Fully Dynamic Set Cover via Hypergraph Maximal Matching: An Optimal Approximation Through a Local Approach

Sepehr Assadi ✉
Rutgers University, New Brunswick, NJ, USA

Shay Solomon ✉
Tel Aviv University, Israel

Abstract

In the (fully) dynamic set cover problem, we have a collection of m sets from a universe of size n that undergo element insertions and deletions; the goal is to maintain an approximate set cover of the universe after each update. We give an $O(f^2)$ update time algorithm for this problem that achieves an f-approximation, where f is the maximum number of sets that an element belongs to; under the unique games conjecture, this approximation is best possible for any fixed f. This is the first algorithm for dynamic set cover with approximation ratio that *exactly* matches f (as opposed to *almost* f in prior work), as well as the first one with runtime *independent of* n, m (for any approximation factor of $o(f^3)$).

Prior to our work, the state-of-the-art algorithms for this problem were $O(f^2)$ update time algorithms of Gupta et al. [STOC'17] and Bhattacharya et al. [IPCO'17] with $O(f^3)$ approximation, and the recent algorithm of Bhattacharya et al. [FOCS'19] with $O(f \cdot \log n/\varepsilon^2)$ update time and $(1+\varepsilon) \cdot f$ approximation, improving the $O(f^2 \cdot \log n/\varepsilon^5)$ bound of Abboud et al. [STOC'19].

The key technical ingredient of our work is an algorithm for maintaining a *maximal* matching in a dynamic hypergraph of rank r – where each hyperedge has at most r vertices – that undergoes hyperedge insertions and deletions in $O(r^2)$ amortized update time; our algorithm is randomized, and the bound on the update time holds in expectation and with high probability. This result generalizes the maximal matching algorithm of Solomon [FOCS'16] with constant update time in ordinary graphs to hypergraphs, and is of independent merit; the previous state-of-the-art algorithms for set cover do not translate to (integral) matchings for hypergraphs, let alone a maximal one. Our quantitative result for the set cover problem is translated directly from this qualitative result for maximal matching using standard reductions.

An important advantage of our approach over the previous ones for approximation $(1+\varepsilon) \cdot f$ (by Abboud et al. [STOC'19] and Bhattacharya et al. [FOCS'19]) is that it is inherently *local* and can thus be distributed efficiently to achieve low amortized round and message complexities.

2012 ACM Subject Classification Theory of computation → Dynamic graph algorithms; Theory of computation → Graph algorithms analysis; Mathematics of computing → Graph algorithms; Mathematics of computing → Approximation algorithms

Keywords and phrases dynamic graph algorithms, hypergraph, maximal matching, matching, set cover

Digital Object Identifier 10.4230/LIPIcs.ESA.2021.8

Related Version *Full Version*: `https://arxiv.org/pdf/2105.06889.pdf`

Funding *Sepehr Assadi*: Research supported in part by a NSF CAREER Grant CCF-2047061, and a gift from Google Research.
Shay Solomon: Partially supported by the Israel Science Foundation grant No.1991/19.

1 Introduction

In the set cover problem, we are given a family $\mathcal{S} = (S_1, \ldots, S_m)$ of m sets on a universe $[n]$. The goal is to find a minimum-size subfamily of sets $\mathcal{F} \subseteq \mathcal{S}$ such that $\mathcal{F}$ covers all the elements of $[n]$. Throughout the paper, we use f to denote the maximum frequency of any element $i \in [n]$ inside the sets in $\mathcal{S}$ (by frequency of element i, we mean the number of sets in $\mathcal{S}$ that contain i).

The set cover problem is one of the most fundamental and well-studied NP-hard problems, with two classic approximation algorithms, both with runtime $O(fn)$: greedy $\ln n$-approximation and primal-dual f-approximation. One cannot achieve approximation $(1-\varepsilon)\ln n$ unless P = NP [26, 17] or approximation $f - \varepsilon$ for any fixed f under the unique games conjecture [20].

In recent years there is a growing body of work on this problem in the dynamic setting, where one would like to efficiently maintain a set cover for a universe that is subject to element insertions and deletions. The holy grail is to coincide with the bounds of the static setting: approximation factor of either $\ln n$ or f with (amortized) update time $O(f)$ (as a static runtime of $\Theta(fn)$ means that we spend $\Theta(f)$ time per each element of the universe on average). Indeed, in any dynamic model where element updates are explicit, update time $\Omega(f)$ is inevitable. Even in stronger models where updates are supported implicitly in constant time, recent SETH-based conditional lower bounds imply that update time $O(f^{1-\varepsilon})$ requires polynomial approximation factor [1].

The dynamic set cover problem was first studied by Gupta et al. [18] who gave an $O(\log n)$-approximation with update time $O(f \log n)$ based on a greedy algorithm. The rest of the known algorithms are primal-dual-based and their approximation factor depends only on f. The state-of-the-art algorithms are: (1) An $O(f^3)$-approximation with $O(f^2)$ update time, by Gupta et al. [18] and independently by Bhattacharya et al. [9], and (2) A $(1+\varepsilon)\cdot f$-approximation with update time $O(f \cdot \log n/\varepsilon^2)$ by Bhattacharya et al. [11], improving the $O(f^2 \cdot \log n/\varepsilon^5)$ bound of Abboud et al. [1] as well as an earlier result by Bhattacharya et al. [10]. Interestingly, the state-of-the-art algorithms for this problem are all *deterministic* (the algorithm of [1] is randomized however).

Our Result

In this work we demonstrate the power of randomization for the dynamic set cover problem by achieving the best possible approximation of f with runtime independent of both m, n:

▶ **Theorem (Informal).** *There is an algorithm for the dynamic set cover problem that achieves an* exact *f-approximation in $O(f^2)$ expected (and with high probability) amortized update time.*

This gives the first algorithm for dynamic set cover with approximation ratio that *exactly* matches f (as opposed to *almost* f in prior work), as well as the first one with runtime *independent of* n, m (for any approximation factor of $o(f^3)$). The bound $O(f^2)$ on the update time of our algorithm holds with high probability and in expectation. As in [1] and in the great majority of randomized dynamic graph algorithms, we assume an oblivious adversary. We shall remark that even for the much simpler problem of dynamic vertex cover (corresponding to $f = 2$ case), no algorithm is known against an adaptive adversary that achieves an exact 2-approximation in time independent of other input parameters (although $(2+\varepsilon)$-approximation algorithms have been known for some time now [13, 9, 6, 8]).

Maximal Hypergraph Matching

The key technical ingredient of our work is an algorithm for maintaining a *maximal* matching in a dynamic hypergraph of rank r – where each hyperedge has at most r vertices – that undergoes hyperedge insertions and deletions in $O(r^2)$ amortized update time. This result generalizes the maximal matching algorithm of Solomon [25] with constant update time for ordinary graphs, and is of independent merit; the previous state-of-the-art algorithms for set cover do not translate to (integral) matchings for hypergraphs, let alone a maximal one (however the algorithm of [1] translates to an integral (non-maximal) matching). The result for set cover follows immediately from this: taking all the matched vertices in a maximal hypergraph matching of rank f yields an f-approximate hypergraph vertex cover, or equivalently, an f-approximate set cover; (see Section 2 for details of this standard reduction).

We stress that there is an inherent difference between a maximal matching (yielding exactly f-approximation) and an almost-maximal matching (yielding $(1+\varepsilon)\cdot f$-approximation), wherein an ε-fraction of the potentially matched vertices may be unmatched; this is true in general but particularly important in the dynamic setting. Despite extensive work on dynamic algorithms for matching and vertex cover in ordinary graphs ($f = 2$), the state-of-the-art deterministic algorithm (or even randomized against adaptive adversary) for 2-approximate matching and vertex cover (via a maximal matching) has update time $O(\sqrt{|E|})$ [21], while $(2+\varepsilon)$-approximate matching and vertex cover can be maintained deterministically in poly-log update time [6, 7]; for approximate vertex cover and *fractional* matching, the update time can be further reduced to $O(1/\varepsilon^2)$ [13]. Therefore, it is only natural that our algorithm, which achieves an integral maximal hypergraph matching, is randomized and assumes an oblivious adversary.

Distributed networks

There is a growing body of work on distributed networks that change dynamically (cf. [22, 24, 15, 3, 19, 14, 2]). A distributed network can be modeled as an undirected (hyper)graph $G(V, E)$, where each vertex $v \in V$ is identified with a unique processor and the edge set E corresponds to direct communication links between the processors. In a static distributed setting all processors wake up simultaneously, and computation proceeds in fault-free synchronous rounds during which every processor exchanges messages of size $O(\log n)$ with its neighbors. We consider the standard $\mathcal{CONGEST}$ model (cf. [23]), which captures the essence of spatial locality and congestion. For hypergraphs, the messages should be of size $O(\log m)$, where m is the number of edges in the graph.

We focus on the standard setting in dynamic graph algorithms – dynamically changing networks that undergo edge updates (both insertions and deletions, a single edge update per step), which initially contain no edges. But in a *distributed* network we are subject to the following *local* constraint: After each edge update, only the affected vertices – the endpoints of the updated edge – are woken up; this is referred to in previous work as the ($\mathcal{CONGEST}$) *local wakeup model*. Those affected vertices initiate an update procedure, which also involves waking up the vertices in the network (beyond the affected ones) that are required to participate in the update procedure, to adjust all outputs to agree on a valid global solution – in our case a maximal hypergraph matching; the output of each vertex is the set of its incident edges that belong to the matching. We make a standard assumption that the edge updates occur in large enough time gaps, and hence the network is always "stable" before the next change occurs (see, e.g., [22, 15, 3]). In this setting, the goal is to optimize (1) the number of communication rounds, and (2) the number of messages, needed for repairing the solution per edge update, over a worst-case sequence of edge updates.

Our dynamic maximal hypergraph matching algorithm can be naturally adapted to *distributed networks*. Note that following an edge update, $O(1)$ communication rounds trivially suffice for updating a maximal hypergraph matching. However, the number of messages sent per update via this naive algorithm may be a factor of r^2 greater than the maximum degree in the hypergraph, which could be $\Omega(\binom{n}{r-1})$, where n is the number of vertices and r is the rank. An important objective is to design a dynamic distributed algorithm that achieves, in addition to low round complexity, a low *message complexity*. The inclusion-maximality of our maintained matching enforces our algorithm to work persistently so there is never any "slack"; that is, the algorithm makes sure that every edge that can be added to the matching is added to it, which stands in contrast to "lazy" approximate-maximal matching (or approximate set cover) algorithms, which may wait to accumulate an ε-factor additive slack in size or weight, and only then run an update procedure. However, such a lazy update procedure is inherently non-local, where, following an edge update e, the required changes to the maintained graph structure may involve edges and vertices that are arbitrarily far from e; indeed, this is the case with the previous algorithms that achieve approximation factor close to f [11, 1]; moreover, the "lazy" feature of these algorithms must rely on a centralized agent that orchestrates the update algorithm with the use of global data structures, and this is, in fact, the key behind the efficiency of these algorithms [11, 1]. Our algorithm, on the other hand, does not employ any global update procedure or global data structures, and as such it is inherently *local* and can be easily distributed, so that the average number of messages sent per update is $O(r^2)$, matching the sequential update time. The number of rounds is clearly upper bounded by the number of messages. Refer to Section 9 in the full version [4] for more details.

Recent related work

Independently and concurrently to our work, Bhattacharya, Henzinger, Nanongkai, and Wu [arXiv'20, SODA'21] obtained an algorithm for dynamic set cover with $O(f^2/\varepsilon^3)$ amortized update time and $O(f \cdot \log^2 n/\varepsilon^3)$ worst case update time and $(1+\varepsilon) \cdot f$ approximation. While closely related, their results and ours are incomparable. Our algorithm can achieve a better approximation ratio of *exactly* f as opposed to $(1+\varepsilon)f$ (which is the first dynamic algorithm with this guarantee) but is randomized, while their result is deterministic and can work for weighted set cover (with extra $\log C$-dependence on the update time where C is the maximum weight of any set). Also, as mentioned, our algorithm is inherently local and can be distributed efficiently, whereas the algorithm of Bhattacharya et al., as the previous aforementioned algorithms, is non-local. In terms of techniques, the two works are disjoint.

1.1 Technical overview

At a high level, all the previous state-of-the-art algorithms [18, 9, 11] (as well as the independent work of [12]) follow a deterministic primal-dual approach by maintaining a fractional packing solution as a dual certificate. The main advantage of this approach over ours, of course, is in being deterministic, and its drawbacks are that (1) the approximation factor is almost f rather than exactly f, (2) it does not give rise to an *integral* matching in the context of hypergraphs, and (3) it is non-local.

Our algorithm generalizes and strengthens the maximal matching algorithm by Solomon [25], which, in turn, builds on and refines the pioneering approach of Baswana et al. [5]. For conciseness, we shall sometimes refer only to [25] in the following (to avoid explaining the differences between [25] and [5]); of course, by that we do not mean to take any credit of [5], on which [25] relies.

Maximal matching algorithm of [5, 25]. Consider a deletion of a matched edge (u, v) from the graph (for $r = 2$), which is the only nontrivial update. Focusing on u, if u has an unmatched neighbor, we need to match u. To avoid a naive scan of the neighbors of u (requiring $O(n)$ time), the key idea is to match u with a randomly sampled (possibly matched) neighbor w. Under the oblivious adversarial model, the expected number of edges incident on u deleted from the graph before the deletion of edge (u, w) is roughly half the "sample space" size of u, i.e., the number of neighbors of u from which we sampled w, which can be viewed as "time token" (or potential value) that is expected to arrive in the future.

To benefit from these tokens, [5, 25] introduces a *leveling scheme* where the *levels* of vertices are exponentially smaller estimates of these potential values: Unmatched vertices have level -1 and matched vertices are assigned the levels of their matched edge, which is roughly the logarithm of the sample space size. This defines a dynamic hierarchical partition of *vertices* into $O(\log n)$ levels.

A key ingredient in the algorithm of [5, 25] is the *sampling rule*: A random mate w is chosen for a vertex u among its neighbors of strictly lower level. Intuitively, if w's level is lower than u's, then w's potential value is much smaller than u's, and the newly created matched edge (u, w) provides enough potential value to cover the cost of deleting the old matched edge on w (if any).

Difficulties of going from ordinary graphs to hypergraphs. The main difficulty of extending the prior work in [5, 25] to hypergraphs of rank $r > 2$ has to do with the "vertex-centric" approach taken by these works [5, 25]. For instance, even at the definition level, it is already unclear how to generalize the sampling rule of the algorithm for hypergraphs, even for $r = 3$, since the hyperedges incident on u may consist of endpoints of various levels, some smaller than that of u, some higher. Ideally we would want to sample the matched hyperedge among those where *all* endpoints have lower level than that of u, but it is a-priori unclear how to maintain this hyperedge set efficiently. In particular, to perform the sampling rule efficiently, the strategy of [25] is to dynamically orient each edge towards the lower level endpoint, and a central obstacle is to efficiently cope with edges where both endpoints are at the same level. For hypergraphs, naturally, these obstacles become more intricate considering there are r different endpoints now.

There are also other hurdles that need to be carefully dealt with, such as the following. Say a matched hyperedge $e = (u_1, \ldots, u_r)$ gets deleted from the hypergraph. When $r = 2$, any edge incident on u_1 is different than any edge incident on u_2, hence informally the update algorithm can handle u_1 and u_2 *independently* of each other. When $r > 2$ (even for $r = 3$), different endpoints of e may share (many) hyperedges in common. Therefore, when choosing random matched hyperedges for the newly unmatched endpoints of e, we need to (i) be careful not to create conflicting matched hyperedges, but at the same time (ii) keep the sample spaces of endpoints sufficiently large so that the potential values can cover the runtime of the update procedure; balancing between these two contradictory requirements is a key challenge in our algorithm.

An $O(r^3)$ update time algorithm. We manage to cope with these and other hurdles by instead switching to a "hyperedge-centric" view of the algorithm. Informally speaking, this means that instead of letting vertices derive the potential values and levels, we assign these values to the hyperedges and use those to define the corresponding level for remaining vertices. Under this new view of the algorithm, we can indeed generalize the approach of [25] to obtain an $O(r^4)$-update time algorithm for rank r hypergraphs. Considering the intricacies in [25], already achieving this $O(r^4)$ time bound turned out to be considerably challenging.

Improving the update time to $O(r^3)$ is based on the following insight. In this hyperedge-centric view, a level-ℓ matched hyperedge e is sampled to the matching from a sample space $S(e)$ of size roughly α^ℓ, where $\alpha = \Theta(r)$. We refer to the hyperedges in $S(e)$ as the *core hyperedges* of e. All the core hyperedges of e are then also assigned a level ℓ. Let us focus on an $e' = (u_1, \ldots, u_r) \in S(e)$. Subsequently, u_1 may initiate the creation of a new matched hyperedge at level $\ell' > \ell$, at which stage we randomly sample a new matched hyperedge, denoted by e_1, among all its incident hyperedges of level lower than ℓ', so $e' \in S(e_1)$, i.e., hyperedge e' is now a core hyperedge of e_1 as well. Perhaps later u_2 may initiate the creation of a new matched hyperedge e_2 at level $\ell'' > \ell'$, and then hyperedge e' will become a core hyperedge of e_2, and so on and so forth. Thus any hyperedge may serve as a core hyperedge of up to r matched hyperedges at any point in time.

To shave a factor of r from the runtime, we need to make sure that each hyperedge serves as a core hyperedge of a *single* matched hyperedge. This is achieved by "freezing" (or *temporarily* deleting) all core hyperedges of a newly created matched hyperedge e; in the sequel (see Section 3.2) we shall refer to these hyperedges as *temporarily deleted hyperedges* (due to e) rather than "core hyperedges", and they will comprise the set $\mathcal{D}(e)$. Then, whenever the matched hyperedge gets deleted from the matching, we need to "unfreeze" these core hyperedges and update all their *ignored* data structures – our analysis shows that this can be carried out efficiently.

Since this algorithm already requires an entirely new view of the previous approaches for ordinary graphs and several nontrivial ideas, we present it as a standalone result in Section 3.

An $O(r^2)$ update time algorithm. The main step is to improve the update time from $O(r^3)$ to $O(r^2)$. We next provide a couple of technical highlights behind this improvement.

In our algorithm, the potential values of level-ℓ matched hyperedges are in the range $[\alpha^\ell, \alpha^{\ell+1})$, for $\alpha = O(r)$, hence they may differ by a factor of $O(r)$. Consider the moment a level-ℓ matched hyperedge $e = (u_1, \ldots, u_r)$ gets deleted by the adversary; the leveling scheme allows us to assume we have an $O(\alpha^{\ell+2})$ "potential time" for handling this hyperedge. To get an update time of $O(r^2)$, we need to handle each of the endpoints $u_i \in e$ within time $O(\alpha^{\ell+1})$ time or instead "contribute" to the potential by creating a new matched hyperedge. If u_i has more than $\alpha^{\ell+1}$ incident hyperedges of level at most ℓ, we can sample a random hyperedge among them to be added to the matching, thereby creating a level-$(\ell+1)$ matched hyperedge; we discuss some issues related to the creation of matched edges later. But if u_i has slightly less than $\alpha^{\ell+1}$ incident edges, this sample space size suffices only for creating a level-ℓ matched edge, but it is crucial that the sample space of a level-ℓ matched edge would consist only of edges where all endpoints have level strictly lower than ℓ. Even checking whether this is the case is too costly, since iterating over all endpoints of all such edges takes time (slightly less than) $O(\alpha^{\ell+2})$, and if we are in the same scenario for each u_i this gives rise to a runtime of $O(\alpha^{\ell+3})$, and thus to an amortized update time of $O(r^3)$. Even if we could check this for free, if most of the edges have endpoints of level ℓ, we need to find those endpoints, and to pass the "ownership" of the edges to those endpoints. To cope with these issues, we maintain a data structure for each hyperedge e that keeps track of all its endpoints of highest level and in $O(1)$ time returns an arbitrary such endpoint or reports that none exists. Of course, now the challenge becomes maintaining these data structures for the edges with $O(r^2)$ update time.

Consider the moment that a level-ℓ matched edge e is created. At this stage, another crucial invariant tells us to "raise" all endpoints of edge e to level ℓ, and then to update the ownership set of each endpoint according to its up-to-date level. One challenging case is when

each endpoint u_i of e now owns slightly less than $\alpha^{\ell+1}$ new edges. This sample space size does not suffice for creating a level-$(\ell+1)$ matched edge yet it is too costly to update each of the endpoints of these edges about the up-to-date level of u_i; summing over all endpoints of e, this again gives rise to update time of $O(r^3)$. However, if we are equipped with the aforementioned data structure, we can efficiently focus on the edges where all endpoints have level lower than ℓ, and can thus create a level-ℓ matched edge. This is not enough, however, since we are merely replacing one level-ℓ matched edge by another, and this process could repeat over and over. Our goal would be to replace one level-ℓ matched edge by *at least two others*, so as to provide enough increase in "potential" to cover for this runtime. Since the total number of edges incident on e in this case is slightly less than $\alpha^{\ell+2}$, the intuition is that we should be able to easily achieve here a fan-out of 2, and therefore a valid charging argument. Alas, there is one significant caveat when working with hypergraphs, which we already mentioned above – dependencies. It is possible that the first level-ℓ matched edge that we randomly sampled for u_1 intersects all the edges incident on e, which will result in fan-out 1. To overcome this final obstacle, we take our hyperedge-centric view to the next level by employing a new sampling method different than [5, 25] altogether. In particular, in this case, our sample space is not necessarily restricted to hyperedges incident on a single vertex, but could be an arbitrary hyperedge set as a function of the deleted hyperedge; however, importantly, to achieve a *local* sampling method, we will make sure that the entire sample space is incident to the endpoints of a single edge. This new sampling method (see Procedure insert-hyperedge in Section 5 of the full version [4]) entails a few technical complications primarily to ensure that we still get enough "potential" from the adversary, but it ultimately enables us to achieve the desired update time bound of $O(r^2)$.

The role of randomness in our algorithm. Our algorithm relies crucially on randomization and on the oblivious adversary assumption, and the probabilistic analysis employed in this work is highly nontrivial; in particular, the usage of randomization for reducing the update time bound from $O(r^3)$ to $O(r^2)$ relies on several new insights. We note that if the entire update sequence is known in advance and is stored in a data structure that allows for fast access, which is sometimes referred to as the "(dynamic) offline setting" (cf. [16]) – then a straightforward variant of our algorithm works *deterministically* with $O(r^2)$ amortized update time. Specifically, whenever a matched edge is randomly sampled by our algorithm (which is always done uniformly) from a carefully chosen sample space of edges – in the offline setting, instead of randomly sampling the matched edge, one can choose the matched edge *deterministically* to be the one that will be deleted last among all edges in the sample space. It is not difficult to verify that this simple tweak translates our probabilistic $O(r^2)$ amortized update time bound into a deterministic $O(r^2)$ time bound, while avoiding the entire probabilistic analysis. The probabilistic ingredients of the analysis are omitted due to space constraints; they appear in Sections 7 and 8 of the full version [4].

2 Preliminaries and Organization

Hypergraph Notation. For a hypergraph $G = (V, E)$, V denotes the set of vertices and E denotes the set of hyperedges. We use $v \in e$ to mean that v is one of the vertices incident on hyperedge e. Rank r of a hypergraph is the maximum number of vertices incident on any edge, i.e., $r := \max\{|e| \mid e \in E\}$. A matching M in G is a collection of vertex-disjoint hyperedges of M. A matching M is called *maximal* if no other edge of G can be added to M without violating its matching constraint. A vertex cover U in G is a collection of vertices so that every hyperedge in E has at least one endpoint in U. We use the next standard fact.

▶ **Fact 1.** *Let G be any hypergraph with rank r. Suppose M is a maximal matching in G and U denotes all vertices incident on M. Then U is an r-approximate vertex cover of G.*

Hypergraph Formulation of Set Cover. Consider a family $\mathcal{S} = \{S_1, \ldots, S_m\}$ of sets over a universe $[n]$. We can represent $\mathcal{S}$ by a hypergraph G on m vertices corresponding to sets of $\mathcal{S}$ and n hyperedges corresponding to elements of $[n]$: Any element $i \in [n]$ is now a hyperedge between vertices corresponding to sets in $\mathcal{S}$ that contain i. It is easy to see that there is a one-to-one correspondence between set covers of $\mathcal{S}$ and vertex covers of G and that the rank r of G is the same as the maximum frequency parameter f in the set cover instance. With this transformation, by Fact 1, obtaining an f-approximation to this instance of set cover reduces to obtaining a maximal matching of G. This is the direction we take in this paper for designing fully dynamic algorithms for set cover by designing a fully dynamic algorithm for hypergraph maximal matching.

Organization. Due to space constraints, in this extended abstract we focus on the $O(r^3)$ update time algorithm and its analysis, where we provide only part of the analysis. The $O(r^2)$ update time algorithm and its analysis are inherently more involved and are entirely omitted. All the missing details appear in the full version [4].

3 An $O(r^3)$-Update Time Algorithm

Throughout, we use M to denote the maximal matching of the underlying hypergraph $G = (V, E)$ maintained by the algorithm. We use the following parameters in our algorithm:

$$\alpha := (4 \cdot r), \qquad L := \lceil \log_\alpha |N| \rceil \tag{1}$$

where N approximates the dynamic number of edges $|E|$ plus the fixed number of vertices $|V|$ from above, so that $\log_\alpha |N| = \Theta(\log_\alpha(|V| + |E|))$. Every $\Omega(N)$ steps we update the value of N, and as a result rebuild all the data structures; this adds a runtime of $O(|V| + |E|) = O(N)$ every $\Omega(N)$ update steps, hence a negligible overhead to the amortized cost of the algorithm. We may henceforth ignore this technical subtlety and treat N as a fixed value in what follows.

3.1 A Leveling Scheme and Hyperedge Ownerships

We use a leveling scheme for the input hypergraph that partitions hyperedges and vertices. This is done by assigning a *level* $\ell(e)$ to each hyperedge $e \in E$ and a *level* $\ell(v)$ to each vertex $v \in V$.

▶ **Invariant 2.** *Our leveling scheme satisfies the following properties:*

1. *For any hyperedge $e \in E$, $0 \leq \ell(e) \leq L$ and for any vertex $v \in V$, $-1 \leq \ell(v) \leq L$; moreover, $\ell(v) = -1$ iff v is **unmatched** by M.*
2. *For any **matched** hyperedge $e \in M$ and any incident vertex $v \in e$, $\ell(v) = \ell(e)$.*
3. *For any **unmatched** hyperedge $e \notin M$, $\ell(e) = \max_{v \in e} \ell(v)$.*

Our leveling scheme needs only to specify $\ell(e)$ for each $e \in M$; the rest are fixed deterministically by Invariant 2. Moreover, this invariant ensures that the matching M obtained by the algorithm is maximal as all unmatched vertices are at level -1 while the level of any hyperedge is at least 0 and at the same time equal to the maximum level of any of its incident vertices. Based on the leveling scheme, we assign each hyperedge e to exactly one of its incident vertices $v \in e$ with $\ell(v) = \ell(e)$ to *own* (the ties between multiple vertices at the same level are broken by the algorithm). We use $\mathcal{O}(v)$ to denote the set of edges owned by v. This definition, combined with Invariant 2, implies the following invariant.

▶ **Invariant 3.** (*i*) *For any vertex $v \in V$, any owned hyperedge $e \in \mathcal{O}(v)$, and any other incident vertex $u \in e$, $\ell(u) \leq \ell(v)$.* (*ii*) *For any vertex $v \in V$, any incident hyperedge $e \in v$ has $\ell(e) \geq \ell(v)$.*

3.2 Temporarily Deleted Hyperedges

To obtain the desired update time of $O(r^3)$, we would need to allow some hyperedges of the hypergraph to be considered *temporarily deleted* and no longer participate in any of the other data structures; moreover, whenever we no longer consider them deleted, we simply treat them as a hyperedge insertion to our hypergraph and handle them similarly (which will take $O(r)$ time per each hyperedge exactly as in the previous algorithm). The role of these deleted hyperedges becomes apparent only in the probabilistic analysis that is omitted due to space constraints; see Section 7 in the full version [4] for this analysis. However, it is the goal of the algorithm itself (rather than the analysis) to cope efficiently with the required deletions. We shall note that these deletions constitute one of several differences of our algorithm with that of [25]. The next invariant allows us to maintain the maximality of our matching.

▶ **Invariant 4.** *Any temporarily deleted hyperedge is incident on some* ***matched*** *hyperedge.*

To maintain Invariant 4, each matched hyperedge $e \in M$ is *responsible* for a set of deleted edges denoted by $\mathcal{D}(e)$ and stored in a linked-list data structure. This set will be *finalized* at the time e joins the matching M and will remain unchanged throughout the algorithm until e is removed from M; at that point, we simply bring back all hyperedges in $\mathcal{D}(e)$ to the graph as new hyperedge insertions. Invariant 4 is crucial for the correctness of our algorithm.

We note that besides this data structure $\mathcal{D}(e)$, these deleted hyperedges *do not appear* in any other data structure of the algorithm and do not (necessarily) satisfy any of the invariants in the algorithm – they are simply treated as if they do not belong to the hypergraph. Invariant 4 ensures that even though we are ignoring temporarily deleted hyperedges in the algorithm, the resulting maximal matching on the hypergraph of undeleted hyperedges is still a maximal matching for the entire hypergraph. As such, throughout the rest of the paper, with a slight abuse of notation, whenever we talk about hyperedges of G, we refer to the hyperedges that are not temporarily deleted (unless explicitly stated otherwise).

3.3 Data Structures

We maintain the following information for each vertex $v \in V$ (again, to emphasize, we ignore the temporarily deleted hyperedges in all the following data structures):

- $\ell(v)$: the level of v in the leveling scheme;
- $M(v)$: the hyperedge in M incident on v (if v is unmatched $M(v) = \perp$);
- $\mathcal{O}(v)$: the set of hyperedges e owned by v – we define $o_v := |\mathcal{O}(v)|$;
- $\mathcal{N}(v)$: the set of hyperedges e incident on v;
- $\mathcal{A}(v, \ell)$ for any integer $\ell \geq \ell(v)$: the set of hyperedges $e \in \mathcal{N}(v)$ that are *not* owned by v and have level $\ell(e) = \ell$ – we define $a_{v,\ell} := |\mathcal{A}(v, \ell)|$.

We also maintain the following information for each hyperedge $e \in E$:

- $\ell(e)$: the level of e in the leveling scheme;
- $O(e)$: the single vertex $v \in e$ that owns e, i.e., $e \in \mathcal{O}(v)$;
- $M(e)$: a Boolean variable to indicate whether or not e is matched.

We also maintain back-and-forth pointers between these different data structures that refer to the same hyperedge or vertex in a straightforward way.

Next, we introduce the main procedures used for updating these data structures. In these procedures, if there is room for confusion, we use superscript $*^{\mathsf{old}}$ to denote a parameter or data structure $*$ before the update and $*^{\mathsf{new}}$ to denote the value of $*$ after the update.

Procedure `set-owner`(e, v). Given a hyperedge e and vertex $v \in e$ where $\ell(v) = \max_{u \in e} \ell(u)$, sets owner of e to be v, i.e., $O(e) = v$.

To implement `set-owner`(e, v), we first set $O^{\mathsf{new}}(e) = v$ and $\ell^{\mathsf{new}}(e) = \ell(v)$, add e to $\mathcal{O}(v)$, and remove e from $\mathcal{O}(O^{\mathsf{old}}(e))$. If $\ell^{\mathsf{new}}(e) = \ell^{\mathsf{old}}(e)$, there is nothing else to do. Otherwise, for any $w \neq v \in e$, we remove e from $\mathcal{A}(w, \ell^{\mathsf{old}}(e))$ and instead insert e in $\mathcal{A}(w, \ell^{\mathsf{new}}(e))$.

▷ Claim 5 (straightforward proof). `set-owner`(e, v) takes $O(r)$ time.

Procedure `set-level`(v, ℓ). Given a vertex $v \in V$ and integer $\ell \in \{-1, 0, \ldots, L\}$, updates the level of v to ℓ, i.e., sets $\ell(v) = \ell$.

The implementation of `set-level`(v, ℓ) is as follows. First, we determine the ownership of all hyperedges $e \in \mathcal{O}^{\mathsf{old}}(v)$ that were previously owned by v. We go over each hyperedge $e \in \mathcal{O}^{\mathsf{old}}(v)$ one by one, find $u := \arg\max_{w \in e} \ell(w)$ (where we use $\ell^{\mathsf{new}}(v) = \ell$ for the computations here), and use the procedure `set-owner`(e, u) to update the owner of e to u.

If $\ell^{\mathsf{old}}(v) > \ell$, nothing is left to do as no new hyperedge needs to be owned by v now that level of v has decreased (Invariant 3). If $\ell^{\mathsf{old}}(v) < \ell$, we should make v the new owner of all hyperedges $e \in \mathcal{N}(v)$ with level between $\ell^{\mathsf{old}}(v)$ to $\ell - 1$. This is done by traversing the lists maintained in $\mathcal{A}(v, \ell^{\mathsf{old}}(v)), \cdots, \mathcal{A}(v, \ell - 1)$, and running `set-owner`$(e, v)$ for each edge e in these lists.

Remark. It will be the responsibility of the procedure calling `set-level` to make sure that the invariants (and particularly Invariant 2) continue to hold.

▷ Claim 6. `set-level`(v, ℓ) takes $O\left(r \cdot (o_v^{\mathsf{old}} + o_v^{\mathsf{new}}) + \ell + 1\right)$ time.

Proof. The algorithm iterates over all hyperedges in $\mathcal{O}^{\mathsf{old}}(v)$ and use `set-owner` that takes $O(r)$ time by Claim 5 for each one. When $\ell^{\mathsf{old}}(v) > \ell$, this is all that is done by the algorithm and the bound on runtime follows. Otherwise, the algorithm also needs to iterate over the lists $\mathcal{A}(v, \ell^{\mathsf{old}}(v)), \cdots, \mathcal{A}(v, \ell - 1)$ one by one which takes $O(\ell)$ time and run `set-owner` for each of them that takes $O(r \cdot o_v^{\mathsf{new}})$ time in total. This gives the bound on runtime. ◁

3.4 The Update Algorithm

There are multiple cases to handle by the update algorithm depending on whether the updated hyperedge is an insertion or deletion, and whether or not it belongs to the maximal matching M. But the only interesting case is when a hyperedge in M is deleted and that is where we start with.

3.4.1 Case 1: a hyperedge $e \in M$ is deleted

There are two things we should take care of upon deletion of a hyperedge e from M; bringing back the temporarily deleted hyperedges in $\mathcal{D}(e)$ to maintain Invariant 4, and more importantly, updating the matching M to ensure its maximality. All the interesting part of the algorithm happen in the second step, but before we get to that, let us quickly mention how the first part is done.

Maintaining Invariant 4

Throughout the course of handling a single hyperedge update, we may need to delete multiple hyperedges $e_1, e_2, \ldots$ from M, starting from the originally deleted hyperedge by the adversary. Each of these hyperedge $e_i \in M$ that are now removed from M is responsible for temporarily deleted hyperedges $\mathcal{D}(e_i)$.

We will maintain a queue of all hyperedges in $\mathcal{D}(e_1), \mathcal{D}(e_2), \ldots$ during the course of this update. At the *very end* of the update, once all other changes are finalized, we will insert the hyperedges in this queue one by one to the hypergraph as if they were inserted by the adversary (using the procedure of Section 3.4.3). This allows us to maintain Invariant 4.

We note that throughout the update we may also temporarily delete some new hyperedges. We'll show that in that case, all these hyperedges are also incident on some matched hyperedge which is responsible for them and thus Invariant 4 holds for these hyperedges as well.

Maintaining maximality of M

Let us now begin the main part of the update algorithm. Suppose $e = (v_1, \ldots, v_r)$ is deleted from M. This makes the vertices $v_1, \ldots, v_r$ *temporarily* free. We will handle each of these vertices using the procedure `handle-free` which is the key ingredient of the update algorithm (we will simply run `handle-free`$(v_1), \ldots,$ `handle-free`(v_r)).

Procedure `handle-free`(v). Handles a given vertex $v \in V$ which is *unmatched* currently in the algorithm (its matched hyperedge may have been deleted via an update or by the algorithm in this time step, or v may simply be unmatched at this point of the algorithm)[a].

[a] To simplify the exposition, we may some time call `handle-free`(v) for a vertex v that is now matched again (while handling other vertices). In that case, this procedure simply aborts.

The execution of Procedure `handle-free`(v) depends on the number of owned hyperedges of v, i.e., o_v (both procedures used within this one are described below): (i) if $o_v < \alpha^{\ell(v)+1}$, we will run `deterministic-settle`(v); and otherwise (ii) if $o_v \geq \alpha^{\ell(v)+1}$, we run `random-settle`$(v)$ instead. We now describe each procedure.

Procedure `deterministic-settle`(v). Handles a given vertex $v \in V$ with $o_v < \alpha^{\ell(v)+1}$.

In `deterministic-settle`(v), we iterate over all hyperedges $e \in \mathcal{O}(v)$ owned by v and check whether all endpoints of e are now unmatched; if so, we add e to M, and run `set-level`$(v, 0)$ and `set-owner`(e, v). Moreover, for any vertex $u \neq v \in e$, we further run `set-level`$(u, 0)$ so all vertices incident on e are now at level 0. If no such hyperedge is found, we run `set-level`$(v, -1)$.

▷ Claim 7. `deterministic-settle`(v) takes $O(r \cdot o_v^{\mathsf{old}}) = O(r \cdot \alpha^{\ell^{\mathsf{old}}(v)+1})$ time and maintains Invariants 2 and 3 for vertex v and hyperedges incident on v.

Proof. Checking if there is any hyperedge that can be added to M takes $O(r \cdot o_v^{\mathsf{old}})$ time. Moreover, running `set-level`$(v, 0)$ or `set-level`$(v, -1)$ take $O(r \cdot o_v^{\mathsf{old}})$ time by Claim 6 as $o_v^{\mathsf{new}} \leq o_v^{\mathsf{old}}$ considering level of v has not increased. If the algorithm finds a hyperedge e to add to M, we run `set-level`$(u, 0)$ for $u \neq v \in e$ as well which takes $O(1)$ time for each u by Claim 6 as $\ell^{\mathsf{old}}(u) = -1$ (u was unmatched and by Invariant 2) and thus $o_u^{\mathsf{old}} = o_u^{\mathsf{new}} = 0$. As there are at most $r - 1$ choices for u, this step takes $O(r) = O(r \cdot \alpha^{\ell^{\mathsf{old}}(v)+1})$ time (we are not being tight here). We also need to run `set-owner`(e, v) in this case which takes another $O(r)$ time by Claim 5. Since `deterministic-settle`(v) is only called when $o_v^{\mathsf{old}} < \alpha^{\ell^{\mathsf{old}}(v)+1}$, the desired time bound follows.

As for maintaining Invariant 2, consider the hyperedge e chosen by the algorithm to be added to M. We set the level of all vertices incident on e to be 0 and making v the owner of e, hence satisfying Invariant 2. On the other hand, if we cannot find such a hyperedge e incident on v, we know that all hyperedges incident on v are already at level at least 0 by Invariant 2 (even after removing v) and hence setting $\ell^{\mathsf{new}}(v) = -1$ keeps Invariant 2. Invariant 3 also holds by maintaining Invariant 2 and the fact that we choose correct owners in the algorithm. ◁

Before defining `random-settle`, we need the following definition. For any vertex $v \in V$ and integer $\ell > \ell(v)$, we define:

- $\widetilde{o}_{v,\ell}$: the number of edges v will own *assuming* we increase its level from $\ell(v)$ to ℓ.

The following claim is a straightforward corollary of procedure `set-level`.

▷ Claim 8. For any $v \in V$ and $\ell > \ell(v)$, $\widetilde{o}_{v,\ell} = \left(\sum_{\ell'=\ell(v)}^{\ell-1} a_{v,\ell'}\right) + o_v$. In particular, $\widetilde{o}_{v,\ell}$ can be obtained from $\widetilde{o}_{v,\ell-1}$ in $O(1)$ time.

We can now describe `random-settle` (this is the main procedure of our update algorithm).

Procedure `random-settle`(v). Handles a given vertex $v \in V$ with $o_v \geq \alpha^{\ell(v)+1}$.

In `random-settle`(v), we first compute the level $\ell^{\mathsf{new}}(v)$ as *minimum* $\ell > \ell(v)$ with $\widetilde{o}_{v,\ell} < \alpha^{\ell+1}$ and run `set-level`$(v, \ell^{\mathsf{new}}(v))$. Then, we sample a hyperedge e uniformly at random from $\mathcal{O}^{\mathsf{new}}(v)$. The next step of the algorithm now depends on this particular hyperedge e.

- **Case (a): for all $u \in e$, $\widetilde{o}_{u,\ell^{\mathsf{new}}(v)} < \alpha^{\ell^{\mathsf{new}}(v)+1}$.** In this case, we add the hyperedge e to M and run `set-level`$(u, \ell^{\mathsf{new}}(v))$ for all $u \in e$ to maintain Invariant 2. This potentially can make M not a matching since some matched hyperedges $e_1, \ldots, e_k$ for $k < r$ incident on e might be in M. We will thus delete these hyperedges from M one by one and *recursively* run the procedure of Section 3.4.1 for each one, treating it *as if* this hyperedge was deleted by the adversary (although we do *not* remove the hyperedge from the hypergraph).
- **Case (b): there exists $u \in e$, s.t. $\widetilde{o}_{u,\ell^{\mathsf{new}}(v)} \geq \alpha^{\ell^{\mathsf{new}}(v)+1}$.** We run `deterministic-settle`$(v)$ to handle v and "switch" the focus to u instead. We then call `set-level`$(u, \ell^{\mathsf{new}}(v))$. If u is matched in M, say by hyperedge e_u, we remove e_u from M, and then *recursively* run the procedure of Section 3.4.1 for hyperedge e_u – we only note that when processing e_u, we start by running `handle-free`(u) first before all other vertices incident on e_u; this is only done for making the analysis more clear and is not needed. If u is *not* matched in M, we will simply run `handle-free`(u).

▷ Claim 9. The first step of `random-settle`(v) before either case (changing level of v and picking hyperedge e) takes $O(r \cdot \alpha^{\ell^{\mathsf{new}}(v)+1})$ time. Additionally, case (a) takes $O(r^2 \cdot \alpha^{\ell^{\mathsf{new}}(v)+1})$ time and case (b) takes $O(r \cdot o_u^{\mathsf{new}})$ time ignoring the recursive calls. Finally, `random-settle`(v) maintains Invariants 2 and 3.

Proof. Finding $\ell^{\mathsf{new}}(v)$ takes $O(\ell^{\mathsf{new}}(v))$ time by Claim 8 and `set-level`$(v, \ell^{\mathsf{new}}(v))$ takes $O(r \cdot o_v^{\mathsf{new}} + \ell^{\mathsf{new}}(v)) = O(r \cdot \alpha^{\ell^{\mathsf{new}}(v)+1})$ time by Claim 6 as level of v is increased (and so is number of its owned edges) and since $o_v^{\mathsf{new}} < \alpha^{\ell^{\mathsf{new}}(v)+1}$ by design. This proves the first part.

The second part for case (a) also follows because for any $u \in e$, `set-level`$(u, \ell^{\mathsf{new}}(v))$ takes $O(r \cdot o_u^{\mathsf{new}} + \ell^{\mathsf{new}}(u)) = O(r \cdot \widetilde{o}_{u,\ell^{\mathsf{new}}(v)} + \ell^{\mathsf{new}}(v))$ by Claim 6 as level of u is increased. This is $O(r \cdot \alpha^{\ell^{\mathsf{new}}(v)+1})$ by the condition $\widetilde{o}_{u,\ell^{\mathsf{new}}(v)} < \alpha^{\ell^{\mathsf{new}}(v)+1}$ in case (a). Multiplying this with at most r vertices $u \in e$ gives the desired bound.

The second part for case (b) holds because v has $o_v \leq \alpha^{\ell^{\mathsf{new}}(v)+1}$ by the definition of $\ell^{\mathsf{new}}(v)$ and thus `deterministic-settle`(v) takes $O(r \cdot \alpha^{\ell^{\mathsf{new}}(v)+1})$ time by Claim 7. Moreover, running `set-level`$(u, \ell^{\mathsf{new}}(v))$ takes $O(r \cdot o_u^{\mathsf{new}})$ by Claim 6 which is at least $O(r \cdot \alpha^{\ell^{\mathsf{new}}(v)+1})$ by the lower bound on $\widetilde{o}_{u,\ell^{\mathsf{new}}(v)}$ in this case.

Finally, Invariants 2 and 3 also hold as explained in the description of the procedure. ◁

The following observation plays a key rule in the recursive analysis of our algorithm.

▶ **Observation 10.** *In* `random-settle`(v)*: (i) the hyperedge e is sampled uniformly at random from at least $\alpha^{\ell^{\mathsf{new}}(v)}$ edges. Moreover, (ii) any hyperedge $e_1, \ldots, e_k$ or e_u deleted from M during this procedure is at level at most $\ell^{\mathsf{new}}(v) - 1$.*

Part (i) of the observation follows from the definition of $\ell^{\mathsf{new}}(v)$. For part (ii), note that any deleted edge e' in the process is incident on e with $\ell^{\mathsf{old}}(e) = \ell^{\mathsf{old}}(v) < \ell^{\mathsf{new}}(v)$. Let u be any vertex incident on both e' and e. Firstly, $\ell(u) \leq \ell^{\mathsf{old}}(v)$ as e was owned by v and not u, and secondly $\ell(e') = \ell(u)$ as e' is a matched hyperedge (see Invariant 2); thus $\ell(e') \leq \ell(e) < \ell^{\mathsf{new}}(v)$ as well.

Temporarily deleting new hyperedges

An astute reader may have noticed that we have not yet temporarily deleted any hyperedge in the update algorithm. The only place that this will be done is in case (a) of `random-settle`. In this case, when we decide to insert e to M, we will temporarily delete all other hyperedges in $\mathcal{O}^{\mathsf{new}}(v)$ from which e was sampled from, and make e responsible for them by adding them to $\mathcal{D}(e)$. The deletions of these hyperedges is done by the procedure of Section 3.4.2 (as they do not belong to M). As the cost of each such hyperedge deletion is $O(r)$ and their total number is $O(\alpha^{\ell^{\mathsf{new}}(v)+1})$, this does not change the asymptotic bounds of Claim 9. Finally, since these edges are incident on e which now belongs to M, Invariant 4 will be maintained by this process. The following observation is now immediate.

▶ **Observation 11.** *Any hyperedge $e \in M$ is responsible for $O(\alpha^{\ell(e)+1})$ hyperedges in $\mathcal{D}(e)$.*

3.4.2 Case 2: a hyperedge $e \notin M$ is deleted

We only need to remove e from the corresponding data structures which can be done in $O(r)$ time for the (at most) r endpoints of e.

3.4.3 Case 3: a hyperedge e is inserted

We need to find $v = \arg\max_{u \in e} \ell(u)$ and run `set-owner`(e, v) which takes $O(r)$ time. Moreover, if all vertices incident on e are unmatched, we should additionally add e to M and run `set-level`$(u, 0)$ for all $u \in e$ which takes $O(r)$ time per vertex by Claim 6.

This concludes the description of our algorithm for update time $O(r^3)$.

4 Part of the Analysis of the $O(r^3)$-Update Time Algorithm

A key definition in analysis is the notion of an *epoch* borrowed from the prior work in [5, 25].

▶ **Definition 12 (Epoch).** *For any time t and any hyperedge e in the matching M at time t, **epoch** of e and t, denoted by* $\mathsf{epoch}(e, t)$*, is the pair $(e, \{t'\})$ where $\{t'\}$ denotes the maximal continuous time period containing t during which e was always present in M (not even deleted temporarily during one step and inserted back at the same time step).*

We further define level of $\mathsf{epoch}(e,t)$ *as the level* $\ell(e)$ *of* e *during the time period of the epoch; we note that our update algorithm never changes level of a hyperedge in* M *without removing it from* M *first and thus level of an epoch is well-defined.*

The update algorithm in Section 3.4 takes $O(r)$ time deterministically for any update step that does *not* change the matching M. However, an update at a time step t that changes M may force the algorithm a large computation time and hence we use amortization to charge the cost of processing such an update at time t to the epochs that are created and removed at time t. In particular, for any time step t, define:

- $\mathsf{epochs}_{create}(t)$: the set of all $\mathsf{epoch}(e,t)$ that are *created* at time t; respectively, $\mathsf{epochs}_{term}(t)$ is defined for epochs *terminated* at time t;
- $C_{create}(\mathsf{epoch}(e,t))$: the computation cost at time t for *creation* of $\mathsf{epoch}(e,t)$; respectively, $C_{term}(\mathsf{epoch}(e,t))$ is defined for the cost of *termination* of $\mathsf{epoch}(e,t)$.

Then, the total cost of update at time t is:

$$C(t) = O(r) + \sum_{\substack{\mathsf{epoch}(e,t)\in \\ \mathsf{epochs}_{create}(t)}} C_{create}(\mathsf{epoch}(e,t)) + \sum_{\substack{\mathsf{epoch}(e,t)\in \\ \mathsf{epochs}_{term}(t)}} C_{term}(\mathsf{epoch}(e,t)). \tag{2}$$

▶ **Lemma 13.** *The total computation cost* $C(t)$ *of an update at time* t *in Equation* (2) *can be charged to the epochs in the RHS so that any level-*ℓ *epoch is charged with* $O(\alpha^{\ell+3})$ *units of cost.*

Proof. We prove this lemma in a sequence of claims. In the following, whenever we say "some computation cost can be charged to $C_{create}(\mathsf{epoch}(e,t))$ or $C_{term}(\mathsf{epoch}(e,t))$" we mean that the cost of computation is $O(\alpha^{\ell+3})$ and there is a level-ℓ epoch $\mathsf{epoch}(e,t)$ at time t that is either created or terminated, respectively, and that this cost is charged to $C_{create}(\mathsf{epoch}(e,t))$ or $C_{term}(\mathsf{epoch}(e,t))$. We emphasize that each epoch during this process is only charged a *constant* number of times. This then immediately satisfies the bounds in Lemma 13.

▷ Claim 14. Cost of a hyperedge-insertion update e at time t can be charged to $C_{create}(\mathsf{epoch}(e,t))$.

Proof. As in Section 3.4.3, this update takes $O(r)$ time and creates a level-0 $\mathsf{epoch}(e,t)$. ◁

From now on, we consider the main case of a hyperedge-deletion update e from M at time t using the procedure in Section 3.4.1. In this case, we remove e from M which results in $\mathsf{epoch}(e,t)$ to be terminated (and hence $\mathsf{epoch}(e,t) \in \mathsf{epochs}_{term}(t)$ in RHS of $C(t)$ in Equation (2)). This step is then followed by running `handle-free`(v) for all $v \in e$. Recall that `handle-free`(v) can be handled either by `deterministic-settle`(v) or `random-settle`(v) (depending on value of o_v).

▷ Claim 15. Let e be a hyperedge in M deleted at time t (either by the adversary or the algorithm in a recursive call). Cost of `handle-free`(v) for all $v \in e$ that are handled by `deterministic-settle`(v) can be charged to $C_{term}(\mathsf{epoch}(e,t))$.

Proof. By Claim 7, for each $v \in e$ that is handled by `deterministic-settle`(v), the runtime of `handle-free`(v) is $O(r \cdot \alpha^{\ell(e)+1})$. As there are at most r such vertices, their total cost is $O(r^2 \cdot \alpha^{\ell(e)+1}) = O(\alpha^{\ell(e)+3})$ by the choice of α in Equation (1). Hence the total cost of all these vertices that are charged to $C_{term}(\mathsf{epoch}(e,t))$ is $O(\alpha^{\ell(e)+3})$. ◁

We now switch to analyzing `random-settle`(v) for a fixed $v \in e$. The easier part, when we can process `random-settle`(v) by case (a), is handled via the next claim.

▷ Claim 16. Cost of case (a) of `random-settle`(v) at time t ignoring the recursive calls can be charged to $C_{create}(\mathsf{epoch}(e_v, t))$, where e_v is the hyperedge inserted to M in this case.

Proof. In case (a) of `random-settle`(v), we create a new edge e_v in M at level $\ell(e_v) = \ell^{\mathsf{new}}(v)$. By Claim 9, ignoring the recursive calls, this case takes $O(r^2 \cdot \alpha^{\ell(e_v)+1}) = O(\alpha^{\ell(e_v)+3})$ time. We can thus charge the cost of `random-settle`(v) to $C_{create}(\mathsf{epoch}(e_v, t))$. ◁

We now analyze case (b) of `random-settle`(v). In this case, instead of handling v, we in fact recursively handle the vertex u (with $\widetilde{o}_{u,\ell^{\mathsf{new}}(v)} \geq \alpha^{\ell^{\mathsf{new}}(v)+1}$) using `handle-free`$(u)$ and only after that will come back to take care of v if needed. Moreover, since we first set level of u to $\ell^{\mathsf{new}}(v)$, we will have $o_u^{\mathsf{new}} \geq \alpha^{\ell^{\mathsf{new}}(u)+1}$. This means `handle-free`$(u)$ will be handled using `random-settle`(u) recursively. It is again possible that `random-settle`(u) hits case (b) and so on and so forth. However, note that when going from v to u, we obtain that $\ell^{\mathsf{new}}(v) < \ell^{\mathsf{new}}(u)$ by Observation 10 (here $\ell^{\mathsf{new}}(u)$ refers to $\ell(u) = \ell^{\mathsf{new}}(v)$ at the beginning of `random-settle`(u)) and hence whenever case (b) happens, the vertex we move on to will have a strictly larger level. As such, this chain of recursive calls to `random-settle` will terminate eventually in some case (a) of `random-settle`.

We'll denote the vertices in the chain of calls to `random-settle` in case (b) for v by $(v =)w_0, w_1, w_2, \ldots, w_k$ for some $k \leq L$ (number of levels); in particular, w_1 is the vertex u in case (b) of `random-settle`(v), w_2 is the vertex u in case (b) of `random-settle`(w_1), etc.

The computation cost of `random-settle`(v) in case (b) then involves two parts: (i) pre-processing before calling each `random-settle`(w_i) (which is bounded in Claim 9), (ii) calling `random-settle`(w_i) itself, and (iii) removing hyperedges e_{w_i} from M (for each one that exist) and recursively handling them using the procedure of Section 3.4.1.

Among these costs, the cost of handling e_{w_i} for $i \in [k]$ is handled separately by recursion (charged either to the termination of $\mathsf{epoch}(e_{w_i}, t)$ or creation of new epochs). We thus need to handle the costs in parts (i) and (ii) in the following claim, whose proof is omitted.

▷ Claim 17. Cost of case (b) of `random-settle`(v) at time t with chain of vertices $w_0, w_1, \ldots, w_k$ ignoring the recursive calls (i.e., part (iii) of costs above) can be charged to $C_{create}(\mathsf{epoch}(e^*, t))$, where e^* is the hyperedge inserted to M in case (a) of `random-settle`(w_k). (Note that by definition, `random-settle`(w_k) finishes in case (a).)

Finally, we also have to handle the cost associated with maintaining Invariant 4, namely, "bringing back" temporarily deleted hyperedges at the very end of the update step (the cost of temporarily deleting new hyperedges is accounted for in Claim 9 already).

▷ Claim 18. Cost of inserting back the temporarily deleted hyperedges in $\mathcal{D}(e)$ for any hyperedge $e \in M$ can be charged to $C_{term}(\mathsf{epoch}(e, t))$.

Proof. Any hyperedge $e \in M$ is responsible for $O(\alpha^{\ell(e)+1})$ hyperedges in $\mathcal{D}(e)$ by Observation 11. As bringing back these hyperedges requires $O(r \cdot \alpha^{\ell(e)+1})$ time in total, this charge can be charged to the termination of $\mathsf{epoch}(e, t)$. ◁

We thus showed that any cost in $C(t)$ can be charged by a factor of $O(\alpha^{\ell+3})$ to some level-ℓ epoch in RHS of Equation (2) without charging any epoch more than a constant number of times, thus finalizing the proof of Lemma 13. ◀

In Lemma 13, we charge at most $O(\alpha^{\ell+3})$ to the creation and/or termination of a level-ℓ epoch. It can be shown that, without loss of generality, we may assume that every epoch created will also be terminated, hence we can re-distribute the charges to creation of each level-ℓ epoch to the termination of the same epoch. We now have an equivalent charging

scheme in which *only termination* of each level-ℓ epoch is charged with $O(\alpha^{\ell+3})$ computation time (and not the creations). Recall that an epoch is terminated when the corresponding hyperedge e is deleted from the maximal matching M. There are two types of hyperedge deletions from M in the algorithm: those that are the result of the adversary deleting a hyperedge from the graph, and those that are the result of the update algorithm to remove a matched hyperedge in favor of another (so the original hyperedge is still part of the hypergraph). Based on this, we differentiate between epochs as follows:

- **Natural epoch:** Ending with the adversary deleting the hyperedge e from G.
- **Induced epoch:** Ending with the update algorithm removing the hyperedge e from M.

With the following lemma, we are going to re-distribute the charge assigned to all induced epochs between the natural epochs. This then will allow us to focus solely on natural epochs.

▶ **Lemma 19** (proof omitted)**.** *The total cost charged to all induced epochs can be charged to natural epochs so that any level-ℓ natural epoch is charged with the costs of at most $r-1$ induced epochs at* lower levels*.*

Next, we move from natural and induced epochs to the corresponding notions in *levels*: A level ℓ is called an **induced level** (resp., **natural level**) if the number of induced level-ℓ epochs is greater than (resp., at most) the number of natural level-ℓ epochs. We will charge the computation costs incurred by any induced level to the computation costs at higher levels, so that in the end, the entire cost of the algorithm will be charged to natural levels. Specifically, in any induced level ℓ, we define a one-to-one mapping from the natural to the induced epochs. For each induced epoch, at most one natural epoch (at the same level) is mapped to it; any natural epoch that is mapped to an induced epoch is called *semi-natural*. For any induced level ℓ, all natural ℓ-level epochs are semi-natural by definition. For any natural level, all natural epochs terminated at that level remain as before; these epochs are called *fully-natural*. By Lemma 19, for any epoch, at most $r-1$ induced epochs at lower levels are charged to it. We define the **recursive cost** of an epoch as the sum of its actual cost and the recursive costs of the at most $r-1$ induced epochs charged to it as well as the (at most) $r-1$ semi-natural epochs mapped to them; the recursive cost of a level-0 epoch is defined as its actual cost. We are now ready to state the following lemma (proof omitted).

▶ **Lemma 20.** *For any $\ell \geq 0$, the recursive cost of any level-ℓ epoch is bounded by $O(\alpha^{\ell+3})$.*

The sum of recursive costs over all fully-natural epochs is equal to the sum of actual costs over all epochs (fully-natural, semi-natural and induced) that have been *terminated* throughout the update sequence, which also bounds the total runtime of the algorithm.

For the omitted proofs in the above statements, we refer the reader to Section 4 in the full version [4]. The final step of the analysis is to use the randomization in the algorithm (and obliviousness of the adversary) to prove a probabilistic upper bound on the amortized cost of the algorithm; this part is handled in detail in Section 7 of the full version [4].

This allows us to conclude the following theorem.

▶ **Theorem 21.** *For any integer $r > 1$, starting from an empty rank-r hypergraph on a fixed set of vertices, a maximal matching can be maintained over any sequence of t hyperedge insertions and deletions in $O(t \cdot r^3)$ time in expectation and with high probability.*

References

1 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*, pages 114–125, 2019.

2 Shiri Antaki, Quanquan C. Liu, and Shay Solomon. Dynamic distributed MIS with improved bounds. *CoRR*, abs/2010.16177, 2020. `arXiv:2010.16177`.

3 Sepehr Assadi, Krzysztof Onak, Baruch Schieber, and Shay Solomon. Fully dynamic maximal independent set with sublinear update time. In *Proceedings of the 50th Annual ACM SIGACT Symposium on Theory of Computing, STOC 2018, Los Angeles, CA, USA, June 25-29, 2018*, pages 815–826, 2018.

4 Sepehr Assadi and Shay Solomon. Fully dynamic set cover via hypergraph maximal matching: An optimal approximation through a local approach. *arXiv preprint arXiv:2105.06889*, 2021.

5 S. Baswana, M. Gupta, and S. Sen. Fully dynamic maximal matching in $O(\log n)$ update time. In *Proceedings of the 52nd IEEE Annual Symposium on Foundations of Computer Science, FOCS 2011, Palm Springs, CA, October 23-25, 2011*, pages 383–392, 2011 (see also *SICOMP'15* version, and subsequent erratum).

6 S. Bhattacharya, M. Henzinger, and G. F. Italiano. Deterministic fully dynamic data structures for vertex cover and matching. In *Proceedings of the 26th Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2015, San Diego, CA, USA, January 4-6, 2015*, pages 785–804, 2015.

7 S. Bhattacharya, M. Henzinger, and D. Nanongkai. New deterministic approximation algorithms for fully dynamic matching. In *STOC*, 2016.

8 S. Bhattacharya, M. Henzinger, and D. Nanongkai. Fully dynamic maximum matching and vertex cover in $O(\log^3 n)$ worst case update time. In *Proceedings of the 28th Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2017, Barcelona, Spain, January 16-19, 2017*, pages 470–489, 2017.

9 Sayan Bhattacharya, Deeparnab Chakrabarty, and Monika Henzinger. Deterministic fully dynamic approximate vertex cover and fractional matching in O(1) amortized update time. In *Proceedings of the 19th International Conference on Integer Programming and Combinatorial Optimization, IPCO 2017, Waterloo, ON, Canada, June 26-28, 2017*, pages 86–98, 2017.

10 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 2015, January 4-6, 2015*, pages 785–804, 2015.

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

12 Sayan Bhattacharya, Monika Henzinger, Danupon Nanongkai, and Xiaowei Wu. An improved algorithm for dynamic set cover. *CoRR*, abs/2002.11171, 2020.

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

14 Matthias Bonne and Keren Censor-Hillel. Distributed detection of cliques in dynamic networks. In Christel Baier, Ioannis Chatzigiannakis, Paola Flocchini, and Stefano Leonardi, editors, *Proc. 46th ICALP*, volume 132 of *LIPIcs*, pages 132:1–132:15, 2019.

15 Keren Censor-Hillel, Elad Haramaty, and Zohar S. Karnin. Optimal dynamic distributed MIS. In *Proceedings of the 2016 ACM Symposium on Principles of Distributed Computing, PODC 2016, Chicago, IL, USA, July 25-28, 2016*, pages 217–226, 2016.

16 Moses Charikar and Shay Solomon. Fully dynamic almost-maximal matching: Breaking the polynomial worst-case time barrier. In Ioannis Chatzigiannakis, Christos Kaklamanis, Dániel Marx, and Donald Sannella, editors, *Proc. 45th ICALP*, volume 107 of *LIPIcs*, pages 33:1–33:14. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2018.

17 Irit Dinur and David Steurer. Analytical approach to parallel repetition. In *Proceedings of the forty-sixth annual ACM symposium on Theory of computing*, pages 624–633, 2014.

18 Anupam Gupta, Ravishankar Krishnaswamy, Amit Kumar, and Debmalya Panigrahi. Online and dynamic algorithms for set cover. In *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.

19 Haim Kaplan and Shay Solomon. Dynamic representations of sparse distributed networks: A locality-sensitive approach. In Christian Scheideler and Jeremy T. Fineman, editors, *Proc. of 30th SPAA 2018*, pages 33–42, 2018.

20 Subhash Khot and Oded Regev. Vertex cover might be hard to approximate to within 2-epsilon. *J. Comput. Syst. Sci.*, 74(3):335–349, 2008.

21 Ofer Neiman and Shay Solomon. Simple deterministic algorithms for fully dynamic maximal matching. In *Proceedings of the 45th Annual ACM SIGACT Symposium on Theory of Computing, STOC 2013, Palo Alto, CA, USA, June 1-4, 2013*, pages 745–754, 2013.

22 Merav Parter, David Peleg, and Shay Solomon. Local-on-average distributed tasks. In *Proceedings of the 27th Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2016, Arlington, VA, USA, January 10-12, 2016*, pages 220–239, 2016.

23 D. Peleg. *Distributed Computing: A Locality-Sensitive Approach*. SIAM, 2000.

24 D. Peleg and S. Solomon. Dynamic $(1 + \epsilon)$-approximate matchings: A density-sensitive approach. In *Proceedings of the 27th Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2016, Arlington, VA, USA, January 10-12, 2016*, 2016.

25 S. Solomon. Fully dynamic maximal matching in constant update time. In *Proceedings of the 57th IEEE Annual Symposium on Foundations of Computer Science, FOCS 2016, New Brunswick, NJ, USA, October 9-11, 2016*, pages 325–334, 2016.

26 David P Williamson and David B Shmoys. *The design of approximation algorithms*. Cambridge university press, 2011.