas 6.26 and 6.27, we have that $|\mathbf{F}| \le O(\log n) \cdot \text{OPT}(L)$, and $|\mathbf{B}_k| \le |\widetilde{\mathbf{B}}_k| \le O(\log n) \cdot \text{OPT}(L)$ for each thread in copy or tail phase. However, we are not done yet because there might be $O(\log n)$ such background threads.

Suppose there are r background threads $T_{k_1}, T_{k_2}, \ldots T_{k_r}$ in the copy or tail phase, ordered by when they entered the copy phase. By the criteria to enter the copy phase, we have $\tau_{k_{i+1}}^{\text{sus}} \le \frac{1}{2}\tau_{k_i}^{\text{sus}}$ for every $i < r$. That is, $\{\tau_{k_i}^{\text{sus}}\}_i$ forms a geometric series with ratio $\le \frac{1}{2}$. We claim that for any thread T_k in the copy or tail phase, $\tau_k^{\text{sus}} \le |\mathbf{B}_k| \le 2.2 \cdot \tau_k^{\text{sus}}$. It follows that $\sum_{i=1}^{r} |\mathbf{B}_{k_i}| \le O(1) \cdot |\mathbf{B}_{k_1}| \le O(\log n) \cdot \text{OPT}(L)$.

Suppose T_k enters suspension phase at time-step t^{sus}, and let t be the time-step indicated by the claim. The lower bound of the claim is simply because $\tau_k^{\text{sus}} = |\mathbf{B}_k(t^{\text{sus}})|$, and T_k cannot remove sets from $\mathbf{B}_k$ after t^{sus}. Next, we prove the upper bound of the claim. By the criteria to enter suspension phase, $|L_k^{\mathbf{F}}(t^{\text{sus}}) \setminus \text{cov}(\mathbf{B}_k(t^{\text{sus}}))| \le \tau_k^{\text{sus}}$. By Corollary 6.11 and lemma 6.12, $t - t^{\text{sus}} \le 0.2 \cdot \tau_k^{\text{sus}}$. After t^{sus}, T_k can only add sets to $\mathbf{B}_k$ through the greedy algorithm during the tail phase. Each set added by the greedy algorithm must cover an element in the uncovered sub-universe. Such an element is either in $L_k^{\mathbf{F}}(t^{\text{sus}}) \setminus \text{cov}(\mathbf{B}_k(t^{\text{sus}}))$, or inserted during $[t^{\text{sus}}+1, t]$. So, the number of sets added to $\mathbf{B}_k$ from t^{sus} to t is at most $1.2 \cdot \tau^{\text{sus}}$. The claim follows. □

6.5 Implementation Details and Update Time

First, we describe the data structures used to maintain the foreground, background, and buffer solutions.

Data Structures. The algorithm maintains three types of data structures:

1. For each solution S maintained by the algorithm (i.e. foreground, background and buffer solutions), its collection of sets are organized by levels, and the sets in each level are stored in a balanced binary search tree. We call these set BSTs and use $\mathbf{T}_{\text{set}}(S, \ell)$ to denote the set BST for solution S at level ℓ. The algorithm maintains a global pointer to the set BST for each level in every solution.

2. For each hierarchical solution S maintained by the algorithm (i.e. foreground and background solutions), its coverage $\text{cov}(S)$ is organized by levels, and the elements in each level of $\text{cov}(S)$ are stored in two balanced binary search trees that store the alive and dormant elements respectively. We call these element BSTs and denote it $\mathbf{T}_{\text{elem}}(S, \ell)$ for solution S at level ℓ. A node in an element BST stores the identity of an element e, its level $\text{lev}^S(e)$, passive level $\text{plev}^S(e)$, and the set whose coverage it belongs to in S, i.e. $s \in S$ such that $e \in \text{cov}^S(s)$. Similar to set BSTs, the algorithm maintains a global pointer to the element BST for each level in every hierarchical solution.

3. As mentioned in Section 5, we also have a priority queue Q_k for each background thread T_k. The purpose of this priority queue is to facilitate efficient execution of a step of the greedy algorithm in the computation and tail phases of T_k. In particular, every node in the priority queue is a set s and its key $Q_k(s)$ is the number of uncovered elements in s, i.e.

$$Q_k(s) = |s \cap (L_k^{\mathbf{F}} \setminus \text{cov}(\mathbf{B}_k))|.$$

   We remark that Q_k only stores sets s with key value $Q_k(s) > 0$. This ensures that the size of the priority queue Q_k is at most the maximum frequency f times the size of uncovered sub-universe $|L_k^{\mathbf{F}} \setminus \text{cov}(\mathbf{B}_k)|$.

   We insert or delete elements from the uncovered sub-universe in T_k implicitly by updating the keys in Q_k. To insert an element e, we add 1 to $Q_k(s)$ for every set s containing e. (If such a set s was not in Q_k prior to this step, we add it to Q_k with key value 1.) Similarly, to delete an element e, we subtract 1 from $Q_k(s)$ for every set s containing e. (If this drops $Q_k(s)$ to 0, we remove s from Q_k.)

4. Finally, for each background thread T_k, we have an array indexed by all sets where for each set s, it stores a BST containing all elements in s that are in the uncovered universe of T_k, i.e. $s \cap (L_k^{\mathbf{F}} \setminus \text{cov}(\mathbf{B}_k))$. We call this the uncovered BST and denote it $\mathbf{T}_{\text{unc}}(k, s)$.[8]

Update Time Bound. We now give details of how the data structures are updated by the algorithm, and derive resulting bounds on the update time of the algorithm.

Lemma 6.29. *In any time-step, the foreground thread or a background thread T_k calls $O(f \log n)$ priority queue and BST operations, and runs for $O(f \log n)$ time outside these calls. Therefore, the worst-case update time of the algorithm is $O(f \log^3 n)$.*

Proof. In bounding update time, we distinguish between two types of operations done at each time-step. The first type of operation is to handle element insertion or deletion in the instance at the current time-step. This is done for both foreground and background threads. The second type of operation applies only to the background threads, where they execute a set of operations based on the phase they are in. We bound the number of data structural steps for these two types of operations separately.

First, we consider an element insertion or deletion:

[8]For space efficiency, if we want to avoid storing an explicit array of size $|\mathcal{S}|$, then we can also store the global pointers in a BST or a hash table that only indexes the sets that actually appear in any priority queue. We ignore this in the rest of the description.

- Insertion of element e: Note that the element belongs to at most f sets.

  We first consider the foreground thread. For each set s that contains e, we query s in each set BST $\mathbf{T}_{\text{set}}(\mathbf{F}, \ell)$. These queries reveal the levels of all sets containing e that are already in $\mathbf{F}$. If there is at least one such set, let s^* be the set at the highest level ℓ^* in $\mathbf{F}$ among all such sets. We add a new node to the element BST $\mathbf{T}_{\text{elem}}(\mathbf{F}, \ell^*)$ containing e, $\text{lev}^{\mathbf{F}}(e) = \ell^*$, $\text{plev}^{\mathbf{F}}(e) = \ell^*$, and s^*. This means we are implicitly adding e to $\text{cov}(s^*)$. Next, suppose e is not in any set $s \in \mathbf{F}$. In this case, we first add an arbitrary set containing e to the set BST $\mathbf{T}_{\text{set}}(\mathbf{F}, 0)$ at level 0. Then, we repeat the above steps for adding e to the element BST $\mathbf{T}_{\text{elem}}(\mathbf{F}, 0)$.

  Now, we consider a background thread T_k at level $k \leq \text{lev}^{\mathbf{F}}(e)$. The first step is identical to the foreground thread, i.e. for each set s that contains e, we query s in each set BST $\mathbf{T}_{\text{set}}(\mathbf{B}_k, \ell)$. If there is at least one set containing e in $\mathbf{B}_k$, then we use the same set of updates as in the foreground solution. If none of the sets in $\mathbf{B}_k$ contains e, then we insert e in Q_k as described in the previous subsection by updating $Q_k(s)$ for $s \ni e$. Correspondingly, we also add e to the uncovered BST $\mathbf{T}_{\text{unc}}(k, s)$ for every $s \ni e$.

  The running time for the operations in any single thread for element insertion is dominated by $O(f \log n)$ BST operations for set BSTs, $O(1)$ BST operations for element BSTs, f BST operations for uncovered BSTs, and f priority queue updates for element addition to Q_k.

- Deletion of element e: For the foreground thread, we query e in the element BSTs $\mathbf{T}_{\text{elem}}(\mathbf{F}, \ell)$ to retrieve $\text{lev}^{\mathbf{F}}(e)$. Then, we update $\text{plev}^{\mathbf{F}}(e)$ in $\mathbf{T}_{\text{elem}}(\mathbf{F}, \text{lev}^{\mathbf{F}}(e))$. We move the element from the alive element BST to the dormant element BST.

  For each background thread T_k, we similarly query the element BSTs $\mathbf{T}_{\text{elem}}(\mathbf{B}_k, \ell)$ to decide whether $e \in \text{cov}(\mathbf{B}_k)$. If yes, the update is identical to the foreground thread. If no, we delete e from Q_k as described in the previous subsection by updating $Q_k(s)$ for $s \ni e$. Correspondingly, we also remove e from the uncovered BST $\mathbf{T}_{\text{unc}}(k, s)$ for every $s \ni e$.

  A special case is when the deleted element e is in $\text{cov}(s)$ for the set s that is currently being added to $\mathbf{B}_k$ by T_k in the computation or tail phases. Since the process of updating data structures for adding s to $\mathbf{B}_k$ does not happen in the single time-step, it is ambiguous as to whether $e \in \text{cov}(\mathbf{B}_k)$ while s is being added. To avoid ambiguity, we do the following. If e is not in the element BSTs $\mathbf{T}_{\text{elem}}(\mathbf{B}_k, \ell)$, then we query e in the uncovered BST $\mathbf{T}_{\text{unc}}(k, s)$. If yes, then we first add e to $\text{cov}(\mathbf{B}_k)$ using the steps described below in the computation and tail phases, and then proceed with the deletion of e (which will now change the passive level of e instead of removing it altogether from the solution $\mathbf{B}_k$).

  The running time for the operations in any single thread for element deletion is dominated by $O(\log n)$ BST operations for element BSTs, f BST operations for uncovered BSTs, and f priority queue updates for element deletion from Q_k.

We now analyze the operations performed by a background thread T_k based on the phase it is in:

- Preparation Phase: Elements in $L_k^{\mathbf{F}}$ form the initial uncovered sub-universe for background thread T_k in the preparation phase. In order to retrieve these elements, the algorithm uses the alive element BSTs $\mathbf{T}_{\text{elem}}(\mathbf{F}, \ell)$ for $\ell \leq k$. For each element e, it is added to the priority queue Q_k as described earlier. Furthermore, the element is added to the uncovered BSTs $\mathbf{T}_{\text{unc}}(k, s)$ for every $s \ni e$. This takes f priority queue and BST operations. Since $O(\log n)$ elements are processed in each time-step, this takes $O(f \log n)$ priority queue and BST operations per time-step.

  Note that the set of elements in $L_k^{\mathbf{F}}$ can change due to insertions, deletions, and because of background solutions at levels $< k$ being switched to the foreground. When an element e is inserted or deleted, the

update algorithm for insertion or deletion given above is applied both to the foreground thread and the background thread T_k. But, if a background thread at level $\ell < k$ is switched to the foreground, it can change the levels of the elements in $L_\ell^{\mathbf{F}}$ (although it does not change the set of elements in $L_k^{\mathbf{F}}$ overall by Claim 6.1). This can create inconsistencies, e.g., if the preparation phase were handling elements by level in the foreground solution.

To avoid such inconsistencies, our goal is to handle elements in order of a fixed index. To do so, the background thread T_k in the preparation phase uses the following strategy to select the next element to process. Recall that the process runs in a consecutive time block, so that the foreground data structures are stable. It runs queries on all the alive element BSTs $\mathbf{T}_{\text{elem}}(\mathbf{F}, \ell)$ at levels $\ell \leq k$ to identify the successor of the last element processed in each BST. Then, it selects the element e with the smallest index among these successors and processes it next. This ensures that elements are processed in order of their index irrespective of their levels in the foreground solution. In one time-step, the process finds c_{spd} elements in $L_k^{\mathbf{F}}$, and each element takes $O(\log n)$ BST operations to query all element BSTs. The process takes $O(\log n)$ BST operations.

- Computation and Tail Phases: In the greedy algorithm, we first retrieve the set s with the largest key in Q_k. This is single priority queue operation. First, we insert set s in the set BST $\mathbf{T}_{\text{set}}(\mathbf{B}_k, \text{lev}(s))$ where $\text{lev}(s)$ is as defined in Line 5.5 in Algorithm 5. Next, we retrieve $\text{cov}(s)$ using $\mathbf{T}_{\text{unc}}(k, s)$. For each element $e \in \text{cov}(s)$, we insert it in the element BST $\mathbf{T}_{\text{elem}}(\mathbf{B}_k, \text{lev}(s))$. At the same time, we remove e from Q_k (by updating the key as described earlier) and also remove it from the uncovered BSTs $\mathbf{T}_{\text{unc}}(k, s)$ for every $s \ni e$.

  To add set s to $\mathbf{B}_k$, we process each element $e \in \text{cov}(s)$ as follows. We remove e from the uncovered sub-universe by deleting it from Q_k, and removing it from $\mathbf{T}_{\text{unc}}(k, s')$ for every $s \ni e$. Then, we insert e to $\mathbf{T}_{\text{elem}}(\mathbf{B}_k, \text{lev}(s))$ with relevant information. ($\text{lev}^{\mathbf{B}_k}(e)$ is set to $\text{lev}(s)$, while $\text{plev}^{\mathbf{B}_k}(e)$ is determined before at preparation phase or when e is inserted.) Since the thread processes c_{spd} elements per time-step, this is $O(f)$ priority queue operations and $O(1)$ BST operations.

- Termination: At the normal termination of a background thread T_k, the switch between $\mathbf{B}_k$ and $\mathbf{F}$ is implemented as follows. We use pointer switch to replace the foreground data structures $\mathbf{T}_{\text{set}}(\mathbf{F}, \ell)$ and $\mathbf{T}_{\text{elem}}(\mathbf{F}, \ell)$ at levels $\ell \leq k$ by the corresponding background data structures $\mathbf{T}_{\text{set}}(\mathbf{B}_k, \ell)$ and $\mathbf{T}_{\text{elem}}(\mathbf{B}_k, \ell)$. Besides this, we merge the data structures for level $k+1$, i.e., we merge $\mathbf{T}_{\text{set}}(\mathbf{F}, k+1)$ with $\mathbf{T}_{\text{set}}(\mathbf{B}_k, k+1)$, and merge $\mathbf{T}_{\text{elem}}(\mathbf{F}, k+1)$ with $\mathbf{T}_{\text{elem}}(\mathbf{B}_k, k+1)$. This is $O(\log n)$ BST operations. □

The above implementation details and running time analysis work for the goal of bounding worst case insertion recourse. To bound the deletion recourse as well, we need to deamortize the deletion recourse according to Lemma 3.6. The garbage collection step can be efficiently implemented as follows. When we switch out some set BST from the foreground solution, we move that to a garbage set. The algorithm performs an additional garbage collection step that removes $O(\log n)$ sets in the garbage set from the output.

Lemma 6.30. *The deamortized algorithm has worst-case update time $O(f \log^3 n)$.*

Proof. Given Lemma 6.29, it remains to prove that the garbage collection step described in Lemma 3.6 can be efficiently implemented. We construct a garbage BST to manage the garbage sets in the output solution. The algorithm can only remove sets from the output solution by switching at the normal termination of a background thread. This step is implemented by pointer switch of BSTs, so we can merge the BSTs of sets that are switched out to the garbage BST in $O(\log n)$ BST operations. At the end of each time-step, we perform the garbage collection step by removing $O(\log n)$ sets from the garbage BST (and the output solution) as required by Lemma 3.6, which takes $O(\log n)$ BST operations. □

One issue is that the size of coverage of some solution might exceed n, since we defined n to be a universal upper bound for the number of live elements. Nevertheless, for any hierarchical solution S maintained by the algorithm, $|\mathrm{cov}(S)|$ can be upper bounded by $O(n)$ by the tidy property of $\mathbf{F}$ and $\widetilde{\mathbf{B}}_k$. So, all data structures maintained by the algorithm have size at most $O(n)$. It follows that each BST or priority queue operation can be implemented in $O(n)$ time. This also validates our claim that $\ell_{\max} = O(\log n)$ when defining $\ell_{\max}$.

Part III

$O(f)$-Competitive Algorithm

Remark. The presentation in Part III is self-contained. In particular, we use slightly different notations and terminologies in Part III than the ones introduced in Section 2 and Part II.

7 Preliminaries

In the Dynamic Set Cover problem, we are given a (fixed) universe U of elements and a collection of sets $\mathcal{S}$ that cover U. At any time-step $t \geq 0$, there is a set of *live* elements $L^{(t)}$ that have to be covered by the algorithm. Elements that are not live are called *dormant* and denoted $D^{(t)} := U \setminus L^{(t)}$. At any time-step, at most one element might change the state of live and dormant. We view such a change as an insertion or deletion to the set $L^{(t)}$. We denote n to be an upper bound of $|L^{(t)}|$, the number of elements to be covered. We denote f to be an upper bound of the frequency of elements, that is the number of sets containing it.

A solution is a collection of sets in a given space $\mathcal{S}$. At time-step t, a solution $S^{(t)} \subseteq \mathcal{S}$ is *feasible* if it covers $L^{(t)}$. We consider the unweighted setting, that is the cost of every set is 1.

This part is devoted to the following theorem.

Theorem 7.1. *There is a deterministic algorithm for the dynamic set cover problem that maintains an $O(f)$-approximate solution with worst-case recourse of $O(\log n)$ and worst-case update time of $O(f \log^3 n)$.*

Definition 4 (Primal-dual solutions)**.** Recall the LP relaxation of unweighted set cover on universe L:

$$(P)\ \min \sum_{s \in \mathcal{S}} x_s \qquad \text{s.t. } \forall e \in L, \sum_{s \in \mathcal{S}: e \in s} x_s \geq 1 \qquad x \geq 0$$

$$(D)\ \max \sum_{e \in L} y_e \qquad \text{s.t. } \forall s \in \mathcal{S}, \sum_{e \in s} y_e \leq 1 \qquad y \geq 0$$

A dual solution defines a dual value $w_e \geq 0$ for all live elements $e \in L^{(t)}$. Although we consider the dual LP on universe $L^{(t)}$, it will be convenient to define dual values for some dormant elements in $D^{(t)}$ as well. Such a dual solution is called an *extended dual solution*. (Let $w_e = 0$ by default for other dormant elements.)

We use $w_s := \sum_{e \in s} w_e$ to denote the total dual value in a set s. As part of the definition of extended dual solution, we require the following invariant:

- (Dual feasibility invariant) $\forall s \in \mathcal{S}, w_s \leq 1$.

We remark that this is stronger than the constraints in dual LP because the sum includes extended dual value of dormant elements.

A set $s \in \mathcal{S}$ is said to be tight if $w_s \geq (1+\varepsilon)^{-1}$. Given an extended dual solution, we define its corresponding primal solution as the collection of all tight sets.

In our algorithm, a primal-dual solution $\mathbf{F}$ consists of two parts:

1. An extended dual solution $\{w_e(\mathbf{F})\}_{e \in U(\mathbf{F})}$. Here, $U(\mathbf{F}) \subseteq U$ is the sub-universe of relevant element of $\mathbf{F}$. The dual values are explicitly maintained for $U(\mathbf{F})$, and are regarded as 0 for other elements.

2. A subset of its corresponding primal solution denoted by $\mathcal{S}(\mathbf{F})$. We remark that the primal solution is not necessarily feasible for the primal LP, but we require the following invariant suggested by complementary slackness.

- (Tight set invariant) $\forall s \in \mathcal{S}(\mathbf{F}), w_s(\mathbf{F}) \geq (1+\varepsilon)^{-1}$.

Lemma 7.2. *If a feasible extended dual solution* $\mathbf{F}$ *satisfies* $w_U(\mathbf{F}) \leq (1+\varepsilon)w_L(\mathbf{F})$*, then the corresponding primal solution is* $((1+\varepsilon)^2 f)$*-competitive.*

Proof sketch.

$$|\mathcal{S}(\mathbf{F})| \leq \sum_{s\in\mathcal{S}(\mathbf{F})} (1+\varepsilon)w_s \leq (1+\varepsilon)\sum_{e\in U}\sum_{s\in\mathcal{S}(\mathbf{F}):e\in s} w_e \leq (1+\varepsilon)f w_U \leq (1+\varepsilon)^2 f \cdot w_L \leq (1+\varepsilon)^2 f \cdot OPT$$

□

Definition 5 (Hierarchical solutions)**.** Based on a primal-dual solution $\mathbf{F}$, we assign a level $\text{lev}_s(\mathbf{F}) \in [0, \ell_{\max}]$ to every set $s \in \mathcal{S}(\mathbf{F})$, and $\text{lev}_e(\mathbf{F}) \in [0, \ell_{\max}]$ for some elements in $U(\mathbf{F})$.[9] Denote $C(\mathbf{F}) \subseteq U(\mathbf{F})$ to be the set of elements whose level is defined in $\mathbf{F}$ (can be live or dormant). Denote $E(\mathbf{F}) := U(\mathbf{F}) \setminus C(\mathbf{F})$ to be the elements in the sub-universe whose levels are not fixed. We call $C(\mathbf{F})$ covered elements and $E(\mathbf{F})$ exposed elements.

As part of the definition of hierarchical solutions, we require the following invariants. First, the levels of elements are related to the dual values.

- (Level invariant) $\forall e \in C(\mathbf{F}), w_e(\mathbf{F}) \leq (1+\varepsilon)^{-\text{lev}_e(\mathbf{F})}$.

Second, a covered element $e \in C(\mathbf{F})$ must be covered by a tight set $s \in \mathcal{S}(\mathbf{F})$. (The converse is not true. It is possible that $e \in s, s \in \mathcal{S}(\mathbf{F})$, but e is exposed.) Moreover, it must be covered by the highest one.

- (Highest level invariant) $\forall e \in C(\mathbf{F}), \text{lev}_e(\mathbf{F}) := \max\{\text{lev}_s(\mathbf{F}) : s \in \mathcal{S}(\mathbf{F}), e \in s\}$.

An element $e \in C(\mathbf{F})$ is called *active* if $w_e(\mathbf{F}) = (1+\varepsilon)^{-\text{lev}_e(\mathbf{F})}$, and *passive* if $w_e(\mathbf{F}) < (1+\varepsilon)^{-\text{lev}_e(\mathbf{F})}$.

We use a subscript k on many notations to denote the corresponding sets defined on levels $[0,k]$. Formally, we denote:

- $\mathcal{S}_k(\mathbf{F}) := \{s \in \mathcal{S}(\mathbf{F}) : \text{lev}_s(\mathbf{F}) \leq k\}$, sets in the primal solution at levels $[0,k]$.
- $C_k(\mathbf{F}) := \{e \in C(\mathbf{F}) : \text{lev}_e(\mathbf{F}) \leq k\}$, the set of covered elements at levels $[0,k]$;
- $L_k(\mathbf{F}) := \{e \in L \cap C_k(\mathbf{F}) : \text{lev}_e(\mathbf{F}) \leq k\}$, the set of live elements at levels $[0,k]$;
- $D_k(\mathbf{F}) := \{e \in D \cap C_k(\mathbf{F}) : \text{lev}_e(\mathbf{F}) \leq k\}$, the set of dormant elements at levels $[0,k]$;
- $A_k(\mathbf{F}) := \{e \in L_k(\mathbf{F}) : w_e(\mathbf{F}) = (1+\varepsilon)^{-\text{lev}_e(\mathbf{F})}\}$, the set of active live elements at levels $[0,k]$;
- $P_k(\mathbf{F}) := \{e \in L_k(\mathbf{F}) : w_e(\mathbf{F}) < (1+\varepsilon)^{-\text{lev}_e(\mathbf{F})}\}$, the set of passive live elements at levels $[0,k]$.

Note that $C_k(\mathbf{F}) = L_k(\mathbf{F}) \uplus D_k(\mathbf{F}), L_k(\mathbf{F}) = A_k(\mathbf{F}) \uplus P_k(\mathbf{F})$.

Definition 6. A hierarchical solution $\mathbf{F}$ is said to be ε-tidy at level k if $|D_k(\mathbf{F})| + |P_k(\mathbf{F})| \leq \varepsilon \cdot (|A_k(\mathbf{F})| + |E(\mathbf{F})|)$, and ε-dirty at level k otherwise. If $\mathbf{F}$ is ε-tidy at every level $k \in [0, \ell_{\max}]$, we say it is ε-tidy.

[9] The notation $[0, \ell_{\max}]$ refers to the set of integers in the interval.

8 Fully Dynamic $O(f)$-Competitive Set Cover Algorithm

The algorithm maintains a set of primal-dual solutions, including a *foreground solution* $\mathbf{F}^{(t)}$, and a set of $\ell_{\max}+1$ *background solutions* $\mathbf{B}_k^{(t)}$ for levels $k \in [0, \ell_{\max}]$. Intuitively, we expect $\mathbf{F}^{(t)}$ to be a feasible primal-dual solution where majority of dual values are assigned to live elements, so that Lemma 7.2 guarantees good approximation. We can handle insertion and deletion by local fix (see Section 8.1). But, doing so may accumulate dual values in dormant elements, as the local fix algorithm need to keep dormant elements to satisfy tight set invariant. To clean up the dormant elements, we repeatedly run a rebuild procedure in parallel for each level k. The procedure rebuilds the levels $[0,k]$ of $\mathbf{F}^{(t)}$ to form the background solution $\mathbf{B}_k$. Since $\mathbf{B}_k$ is identical to $\mathbf{F}$ at levels $> k$, the algorithm only needs to maintain a partial solution of $\mathbf{B}_k$ at levels $[0,k]$. We denote $\mathcal{S}^\star(\mathbf{B}_k) \subseteq \mathcal{S}(\mathbf{B}_k)$ to be the sets in the partial solution. Ideally, the procedure can remove all dormant elements in $D_k(\mathbf{F})$ and raise all remaining passive elements in $P_k(\mathbf{F})$ to level $k+1$. After the rebuild finishes, the algorithm performs a switch step which replaces $\mathbf{F}$ with the new solution $\mathbf{B}_k$. Then, it aborts the rebuild procedures on lower levels, since they are based on an outdated version of $\mathbf{F}$.

One issue here is that the switch step can replace a large subset of $\mathcal{S}(\mathbf{F})$, yet must be executed in one time-step. To avoid large recourse, we have to gradually add $\mathcal{S}^\star(\mathbf{B}_k)$ to the output solution before the switch step. It is tempting to include all background solutions in the output, but doing so will harm the approximation factor since there can be $O(\log n)$ background solutions. To resolve this, we only include a subset of background solutions in the output solution, and only allow these included background solutions to switch.

Algorithmically, we use a set of *buffer solutions* $\mathcal{R}_k$ for $k \in [0, \ell_{\max}]$ to store copies of $\mathcal{S}^\star(\mathbf{B}_k)$ before the potential switch steps. The output solution consists of the foreground solution and all buffer solutions. We remark that the buffer solutions are not primal-dual solutions; they only store primal solutions in order to control recourse. The switch steps still happen between the foreground and background solutions.

Initialization. The algorithm initializes the foreground, background, and buffer solutions to be all empty. (We assume $L^{(0)} = \emptyset$.)

Multi-threading. It will be convenient to view the algorithm as independent threads. The threads include:

1. a foreground thread that handles updates in the foreground solution, and
2. $\ell_{\max}+1$ background threads T_k, one for each level k, that gradually execute a primal-dual rebuild algorithm on the sub-universe $L_k(\mathbf{F}^{(t)})$. The background threads also need to handle updates in the set $L_k(\mathbf{F}^{(t)})$ because of changes in $\mathbf{F}^{(t)}$. The actual sub-universe involved in the background thread is $L_k(\mathbf{F}^{(t_0)})$ at the time t_0 when T_k is created, and all elements inserted to levels $\le k$ of $\mathbf{F}$ during the lifetime of T_k.

The foreground solution can only be modified by the foreground thread, or by the switch step at the termination of a background thread.

Sequential Behavior. Although we describe the algorithm as independent threads, the actual behavior of the algorithm within a time-step is sequential. A time-step is organized as follows:

1. The algorithm receives the update (insertion or deletion).
2. The foreground thread executes.
3. The background threads execute one by one, from higher level to lower level. If a background thread is not running, the algorithm creates that thread. If a background thread terminates, the algorithm

processes the switch triggered by the termination, and aborts all lower threads. The first thread is said to normally terminate.

8.1 Foreground Thread

On deletion of element e, it becomes dormant, but we keep its dual value in the foreground solution. The solution does not change.

On insertion of element e, let s be the highest set in $\mathcal{S}(\mathbf{F})$ covering e, i.e. $s = \arg\max_{s\in\mathcal{S}(\mathbf{F}),e\in s} \mathrm{lev}_s(\mathbf{F})$. If such a set s exists, we assign e to s by setting $\mathrm{lev}_e(\mathbf{F}) = \mathrm{lev}_s(\mathbf{F})$ and $w_e(\mathbf{F}) = 0$. Otherwise, e is not covered by $\mathbf{F}$. In this case, we pick the set s with smallest dual slackness $1 - w_s$ among the sets that cover e. We set $w_e(\mathbf{F}) = 1 - w_s(\mathbf{F})$, so that s becomes tight and dual feasibility is preserved. We add s to $\mathcal{S}(\mathbf{F})$ at level 0.

8.2 Background Threads

The background thread intends to run a primal-dual rebuild described in Algorithm 11 at the pace of processing c_{spd} elements per time-step to rebuild the sub-universe. Before the rebuild, the thread performs a initialization procedure described in Algorithm 10. This is called the preparation phase of the thread. During the execution of Algorithms 10 and 11, the sub-universe might be updated due to the foreground thread. Such an update must be insertion or deletion of an element in the sub-universe. The updates are handled by Algorithms 12 and 13 after the preparation phase. (The behavior in preparation phase will be described later.) Within a time-step, a background thread first handles the update, then continues the rebuild process.

The algorithm runs primal-dual algorithm by lazily raising the dual value of the exposed elements in sub-universe. Denote p_k to be a level pointer that decreases from $k+1$ to 0. Conceptually, all exposed elements in $E(\mathbf{B}_k)$ have dual values raised to $(1+\varepsilon)^{-p_k}$. Whenever a set becomes tight during this process, we place the set and all exposed element in the set at level p_k. By doing so, the elements become covered and their dual values are fixed.

Data Structures. The background solution $\mathbf{B}_k$ should be viewed as a complete primal-dual solution that is identical to $\mathbf{F}$ for elements with $\mathrm{lev}_e(\mathbf{F}) \geq k+1$. However, the algorithms only maintain a partial solution that differs from $\mathbf{F}$. Denote $U^\star(\mathbf{B}_k) \subseteq U(\mathbf{B}_k)$ to be a sub-universe of elements that we explicitly maintain. [10] Denote $C^\star(\mathbf{B}_k)$ to be elements in $U^\star(\mathbf{B}_k)$ with level defined, i.e., $C^\star(\mathbf{B}_k) := U^\star(\mathbf{B}_k) \cap C(\mathbf{B}_k)$. For each element $e \in U^\star(\mathbf{B}_k)$, we maintain its dual value $w_e(\mathbf{B}_k)$; For each element $e \in C^\star(\mathbf{B}_k)$, we maintain its level $\mathrm{lev}_e(\mathbf{B}_k)$. The information of elements outside the sub-universe will only be accessed aggregately when the algorithm queries the total dual value of a set $s \in \mathcal{S}$. Such queries can be efficiently retrieved from data structure of $\mathbf{F}$.

We denote $\mathcal{S}^\star(\mathbf{B}_k)$ to be the sets in the partial solution. Sets in $\mathcal{S}^\star(\mathbf{B}_k)$ are tight sets of $\mathbf{B}_k$ at levels $\leq k+1$, and they can be combined with $\mathcal{S}(\mathbf{F}) \setminus \mathcal{S}_k(\mathbf{F})$ to form $\mathcal{S}(\mathbf{B}_k)$. These sets are explicitly maintained by the background thread T_k

The exposed elements $E(\mathbf{B}_k)$ are viewed to participate an ongoing primal-dual algorithm, so that their dual values are regarded as $(1+\varepsilon)^{-p_k}$ but their levels are not decided yet. The algorithm explicitly maintains $E(\mathbf{B}_k)$ and guarantee $E(\mathbf{B}_k) = U^\star(\mathbf{B}_k) \setminus C^\star(\mathbf{B}_k)$. Denote $\mathcal{S}^{\mathrm{in}}(E)$ to be the collection of sets incident to E. The algorithm explicitly maintains $\mathcal{S}^{\mathrm{in}}(E(\mathbf{B}_k))$ in the sense that for each insertion and deletion of an element e in $E(\mathbf{B}_k)$, the algorithm updates $\mathcal{S}^{\mathrm{in}}(E(\mathbf{B}_k))$ accordingly by checking all sets incident to e.

The total dual value $w_s(\mathbf{B}_k)$ of a set $s \in \mathcal{S}$ consists of three parts:

1. Elements at levels $\geq k+1$ of $\mathbf{F}$ contribute their dual values in $\mathbf{F}$.

[10] Precisely speaking, the sub-universe consists of elements in the initial sub-universe $L_k(\mathbf{F}^{(t_0)})$ at the time-step t_0 when T_k is created, and elements inserted to $L_k(\mathbf{F})$ after t_0. In addition, deleted elements may be removed from the sub-universe.

2. Elements in $C^\star(\mathbf{B}_k)$ contribute their dual values in $\mathbf{B}_k$.

3. Exposed elements contribute $|s \cap E(\mathbf{B}_k)| \cdot (1+\varepsilon)^{-p_k}$. This part is maintained lazily in the sense that the algorithm maintains $|s \cap E(\mathbf{B}_k)|$ in appropriate data structures for each set $s \in \mathcal{S}^{\text{in}}(E(\mathbf{B}_k))$.

We maintain $w_s(\mathbf{B}_k)$ for sets $s \in \mathcal{S}^{\text{in}}(U^\star(\mathbf{B}_k))$ explicitly, in the sense that an update on w_e or whether $e \in E(\mathbf{B}_k)$ triggers update on w_s for all incident sets s. For other sets $s \notin \mathcal{S}^{\text{in}}(U^\star(\mathbf{B}_k))$, we have $w_s(\mathbf{B}_k) = w_s(\mathbf{F})$, and we can retrieve its total dual value from the foreground data structure.

Algorithm 10: Initialization of primal-dual algorithm run by background thread T_k

10.1 $p_k \leftarrow k+1$
10.2 **foreach** $e \in A_k(\mathbf{F})$ **do**
10.3 $\quad w_e(\mathbf{B}_k) \leftarrow (1+\epsilon)^{-(k+1)}$ // e is added to $E(\mathbf{B}_k)$.
10.4 **foreach** $e \in P_k(\mathbf{F})$ **do**
10.5 $\quad$ Run Algorithm 13 on element e.

Algorithm 11: Primal-dual rebuild algorithm run by background thread T_k

11.1 **for** $p_k \leftarrow k+1$ *downto* 0 **do**
11.2 $\quad$ Raise $w_e(\mathbf{B}_k) \leftarrow (1+\varepsilon)^{-p_k}$ for each $e \in E(\mathbf{B}_k)$
11.3 $\quad$ **while** $\exists s \in \mathcal{S}^{in}(E(\mathbf{B}_k))$ *s.t.* $w_s(\mathbf{B}_k) \geq (1+\varepsilon)^{-1}$ **do**
11.4 $\quad\quad$ Add s to $\mathcal{S}^\star(\mathbf{B}_k)$ at level p_k
11.5 $\quad\quad$ **foreach** $e \in s \cap E(\mathbf{B}_k)$ **do**
11.6 $\quad\quad\quad$ $\text{lev}_e(\mathbf{B}_k) \leftarrow p_k, w_e(\mathbf{B}_k) \leftarrow (1+\varepsilon)^{-p_k}$ // e is moved from $E(\mathbf{B}_k)$ to $C^\star(\mathbf{B}_k)$.
$\quad\quad$ // s is removed from $\mathcal{S}^{\text{in}}(E(\mathbf{B}_k))$.
11.7 Switch $\mathbf{F}$ with $\mathbf{B}_k$.

Algorithm 12: Handling deletion of an element e in $L_k(\mathbf{F})$

12.1 **if** $e \in s$ *for some set* $s \in \mathcal{S}^\star(\mathbf{B}_k)$ **then**
12.2 $\quad$ **if** $e \in E(\mathbf{B}_k)$ **then**
12.3 $\quad\quad$ $\text{lev}_e(\mathbf{B}_k) \leftarrow p_k, w_e(\mathbf{B}_k) \leftarrow (1+\varepsilon)^{-p_k}$ // e is moved from $E(\mathbf{B}_k)$ to $C^\star(\mathbf{B}_k)$.
12.4 $\quad$ e becomes dormant in $\mathbf{B}_k$ // $w_e(\mathbf{B}_k)$ and $\text{lev}_e(\mathbf{B}_k)$ do not change.
12.5 **else**
12.6 $\quad$ Remove e from $E(\mathbf{B}_k)$ and $U^*(\mathbf{B}_k)$ // $w_e(\mathbf{B}_k)$ becomes 0.

Next, we describe the phases of a background thread T_k. Let c_{spd} be a large constant that will be decided later. Specially, if the sub-universe is small, we do not separate the phases.

- (Rule of base threads.) If $|L_k(\mathbf{F})| \leq c_{\text{spd}}$ when T_k is created, all work of T_k can be finished in one time-step. In this case, we execute Algorithms 10 and 11 in one time-step. T_k is not viewed to enter any phase.

  (According to the sequential behavior of threads and since $|L_k(\mathbf{F})|$ is monotone, the algorithm only processes the highest basic thread in a time-step.)

Algorithm 13: Handling insertion of an element e in $L_k(\mathbf{F})$

13.1 **if** $e \in s$ *for some set* $s \in \mathcal{S}^\star(\mathbf{B}_k)$ **then**
13.2 Let s be the highest set in $\mathbf{B}_k$ that contains e
13.3 $\mathrm{lev}_e(\mathbf{B}_k) \leftarrow \mathrm{lev}_s(\mathbf{B}_k), w_e(\mathbf{B}_k) \leftarrow 0$ // e is added to $C^\star(\mathbf{B}_k)$.
13.4 **else if** $1 - \max_{s \in \mathcal{S}: e \in s}\{w_s(\mathbf{B}_k)\} < (1+\varepsilon)^{-p_k}$ **then**
13.5 $s \leftarrow \arg\max_{s \in \mathcal{S}: e \in s}\{w_s(\mathbf{B}_k)\}$
13.6 Add s to $\mathcal{S}^\star(\mathbf{B}_k)$ at level p_k
13.7 $\mathrm{lev}_e(\mathbf{B}_k) \leftarrow p_k, w_e(\mathbf{B}_k) \leftarrow 1 - w_s(\mathbf{B}_k)$ // e is added to $C^\star(\mathbf{B}_k)$.
13.8 **else**
13.9 $w_e(\mathbf{B}_k) \leftarrow (1+\epsilon)^{-p_k}$ // e is added to $E(\mathbf{B}_k)$.

Preparation Phase. In the preparation phase, T_k executes Algorithm 10 at the speed of processing c_{spd} elements per time-step.

During the preparation phase, $L_k(\mathbf{F})$ might be modified by insertion and deletion (after the foreground thread decided that the update affects $L_k(\mathbf{F})$). A deletion in $L_k(\mathbf{F})$ is handled by Algorithm 12. An insertion in $L_k(\mathbf{F})$ during the second for loop for passive elements is handled by Algorithm 13. Specially, on insertion of element e during the first loop for active elements, we view e as a passive element in $P_k(\mathbf{F})$ and simply append the new element to the list of second for loop. This special behavior is because we have not initialized the dual values of active elements during the first loop, which is necessary for Algorithm 13.

Computation Phase. In the computation phase, T_k executes Algorithm 11. But, once $|E(\mathbf{B}_k)| \le |\mathcal{S}^\star(\mathbf{B}_k)|$ after finishing a while loop (adding a set), it pauses Algorithm 11 and enters the suspension phase.

The bottleneck of each while loop is maintaining the data structures for elements in $s \cap E(\mathbf{B}_k)$. The algorithm processes c_{spd} elements per time-step.

In computation phase and later phases, updates in $L_k(\mathbf{F})$ are handled by Algorithms 12 and 13.

Specially, if the solution size is small, we do not pause the algorithm.

- (Rule of shortcut threads.) If $|E(\mathbf{B}_k)| \le c_{\mathrm{spd}}$ and $|\mathcal{S}^\star(\mathbf{B}_k)| \le c_{\mathrm{spd}}$ during computation phase of T_k, all remaining work can be done in one time-step. In this case, we continue executing Algorithm 11 to the end, and copy all sets in $\mathcal{S}^\star(\mathbf{B}_k)$ to $\mathcal{R}_k$ in one time-step. T_k is viewed to terminate in computation phase, and not viewed to enter later phases.

Suspension Phase. When a thread T_k enters the suspension phase, we take a snapshot of its solution size and denote it τ_k^{sus}. (We have $\tau_k^{\mathrm{sus}} > c_{\mathrm{spd}}$ by the rule of shortcut threads.)

During the suspension phase, T_k checks whether $\tau_k^{\mathrm{sus}} \le \frac{1}{2} \cdot \tau$, where τ is the minimum of τ_j^{sus} among the threads T_j in copy and tail phases. If this condition holds, T_k enters copy phase; otherwise, T_k remains in suspension phase.

According to the sequential behavior of background threads, after a higher thread T_k enters the copy phase, lower threads in suspension phase will compare to the updated threshold $\tau = \tau_k^{\mathrm{sus}}$.

Copy Phase. During the copy phase, T_k copies sets in $\mathcal{S}^\star(\mathbf{B}_k)$ to $\mathcal{R}_k$ at the speed of c_{spd} sets per time-step. ($\mathcal{S}^\star(\mathbf{B}_k)$ may grow by one set per time-step due to Algorithm 13, but cannot shrink. So, the order to copy sets is arbitrary since we can catch up the updates.)

Tail Phase. During the tail phase, T_k resumes the primal-dual algorithm. In addition, we maintain $\mathcal{R}_k$ to be a faithful copy of $\mathcal{S}^\star(\mathbf{B}_k)$.

If $|E(\mathbf{B}_k)| \le c_{\text{spd}}$ during tail phase, all remaining work can be done in one time-step. In this case, we continue executing Algorithm 11 to the end in the current time-step.

Termination. When a background thread normally terminates, at the end of Algorithm 11, the algorithm switches the $\mathbf{F}$ with $\mathbf{B}_k$. In terms of primal solution, the algorithm removes sets $\mathcal{S}_k(\mathbf{F})$ and moves $\mathcal{R}_k$ to the foreground, which costs no insertion recourse. In terms of data structure for dual values, the algorithms discards the foreground data structures at levels $\le k$, moves the data structures at levels $\le k$ of $\mathbf{B}_k$ to foreground, and merges the data structure at level $k+1$ of $\mathbf{B}_k$ with the foreground data structure at level $k+1$. All these operations can be done in one time-step.

9 Analysis of the Fully Dynamic $O(f)$-Competitive Set Cover Algorithm

9.1 Invariants

Fact 9.1. *When a background thread T_k normally terminates, $E(\mathbf{B}_k) = \emptyset$.*

Proof. Assume for contradiction that $e \in E(\mathbf{B}_k)$ at the end of Algorithm 11. Then, there exists an set s containing e, and $s \in \mathcal{S}^{\text{in}}(E(\mathbf{B}_k))$ by definition of $\mathcal{S}^{\text{in}}(E(\mathbf{B}_k))$. Since $w_s(\mathbf{B}_k) \ge w_e(\mathbf{B}_k) = (1+\varepsilon)^{-p_k} = 1$ when p_k decreases to 0, s should be selected by the while loop to cover e, a contradiction. □

Lemma 9.2. *When a background thread T_k normally terminates, the switch step cannot change $L_j(\mathbf{F})$ at any level $j \ge k+1$.*

Proof. Assume for contradiction that the earliest violation happens at time-step t for levels $j > k$. Let t^-, t^+ respectively denote the time before and after the switch step at termination of T_j. Suppose T_j is created at time-step t^{prep}.

The switch step removes all elements in $L_k(\mathbf{F}^{(t^-)})$ from $\mathbf{F}$. We first show that any element $e \in L_k(\mathbf{F}^{(t^-)})$ is added back to $L_j(\mathbf{F}^{(t^+)})$. By the assumption that the lemma holds before t, $L_j(\mathbf{F})$ cannot be modified by the termination of lower-level threads during $[t^{\text{prep}}, t]$. So, e is either in the initial sub-universe $L_j(\mathbf{F}^{(t^{\text{prep}})})$ or inserted to the sub-universe after t^{prep}. According to Algorithms 10 and 13, at initialization or insertion, e is either assigned $\text{lev}_e(\mathbf{B}_k) \le k+1 \le j$ or added to $E(\mathbf{B}_k)$. In the latter case, e must be removed from $E(\mathbf{B}_k)$ before the termination by Fact 9.1. Since $e \in L^{(t)}$, e can only be removed from $E(\mathbf{B}_k)$ by Algorithm 11, and e must be assigned $\text{lev}_e(\mathbf{B}_k) \le k+1 \le j$ when removed from $E(\mathbf{B}_k)$. The algorithm cannot modify $\text{lev}_e(\mathbf{B}_k)$ after it is assigned. So, we have $\text{lev}_e(\mathbf{B}_k) \le j$ before the switch. Since $\text{lev}_e(\mathbf{F}) = \text{lev}_e(\mathbf{B}_k)$ after the switch, we have $e \in L_j(\mathbf{F}^{(t^+)})$.

The switch step inserts the partial solution of $\mathbf{B}_k$ to foreground. Since $E(\mathbf{B}_k) = \emptyset$ by Fact 9.1 and $U^\star(\mathbf{B}_k) = C^\star(\mathbf{B}_k) \cup E(\mathbf{B}_k)$, the switch step can only add elements in $C^\star(\mathbf{B}_k)$ to foreground. Among these elements, the dormant elements in $D^{(t)}$ are not involved in the lemma. We next show that the live elements in $C^\star(\mathbf{B}_k)$ must belong to $L_k(\mathbf{F}^{(t^-)})$ before the switch. An element e can be added to $C^\star(\mathbf{B}_k)$ in three ways: (1) in Algorithm 10 as a passive element, (2) in the inner loop of Algorithm 11, (3) in Algorithm 13 as a passive element. Let $t_e \in [t^{\text{prep}}, t]$ be the time-step when e is added to $C^\star(\mathbf{B}_k)$. In all three cases, e is either in $L_k(\mathbf{F})$ or inserted into $L_k(\mathbf{F})$ at t_e. By the assumption that the lemma holds before t, $L_k(\mathbf{F})$ can only be modified by the foreground thread during $[t^{\text{prep}}, t-1]$. This is also true at time-step t before the switch, since there cannot be another normal termination at time-step t by the sequential behavior of threads. So, $\text{lev}_e(\mathbf{F})$ cannot change from t_e to t^-, and $e \in L_k(\mathbf{F}^{(t^-)})$. □

The above lemma implies that during the lifetime of a background thread T_k, $L_k(\mathbf{F})$ is stable in the sense that $L_k(\mathbf{F})$ can only be modified by the foreground thread, and cannot be modified by termination of lower background threads. (It also cannot be modified by higher background threads since termination of higher threads will abort T_k.) Since all modification of $L_k(\mathbf{F})$ due to foreground thread are handled by Algorithms 12 and 13, it is valid to explicitly maintain $E(\mathbf{B}_k)$.

Lemma 9.3. *For any background thread T_k, we have $E(\mathbf{B}_k) \subseteq L_k(\mathbf{F})$, and any element $e \in E(\mathbf{B}_k)$ cannot belong to any set in $\mathcal{S}^\star(\mathbf{B}_k)$ at levels $> p_k$.*

Proof. For the first statement, we prove that (1) any element added to $E(\mathbf{B}_k)$ is from $L_k(\mathbf{F})$. This is easy to check in Algorithms 10 and 13. (2) Any deletion from $E(\mathbf{B}_k) \cap L_k(\mathbf{F})$ in $L_k(\mathbf{F})$ is also removed from $E(\mathbf{B}_k)$. By Lemma 9.2, $L_k(\mathbf{F})$ can only be modified due to insertion and deletion during the lifetime of T_k. Whenever an element $e \in E(\mathbf{B}_k)$ is removed from $L_k(\mathbf{F})$ due to a deletion, Algorithm 12 will remove e from $E(\mathbf{B}_k)$.

We prove the second statement by induction on time for each element e. For the base case, we prove that e does not belong to any set in $\mathcal{S}^\star(\mathbf{B}_k)$ when added to $E(\mathbf{B}_k)$. If e is added to $E(\mathbf{B}_k)$ in the first loop of Algorithm 10, $\mathcal{S}^\star(\mathbf{B}_k)$ is empty; If e is added to $E(\mathbf{B}_k)$ by Algorithm 13 or in the second loop of Algorithm 10, the algorithm guarantees that e does not belong to any set in $\mathcal{S}^\star(\mathbf{B}_k)$.

For the inductive step, since the levels in $\mathbf{B}_k$ cannot change once determined, and the algorithm is only allowed to add sets at level p_k, the only nontrivial case is when p_k decreases. Assume for contradiction that e is contained in a set $s \in \mathcal{S}^\star(\mathbf{B}_k)$ at level p_k when the outer loop of p_k finishes in Algorithm 11. s can be added to $\mathcal{S}^\star(\mathbf{B}_k)$ by Algorithm 11, or by Algorithms 10 and 13. In the former case, every element in $s \cap E(\mathbf{B}_k)$ should be removed from $E(\mathbf{B}_k)$, a contradiction. In the latter case, s must be a tight set when added to $\mathcal{S}^\star(\mathbf{B}_k)$ by Algorithms 10 and 13. Since the algorithms cannot decrease $w_s(\mathbf{B}_k)$, s is still tight at the end of outer loop of p_k. Also, $s \in \mathcal{S}^{\text{in}}(E(\mathbf{B}_k))$ since $e \in E(\mathbf{B}_k)$. This contradicts the termination condition of while loop in Algorithm 11. □

Next, we establish the invariants required in the definition of primal-dual solution and hierarchical solution.

Lemma 9.4 (Level invariant)**.** *In the foreground solution and all background solutions, any element $e \in C(\mathbf{B}_k)$ satisfies $w_e(\mathbf{B}_k) \le (1+\varepsilon)^{-\text{lev}_e(\mathbf{B}_k)}$.*

Proof. We prove the invariant by induction on time. Initially the statement is true.

The foreground solution can be modified in the following cases.

- Insertion in foreground thread: The algorithm either sets $w_e(\mathbf{F}) = 0$, or sets $\text{lev}_e(\mathbf{F}) = 0$. The invariant holds in both cases.
- Deletion in foreground thread: $\text{lev}_e(\mathbf{F})$ and $w_e(\mathbf{F})$ do not change.
- Switch step: $\text{lev}_e(\mathbf{F})$ and $w_e(\mathbf{F})$ can only be modified by copying from the background solution. The invariant holds by inductive hypothesis for background solutions before the switch.

For a background thread T_k, initially the invariant for elements outside $L_k(\mathbf{F})$ follows inductive hypothesis for the foreground solution. Whenever T_k determine $\text{lev}_e(\mathbf{B}_k)$ for an element e (in the inner loop of Algorithm 11, or in Algorithms 10 and 13 as a passive element), we have $\text{lev}_e(\mathbf{B}_k) = p_k$, $w_e(\mathbf{B}_k) \le (1+\varepsilon)^{-p_k}$ in all cases. The levels and dual values cannot change for elements in $C^\star(\mathbf{B}_k)$. □

Lemma 9.5 (Highest level invariant)**.** *For any element $e \in C(\mathbf{F})$ (resp. $C^\star(\mathbf{B}_k)$), $\text{lev}_e(\mathbf{F})$ (resp. $\text{lev}_e(\mathbf{B}_k)$) is equal to the maximum level of sets in $\mathcal{S}(\mathbf{F})$ (resp. $\mathcal{S}^\star(\mathbf{B}_k)$) that contains it.*

Proof. We prove the invariant by induction on time. Initially the statement is true.

The foreground solution can be modified in the following cases.

- Insertion in foreground thread: The algorithm directly guarantees the invariant.
- Deletion in foreground thread: The solution does not change.
- Switch step at termination of T_k: Let t^-, t^+ be the time before and after switch. The elements e with $\text{lev}_e(\mathbf{F}^{(t^-)}) \geq k+1$ and the highest set containing e are not modified. Next, consider elements $e \in C_k(\mathbf{F}^{(t^-)}) \cap C(\mathbf{F}^{(t^+)})$. By inductive hypothesis, e cannot belong to any set at levels $\geq k+1$ of $\mathbf{F}^{(t^-)}$. As argued in Lemma 9.2, e must be covered by a set in $\mathcal{S}^\star(\mathbf{B}_k^{(t^-)})$ at levels $\leq k+1$. By the inductive hypothesis for $\mathbf{B}_k^{(t^-)}$, $\text{lev}_e(\mathbf{B}_k^{(t^-)})$ is equal to the highest level of sets covering e in $\mathcal{S}^\star(\mathbf{B}_k^{(t^-)})$, which is also the highest level of sets covering e in $\mathcal{S}(\mathbf{F}^{(t^+)})$.

For a background thread T_k, initially $\mathcal{S}^\star(\mathbf{B}_k)$ is empty. A background solution $\mathbf{B}_k$ can be modified in the following cases.

- Adding a passive element in Algorithms 10 and 13: The algorithm directly guarantees the invariant.
- Deletion in Algorithm 12: The solution does not change.
- Adding a tight set in Algorithm 11: For each $e \in s \cap E(\mathbf{B}_k)$, we set $\text{lev}_e = \text{lev}_s = p_k$. By Lemma 9.3, e cannot belong to an existing higher set in $\mathcal{S}^\star(\mathbf{B}_k)$. Since the algorithms of T_k can only add sets at level p_k and the level pointer p_k is monotone decreasing, there cannot be a higher set added to $\mathcal{S}^\star(\mathbf{B}_k)$ in the future.

□

Lemma 9.6 (Dual feasibility invariant)**.** *For the dual value w defined by the foreground solution or any background solution, $w_s \leq 1$ for all sets $s \in \mathcal{S}$.*

Proof. We prove the invariant by induction on time. Initially the statement is true.

The foreground solution can be modified in the following cases.

- Insertion in foreground thread: The foreground algorithm guarantees the invariant.
- Deletion in foreground thread: The solution does not change.
- Switch step at termination of T_k: The invariant holds by inductive hypothesis for the background solution before the switch.

For a background solution $\mathbf{B}_k$, we first show that the initialization step in Algorithm 10 can only decrease the dual values of elements compared to $\mathbf{F}$, so that the invariant follows inductive hypothesis for the foreground solution before the initialization step. Elements in $A_k(\mathbf{F})$ have $w_e(\mathbf{F}) = (1+\varepsilon)^{-\text{lev}_e(\mathbf{F})}$ by definition of active elements, which is higher then their initial value $(1+\varepsilon)^{-(k+1)}$ in $\mathbf{B}_k$. After Algorithm 10 handles all active elements, the passive elements are initialized to satisfy the invariant.

During the background thread, $w_s(\mathbf{B}_k)$ may increase due to insertion in $L_k(\mathbf{F})$, or the change of level p_k. In the former case, Algorithm 13 guarantees the invariant. In the latter case, when Algorithm 11 decrease p_k by one to increase $w_e(\mathbf{B}_k)$ by a factor of $1+\varepsilon$ for all elements in $e \in E(\mathbf{B}_k)$, only the sets in $\mathcal{S}^{\text{in}}(E(\mathbf{B}_k))$ have their w_s increased, and they increase by at most by a factor of $1+\varepsilon$. Since the termination condition of the while loop rules out any $s \in \mathcal{S}^{\text{in}}(E(\mathbf{B}_k))$ with $w_s \geq (1+\varepsilon)^{-1}$, this increase will not violate the invariant. □

Lemma 9.7 (Tight set invariant)**.** *For the foreground solution* $\mathbf{F}$ *and all background solutions* $\mathbf{B}_k$*, all sets in* $\mathcal{S}(\mathbf{F})$ *or* $\mathcal{S}^\star(\mathbf{B}_k)$ *are tight (i.e.,* $w_s \geq (1+\varepsilon)^{-1}$*).*

Proof. We first prove the invariant for a background solution $\mathbf{B}_k$. Whenever a set s is added to $\mathcal{S}^\star(\mathbf{B}_k)$ by Algorithms 10, 11 and 13, s must be tight. s remains tight since the background algorithms cannot decrease $w_s(\mathbf{B}_k)$.

We next prove the invariant for the foreground solution by induction on time. Initially the statements is true. The foreground solution can be modified in the following cases.

- Insertion in foreground thread: The algorithm assigns the new element e to an existing set $s \in \mathcal{S}(\mathbf{F})$ or adds a new set s to $\mathcal{S}(\mathbf{F})$. The choice of w_e guarantees that s remains tight in the former case and s is tight in the latter case.
- Deletion in foreground thread: The solution does not change.
- Switch step at termination of T_k: Let t^-, t^+ be the time before and after switch. All elements and sets at levels $\geq k+1$ of $\mathbf{F}$ before the switch are not affected. By Lemma 9.5, removing elements at levels $\leq k$ does not affect the total values of sets at levels $\geq k+1$. So, the invariant continues to hold for sets at levels $\geq k+1$ before the switch. The other sets are replaced by $\mathcal{S}^\star(\mathbf{B}_k)$. For a new set $s \in \mathcal{S}^\star(\mathbf{B}_k)$, we have $w_s(\mathbf{F}^{(t^+)}) = w_s(\mathbf{B}_k^{(t^-)})$. So, the invariant for these sets follows the inductive hypothesis for the background solution before the switch.

□

9.2 Feasibility

Lemma 9.8. *The foreground solution (and hence the output solution) is a feasible set cover at the end of every time-step.*

Proof. We prove the lemma by induction on time. Initially the lemma is trivial. The foreground solution can be modified in the following cases.

- Insertion in foreground thread: The foreground algorithm either assigns e to an existing set $s \in \mathcal{S}(\mathbf{F})$ or adds a new set s that cover e to $\mathcal{S}(\mathbf{F})$.
- Deletion in foreground thread: The solution does not change.
- Switch step at termination of T_k: Denote t to be the time-step of switch step and t^-, t^+ to be the times before and after the switch step. By the inductive hypothesis that $\mathbf{F}$ is a feasible set cover before the switch, $L_{\ell_{\max}}(\mathbf{F}^{(t^-)}) = L^{(t)}$. By Lemma 9.2, $L_{\ell_{\max}}(\mathbf{F}^{(t^-)}) = L_{\ell_{\max}}(\mathbf{F}^{(t^+)})$. So, $L_{\ell_{\max}}(\mathbf{F}^{(t^+)}) = L^{(t)}$. By Lemma 9.5, all elements in $L_{\ell_{\max}}(\mathbf{F})$ are contained in sets in $\mathcal{S}(\mathbf{F})$. So, $\mathbf{F}$ is feasible after the switch.

□

9.3 Recourse

Lemma 9.9. *The insertion recourse is* $O(\log n)$*.*

Proof. The output solution consists of the foreground solution $\mathbf{F}$ and all buffer solutions $\mathcal{R}_k$. The algorithm can add sets to $\mathbf{F}$ in two ways:

1. Handling an insertion by the foreground thread. This causes at most one insertion recourse.
2. Switch steps at normal terminations of background threads. This causes no insertion recourse.

For each level k, the algorithm can add sets to $\mathcal{R}_k$ in two ways:

1. In the copy and tail phases of T_k, or at the termination of T_k when T_k is a base or shortcut thread, T_k copies sets from $\mathcal{S}^\star(\mathbf{B}_k)$ to $\mathcal{R}_k$ at the speed of c_{spd} sets per time-step. This causes at most $2c_{\text{spd}} = O(1)$ recourse, because T_k can go through each phase at most once in a time-step.
2. Handling an insertion in the sub-universe of $\mathbf{B}_k$ when T_k is in the copy and tail phases. This may add a new set to $\mathcal{S}^\star(\mathbf{B}_k)$, which is repeated in $\mathcal{R}_k$ in the copy and tail phases. This causes at most one insertion recourse.

In conclusion, the overall insertion recourse is $O(\log n)$ since there are $\ell_{\max} + 1 = O(\log n)$ background threads. □

To bound the deletion recourse by $O(\log n)$, we call Lemma 3.6, which designs a garbage collection process to deamortize the deletion recourse.

Lemma 3.6. *Consider an online unweighted minimization problem, where the size of the optimal solution changes by at most δ in each time-step. Suppose there exists an algorithm $\mathcal{A}$ that maintains an α-approximate solution and has worst-case insertion recourse at most β, for some parameters $\alpha, \beta \geq 1$. (The worst-case deletion recourse of algorithm $\mathcal{A}$ can be arbitrarily large.) Then, there exists an alternative algorithm $\mathcal{A}'$ that also maintains an α-approximate solution and has worst-case recourse (both insertion and deletion) at most $2\beta + \delta\alpha$.*

9.4 Lifetime Bounds

In this section, we bound the lasting time of a background thread in each phase. Recall that a base thread T_k with $|L_k(\mathbf{F}^{(t^{\text{prep}})})| \leq c_{\text{spd}}$ when created at t^{prep} will terminate in one time-step and will not enter any phase. So, for a background thread T_k in any phase, we have $|L_k(\mathbf{F}^{(t^{\text{prep}})})| > c_{\text{spd}}$.

A background thread T_k is allow to undergo many phases in a time-step. So, the last time-step t of the previous phase is also viewed as the first time-step of the next phase. For each phase T_k enters within a time-step, the algorithm performs the individually scheduled work amount (e.g., processing c_{spd} elements). When T_k normally terminates or gets aborted, it will restart at the next time-step.

We set $c_{\text{spd}} = 500$ to be a sufficiently large constant.

Fact 9.10. *Suppose that a background thread T_k is in the preparation phase at time-step t. Then, T_k will finish preparation phase (enter computation phase or get aborted) by time-step $t + \frac{1.1}{c_{\text{spd}}} \cdot |L_k(\mathbf{F}^{(t)})|$.*

Proof. Suppose T_k is created at time-step t^{prep}, and finishes preparation phase at t^{comp}. During the preparation phase, T_k runs Algorithm 10 and scans elements in the sub-universe $L_k(\mathbf{F}) = A_k(\mathbf{F}) \uplus P_k(\mathbf{F})$, at a speed of c_{spd} elements per time-step, except for the last time-step t^{comp}. On the other hand, $L_k(\mathbf{F}^{(t)})$ may be modified due to insertion/deletion of elements, but at the speed of at most one element per time-step. So, $t^{\text{comp}} - t^{\text{prep}} \leq \frac{1}{c_{\text{spd}}-1} \cdot |L_k(\mathbf{F}^{(t^{\text{prep}})})|$. Since $t \in [t^{\text{prep}}, t^{\text{comp}}]$, we have

$$|L_k(\mathbf{F}^{(t)})| \geq |L_k(\mathbf{F}^{(t^{\text{prep}})})| - (t - t^{\text{prep}}) \geq \frac{c_{\text{spd}} - 2}{c_{\text{spd}} - 1} \cdot |L_k(\mathbf{F}^{(t^{\text{prep}})})|.$$

Combine the above inequalities with $|L_k(\mathbf{F}^{(t^{\text{prep}})})| > c_{\text{spd}}$ from the rule of base threads, we conclude

$$t^{\text{comp}} - t \le t^{\text{comp}} - t^{\text{prep}} \le \frac{1}{c_{\text{spd}} - 1} \cdot |L_k(\mathbf{F}^{(t^{\text{prep}})})| \le \frac{1}{c_{\text{spd}} - 2} \cdot |L_k(\mathbf{F}^{(t)})| \le \frac{1.1}{c_{\text{spd}}} \cdot |L_k(\mathbf{F}^{(t)})|.$$

□

Fact 9.11. *Suppose that a background thread T_k is in the computation phase at time-step t. Then, T_k will finish computation phase by time-step $t + \left\lceil \frac{1.1}{c_{\text{spd}}} \cdot |E(\mathbf{B}_k^{(t)})| \right\rceil$.*

Proof. During the computation phase, T_k runs Algorithm 11 at a speed of removing c_{spd} elements from $E(\mathbf{B}_k)$ per time-step, except for the last time-step. The only way to add elements to $E(\mathbf{B}_k)$ during the computation phase is due to Algorithm 13 handling insertion, which can only add one element in a time-step. (By Lemmas 9.2 and 9.3, termination of lower threads cannot add elements to $E(\mathbf{B}_k)$.) The algorithm terminates when $E(\mathbf{B}_k) = \varnothing$. So, the computation phase finishes in $\left\lceil \frac{1}{c_{\text{spd}}-1} \cdot \left| E(\mathbf{B}_k^{(t)}) \right| \right\rceil \le \left\lceil \frac{1.1}{c_{\text{spd}}} \cdot \left| E(\mathbf{B}_k^{(t)}) \right| \right\rceil$ time-steps after t. □

Fact 9.12. *Suppose that a background thread T_k is in the tail phase at time-step t. Then, T_k will terminate by time-step $t + \frac{1.1}{c_{\text{spd}}} \cdot |E(\mathbf{B}_k^{(t)})|$.*

Proof. During the tail phase, T_k runs Algorithm 11 at a speed of removing c_{spd} elements from $E(\mathbf{B}_k)$ per time-step, including the last time-step because of the shortcut rule. The only way to add elements to $E(\mathbf{B}_k)$ during the computation phase is due to Algorithm 13 handling insertion, which can only add one element in a time-step. The algorithm terminates when $E(\mathbf{B}_k) = \varnothing$. So, the tail phase finishes in $\frac{1}{c_{\text{spd}}-1} \cdot \left| E(\mathbf{B}_k^{(t)}) \right| \le \frac{1.1}{c_{\text{spd}}} \cdot \left| E(\mathbf{B}_k^{(t)}) \right|$ time-steps after t. □

Fact 9.13. *Suppose that a background thread T_k is in the copy phase at time-step t. Then, T_k will finish copy phase by time-step $t + \frac{2.1}{c_{\text{spd}}} \cdot \tau_k^{\text{sus}}$.*

Proof. During each time-step within the copy phase, the thread T_k copies c_{spd} sets from $\mathcal{S}^\star(\mathbf{B}_k^{(t)})$ to the buffer solution $\mathcal{R}_k$. On the other hand, $\mathcal{S}^\star(\mathbf{B}_k)$ may grow by one set per time-step due to Algorithm 13. So, the copy phase finishes in $\left\lceil \frac{1}{c_{\text{spd}}-1} \cdot \tau_k^{\text{sus}} \right\rceil \le \frac{1.1}{c_{\text{spd}}} \cdot \tau_k^{\text{sus}} + 1 \le \frac{2.1}{c_{\text{spd}}} \cdot \tau_k^{\text{sus}}$ time-steps after t. Here, the last step uses $\tau_k^{\text{sus}} \ge c_{\text{spd}}$ by rule of shortcut threads. □

Lemma 9.14. *Suppose a background thread T_k is in the copy or tail phase at time-step t, T_k spent at most $\frac{30}{c_{\text{spd}}} \cdot \tau_k^{\text{sus}}$ time-steps in suspension phase. Then, T_k will terminate by time-step $t + \frac{5}{c_{\text{spd}}} \cdot \tau_k^{\text{sus}}$.*

Proof. Suppose T_k enters suspension, copy and tail phases at $t^{\text{sus}}, t^{\text{copy}}, t^{\text{tail}}$ respectively, and terminates at t^{end}. (If T_k is aborted in copy phase, let $t^{\text{tail}} = t^{\text{end}}$.) Since $t \in [t^{\text{copy}}, t^{\text{end}}]$, it suffices to prove $t^{\text{end}} - t^{\text{copy}} \le \frac{5}{c_{\text{spd}}} \cdot \tau_k^{\text{sus}}$. During the suspension and copy phases, $E(\mathbf{B}_k)$ can only be modified by insertion or deletion in one element per time-step. That is,

$$||E(\mathbf{B}_k(t_1))| - |E(\mathbf{B}_k(t_2))|| \le |t_1 - t_2|, \forall t_1, t_2 \in [t^{\text{sus}}, t^{\text{tail}}] \tag{33}$$

The assumption gives $t^{\text{copy}} - t^{\text{sus}} \le \frac{30}{c_{\text{spd}}} \cdot \tau_k^{\text{sus}}$. By Fact 9.13, $t^{\text{tail}} - t^{\text{copy}} \le \frac{2.1}{c_{\text{spd}}} \cdot \tau_k^{\text{sus}}$. If $t^{\text{end}} = t^{\text{tail}}$, we are done. Otherwise, it remains to bound $t^{\text{end}} - t^{\text{tail}} \le \frac{2.9}{c_{\text{spd}}} \cdot \tau_k^{\text{sus}}$. From (33), we have

$$|E(\mathbf{B}_k(t^{\text{tail}}))| \le |E(\mathbf{B}_k(t^{\text{sus}}))| + (t^{\text{tail}} - t^{\text{sus}}) \le \left(1 + \frac{32.1}{c_{\text{spd}}}\right) \tau_k^{\text{sus}} \le 1.1 \tau_k^{\text{sus}}.$$

By Fact 9.12, we have that $t^{\text{end}} - t^{\text{tail}} \le \frac{1.1}{c_{\text{spd}}}|E(\mathbf{B}_k(t^{\text{tail}}))| \le \frac{1.3\tau_k^{\text{sus}}}{c_{\text{spd}}}$.

□

Lemma 9.15. *Suppose a background thread T_k is in the copy or tail phase at time-step t. Then, T_k either enters copy phase or gets aborted in suspension phase by time-step $t + \frac{30}{c_{\text{spd}}} \cdot \tau_k^{\text{sus}}$.*

Proof. We prove the lemma by induction first on time in increasing order, then on level k in decreasing order. That is, in our inductive step, the assumption of Lemma 9.14 holds for all threads at levels higher than k, and also holds for all threads that entered copy phase before t.

If T_k is in suspension phase but does not enter copy phase at any time-step t', one of the following reasons must hold:

1. (stopped from above) There exists a higher thread T_j, $j > k$ in copy or tail phase such that $\tau_j^{\text{sus}} < 2\tau_k^{\text{sus}}$.
2. (stopped from below) There exists a lower thread T_ℓ, $\ell < k$ in copy or tail phase such that $\tau_\ell^{\text{sus}} < 2\tau_k^{\text{sus}}$. By the sequential behavior of threads, T_ℓ must have entered copy phase before t'.

If T_k is stopped from above at time-step t, then T_k cannot enter copy phase before T_j terminates since $\tau_j^{\text{sus}}, \tau_k^{\text{sus}}$ cannot change until the threads terminate. But, termination of T_j will abort T_k. So, T_k will get aborted in suspension phase. By Lemma 9.14, this will happen by time-step $t + \frac{5}{c_{\text{spd}}} \cdot \tau_j^{\text{sus}} \le t + \frac{10}{c_{\text{spd}}}\tau_k^{\text{sus}}$.

The remaining case is that T_k is stopped from below at time-step t. In this case, we define a sequence of levels $(\ell_1, \ell_2, \ldots, \ell_r)$ and time-steps $(t_0, t_1, t_2, \ldots, t_r)$ as follows. We start with $t_0 = t$. For each $i \ge 0$, we assume T_k is stopped from below at time-step t_i, and let ℓ_{i+1} be the highest thread in copy or tail phase at t_i such that $\ell < k, \tau_\ell^{\text{sus}} < 2\tau_k^{\text{sus}}$. Then, let t_{i+1} be the time-step when $T_{\ell_{i+1}}$ terminates. The sequence ends at r if T_k is not stopped from below or not in suspension phase after T_{ℓ_r} terminates at t_r.

By Lemma 9.14, $t_{i+1} - t_i \le \frac{5}{c_{\text{spd}}} \cdot \tau_{\ell_i}^{\text{sus}}$ for each i in the sequence. To bound the total waiting time $t_r - t$, we bound the sum of $\tau_{\ell_i}^{\text{sus}}$ as follows. For each $i \in [2, r]$, ℓ_i is higher than ℓ_{i-1} since T_{ℓ_i} is not aborted at the termination of $T_{\ell_{i-1}}$. Hence, T_{ℓ_i} is not in copy or tail phase at t_{i-2}, otherwise we should not have chosen ℓ_{i-1}. That is, T_{ℓ_i} enters copy phase during $[t_{i-2} + 1, t_{i-1}]$ when T_{i-1} is in copy or tail phase. By the rule to enter copy phase, $\tau_{\ell_i}^{\text{sus}} \le \frac{1}{2}\tau_{\ell_{i-1}}^{\text{sus}}$. Since this holds for all $i \in [2, r]$, $\tau_{\ell_i}^{\text{sus}}$ form a geometric series. We conclude

$$t_r - t \le \frac{5}{c_{\text{spd}}} \sum_{i=1}^{r} \tau_{\ell_i}^{\text{sus}} \le \frac{10}{c_{\text{spd}}} \cdot \tau_{\ell_1}^{\text{sus}} \le \frac{20}{c_{\text{spd}}} \cdot \tau_k^{\text{sus}}.$$

Note that T_k cannot enter copy phase before t_r since there is always some T_{ℓ_i} stopping T_k from below during $[t, t_r]$. Also, T_k is not stopped from below after t_r. So, after t_r, there are the following cases.

1. T_k gets aborted.
2. T_k enters copy phase.
3. T_k is stopped from above. We can use the same argument as the case of stopped from above at t, with an additional waiting time $t_r - t$. Then, T_k gets aborted in suspension phase by $t + \frac{30}{c_{\text{spd}}} \cdot \tau_k^{\text{sus}}$.

The lemma holds in all cases. □

We combine the results in this section as follows.

Lemma 9.16. *Suppose a background thread T_k is running at time-step t. Then, the lifetime of T_k is at most $0.1|L_k(\mathbf{F}^{(t)})|$ time-steps.*

Proof. Suppose T_k starts at t^{prep} and terminates at t^{end}. Suppose T_k enters computation phase, suspension phase and copy phase at $t^{\text{comp}}, t^{\text{sus}}, t^{\text{copy}}$ respectively. If T_k terminates before these phases, we collapse the phases by redefining the notations to be t^{end}. Any time bound for collapsed phases is trivially true since the time gap is 0 under this definition, even though the assumptions for corresponding lemmas may not hold. By the rule of base threads, we may assume $|L_k(\mathbf{F}^{(t^{\text{prep}})})| \geq c_{\text{spd}}$.

We have the following time bounds:

$$t^{\text{comp}} - t^{\text{prep}} \leq \frac{1.1}{c_{\text{spd}}} \cdot |L_k(\mathbf{F}^{(t^{\text{prep}})})| \qquad \text{(Fact 9.10)}$$

$$t^{\text{sus}} - t^{\text{comp}} \leq \frac{1.1}{c_{\text{spd}}} \cdot |L_k(\mathbf{F}^{(t^{\text{comp}})})| + 1 \qquad \text{(Fact 9.11 and lemma 9.3)}$$

$$t^{\text{end}} - t^{\text{sus}} \leq \frac{35}{c_{\text{spd}}} \cdot \tau_k^{\text{sus}} \qquad \text{(Lemmas 9.14 and 9.15)}$$

Let $\Delta = 0.1|L_k(\mathbf{F}^{(t^{\text{prep}})})|$. During $t \in [t^{\text{prep}}, t^{\text{prep}} + \Delta]$, $|L_k(\mathbf{F}^{(t)})|$ can change by at most 1 per time-step, so we have $|L_k(\mathbf{F}^{(t)})| \in [0.9|L_k(\mathbf{F}^{(t^{\text{prep}})})|, 1.1|L_k(\mathbf{F}^{(t^{\text{prep}})})|]$. To bound $\tau_k^{\text{sus}} = |\mathcal{S}^\star(\mathbf{B}_k^{t^{\text{sus}}})|$, we consider a bijection from each set $s \in \mathcal{S}^\star(\mathbf{B}_k^{t^{\text{sus}}})$ to an element e added to $C^\star(\mathbf{B}_k)$ accompanied by s in Algorithms 10, 11 and 13. These elements are distinct and belong to $L_k(\mathbf{F})$ when added to $C^\star(\mathbf{B}_k)$. So, $\tau_k^{\text{sus}} \leq 1.1|L_k(\mathbf{F}^{(t^{\text{prep}})})|$. In conclusion,

$$t^{\text{end}} - t^{\text{prep}} \leq (1.1 + 1.1 \times 1.1 + 1 + 35 \times 1.1)\frac{1}{c_{\text{spd}}} \cdot |L_k(\mathbf{F}^{(t^{\text{prep}})})| \leq \frac{45}{c_{\text{spd}}} \cdot |L_k(\mathbf{F}^{(t^{\text{prep}})})|$$

Since $\frac{45}{c_{\text{spd}}} < 0.1$, our above argument is valid for $[t^{\text{prep}}, t^{\text{end}}] \subseteq [t^{\text{prep}}, t^{\text{prep}} + \Delta]$. Because $|L_k(\mathbf{F}^{(t)})| > 0.9|L_k(\mathbf{F}^{(t^{\text{prep}})})|$, we conclude that $t^{\text{end}} - t^{\text{prep}} \leq \frac{50}{c_{\text{spd}}}|L_k(\mathbf{F}^{(t)})| = 0.1|L_k(\mathbf{F}^{(t)})|$. □

Lemma 9.17. *Suppose a background thread T_k is in the computation phase and $E(\mathbf{B}_k) > |\mathcal{S}^\star(\mathbf{B}_k)|$ at time-step t. Then, T_k will terminate by $t + 0.1|E(\mathbf{B}_k^{(t)})|$.*

Proof. Let t^{sus} be the time when T_k finishes computation phase, and t^{end} be the time when T_k terminates. By Fact 9.11, $t^{\text{sus}} - t \leq \frac{1.1}{c_{\text{spd}}} \cdot |E(\mathbf{B}_k^{(t)})| + 1 \leq \frac{2.1}{c_{\text{spd}}} \cdot |E(\mathbf{B}_k^{(t)})|$. Here, the last step uses the rule of shortcut threads. By Lemmas 9.14 and 9.15, $t^{\text{end}} - t^{\text{sus}} \leq \frac{35}{c_{\text{spd}}}\tau_k^{\text{sus}}$ (trivially true if T_k does not enter suspension phase). Since sets in $\mathcal{S}^\star(\mathbf{B}_k)$ cannot be removed, we have $\tau_k^{\text{sus}} \leq |\mathcal{S}^\star(\mathbf{B}_k^{(t)})| + (t^{\text{sus}} - t) \leq (1 + \frac{2.1}{c_{\text{spd}}}) \cdot |E(\mathbf{B}_k^{(t)})|$. In conclusion, $t^{\text{end}} - t \leq (\frac{2.1}{c_{\text{spd}}} + \frac{35}{c_{\text{spd}}}(1 + \frac{2.1}{c_{\text{spd}}})) \cdot |E(\mathbf{B}_k^{(t)})| \leq 0.1|E(\mathbf{B}_k^{(t)})|$. □

9.5 Tidy Property

Recall that we say a hierarchical solution $\mathbf{F}$ is ε-tidy if $|D_k(\mathbf{F})| + |P_k(\mathbf{F})| \leq \varepsilon \cdot (|A_k(\mathbf{F})| + |E(\mathbf{F})|)$ for all $k \in [0, \ell_{\max}]$. We next establish the tidy property for the foreground threads and all background threads. It will be useful in bounding approximation factor. Intuitively, we can expect from the lifetime bounds that there cannot accumulate too many dormant and passive elements.

Lemma 9.18. *Suppose a background thread T_k is in computation or later phases at time-step t, and $\mathbf{F}$ was ε-tidy when T_k was created at time-step t^{prep}. Then, $\mathbf{B}_k^{(t)}$ is 0.1-tidy at levels $j \leq k$, and $(1.2\varepsilon + 0.2)$-tidy at levels $j > k$.*

Proof. First, we consider levels $j \leq k$. Notice that any element added to $C(\mathbf{B}_k)$ by T_k must have $\text{lev}_e(\mathbf{B}_k) = p_k$, and $\text{lev}_e(\mathbf{B}_k)$ cannot change once decided. So, before p_k is decreased to j, $D_j(\mathbf{B}_k) = P_j(\mathbf{B}_k) = \emptyset$. Suppose p_k

is decreased to j at time t_j. After t_j, the only way to generate a dormant element in $D_j(\mathbf{B}_k)$ is by Algorithm 12, and the only way to generate a passive element in $P_j(\mathbf{B}_k)$ is by Algorithm 13. So, $|D_j(\mathbf{B}_k)| + |P_j(\mathbf{B}_k)|$ can increase by at most 1 per time-step after time-step t_j. That is,

$$|D_j(\mathbf{B}_k^{(t)})| + |P_j(\mathbf{B}_k^{(t)})| \le t - t_j.$$

We claim that $t - t_j \le 0.1|E(\mathbf{B}_k^{(t_j)})|$, which implies the tidy property for level j.

We now prove the claim. Just before p_k decrease to $j \le k$, Algorithm 11 have finished the last set at level $j+1$. At that time, there are two cases: either $|E(\mathbf{B}_k)| > |\mathcal{S}^\star(\mathbf{B}_k)|$, or $|E(\mathbf{B}_k)| \le |\mathcal{S}^\star(\mathbf{B}_k)|$. In the former case, the claim follows Lemma 9.17. In the latter case, T_k will first enter suspension phase, and must wait until the tail phase to decrease p_k to j. Then, the claim follows Fact 9.12.

Next, we consider levels $j > k$. By the tidy property of $\mathbf{F}^{(t^{\text{prep}})}$,

$$|D_j(\mathbf{F}^{(t^{\text{prep}})})| + |P_j(\mathbf{F}^{(t^{\text{prep}})})| \le \varepsilon \cdot |A_j(\mathbf{F}^{(t^{\text{prep}})})|.$$

At the beginning of $\mathbf{B}_k$, we view all elements in $A_k(\mathbf{F})$ are exposed in $E(\mathbf{B}_k)$ and $A(\mathbf{B}_k)$ is empty. With a slight abuse of notation, we view dormant and passive elements from $\mathbf{F}$ to be in $D_j(\mathbf{B}_k)$ and $P_j(\mathbf{B}_k)$. The position of an element in $E(\mathbf{B}_k), A(\mathbf{B}_k), P(\mathbf{B}_k)$ or $D(\mathbf{B}_k)$ follows the algorithm after it gets processed by Algorithm 10 in the preparation phase. (The alternate view is valid because the lemma applies after the preparation phase.) Then, we have

$$|D_j(\mathbf{B}_k^{(t^{\text{prep}})})| + |P_j(\mathbf{B}_k^{(t^{\text{prep}})})| \le \varepsilon(|A_j(\mathbf{B}_k^{(t^{\text{prep}})})| + |E(\mathbf{B}_k^{(t^{\text{prep}})})|).$$

We next discuss how this relation changes from t^{prep} to t. A new dormant element in $D_j(\mathbf{B}_k)$ (not from $D_j(\mathbf{F}^{(t^{\text{prep}})})$) can only be generated by Algorithm 12 or the foreground thread deletion algorithm. Similarly, a new passive element in $P_j(\mathbf{B}_k)$ (not from $P_j(\mathbf{F}^{(t^{\text{prep}})})$) can only be generated by Algorithm 13 or the foreground thread insertion algorithm. In addition, Algorithm 10 in preparation phase may move a passive element in $P_k(\mathbf{F})$ to level $k+1$ or to $E(\mathbf{B}_k)$, which cannot increase $|D_j(\mathbf{B}_k^{(t^{\text{prep}})})| + |P_j(\mathbf{B}_k^{(t^{\text{prep}})})|$. In conclusion, $|D_j(\mathbf{B}_k^{(t^{\text{prep}})})| + |P_j(\mathbf{B}_k^{(t^{\text{prep}})})|$ can only increase by at most 1 per time-step due to insertion/deletion. Algorithms 10 and 11 may assign an exposed element to be active at levels $p_k \le k+1$, which does not change $|A_j(\mathbf{B}_k)| + |E(\mathbf{B}_k)|$. Algorithms 12 and 13 may remove or add an element to $E(\mathbf{B}_k)$ due to deletion or insertion. So, $|A_j(\mathbf{B}_k)| + |E(\mathbf{B}_k)|$ may decrease by at most 1 per time-step. By Lemma 9.16, $\Delta_t := t - t^{\text{prep}} \le 0.1|L_k(\mathbf{F}^{(t^{\text{prep}})})| = 0.1(|A_j(\mathbf{B}_k^{(t^{\text{prep}})})| + |E(\mathbf{B}_k^{(t^{\text{prep}})})|)$. So,

$$\begin{aligned}
|D_j(\mathbf{B}_k^{(t)})| + |P_j(\mathbf{B}_k^{(t)})| &\le |D_j(\mathbf{B}_k^{(t^{\text{prep}})})| + |P_j(\mathbf{B}_k^{(t^{\text{prep}})})| + \Delta_t \\
&\le (\varepsilon + 0.1)(|A_j(\mathbf{B}_k^{(t^{\text{prep}})})| + |E(\mathbf{B}_k^{(t^{\text{prep}})})|) \\
&\le (\varepsilon + 0.1)(|A_j(\mathbf{B}_k^{(t)})| + |E(\mathbf{B}_k^{(t)})| + \Delta_t) \\
&\le (\varepsilon + 0.1)(1 + \frac{0.1}{1 - 0.1})(|A_j(\mathbf{B}_k^{(t)})| + |E(\mathbf{B}_k^{(t)})|) \\
&\le (1.2\varepsilon + 0.2)(|A_j(\mathbf{B}_k^{(t)})| + |E(\mathbf{B}_k^{(t)})|)
\end{aligned}$$

□

Lemma 9.19. *The foreground solution* $\mathbf{F}$ *is always* 0.5*-tidy.*

Proof. We prove by induction on time-step t. As the base case, the initial greedy solution is 0-tidy. For the inductive case, it suffices to prove the following claim for every level $k \in [0, \ell_{\max}]$: Suppose $\mathbf{F}$ is 0.2-tidy at level k when T_k starts. Then, $\mathbf{F}$ is 0.2-tidy at level k when T_k terminates, and $\mathbf{F}$ is 0.5-tidy at level k throughout

the lifetime of T_k.

When T_k terminates, it either normally terminates or gets aborted due to a normal termination of a higher thread. In both cases, Lemma 9.18 guarantees that the $\mathbf{F}$ is 0.1-tidy at level k after the switch step. This establishes the first part of the claim.

For the second part of the claim, suppose T_k is created at time-step t^{prep} and terminates at time-step t^{end}. By Lemma 9.16, for any $t \in [t^{\text{prep}}, t^{\text{end}}]$,

$$\Delta_t := t^{\text{end}} - t^{\text{prep}} \le 0.1|L_k(\mathbf{F}^{(t)})| = 0.1(|A_k(\mathbf{F}^{(t)})| + |P_k(\mathbf{F}^{(t)})|).$$

Notice that $E(\mathbf{F})$ is always empty since $\mathbf{F}$ is feasible. Since $\mathbf{F}$ is 0.2-tidy when T_k is created,

$$|D_k(\mathbf{F}^{(t^{\text{prep}})})| + |P_k(\mathbf{F}^{(t^{\text{prep}})})| \le 0.2|A_k(\mathbf{F}^{(t^{\text{prep}})})|.$$

Combining the above inequalities gives

$$\Delta_t \le |P_j(\mathbf{B}_k^{(t^{\text{prep}})})| \tag{34}$$

During the lifetime of T_k, a new dormant element in $D_k(\mathbf{F})$ can be generated by the foreground thread deletion algorithm, a new passive element in $P_k(\mathbf{F})$ can be generated by the foreground thread insertion algorithm, and an element can be removed from $A_k(\mathbf{F})$ by the foreground thread deletion algorithm. These events can happen for one element per time-step due to insertion/deletion. In addition, termination of a lower thread T_j, $j < k$ may remove some dormant elements, or move some passive elements to be either passive or active with levels $j+1 \le k$. These events cannot increase $|D_k(\mathbf{F})|$, $|P_k(\mathbf{F})|$ or decrease $|A_k(\mathbf{F})|$. For any $t \in [t^{\text{prep}}, t^{\text{end}}]$, we have the following.

$$\begin{aligned}
|D_k(\mathbf{F}^{(t)})| + |P_k(\mathbf{F}^{(t)})| &\le |D_k(\mathbf{F}^{(t^{\text{prep}})})| + |P_k(\mathbf{F}^{(t^{\text{prep}})})| + \Delta_t \\
&\le 0.2(|A_k(\mathbf{F}^{(t)})| + \Delta_t) + \Delta_t \\
&\le 0.2|A_k(\mathbf{F}^{(t)})| + 0.12(|A_k(\mathbf{F}^{(t)})| + |P_k(\mathbf{F}^{(t)})|) \\
|D_k(\mathbf{F}^{(t)})| + |P_k(\mathbf{F}^{(t)})| &\le \frac{0.2 + 0.12}{1 - 0.12}|A_k(\mathbf{F}^{(t)})| \le 0.5|A_k(\mathbf{F}^{(t)})|
\end{aligned}$$

□

9.6 Approximation Factor

The $O(f)$-competitiveness of the foreground solution and each (non-empty) background solution follows Lemma 9.20.

The buffer solutions $\mathcal{R}_k$ are empty for background threads before copy phase, and have size $O(\tau_k^{\text{sus}})$ after entering copy phase. Since τ_k^{sus} of threads in copy and tail phases must form a geometric series, the total size of buffer solutions is at most constant times larger than the highest thread, which is $O(f)$-competitive.

Definition 7. The foreground solution $\mathbf{F}$ is said to be ε-tidy at level k if $|D_k^{\mathbf{F}}| + |P_k^{\mathbf{F}}| \le \varepsilon \cdot |A_k^{\mathbf{F}}|$, and ε-dirty at level k otherwise. If $\mathbf{F}$ is ε-tidy at every level $k \in [0, \ell_{\max}]$, we say it is ε-tidy.

A background solution $\mathbf{B}_k$ is said to be ε-tidy at level $j \ge p_k$ if $|D_j^{\mathbf{B}_k}| + |P_j^{\mathbf{B}_k}| \le \varepsilon \cdot |A_j^{\mathbf{B}_k}| + |E(\mathbf{B}_k)|$, and ε-dirty at level j otherwise. If $\mathbf{B}_k$ is ε-tidy at every level $j \in [p_k, \ell_{\max}]$, we say it is ε-tidy.

Lemma 9.20. *If $\mathbf{F}$ or $\mathbf{B}_k$ is an ε-tidy hierarchical solution for some $\varepsilon = \Omega(1)$, then it is $O(f)$-competitive.*

Proof. Given Lemmas 7.2 and 9.4 to 9.7, it remains to prove $w_D \le \varepsilon w_L$ for $\mathbf{F}$ or $\mathbf{B}_k$.

Extend the notation $D_j(\mathbf{F})$ to all integers j, so that $D_j(\mathbf{F}) = D_{\ell_{\max}}(\mathbf{F})$ when $j > \ell_{\max}$, and $D_j(\mathbf{F}) = \emptyset$ when $j < 0$.

$$
\begin{aligned}
w_D &= \sum_{j=0}^{\infty} w_{D_j(\mathbf{F})} - w_{D_{j-1}(\mathbf{F})} && \leq \sum_{j=0}^{\infty} (1+\varepsilon)^{-j} (|D_j(\mathbf{F})| - |D_{j-1}(\mathbf{F})|) && \text{(level invariant)} \\
&= \sum_{j=0}^{\infty} ((1+\varepsilon)^{-j} - (1+\varepsilon)^{-(j+1)}) |D_j(\mathbf{F})| && \leq \sum_{j=0}^{\infty} \varepsilon (1+\varepsilon)^{-(j+1)} \cdot \varepsilon |A_j(\mathbf{F})| && \text{(tidy property)} \\
&= \varepsilon \sum_{j=0}^{\infty} (1+\varepsilon)^{-j} (|A_j(\mathbf{F})| - |A_{j-1}(\mathbf{F})|) && = \varepsilon \cdot w_A \leq \varepsilon \cdot w_L && \text{(definition of active elements)}
\end{aligned}
$$

The proof for background solutions is similar if we move $E(\mathbf{B}_k)$ to $A_{p_k}(\mathbf{B}_k)$ and take the sum from p_k instead of 0. This is valid because no element or set in $\mathbf{B}_k$ can have level $< p_k$. □

Lemma 9.21. *Suppose that a background thread T_k is in the copy or tail phase at time-step t. Then, $|\mathcal{S}^\star(\mathbf{B}_k^{(t)})| \leq 3\tau_k^{\text{sus}}$.*

Proof. Suppose T_k enters suspension phase at t^{sus}. Define potential function $\phi := |\mathcal{S}^\star(\mathbf{B}_k)| + |E(\mathbf{B}_k)|$. We have $\phi^{(t^{\text{sus}})} \leq 2\tau_k^{\text{sus}}$ by the criteria to enter suspension phase. During the suspension, copy and tail phases, an insertion in Algorithm 13 either adds a new exposed element, or adds a set to cover a new passive element. In both cases, ϕ increase by 1. A deletion does not increase ϕ. When Algorithm 11 adds a set s to $\mathcal{S}^\star(\mathbf{B}_k)$, the algorithm must remove at least one exposed element, and ϕ cannot increase.

We have $t - t^{\text{sus}} \leq 0.1\tau_k^{\text{sus}}$ by Lemmas 9.14 and 9.15. In conclusion,

$$|\mathcal{S}^\star(\mathbf{B}_k^{(t)})| \leq \phi^{(t)} \leq \phi^{(t^{\text{sus}})} + (t - t^{\text{sus}}) \leq 3\tau_k^{\text{sus}}.$$

□

Lemma 9.22. *The output is $O(f)$-competitive.*

Proof. The foreground solution is $O(f)$-competitive by Lemmas 9.19 and 9.20. For each background thread after preparation phase, $\mathcal{S}^\star(\mathbf{B}_k)$ is $O(f)$-competitive by Lemmas 9.18 and 9.20. Since buffer solutions $\mathcal{R}_k$ are empty in preparation phase and always a subset of $\mathcal{S}^\star(\mathbf{B}_k)$, each $\mathcal{R}_k$ is $O(f)$-competitive.

By Lemma 9.21, $|\mathcal{R}_k| \leq 3\tau_k^{\text{sus}}$ for each buffer solution. By the criteria to enter copy phase, τ_k^{sus} are upper bounded by a geometric series with ratio 0.5. So, the total size of buffer solutions is upper bounded by the largest τ_k^{sus}, which is $O(f)$-competitive by Lemmas 9.18 and 9.20 at the time T_k enters suspension phase. □

9.7 Implementation Details and Update Time

First, we describe the data structures used to maintain the foreground, background, and buffer solutions.

Data Structures. The algorithm maintains the following data structures:

1. The primal solutions maintained by the algorithm, including $\mathcal{S}(\mathbf{F})$ for the foreground solution, $\mathcal{S}^\star(\mathbf{B}_k)$ for the background solutions, and $\mathcal{R}_k$ for the buffer solutions, are organized by levels, and the sets in each level are stored in a balanced binary search tree. We call these set BSTs and use $\mathbf{T}_{\text{set}}(S, \ell)$ to denote the set BST for solution S at level ℓ. The algorithm maintains a global pointer to each set BST.

2. The covered elements in hierarchical solutions maintained by the algorithm, including $C(\mathbf{F})$ for the foreground solution and $C^\star(\mathbf{B}_k)$ for the background solutions, are partitioned into active, passive and dormant elements, and organized by levels. The elements in each level of each part are stored in a balanced binary search trees. We call these element BSTs and use $\mathbf{T}_{\mathrm{A}}(S,\ell), \mathbf{T}_{\mathrm{P}}(S,\ell), \mathbf{T}_{\mathrm{D}}(S,\ell)$ to denote the three BSTs for solution S at level ℓ. A node in an element BST stores the identity of an element e, its level $\mathrm{lev}_e(S)$ and dual value $w_e(S)$. The algorithm maintains a global pointer to each element BST.

3. For each background thread T_k, we have an array indexed by all sets where for each set s, it stores a BST containing all exposed elements in $s \cap E(\mathbf{B}_k)$. We call these exposed BSTs and denote them $\mathbf{T}_{\mathrm{E}}(\mathbf{B}_k, s)$. In contrast to other element BSTs, the exposed BSTs do no maintain levels or dual values (since they are undefined).

4. For the foreground solution and each level ℓ, we have a BST maintaining the total dual value of elements in s at level ℓ of $\mathbf{F}$ for all sets s incident to level ℓ of $\mathbf{F}$. We call this foreground incident BSTs and denote it $\mathbf{T}_{\mathrm{in}}(\mathbf{F},\ell)$. Similarly, for each background solution $\mathbf{B}_k$ and each level $\ell \le k$, we have a BST maintaining the total dual value of elements in s at levels ℓ of $\mathbf{B}_k$ for all sets s incident to level k of $\mathbf{F}$. We call this background incident BSTs and denote it $\mathbf{T}_{\mathrm{in}}(\mathbf{B}_k,\ell)$. The algorithm maintains a global pointer to each incident BST. In contrast to the set BSTs, we allow duplicate sets in the incident BSTs.

5. For each background thread T_k, we use a priority queue Q_k to maintain the dual values $w_s(\mathbf{B}_k)$ for all sets $s \in \mathcal{S}^{\mathrm{in}}(E(\mathbf{B}_k))$. Recall that $w_s(\mathbf{B}_k)$ consists of three parts: the total dual value in $C^\star(\mathbf{B}_k)$, the total dual value at levels $\ge k+1$ in $\mathbf{F}$, and $(1+\varepsilon)^{-p_k} \cdot |s \cap E(\mathbf{B}_k)|$. Denote $w_s^\star$ to be the sum of first two parts. Since Algorithm 11 needs to decide whether there exists a set with $w_s \ge (1+\varepsilon)^{-1}$ for changing p_k, the Q_k actually maintains a target value $t_s := \frac{(1+\varepsilon)^{-1} - w_s^\star}{|s \cap E(\mathbf{B}_k)|}$ for sets s, and converts the maximum query on w_s to minimum query on the target values. It is straightforward to check that the condition $w_s \ge (1+\varepsilon)^{-1}$ is equivalent to $t_s \le (1+\varepsilon)^{-p_k}$. [11]

Update Time Bound. We now give details of how the data structures are updated by the algorithm, and derive resulting bounds on the update time of the algorithm.

Lemma 9.23. *Given the index of an element e or set s and a hierarchical solution $\mathbf{F}$ or $\mathbf{B}_k$, the data structures can answer in $O(\log^2 n)$ time:*

1. *Whether $e \in C(\mathbf{F}), s \in \mathcal{S}(\mathbf{F}), e \in C^\star(\mathbf{B}_k), e \in E(\mathbf{B}_k)$ or $s \in \mathcal{S}^\star(\mathbf{B}_k)$.*
2. *$\mathrm{lev}_e(\mathbf{F})$ and $w_e(\mathbf{F})$ if $e \in C(\mathbf{F})$, or $\mathrm{lev}_e(\mathbf{B}_k)$ and $w_e(\mathbf{B}_k)$ if $e \in C^\star(\mathbf{B}_k)$.*
3. *$w_s(\mathbf{F})$ or $w_s(\mathbf{B}_k)$.*

Proof. Since the input is only the index, we first need to find the node representing the element or set in the corresponding element BSTs, set BSTs or priority queues. Search a given index in a given BST or priority queue takes $O(\log n)$ time. Because we do not know the level of the input element or set, we need to search in $O(\log n)$ BSTs for all levels. This takes $O(\log^2 n)$ time in total.

Queries 1 and 2 can be answered by information stored in the node. Next, we consider query 3 on total dual value w_s. For the foreground solution, the value is maintained in the foreground incident BSTs. For the background solution, the value consists of three parts, which can be found in the foreground incident BSTs $\mathbf{T}_{\mathrm{in}}(\mathbf{F},\ell), \ell > k$, background incident BSTs $\mathbf{T}_{\mathrm{in}}(\mathbf{B}_k,\ell), \ell \le k$, and exposed BST $\mathbf{T}_{\mathrm{E}}(k,s)$ respectively. Here, we make $O(\log n)$ queries on $O(\log n)$ BSTs, which costs $O(\log^2 n)$ time. □

[11] Here, dividing by 0 outputs $+\infty$ when the nominator is positive and $-\infty$ when the nominator is nonpositive.

Lemma 9.24. *The data structures can be maintained in $O(f \log^2 n)$ time for the following operations:*

1. *Adding or removing a set in $\mathcal{S}(\mathbf{F}), \mathcal{S}^\star(\mathbf{B}_k), \mathcal{R}_k$ or $\mathcal{S}^{in}(E(\mathbf{B}_k))$.*
2. *Changing element status of live/dormant, active/passive, covered/exposed in $C(\mathbf{F})$, $C^\star(\mathbf{B}_k)$ or $E(\mathbf{B}_k)$.*
3. *Updating $w_e(\mathbf{F}), e \in C(\mathbf{F})$ or $w_e(\mathbf{B}_k), e \in C^\star(\mathbf{B}_k)$.*
4. *Adding or removing an element in $C(\mathbf{F})$, $C^\star(\mathbf{B}_k)$ or $E(\mathbf{B}_k)$.*

The input to the operations is the hierarchical solution, index of the element or set, and its new level and weight if they are involved.

Proof. As argued in Lemma 9.23, we can find the node representing the element or set in the corresponding element BSTs, set BSTs or priority queues in $O(\log^2 n)$ time. We now bound the running time of each operation.

1. Adding or removing a set in $\mathcal{S}(\mathbf{F}), \mathcal{S}^\star(\mathbf{B}_k)$ or $\mathcal{R}_k$ can be handled by adding or removing a node in the corresponding set BST. Adding or removing a set in $\mathcal{S}^{\text{in}}(E(\mathbf{B}_k))$ can be handled by adding or removing a node in the priority queue. These operations take $O(\log n)$ time. The total running time is $O(\log^2 n)$ for finding the node.

2. For all types of change of status, we can maintain the data structures by removing the node representing the element from the corresponding element BST ($\mathbf{T}_\text{A}$, $\mathbf{T}_\text{P}$, $\mathbf{T}_\text{D}$ or $\mathbf{T}_\text{E}$), and add a new node representing the element to another element BST. This takes $O(\log n)$ time.

   If the operations involves $E(\mathbf{B}_k)$, we check whether each incident sets is added or removed from $\mathcal{S}^{\text{in}}(E(\mathbf{B}_k))$. By item 1, this takes $O(f \log^2 n)$ time in total. We also need to update the target values of all incident sets in Q_k to reflect the changes in $|s \cap E(\mathbf{B}_k)|$ (which can be queried from $\mathbf{T}_\text{E}(k, s)$). This is at most f priority queue operations. The total running time is $O(f \log^2 n)$.

3. For change of dual value of an element e in $C(\mathbf{F})$ (resp. $C^\star(\mathbf{B}_k)$), we first update its node in the corresponding element BST, and then update the total dual value of all incident sets in $\mathbf{T}_{\text{in}}(\mathbf{F}, \text{lev}_e(\mathbf{F}))$ (resp. $\mathbf{T}_{\text{in}}(\mathbf{B}_k, \text{lev}_e(\mathbf{B}_k))$). After that, for each incident set s, we update its target value in Q_ℓ for all levels ℓ. (A change of dual value in foreground may affect the target value in all background threads.) These operations take $O(f \log^2 n)$ time.

4. Adding or removing an element can be handled by adding or removing a node in the corresponding element BST ($\mathbf{T}_\text{A}$, $\mathbf{T}_\text{P}$, $\mathbf{T}_\text{D}$ or $\mathbf{T}_\text{E}$). Since this changes the dual value of the element, we repeat the argument in item 3. If the operation involves $E(\mathbf{B}_k)$, we also repeat the argument in item 2. The total running time is $O(f \log^2 n)$. □

Lemma 9.25. *In any time-step, the foreground thread or a background thread T_k calls runs for $O(f \log^2 n)$ time. Therefore, the worst-case update time of the algorithm is $O(f \log^3 n)$.*

Proof. First, we consider an element insertion or deletion. This is handled by the foreground thread, and Algorithms 12 and 13 for each background thread. All these algorithms executes $O(f)$ queries described in Lemma 9.23 and $O(1)$ operations described in Lemma 9.24. So, the running time is $O(f \log^2 n)$.

Next, we analyze the operations performed by a background thread T_k based on the phase it is in. This includes Algorithm 10 in the preparation phase and Algorithm 11 in the computation and tail phases. We limit the work in each phase of a time-step to be processing c_{spd} elements and copying c_{spd} sets. Processing an element

or copying a set calls $O(f)$ queries described in Lemma 9.23 and $O(1)$ operations described in Lemma 9.24. So, these work take $O(f \log^2 n)$ time. The remaining step that cannot be bounded by these operations is when p_k decreases in Algorithm 11. In this situation, the data structure need no update, because we do not maintain the value of exposed elements, and the priority queue maintains target values instead of total dual values. The condition of the while loop in Algorithm 11 can be checked using the minimum target value in Q_k, since $w_s \geq (1+\varepsilon)^{-1}$ is equivalent to $t_s \leq (1+\varepsilon)^{-p_k}$.

Finally, we bound the running time of a termination step at the normal termination of a background thread T_k. We use pointer switch to replace the foreground data structures $(\mathbf{T}_{\text{set}}(\mathbf{F}, \ell), \mathbf{T}_{\text{A}}(\mathbf{F}, \ell), \mathbf{T}_{\text{P}}(\mathbf{F}, \ell), \mathbf{T}_{\text{D}}(\mathbf{F}, \ell)$ and $\mathbf{T}_{\text{in}}(\mathbf{F}, \ell))$ at levels $\ell \leq k$ by the corresponding background data structures $(\mathbf{T}_{\text{set}}(\mathbf{B}_k, \ell), \mathbf{T}_{\text{A}}(\mathbf{B}_k, \ell), \mathbf{T}_{\text{P}}(\mathbf{B}_k, \ell), \mathbf{T}_{\text{D}}(\mathbf{B}_k, \ell)$ and $\mathbf{T}_{\text{in}}(\mathbf{B}_k, \ell))$. Besides this, we merge the data structures for level $k+1$. This is $O(\log n)$ BST operations and takes $O(\log^2 n)$ time.

By highest level invariant, the merge cannot create duplicate nodes in element BSTs or set BSTs. Merging incident BSTs at level $k+1$ may create duplicate sets in $\mathbf{T}_{\text{in}}(\mathbf{F}, k+1)$. We allow duplicate in incident BSTs. We claim that the size of each incident BST is at most $\widetilde{O}(nf)$, so that each BST operation takes $O(\log n)$ time. Note that duplicate sets cannot be created within a background solution, in particular the highest one $\mathbf{B}_{\ell_{\max}}$. So, whenever $T_{\ell_{\max}}$ terminates, the switch step replaces all foreground incident BSTs with no duplicate sets. Since each element is incident to at most f sets, this is at most nf sets. Between two termination of $T_{\ell_{\max}}$, each step can process $\widetilde{O}(1)$ elements, which adds $\widetilde{O}(f)$ sets to all incident BSTs. By Lemma 9.16, the lifetime of $T_{\ell_{\max}}$ is $O(n)$. Since $T_{\ell_{\max}}$ cannot be aborted, we must witness its termination every $O(n)$ time-steps, and the total size of each incident BSTs cannot exceed $\widetilde{O}(nf)$. □

The above implementation details and running time analysis work for the goal of bounding worst case insertion recourse. To bound the deletion recourse as well, we need to deamortize the deletion recourse according to Lemma 3.6. The garbage collection step can be efficiently implemented as follows. When we switch out some set BST from the foreground solution, we move that to a BST representing all garbage sets. The algorithm performs an additional garbage collection step that removes $O(\log n)$ sets in the garbage set from the output.

Lemma 9.26. *The deamortized algorithm has worst-case update time $O(f \log^3 n)$.*

We have concluded the proof of Theorem 7.1 by combining Lemmas 9.8, 9.9, 9.22 and 9.26.

References

[AAG+19] Amir Abboud, Raghavendra Addanki, Fabrizio Grandoni, Debmalya Panigrahi, and Barna Saha. Dynamic set cover: improved algorithms and lower bounds. In *Proceedings of the 51st Annual ACM SIGACT Symposium on Theory of Computing (STOC)*, pages 114–125. ACM, 2019.

[ACC+18] Moab Arar, Shiri Chechik, Sarel Cohen, Cliff Stein, and David Wajc. Dynamic matching: Reducing integral algorithms to approximately-maximal fractional algorithms. In *45th International Colloquium on Automata, Languages, and Programming (ICALP)*, volume 107 of *LIPIcs*, pages 7:1–7:16, 2018.

[AKK25] Sepehr Assadi, Sanjeev Khanna, and Peter Kiss. Improved bounds for fully dynamic matching via ordered ruzsa-szemerédi graphs. In *Proceedings of the 2025 Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 2971–2990, 2025.

[AS21] Sepehr Assadi and Shay Solomon. Fully dynamic set cover via hypergraph maximal matching: An optimal approximation through a local approach. In *29th Annual European Symposium on Algorithms (ESA)*, volume 204 of *LIPIcs*, pages 8:1–8:18. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021.

[BCH17] Sayan Bhattacharya, Deeparnab Chakrabarty, and Monika Henzinger. Deterministic fully dynamic approximate vertex cover and fractional matching in O(1) amortized update time. In Friedrich Eisenbrand and Jochen Könemann, editors, *Integer Programming and Combinatorial Optimization - 19th International Conference (IPCO)*, volume 10328 of *Lecture Notes in Computer Science*, pages 86–98, 2017.

[BDL21] Aaron Bernstein, Aditi Dudeja, and Zachary Langley. A framework for dynamic matching in weighted graphs. In *53rd Annual ACM SIGACT Symposium on Theory of Computing (STOC)*, pages 668–681, 2021.

[Beh23] Soheil Behnezhad. Dynamic algorithms for maximum matching size. In *Proceedings of the 2023 ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 129–162, 2023.

[BG24] Soheil Behnezhad and Alma Ghafari. Fully dynamic matching and ordered ruzsa-szemerédi graphs. In *65th IEEE Annual Symposium on Foundations of Computer Science (FOCS)*, pages 314–327, 2024.

[BGS11] Surender Baswana, Manoj Gupta, and Sandeep Sen. Fully dynamic maximal matching in O (log n) update time. In *IEEE 52nd Annual Symposium on Foundations of Computer Science (FOCS)*, pages 383–392, 2011.

[BHI15a] Sayan Bhattacharya, Monika Henzinger, and Giuseppe F. Italiano. Design of dynamic algorithms via primal-dual method. In *Automata, Languages, and Programming - 42nd International Colloquium, ICALP 2015, Kyoto, Japan, July 6-10, 2015, Proceedings, Part I*, volume 9134 of *Lecture Notes in Computer Science*, pages 206–218, 2015.

[BHI15b] Sayan Bhattacharya, Monika Henzinger, and Giuseppe F. Italiano. Deterministic fully dynamic data structures for vertex cover and matching. In *Proceedings of the Twenty-Sixth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 785–804, 2015.

[BHN17] 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 *Proceedings of the Twenty-Eighth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 470–489, 2017.

[BHN19] Sayan Bhattacharya, Monika Henzinger, and Danupon Nanongkai. A new deterministic algorithm for dynamic set cover. In David Zuckerman, editor, *60th IEEE Annual Symposium on Foundations of Computer Science (FOCS)*, pages 406–423. IEEE Computer Society, 2019.

[BHNW21] Sayan Bhattacharya, Monika Henzinger, Danupon Nanongkai, and Xiaowei Wu. Dynamic set cover: Improved amortized and worst-case update time. In Dániel Marx, editor, *Proceedings of the 2021 ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 2537–2549. SIAM, 2021.

[BK19] Sayan Bhattacharya and Janardhan Kulkarni. Deterministically maintaining a $(2+\epsilon)$-approximate minimum vertex cover in $o(1/\epsilon^2)$ amortized update time. In *Proceedings of the Thirtieth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 1872–1885, 2019.

[BKSW23] Sayan Bhattacharya, Peter Kiss, Thatchaphol Saranurak, and David Wajc. Dynamic matching with better-than-2 approximation in polylogarithmic update time. In *Proceedings of the 2023 ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 100–128, 2023.

[BS16] Aaron Bernstein and Cliff Stein. Faster fully dynamic matchings with small approximation ratios. In *Proceedings of the Twenty-Seventh Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 692–711, 2016.

[BSZ25] Anton Bukov, Shay Solomon, and Tianyi Zhang. Nearly optimal dynamic set cover: Breaking the quadratic-in-f time barrier. In Yossi Azar and Debmalya Panigrahi, editors, *Proceedings of the 2025 Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 824–863. SIAM, 2025.

[DS14] Irit Dinur and David Steurer. Analytical approach to parallel repetition. In *Symposium on Theory of Computing (STOC)*, pages 624–633, 2014.

[GKKP17] Anupam Gupta, Ravishankar Krishnaswamy, Amit Kumar, and Debmalya Panigrahi. Online and dynamic algorithms for set cover. In Hamed Hatami, Pierre McKenzie, and Valerie King, editors, *Proceedings of the 49th Annual ACM SIGACT Symposium on Theory of Computing, STOC 2017, Montreal, QC, Canada, June 19-23, 2017*, pages 537–550, 2017.

[KR03] Subhash Khot and Oded Regev. Vertex cover might be hard to approximate to within 2-\varepsilon. In *18th Annual IEEE Conference on Computational Complexity (CCC)*, page 379, 2003.

[NS13] Ofer Neiman and Shay Solomon. Simple deterministic algorithms for fully dynamic maximal matching. In *Symposium on Theory of Computing Conference (STOC)*, pages 745–754, 2013.

[OR10] Krzysztof Onak and Ronitt Rubinfeld. Maintaining a large matching and a small vertex cover. In *Proceedings of the 42nd ACM Symposium on Theory of Computing (STOC)*, pages 457–464. ACM, 2010.

[SU23] Shay Solomon and Amitai Uzrad. Dynamic ((1+ϵ) ln n)-approximation algorithms for minimum set cover and dominating set. In Barna Saha and Rocco A. Servedio, editors, *Proceedings of the 55th Annual ACM Symposium on Theory of Computing, STOC 2023, Orlando, FL, USA, June 20-23, 2023*, pages 1187–1200. ACM, 2023.

[SUZ24] Shay Solomon, Amitai Uzrad, and Tianyi Zhang. A lossless deamortization for dynamic greedy set cover. In *65th IEEE Annual Symposium on Foundations of Computer Science, FOCS 2024, Chicago, IL, USA, October 27-30, 2024*, pages 264–290. IEEE, 2024.

[Waj20] David Wajc. Rounding dynamic matchings against an adaptive adversary. In *Proceedings of the 52nd Annual ACM SIGACT Symposium on Theory of Computing (STOC)*, pages 194–207, 2020.

[WS11] David P. Williamson and David B. Shmoys. *The Design of Approximation Algorithms*. Cambridge University Press, 2011.