Fully Dynamic Maximal Independent Set with Sublinear Update Time

Sepehr Assadi*
University of Pennsylvania
Philadelphia, PA, USA
sassadi@cis.upenn.com

Krzysztof Onak
IBM Research
Yorktown Heights, NY, USA
konak@us.ibm.com

Baruch Schieber
IBM Research
Yorktown Heights, NY, USA
sbar@us.ibm.com

Shay Solomon†
IBM Research
Yorktown Heights, NY, USA
solo.shay@gmail.com

ABSTRACT

A maximal independent set (MIS) can be maintained in an evolving m-edge graph by simply recomputing it from scratch in $O(m)$ time after each update. But can it be maintained in time sublinear in m in fully dynamic graphs?

We answer this fundamental open question in the affirmative. We present a *deterministic* algorithm with amortized update time $O(\min\{\Delta, m^{3/4}\})$, where Δ is a fixed bound on the maximum degree in the graph and m is the (dynamically changing) number of edges.

We further present a distributed implementation of our algorithm with $O(\min\{\Delta, m^{3/4}\})$ amortized message complexity, and $O(1)$ amortized round complexity and adjustment complexity (the number of vertices that change their output after each update). This strengthens a similar result by Censor-Hillel, Haramaty, and Karnin (PODC'16) that required an assumption of a non-adaptive oblivious adversary.

CCS CONCEPTS

• **Mathematics of computing → Graph algorithms**; • **Theory of computation → Dynamic graph algorithms**;

KEYWORDS

dynamic graph algorithms, dynamic distributed algorithms, maximal independent set

ACM Reference Format:
Sepehr Assadi, Krzysztof Onak, Baruch Schieber, and Shay Solomon. 2018. Fully Dynamic Maximal Independent Set with Sublinear Update Time. In *Proceedings of 50th Annual ACM SIGACT Symposium on the Theory of Computing (STOC'18).* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3188745.3188922

*Partially supported by NSF grant CCF-1617851.
†Partially supported by the IBM Herman Goldstine Postdoctoral Fellowship.

STOC'18, June 25–29, 2018, Los Angeles, CA, USA

ACM ISBN 978-1-4503-5559-9/18/06…$15.00
https://doi.org/10.1145/3188745.3188922

1 INTRODUCTION

Dynamic graph algorithms constitute an active area of research in theoretical computer science. Their objective is to maintain a solution to a combinatorial problem in an input graph—for example, a minimum spanning tree or maximal matching—under insertion and deletion of edges. The research on dynamic graph algorithms addresses the natural question of whether one essentially needs to recompute the solution from scratch after every update.

This question has been asked over the years for a wide range of problems such as connectivity [30, 31], minimum spanning tree [20, 22, 29, 31, 45], maximal matching [7, 32, 37, 43], approximate matching and vertex cover [10–12, 14–16, 25, 32, 37, 38], shortest paths [1, 8, 9, 19, 21, 26–28, 34, 42, 44], and graph coloring [3, 6, 13] (this is by no means a comprehensive summary of previous results).

Surprisingly however, almost no work has been done for the prominent problem of maintaining a *maximal independent set (MIS)* in dynamic graphs. Indeed, the only previous result for this problem that we are aware of is due to Censor-Hillel *et al.* [17], who developed a *randomized* algorithm for this problem in *distributed* dynamic networks and left the *sequential* case (the main focus of this paper) as a major open question. We note that implementing their distributed algorithm in the sequential setting requires $\Omega(\Delta)$[1] update time in *expectation*, where Δ is a fixed upper bound on the degree of vertices in the graph and can be as large as $\Theta(m)$ in sparse graphs.

The maximal independent set problem is of fundamental importance in graph theory with natural connections to a plethora of other basic problems, such as vertex cover, matching, vertex coloring, and edge coloring (in fact, all these problems can be solved approximately by finding an MIS, see, e.g., the paper of Linial [35]). As a result, this problem has been studied extensively in different settings, in particular in parallel and distributed algorithms [2, 4, 5, 18, 23, 33, 35, 36, 39]. (We refer the interested reader to the papers of Barenboim *et al.* [5] and Ghaffari [23] for the story of this problem in these settings and a comprehensive summary of previous work.)

In this paper, we concentrate on sequential algorithms for maintaining a maximal independent set in a dynamic graph. Our results

[1] It is *not* clear whether $O(\Delta)$ time is also sufficient for this algorithm or not; see Section 6 of their paper.

are also applicable to the dynamic distributed setting and improve upon the previous work of Censor-Hillel *et al.* [17].

1.1 Problem Statement and Our Results

Recall that a maximal independent set (MIS) of an undirected graph, is a *maximal* collection of vertices subject to the restriction that no pair of vertices in the collection are *adjacent*. In the maximal independent set problem, the goal is to compute an MIS of the input graph.

We study the *fully dynamic* variant of the maximal independent set problem in which the goal is to maintain an MIS of a dynamic graph G, denoted by $\mathcal{M} := \mathcal{M}(G)$, subject to a sequence of edge insertions and deletions. When an edge change occurs, the goal is to maintain $\mathcal{M}$ in time significantly faster than simply recomputing it from scratch. Our main result is the following:

THEOREM 1. *Starting from an empty graph on n fixed vertices, a maximal independent set can be maintained deterministically over any sequence of edge insertions and deletions in $O(m^{3/4})$ amortized update time, where m denotes the dynamic number of edges.*

As a warm-up to our main result in Theorem 1, we also present an extremely simple *deterministic* algorithm for maintaining an MIS with $O(\Delta)$ amortized update time, where Δ is a fixed upper bound on the maximum degree of the graph. Our algorithms can be combined together to achieve a *deterministic* $O(\min\left\{\Delta, m^{3/4}\right\})$ *amortized update time algorithm for maintaining an MIS in dynamic graphs*. This constitutes the first improvement on the update time required for this problem in fully dynamic graphs over the naïve $O(m)$ bound for all possible values of m. We now elaborate more on the details of our algorithm in Theorem 1.

Deterministic Algorithm. An important feature of our algorithm in Theorem 1 is that it is deterministic. The distinction between deterministic and randomized algorithms is particularly important in the dynamic setting as almost all existing randomized algorithms require the assumption of a *non-adaptive oblivious adversary* who is not allowed to learn anything about the algorithm's random bits. Alternately, this setting can be viewed as the requirement that the entire sequence of updates be fixed in advance, in which case the adversary cannot use the solution maintained by the algorithm in order to break its guarantees. While these assumptions can be naturally justified in many settings, they can render randomized algorithms entirely unusable in certain scenarios (see, e.g., [8, 10, 15] for more details).

As a result of this assumption, obtaining a deterministic algorithm for most dynamic problems is considered a distinctively harder task compared to finding a randomized one. This is evident by the polynomial gap between the update time of best known deterministic algorithms compared to randomized ones for many dynamic problems. For example, a maximal matching can be maintained in a fully dynamic graph with $O(1)$ update time via a randomized algorithm [43], assuming a non-adaptive oblivious adversary, while the best known deterministic algorithm for this problem requires $\Theta(\sqrt{m})$ update time [37] (see [13] for a similar situation for $(\Delta+1)$-coloring of vertices of a graph).

$O(1)$-Amortized Adjustment Complexity. An important performance measure of a dynamic algorithm is its *adjustment complexity* (sometimes called *recourse*) that counts the number of vertices (or edges) that need to be inserted to or deleted from the maintained solution after each update (see, e.g. [3, 13, 17, 24]). For many natural graph problems such as maintaining a maximal matching, constant worst-case adjustment complexity can be trivially achieved since one edge update cannot ever necessitate more than a constant number of changes in the maintained solution. This is, however, *not* the case for the MIS problem: by inserting an edge between two vertices already in $\mathcal{M}$, the adversary can force the algorithm to delete at least one end point of this edge from $\mathcal{M}$, which in turn forces the algorithm to pick *all* neighbors of this deleted vertex to ensure maximality (this phenomena also highlights a major challenge in the treatment of this problem compared to the maximal matching problem which we discuss further below).

Nevertheless, we prove that the adjustment complexity of our algorithm in Theorem 1 is $O(1)$ on average which is clearly optimal. Can we further strengthen our results to achieve an $O(1)$ *worst-case* adjustment complexity or even $o(n)$ *worst-case* adjustment complexity? We claim that this is not possible by showing that the worst-case adjustment complexity of any algorithm for maintaining an MIS is $\Omega(n)$. To this end we adapt an example proposed originally by [17] for proving a similar lower bound in distributed settings where vertex deletions by the adversary are also allowed. Adapting this example to settings that only allow the adversary to perform edge (rather than both vertex and edge) updates is not immediate. The description of this adaptation is deferred to the full version of this paper.

Distributed Implementation. Finding a maximal independent set is one of the most studied problems in distributed computing. In the distributed computing model, there is a processor on each vertex of the graph. Computation proceeds in synchronous rounds during which every processor can communicate messages of size $O(\log n)$ with its neighbors (this corresponds to the $\mathcal{CONGEST}$ model of distributed computation; see Section 5 for further details). In the dynamic setting, both edges and vertices can be inserted to or deleted from the graph and the goal is to update the solution in a small number of rounds of communication, with small communication cost and adjustment complexity.

Our results in the sequential setting also imply a *deterministic distributed algorithm for maintaining an MIS in a dynamic network with $O(1)$ amortized round complexity, $O(1)$ amortized adjustment complexity, and $O(\min\left\{\Delta, m^{3/4}\right\})$ amortized message complexity per each update.* This result achieves an improved message complexity compared to the distributed algorithm of [17] with asymptotically the same round and adjustment complexity (albeit in amortized sense as opposed to in expectation; see Section 5). More importantly, our result is achieved via a *deterministic* algorithm and does not require the assumption of a non-adaptive oblivious adversary. Similar to [17], our algorithm can also be implemented in the asynchronous

model, where there is no global synchronization of communication between nodes. We elaborate more on this result in Section 5.

Maximal Independent Set vs. Maximal Matching. We conclude this section by comparing the maximal independent set problem to the closely related problem of maintaining a maximal matching[2] in dynamic graphs. We discuss additional challenges that one encounters for the maximal independent set problem.

In sharp contrast to the maximal independent set problem, maintaining maximal matchings in dynamic graphs has been studied extensively, culminating in an $O(\sqrt{m})$ worst-case update time deterministic algorithms [37] and $O(1)$ expected update time randomized algorithm [43] (assuming a non-adaptive oblivious adversary).

Maintaining an MIS in a dynamic graph seems inherently more complicated than maintaining a maximal matching. One simple reason is that as argued before, a single update can only change the status of $O(1)$ edges/vertices in the maximal matching, while any algorithm can be forced to make $\Omega(n)$ changes to the MIS for a single edge update in the worst case. As a result, a maximal matching can be maintained with an $O(\Delta)$ worst-case update time via a straightforward algorithm (see, e.g. [32, 38]), while the analogous approach for MIS only results in $O(m)$ update time.

Another, perhaps more fundamental difference between the two problems lies in their different level of "locality." To adjust a maximal matching after an update, we only need to consider the neighbors of the recently unmatched vertices (to find another unmatched vertex to match with), while to fix an MIS, we need to consider the two-hop neighborhood of a recently removed vertex from the MIS (to add to the MIS the neighbors of this vertex which themselves do not have another neighbor in the MIS). We note that this difficulty is similar-in-spirit to the barrier for maintaining a *better than* 2 approximate matching via *excluding length-3 augmenting paths* in dynamic graphs. Currently, the best known algorithm for achieving a better than 2-approximation to matching in dynamic graphs requires $O(m^{1/4})$ update time [10, 11]. Achieving sub-polynomial in m update time—even using randomness and assuming a non-adaptive oblivious adversary—remains a major open problem in this area (we refer the interested reader to [15] for more details).

We emphasize that even for the seemingly easier problem of maximal matching, the best upper bound on update time using a deterministic algorithm (the focus of our paper) is only $O(\sqrt{m})$ [37].

1.2 Overview of Our Techniques

$O(\Delta)$-Amortized Update Time. Consider the following simple algorithm for maintaining an MIS $\mathcal{M}$ of a dynamic graph: for each vertex, maintain the number of its neighbors in $\mathcal{M}$ in a counter, and for each update in the graph or $\mathcal{M}$, spend $O(\Delta)$ time to update this counter for the neighbors of the updated vertex. What is the complexity of this algorithm? Unfortunately, as argued before, an update to the graph may inevitably result in an update of size $\Omega(n)$ to $\mathcal{M}$. Processing it may take $\Omega(n \cdot \Delta)$ time as we have to update all neighbors of every updated vertex. However, all we need to handle this case is the following basic observation: while a single update can force the algorithm to insert up to $\Omega(n)$ vertices to $\mathcal{M}$, it can never force the algorithm to remove more than one vertex from $\mathcal{M}$. We therefore charge the $O(\Delta)$ time needed to insert a vertex into $\mathcal{M}$ (and there can be many such vertices per one update) to the time spent in a previous update in which the same vertex was (the only one) removed from $\mathcal{M}$. This allows us to argue that on *average*, we only spend $O(\Delta)$ time per update.

$O(m^{3/4})$-Amortized Update Time. Achieving an $o(\Delta)$ amortized update time however is distinctly more challenging. On the one hand, we cannot afford to update all neighbors of a vertex after every change in the graph. On the other hand, we do not have enough time to iterate over all neighbors of an updated vertex to even check whether or not they should be added to $\mathcal{M}$ and hence need to maintain this information, which is a function of vertices in the two-hop neighborhood of a vertex, explicitly for every vertex.

To bypass these challenges, we relax our requirement for knowing the status of *all* vertices in the neighborhood of a vertex, and instead maintain the status of some vertices that are in the two-hop neighborhood of a vertex. More concretely, we allow "high" degree vertices to *not* update their "low" degree neighbors about their status (as the number of low degree neighbors can be very large), while making every "low" degree vertex update not only all its neighbors but even some of its neighbor's neighbors, using the extra time available to this vertex (as its degree is small). This approach allows us to maintain a "noisy" version of the information described above. Note that this information is not completely accurate as the status of some vertices in $\mathcal{M}$ would be unknown to their neighbors and their neighbor's neighbors (in the actual algorithm, we use a more fine-grained partition of vertices based on their degree into more than two classes, not only "high" and "low").

We now need to address a new challenge introduced by working with this "noisy" information: we may decide that a vertex is ready to join $\mathcal{M}$ based on the information stored in the algorithm and insert this vertex to $\mathcal{M}$, only to find out that there are already some vertices in $\mathcal{M}$ adjacent to this vertex. To handle this, we also relax the property of the basic algorithm above that only allowed for deleting one vertex from $\mathcal{M}$ per each update. This allows us to insert multiple vertices to $\mathcal{M}$ as long as a large portion (but not all) of their neighbors are known to be not in $\mathcal{M}$. Then we go back and delete a small number of "violating" vertices from $\mathcal{M}$ to make sure it is indeed an independent set. Note that deleting those vertices may now require inserting a new set vertices in their neighborhood to $\mathcal{M}$ to ensure maximality.

In order to be able to perform all those operations and recursively treat the newly deleted vertices in a timely manner, we maintain the invariant that whenever we need to remove more than one vertex from $\mathcal{M}$, the number of inserted vertices leading to this case is much larger than the number of removed vertices. This allows us to extend the simple charging scheme used in the analysis of the basic algorithm above to this new algorithm and prove our upper bound on the amortized update time of the algorithm.

We point out that despite the multiple challenges along the way that are described above, our algorithm turned to be quite simple in hindsight. The main delicate matters are in the choice of parameters and in the analysis. This in turn makes the implementation of our results in sequential and distributed settings quite practical.

[2] A maximal matching in a graph G can be obtained by computing an MIS of the line graph of G.

Organization. We introduce our notation and preliminaries in Section 2. We then present a simple proof of the $O(\Delta)$-amortized update time algorithm in Section 3 as a warm-up to our main result. Section 4 contains the proof of our main result in Theorem 1. The distributed implementation of our result and a detailed comparison of our results with that of Censor-Hillel *et al.* [17] appear in Section 5.

2 PRELIMINARIES

Notation. We denote the static vertex set of the input graph by V. Let $\mathcal{G} = \langle G_0, G_1, \ldots \rangle$ be the sequence of graphs that are given to the algorithm: initial graph G_0 is empty and each graph G_t is obtained from the previous graph G_{t-1} by either inserting or deleting a single edge $e_t = (u_t, v_t)$. We use $G_t(V, E_t)$ to denote the graph at step t and define $m_t := |E_t|$. Finally, throughout the paper, $\mathcal{M}$ denotes the maximal independent set maintained by the algorithm at every time step.

Greedy MIS Algorithm. Consider the following algorithm for computing an MIS of a given graph: Fix an arbitrary ordering of the vertices in the graph, add the first vertex to the MIS, remove all its neighbors from the list, and continue. This algorithm clearly computes an MIS of the input graph. In the rest of the paper, we refer to this algorithm as *the greedy MIS algorithm.*

Fact 2.1. *For an n-vertex graph G with maximum degree Δ, the greedy MIS algorithm computes an MIS of size at least $n/(\Delta+1)$.*

3 WARM-UP: A SIMPLE $O(\Delta)$-UPDATE-TIME DYNAMIC ALGORITHM

As a warm-up to our main result, we describe a straightforward algorithm for maintaining an MIS $\mathcal{M}$ in a dynamic graph with $O(\Delta)$ amortized update time, where Δ is a fixed upper bound on the maximum degree in the graph. For every vertex v in the graph, we simply maintain a counter $\texttt{MISCounter}[v]$, counting number of its neighbors in $\mathcal{M}$. In the following, we consider updating $\mathcal{M}$ and this counter after each edge update.

Let $e_t = (u_t, v_t)$ be the updated edge. Suppose first that we delete this edge. In this case, u_t and v_t cannot both be in $\mathcal{M}$ by definition of an independent set. Also, if none of them belong to $\mathcal{M}$, there is nothing to do. The interesting case is thus when exactly one of u_t or v_t belongs to $\mathcal{M}$; without loss of generality, we assume this vertex is u_t. We first subtract one from $\texttt{MISCounter}[v_t]$ (as it is no longer adjacent to u_t). If $\texttt{MISCounter}[v_t] > 0$ still, it means that v_t is adjacent to some vertex in $\mathcal{M}$ and hence we are done. Otherwise, we add v_t to $\mathcal{M}$ and update the counter of all its neighbors in $O(\Delta)$ time. Clearly, this step takes $O(\Delta)$ time in the worst case, after that $\mathcal{M}$ is indeed an MIS.

Now suppose e_t was inserted to the graph. The only interesting case here is when both u_t and v_t belong to $\mathcal{M}$ (we do not need to do anything in the remaining cases, other than perhaps updating the neighbor list of u_t and v_t in $O(1)$ time). To ensure that $\mathcal{M}$ remains an independent set, we need to remove one of these vertices, say u_t, from $\mathcal{M}$. After this, to ensure the maximality, we have to insert to $\mathcal{M}$ any neighbor of u_t that can now join $\mathcal{M}$. To do this, we first update the $\texttt{MISCounter}[\cdot]$ of all neighbors of u_t in $O(\Delta)$ time. Next (using the updated counter), we iterate over all neighbors of u_t and for each one check if they can be inserted to $\mathcal{M}$ now or not. If so, we add this new vertex to $\mathcal{M}$ and inform all its neighbors in $O(\Delta)$ time to update their $\texttt{MISCounter}[\cdot]$. It is easy to see that in this case, we spend $O(k \cdot \Delta)$ time in the worst case, where k is the number of vertices added to $\mathcal{M}$.

The correctness of this algorithm is straightforward to verify. We now prove that the amortized running time of the algorithm is $O(\Delta)$. The crucial observation is that whenever we change $\mathcal{M}$, we may increase its size without any restriction, but we never decrease its size by more than one. We use the following straightforward charging scheme.

Initially, we start with all vertices being in $\mathcal{M}$ as the original graph is empty. Whenever we delete one vertex from $\mathcal{M}$, we spend $O(\Delta)$ time to handle this vertex (including updating its neighbors and checking which ones can join $\mathcal{M}$), and place $O(\Delta)$ "extra budget" on this vertex to be spent later. We use this budget when this vertex is being inserted to $\mathcal{M}$ again. Whenever we want to bring this vertex back to $\mathcal{M}$, we only need to spend this extra budget and hence the $O(\Delta)$ time spent for inserting this vertex back to $\mathcal{M}$ can be charged to the time spent for this vertex when we removed it from $\mathcal{M}$. This implies that the update time is $O(\Delta)$ in average. We can therefore conclude the following lemma.

Lemma 3.1. *Starting from an empty graph on n vertices, a maximal independent set can be maintained* deterministically *over any sequence of K edge insertions and deletions in $O(K \cdot \Delta)$ time where Δ is a fixed bound on the maximum degree in the graph.*

4 AN $O(m^{3/4})$-UPDATE-TIME DYNAMIC ALGORITHM

We present our fully dynamic algorithm for maintaining a maximal independent set in this section and prove Theorem 1. The following lemma is a somewhat weaker looking version of Theorem 1. However, we prove next that this lemma is all we need to prove Theorem 1.

Lemma 4.1. *Starting with any arbitrary graph on n vertices and m edges, a maximal independent set $\mathcal{M}$ can be maintained* deterministically *over any sequence of $K = \Omega(m)$ edge insertions and deletions in $O(K \cdot m^{3/4})$ time, as long as the number of edges remains within a factor 2 of m.*

We first show that this lemma implies Theorem 1.

Proof of Theorem 1. For simplicity, we define $m = 1$ in case of empty graphs. We start from the empty graph and run the algorithm in Lemma 4.1 until the number of edges m_t in the graph differs from m by a factor more than 2. This crucially implies that the total number of updates before terminating the algorithm (the parameter K in Lemma 4.1), is $\Omega(m)$. As such, we can invoke Lemma 4.1 to obtain an upper bound of $O(m^{3/4})$ on the amortized update time of the algorithm throughout these updates. We then update $m = m_t$ and start running the algorithm in Lemma 4.1 on the current graph using the new choice of m. Clearly, this results in an amortized update time of $O(m^{3/4})$ where m now denotes the number of dynamic edges in the graph. As the algorithm in Lemma 4.1 always maintain an MIS of the underlying graph, we obtain the final result. □

The rest of this section is devoted to the proof of Lemma 4.1. In the following, we first describe the data structure maintained in the algorithm for storing the required information and its main properties and then present our update algorithm.

4.1 The Data Structure

For every vertex v, we maintain the following information:

- `neighbors`$[v]$: a list of current neighbors of v in the graph.
- `degree`$[v]$: an estimate of the degree of v to within a factor of two.
- `neighbors-degree`$[v]$: a list containing `degree`$[u]$ for every vertex u in `neighbors`$[v]$.
- `MIS-flag`$[v]$: a boolean entry indicating whether or not v belongs to $\mathcal{M}$.
- `MIS-neighbors`$[v]$: a counter denoting the size of a suitable subset of current neighbors of v in $\mathcal{M}$. Any vertex counted in `MIS-neighbors`$[v]$ belongs to $\mathcal{M}$ but not all neighbors of v in $\mathcal{M}$ are (necessarily) counted in `MIS-neighbors`$[v]$ (see Invariant 1 for more detail).
- `MIS-2hop-neighbors`$[v]$: a list, containing for every vertex w in `neighbors`$[v]$, a counter that counts the size of a suitable subset of current neighbors of w in $\mathcal{M}$. Any vertex counted in `MIS-2hop-neighbors`$[v][w]$ is also counted in `MIS-neighbors`$[w]$ but *not* vice versa (see Invariant 2 below for more detail).

Additionally, we maintain a partition of the vertices into four sets, $(V_{\text{High}}, V_{\text{Med-High}}, V_{\text{Med-Low}}, V_{\text{Low}})$, based on their current approximate degree, namely, `degree`$[v]$. In particular, v belongs to V_{High} iff `degree`$[v] \geq m^{3/4}$, to $V_{\text{Med-High}}$ iff $m^{3/4} >$ `degree`$[v] \geq m^{1/2}$, to $V_{\text{Med-Low}}$ iff $m^{1/2} >$ `degree`$[v] \geq m^{1/4}$, and to V_{Low} iff `degree`$[v] < m^{1/4}$. We refer to the vertices of V_{Low} as the *low-degree* vertices. Throughout, we assume that in any of the lists maintained for a vertex by the algorithm, we can directly iterate over vertices of a particular subset in $(V_{\text{High}}, V_{\text{Med-High}}, V_{\text{Med-Low}}, V_{\text{Low}})$. (This can be done, for example, by storing these lists as four separate linked lists, one per each such subset.)

The following invariant is concerned with the information we need from `MIS-neighbors`$[v]$.

Invariant 1. *For any $v \in V \setminus V_{\text{Low}}$,* `MIS-neighbors`$[v]$ *counts the number of all neighbors of v in $\mathcal{M}$. For any vertex $v \in V_{\text{Low}}$,* `MIS-neighbors`$[v]$ *counts the number of neighbors of v that are in $\mathcal{M}$ but* not *in V_{High}, i.e., are in $\mathcal{M} \setminus V_{\text{High}}$.*

By Invariant 1, any vertex either knows the number of *all* its neighbors in $\mathcal{M}$ or is a low-degree vertex and can iterate over all its neighbors in $O(m^{1/4})$ time to count this number. Moreover, even a low-degree vertex knows the number of its neighbors in $\mathcal{M} \setminus V_{\text{High}}$. This is crucial for our algorithm as in some cases, we need to iterate over *many* vertices that belong to V_{Low} and decide if they can join $\mathcal{M}$ and hence cannot spend $O(m^{1/4})$ time per each vertex to determine this information. Note that the information we obtain in this way is "noisy", as we ignore some neighbors of vertices in V_{Low} that are potentially in $\mathcal{M}$. We shall address this problem using a post-processing step that exploits the fact that the total number of ignored vertices, i.e., vertices in V_{High}, is small.

The following invariant is concerned with the information we need from `MIS-2hop-neighbors`$[v]$.

Invariant 2. *For all $v \in V$ and $u \in$* `neighbors`$[v] \cap V_{\text{Low}}$, `MIS-2hop-neighbors`$[v][u]$ *counts the number of vertices in $V_{\text{Med-Low}} \cup V_{\text{Low}}$ that belong to $\mathcal{M}$ and are neighbors of u (the entry in* `MIS-2hop-neighbors`$[v][u]$ *for any vertex $u \notin V_{\text{Low}}$ is $\perp$).*

Invariant 2 allows us to infer some nontrivial information about the two-hop neighborhood of any vertex. We use Invariant 2 to quickly determine which neighbors of a vertex v can be added to $\mathcal{M}$ in case v is deleted from it. Similar to the one-hop information we obtain through maintaining Invariant 1, the information we obtain in this way is also "noisy".

We show how to update the information per each vertex after a change in the topology or $\mathcal{M}$. Maintaining `neighbors`$[v]$ under edge updates is straightforward. To maintain `degree`$[v]$, each vertex simply keeps a 2-approximation of its degree in `degree`$[v]$. Whenever the current actual degree of v differs from `degree`$[v]$ by more than a factor of two, v updates `degree`$[v]$ to its actual degree and informs all its neighbors $u \in$ `neighbors`$[v]$ to update `neighbors-degree`$[u]$. This requires only $O(1)$ amortized time.

The above information is a function of the underlying graph and not $\mathcal{M}$. We also need to update the information per each vertex that are functions of $\mathcal{M}$ whenever $\mathcal{M}$ changes. Maintaining `MIS-flag`$[v]$ is trivial for any vertex v, hence in the following we focus on the remaining two parts.

Once a vertex u changes its status in $\mathcal{M}$, we apply the following algorithm to update the value of `MIS-neighbors`$[v]$ for every vertex v (we only need to update this for $v \in$ `neighbors`$[u]$).

Algorithm UpdateNeighbors(u)**.** An algorithm called whenever a vertex u enters or exists $\mathcal{M}$ to update `MIS-neighbors`$[v]$ for neighbors of u.

(1) If $u \in V_{\text{High}}$, update `MIS-neighbors`$[v]$ for any vertex $v \in$ `neighbors`$[u]$ *not* in V_{Low} accordingly (i.e., add or subtract one depending on whether u joined or left $\mathcal{M}$).
(2) If $u \notin V_{\text{High}}$, update `MIS-neighbors`$[v]$ for every vertex $v \in$ `neighbors`$[u]$.

It is immediate to see that by running UpdateNeighbors(u) in our main algorithm whenever a vertex u is updated in $\mathcal{M}$, we can maintain Invariant 1. Also each call to UpdateNeighbors(u) takes $O(m^{3/4})$ time in worst-case since in both cases of the algorithm, we only need to update $O(m^{3/4})$ vertices: (i) if $u \in V_{\text{High}}$, the algorithm only updates the vertices in $V \setminus V_{\text{Low}}$ whose size is $O(m^{3/4})$, and (ii) if $u \notin V_{\text{High}}$, u only has $O(m^{3/4})$ neighbors to update. We also point out that `MIS-neighbors`$[v]$ can be updated easily whenever an edge incident on (u, v) is inserted or deleted in $O(1)$ time by simply visiting `MIS-flag`$[u]$ and updating `MIS-neighbors`$[v]$ accordingly.

Now consider updating MIS-2hop-neighbors[·]. We use the following algorithm on a vertex u that has changed its status in $\mathcal{M}$ to update MIS-2hop-neighbors[v] for every vertex v in the graph (we only need to update this information for the two-hop neighborhood of u).

Algorithm UpdateTwoHopNeighbors(u)**.** An algorithm called when a vertex u enters or exists $\mathcal{M}$ to update MIS-2hop-neighbors[v] for the two-hop neighborhood of u.

(1) If $u \in V_{\text{Med-Low}} \cup V_{\text{Low}}$, for any vertex $w \in$ neighbors[u]:
   (a) If w belongs to V_{Low}, iterate over all vertices $v \in$ neighbors[w].
   (b) For any such v, update MIS-2hop-neighbors[v][w] accordingly (i.e., add or subtract one depending on whether u joined or left $\mathcal{M}$).

Each call to UpdateTwoHopNeighbors(u) takes $O(m^{3/4})$ time in the worst case. This is because u only updates its neighbors if it has $O(m^{1/2})$ neighbors as u should be in $V_{\text{Med-Low}} \cup V_{\text{Low}}$ and when it updates its neighbors, it changes the counter of $O(m^{1/4})$ vertices (as w should be in V_{Low}). This ensures that the running time of the algorithm is $O(m^{1/2} \cdot m^{1/4}) = O(m^{3/4})$. Whenever an edge (u, v) is updated in the graph, we can run a similar algorithm to update the two-hop neighborhood of u and v in the same way in $O(m^{3/4})$ time; we omit the details. It is also straightforward to verify that by running UpdateTwoHopNeighbors(u) in our main algorithm whenever a vertex u is updated in $\mathcal{M}$, we preserve Invariant 2.

Finally, recall that we also need a preprocessing step that given a graph G initializes this data structure. We can implement this step by first initializing all non-MIS-related information in this data structure in $O(m)$ time (we do not need to handle isolated vertices at this point). Next, we run the greedy MIS algorithm to compute an MIS $\mathcal{M}$ of this graph in $O(m)$ time (again only on non-isolated vertices). Finally, we update the information for every vertex in $\mathcal{M}$ using the two procedures above which takes $O(m \cdot m^{3/4})$ time in total. (We note that a more efficient implementation for this initial stage is possible.) As $K = \Omega(m)$ in Lemma 4.1, this (one time only) initialization cost is within the bounds stated in the lemma statement.

We summarize the results in this section in the following two lemmas.

Lemma 4.2. *After updating any single edge, the data structure stored in the algorithm can be updated in $O(m^{3/4})$ amortized time. Moreover, Invariants 1 and 2 hold after this update.*

Lemma 4.3. *After updating any single vertex in $\mathcal{M}$, the data structure stored in the algorithm can be updated in $O(m^{3/4})$ worst case time. Moreover, Invariants 1 and 2 hold after this update.*

4.2 The Update Algorithm

The update algorithm is applied following edge insertions and deletions to and from the graph. After any edge update, the algorithm updates the data structure and $\mathcal{M}$. In order to do the latter task, the algorithm may need to remove and/or insert multiple vertices from and to $\mathcal{M}$. Since we already argued that maintaining the data structure requires $O(m^{3/4})$ amortized time (by Lemma 4.2), from now on, without loss of generality, we only measure the time needed to fix $\mathcal{M}$ after any edge update and ignore the additive term needed to update the data structure. The following is the core invariant that we aim to maintain in our algorithm.

Invariant 3 (Core Invariant). *Following every edge update, the set $\mathcal{M}$ maintained by the algorithm is an MIS of the input graph. Moreover,*
(i) if only a single vertex leaves $\mathcal{M}$, then there is no restriction on the number of vertices joining $\mathcal{M}$ (which could be zero).
(ii) if at least two vertices leave $\mathcal{M}$, then at least twice as many vertices join $\mathcal{M}$.
In either case, the total time spent by the algorithm to fix $\mathcal{M}$ for an edge update is at most an $O(m^{3/4})$ factor larger than the total number of vertices leaving and joining $\mathcal{M}$.

Before showing how to maintain Invariant 3, we present the proof of Lemma 4.1 using this invariant.

Proof. The main idea behind the proof is as follows. By Invariant 3, after each step, the size of $\mathcal{M}$ either decreases by at most one, or it will increase. At the same time, $\mathcal{M}$ cannot grow more than n, the number of vertices in the graph. It then follows that the *average* number of changes to $\mathcal{M}$ per each update is $O(1)$. As we only spend $O(m^{3/4})$ per each update, we obtain the final result. We now present the formal proof using the following charging scheme.

Recall that we compute an MIS $\mathcal{M}$ of the initial graph in the preprocessing step and that the initialization phase takes $O(m \cdot m^{3/4})$ time in total. We place $O(m^{3/4})$ "extra budget" on vertices in the initial graph that do not belong to $\mathcal{M}$ to be spent later when these vertices are inserted to $\mathcal{M}$. As the number of such vertices is $O(m)$, this extra budget can be charged to the time spent in the initialization phase. Note that at this point, an extra budget is allocated to any vertex not in $\mathcal{M}$ and we maintain this throughout the algorithm.

Whenever an update results in only a single vertex leaving $\mathcal{M}$ (corresponding to Part (i) of Invariant 3), we spend $O(m^{3/4})$ time to handle this vertex and additionally place $O(m^{3/4})$ budget on this vertex and then for the vertices inserted to $\mathcal{M}$, we simply use the extra budget allocated to these vertices before to charge for the $O(m^{3/4})$ time needed to handle each. If an update results in removing $k > 1$ vertices from $\mathcal{M}$, we know that at least $2 \cdot k$ vertices would be added to $\mathcal{M}$ after this update (corresponding to Part (ii) of Invariant 3). In this case, we use the $O(m^{3/4})$ extra budget on these (at least) $2 \cdot k$ vertices that are joining $\mathcal{M}$ to charge for the time needed to insert these vertices to $\mathcal{M}$, remove the k initial vertices from $\mathcal{M}$, and place $O(m^{3/4})$ extra budget on every removed vertex. As a result, this type of updates can be handled free of charge. Finally, if an update only involves inserting some vertices to $\mathcal{M}$, we simply use the budgets on these vertices to handle them free of charge. This finalizes the proof of Lemma 4.1.

We point out that using the above charging scheme, we can also argue that the average number of changes to $\mathcal{M}$ is $O(1)$ in each update. □

Fix a time step t and suppose the invariant holds up until this time step. Let $e_t = (u_t, v_t)$ be the edge updated at this time step. In the remainder of this section, we describe one round of the update algorithm to handle this single edge update and preserve Invariant 3.

4.2.1 Edge Deletions. We start with the easier case of deleting an edge $e_t = (u_t, v_t)$.

Case 1: *Neither u_t nor v_t belong to $\mathcal{M}$.* In this case, there is nothing to do.

Case 2: *u_t belongs to $\mathcal{M}$ but not v_t (or vice versa).* After deleting the edge e_t, it is possible that v_t may need to join $\mathcal{M}$ as well. We first check whether $\texttt{MIS-neighbors}[v_t] = 0$. If not, there is nothing else to do as v_t is still adjacent to some vertex in $\mathcal{M}$. Otherwise, we need to ensure that v_t does not have any neighbor in $\mathcal{M}$ (outside those vertices counted in $\texttt{MIS-neighbors}[v_t]$). If $v_t \in V \setminus V_{\text{Low}}$, by Invariant 1, $\texttt{MIS-neighbors}[v_t]$ counts *all* neighbors of v_t and hence there is nothing more to check. If $v_t \in V_{\text{Low}}$, we can go over all the $O(m^{1/4})$ vertices in the neighborhood of v_t and check whether v_t has a neighbor in $\mathcal{M}$ or not. This only takes $O(m^{1/4})$ time in the worst case. Again, if we find a neighbor in $\mathcal{M}$ there is nothing else to do. Otherwise, we add v_t to $\mathcal{M}$ and update the data structure which takes $O(m^{3/4})$ time in the worst case by Lemma 4.3. After this step, $\mathcal{M}$ is again a valid MIS and hence Invariant 3 is preserved as we only spent $O(m^{3/4})$ time and inserted at most one vertex to $\mathcal{M}$ without deleting any vertex from it.

Case 3: *Both u_t and v_t belong to $\mathcal{M}$.* This case is not possible in the first place by Invariant 3 as otherwise $\mathcal{M}$ maintained by the algorithm before this edge update was not an MIS.

4.2.2 Edge Insertions. We now consider the by far more challenging case of edge insertions where we concentrate bulk of our efforts. It is immediate to see that the only time we need to handle an edge insertion is when the inserted edge $e_t = (u_t, v_t)$ connects two vertices already in $\mathcal{M}$ (there is nothing to do in the remaining cases). Hence, in the following, *we assume both u_t and v_t belong $\mathcal{M}$.*

To ensure that $\mathcal{M}$ is an independent set, we first need to remove one of u_t or v_t from it and then potentially insert some of the neighbors of the deleted vertex to $\mathcal{M}$ to ensure its maximality. Let u_t be the deleted vertex (the choice of which vertex to delete is arbitrary). After deleting u_t, we update the algorithm's data structure in $O(m^{3/4})$ time by Lemma 4.3.

Let $L := \texttt{neighbors}[u_t] \cap V_{\text{Low}}$ denote the set of low degree neighbors of u_t. We first show that one can easily handle all neighbors of u_t which are *not* in L. To do so, we can iterate over these vertices as there are $O(m^{3/4})$ of them and for any vertex w, by Invariant 1, we know whether w can be added $\mathcal{M}$ or not by simply checking $\texttt{MIS-neighbors}[w]$. Hence, we can add the necessary vertices to $\mathcal{M}$ and spend $O(m^{3/4})$ time for each inserted one using Lemma 4.3. As such, we spend $O(m^{3/4})$ time for iterating the vertices which did not join $\mathcal{M}$ and $k \cdot O(m^{3/4})$ time for the k vertices that joined $\mathcal{M}$. Hence, Invariant 3 is preserved after this step.

We now consider the challenging case of updating the neighbors of u_t that belong to L. As the number of such vertices is potentially very large, we cannot iterate over all of them anymore. Define the following subsets of L:

- $L_{\text{MIS}} \subseteq L$: the set of vertices in L that do not have any neighbor in $\mathcal{M}$.
- $L_{\text{1-hop}} \supseteq L_{\text{MIS}}$: all vertices $w \in L$ where $\texttt{MIS-neighbors}[w] = 0$ i.e., our algorithm did not count any neighbor for them in $\mathcal{M}$. Recall that $\texttt{MIS-neighbors}[w]$ does *not* count all neighbors of w in $\mathcal{M}$; it is missing the vertices in V_{High} by Invariant 1.
- $L_{\text{2-hop}} \supseteq L_{\text{1-hop}} \supseteq L_{\text{MIS}}$: all vertices $w \in L$, where $\texttt{MIS-2hop-neighbors}[u_t][w] = 0$. Again, recall that $\texttt{MIS-2hop-neighbors}[u_t][w_t]$ does *not* count all neighbors of $w \in \texttt{MIS-neighbors}[w]$ (and consequently in $\mathcal{M}$); it misses the vertices in $V_{\text{Med-High}}$ in $\texttt{MIS-neighbors}[w]$ (and additionally V_{High} in $\mathcal{M}$) by Invariant 2.

Let $\ell_{\text{MIS}} := |L_{\text{MIS}}|$, $\ell_{\text{1-hop}} := \left|L_{\text{1-hop}}\right|$ and $\ell_{\text{2-hop}} := \left|L_{\text{2-hop}}\right|$, where $\ell_{\text{MIS}} \leq \ell_{\text{1-hop}} \leq \ell_{\text{2-hop}}$. Our algorithm does not know the sets L_{MIS} and $L_{\text{1-hop}}$ or even their sizes. However, the update algorithm knows the value of $\ell_{\text{2-hop}}$ and has access to vertices in $L_{\text{2-hop}}$ through the list $\texttt{MIS-2hop-neighbors}[u_t]$ and can iterate over them in $O(1)$ time per each vertex in $L_{\text{2-hop}}$ (notice that even this can be potentially too time consuming as size of this list can be too large). We consider different cases based on the value of these parameters.

Case 1: *when $\ell_{\text{2-hop}}$ is small, i.e., $\ell_{\text{2-hop}} \leq 4 \cdot m^{3/4}$.* In this case, we iterate over vertices $w \in L_{\text{2-hop}}$ in $O(\ell_{\text{2-hop}}) = O(m^{3/4})$ time and check whether $\texttt{MIS-neighbors}[w] = 0$ or not. This allows us to compute the set $L_{\text{1-hop}}$ and $\ell_{\text{1-hop}}$ as well. We further distinguish between two cases.

Case 1-a: *when $\ell_{\text{1-hop}}$ is very small, i.e., $\ell_{\text{1-hop}} \leq 4 \cdot m^{1/2}$.* We iterate over vertices $w \in L_{\text{1-hop}}$ and for each vertex, spend $O(m^{1/4})$ time to go over all its neighbors and decide whether w has any neighbor in $\mathcal{M}$ or not (degree of w is $O(m^{1/4})$ since it belongs to V_{Low}). Hence, in this case, we can obtain the set L_{MIS} fully in $O(m^{3/4})$ time in total.

We then iterate over vertices in L_{MIS}, insert each one greedily to $\mathcal{M}$, and update the data structure in $O(m^{3/4})$ time using Lemma 4.3. It is possible that some vertices in L_{MIS} are adjacent to each other and hence before inserting any vertex w, we first need to check $\texttt{MIS-neighbors}[w]$ to make sure it is zero still (by Invariant 1 and since all vertices in L_{MIS} belong to V_{Low}, any vertex added to $\mathcal{M}$ here would update $\texttt{MIS-neighbors}[w]$ for any neighbor w). Hence, in this case, we spend $O(m^{3/4})$ time for each vertex inserted to $\mathcal{M}$ and did not delete any vertex from it. Therefore, Invariant 3 is preserved after the edge update in this case.

Case 1-b: *when $\ell_{\text{1-hop}}$ is* not *very small, i.e., $\ell_{\text{1-hop}} > 4 \cdot m^{1/2}$.* In this case, we cannot afford to compute L_{MIS} explicitly. Rather, we simply add the vertices in $L_{\text{1-hop}}$ to $\mathcal{M}$ directly, without considering whether they are adjacent to vertices already in $\mathcal{M}$ or not at all (although we check that they are not adjacent to the previously inserted vertices from $L_{\text{1-hop}}$). As a result, it is possible that after this process, $\mathcal{M}$ is not an independent set of the graph anymore. To fix this, we perform a post processing step in which we delete

some vertices from $\mathcal{M}$ to ensure that the remaining vertices indeed form an MIS of the original graph.

Concretely, we go over vertices in $L_{1\text{-hop}}$ and insert each to $\mathcal{M}$ if none of its neighbors have been added to $\mathcal{M}$ *in this step*, and then invoke Lemma 4.3 to update the algorithm's data structure. Since in this step, we are only adding vertices that are in V_{Low}, we can check in $O(1)$ time whether a vertex has a neighbor in $\mathcal{M}$ (that has been added in this step) or not by Invariant 1. This step clearly takes $O(m^{3/4})$ time per each vertex inserted to the MIS.

At this point, it is possible that there are some vertices in $\mathcal{M}$ which are adjacent to the newly inserted vertices. By Invariant 1, we know that these vertices can only belong to V_{High} and hence there are at most $m^{1/4}$ of them. We iterate over all vertices in V_{High} and check whether they have a neighbor in $\mathcal{M}$ (by Invariant 1, we stored this information for these vertices) and mark all such vertices. Next, we remove all these marked vertices from $\mathcal{M}$ simultaneously and update the algorithm's state by Lemma 4.3. We are not done yet though because after removing these vertices, it is possible that we may need to bring some of their neighbors back to $\mathcal{M}$. We solve this problem recursively using the same update algorithm by treating these marked vertices the same as u_t.

We argue that Invariant 3 is preserved. As the degree of vertices in $L_{1\text{-hop}}$ is bounded by $m^{1/4}$, the number of vertices added to $\mathcal{M}$ in this part is at least $\frac{\ell_{1\text{-hop}}}{(m^{1/4}+1)} \geq 2 \cdot m^{1/4}$ (by Fact 2.1 and the assumption on $\ell_{1\text{-hop}}$ in this case). On the other hand, the number of vertices removed from $\mathcal{M}$ is at most equal to size of V_{High} which is $m^{1/4}$. As a result, in this specific step, the number of vertices inserted to $\mathcal{M}$ is at least twice as many as the vertices removed from it. For any vertex inserted or deleted from $\mathcal{M}$ also, we spent $O(m^{3/4})$ time. As we are performing the recursive step using the same algorithm, we can argue inductively that for any vertex deleted in those recursive calls, at least twice as many vertices would be added to $\mathcal{M}$ and that the total running time would be proportional to the number of vertices added or removed from $\mathcal{M}$ times $O(m^{3/4})$. We point out that any recursive call that leads to another one in this algorithm necessarily increase the number of vertices in $\mathcal{M}$ and hence the algorithm does indeed terminate (see also case *2*).

Case 2: *when* $\ell_{2\text{-hop}}$ *is* not *small, i.e.,* $\ell_{2\text{-hop}} > 4 \cdot m^{3/4}$. We use a similar strategy as case *1-b* here as well. We iterate over all vertices in $L_{2\text{-hop}}$, greedily add each vertex to $\mathcal{M}$ as long as this vertex is not adjacent to any of the newly added vertices (which can be checked in $O(1)$ time by Invariant 1), and update the data structure using Lemma 4.3. As the maximum degree of vertices in $L_{2\text{-hop}}$ is at most $m^{1/4}$, we add at least $\frac{\ell_{2\text{-hop}}}{m^{1/4}+1} > 2 \cdot m^{1/2}$ vertices to $\mathcal{M}$ by Fact 2.1. By Invariant 2, if a vertex belongs to $L_{2\text{-hop}}$, the only neighbors of this vertex in $\mathcal{M}$ belong to V_{High} or $V_{\text{Med-High}}$ and hence has degree at least $m^{1/2}$. We go over these vertices next and mark them. Then, we remove all of them from $\mathcal{M}$ simultaneously and update the algorithm by Lemma 4.3. Similar to case *1-b*, we now also have to consider bringing some of the neighbors of these vertices to $\mathcal{M}$ which is handled recursively exactly the same way as in case *1-b*.

We first analyze the time complexity of this step. Iterating over $L_{2\text{-hop}}$ takes $O(|\ell_{2\text{-hop}}|) = O(m)$ time and since we are inserting at least $m^{1/2}$ vertices from $L_{2\text{-hop}}$ to $\mathcal{M}$, we can charge the time needed for this step to the time allowed for inserting these vertices to $\mathcal{M}$. Moreover, we inserted at least $2 \cdot m^{1/2}$ vertices to $\mathcal{M}$ and would remove at most $m^{1/2}$ vertices after considering violating vertices in V_{High} and $V_{\text{Med-High}}$. Hence, number of inserted vertices is at least twice the number of removed ones at this step. We can also argue inductively that this property hold for each recursive call similar to the case *1-b*. This finalizes the proof of this case.

To conclude, we proved that Invariant 3 is preserved after any edge insertion or deletion in the algorithm, which finalizes the proof of Lemma 4.1.

5 MAXIMAL INDEPENDENT SET IN DYNAMIC DISTRIBUTED NETWORKS

We consider the $\mathcal{CONGEST}$ model of distributed computation (cf. [41]) which captures the essence of both spatial locality and congestion. The network is modeled by an undirected graph $G(V, E)$ where the vertex-set is V, and E corresponds to both the edge-set in the current graph and also the vertex pairs that can directly communicate with each other. We assume a synchronous communication model, where time is divided into rounds and in each round, each vertex can send a message of size $O(\log n)$ bits to any of its neighbors, where $n = |V|$. The goal is to maintain an MIS $\mathcal{M}$ in G in a way that each vertex is able to output whether or not it belongs to $\mathcal{M}$.

We focus on dynamically changing networks where both edges and vertices can be inserted to or deleted from the network. For deletions, we consider *graceful deletions* where the deleted vertex/edge may be used for passing messages between its neighbors (endpoints), and is only deleted completely once the network is stable again. After each change, the vertices communicate with each other to adjust their outputs, namely make the network *stable* again. We make the standard assumption that the changes occur in large enough time gaps, and hence the network is always stable before the next change occurs (see, e.g., [17, 40]). We further assume that each change in the network is indexed and vertices affected by this change know how many updates have happened before[3].

There are three complexity measures for the algorithms in this model. The first is the so-called *adjustment complexity*, which measures the number of vertices that change their output as a result of a recent topology change. The second is the *round complexity*, the number of rounds required for the network to become stable again after each update. The third is the *message complexity*, measuring the total number of $O(\log n)$-length messages communicated by the algorithm.

Our main result in this section is an implementation of Theorem 1 in this distributed setting for maintaining an MIS in a dynamically changing network.

Theorem 2. *Starting from an empty distributed network on n vertices, a maximal independent set can be maintained* deterministically *in a distributed fashion (in the* $\mathcal{CONGEST}$ *communication model) over any sequence of vertex/edge insertions and (graceful) deletions with* (*i*) $O(1)$ amortized adjustment complexity, (*ii*) $O(1)$ amortized

[3]This is only needed by our algorithm in Theorem 2 to have an approximation of the number of edges in the graph, which is a global quantity and cannot be maintained by each vertex locally

round complexity, *and* (*iii*) $O(m^{3/4})$ amortized message complexity. *Here, m denotes the number of dynamic edges.*

The algorithm in Lemma 3.1 can also be trivially implemented in this distributed setting, resulting in *an extremely simple deterministic distributed algorithm for maintaining an MIS of a dynamically changing graph in* $O(1)$ *amortized adjustment complexity and round complexity, and* $O(\Delta)$ *amortized message complexity.* As argued before, this simple algorithm already strengthens the previous randomized algorithm of Censor-Hillel *et al.* [17] by virtue of being deterministic and not requiring an assumption of a non-adaptive oblivious adversary. In the following, we compare our results in the distributed setting with those of [17].

Amortized vs in Expectation Guarantee. The guarantees on the complexity measures provided by our deterministic algorithms in this setting are *amortized*, while the randomized algorithm in [17] achieves its bound *in expectation* which may be considered somewhat stronger than our guarantee. To achieve this guarantee however, the algorithm in [17], besides using randomization, also assumes a non-adaptive oblivious adversary. An adaptive adversary (the assumption supported by all our algorithms in this paper) can force the algorithm in [17] to adjust the MIS by $\Omega(n)$ vertices in every round, which in turn blows up all the complexity measures in [17] by a factor of $\Omega(n)$. It is also worth mentioning that the guarantee achieved by [17] only holds in expectation and not *with high probability* and for a fundamental reason: It was shown in [17] that for every value of k, there exists an instance for which at least $\Omega(k)$ adjustments are needed for any algorithm with probability at least $1/k$ (see Section 1.1 of their paper).

Broadcast vs Unicast. The communication in algorithm of [17] in each round is $O(1)$ broadcast messages in expectation that requires only $O(1)$ bits on every edge (i.e., each vertex communicates the same $O(1)$ bits to every one of its neighbors). As such, the total communication at every round of this algorithm is $O(\Delta)$ bits in expectation. Our amortized $O(\Delta)$-message complexity algorithm (distributed implementation of Lemma 3.1) also works with the same guarantee: indeed, every vertex simply needs to send $O(1)$ bits to all its neighbors in a broadcast manner so that their neighbors know whether to add or subtract the contribution of this vertex to or from their counter. This is however not the case for our main algorithm in Theorem 2 which requires a processor to communicate differently to its neighbor over each edge (in general, one cannot hope to achieve $o(\Delta)$ communication with only broadcast messages). Additionally, this algorithm now requires to communicate $O(\log n)$ bits (as opposed to $O(1)$ in the previous two algorithms) over every edge. This is mainly due to the fact that in this new algorithm we need to communicate with vertices which are at distance 2 of the current vertex and hence we need to carry the ID of original senders in the messages also.

Graceful vs Abrupt Deletions. A stronger notion of deletion in the dynamic setting is *abrupt deletion* in which the neighbors of the deleted vertex/edge simply discover that this vertex/edge is being deleted and the deleted vertex/edge cannot be used for communication anymore right after the deletion happens. Censor-Hillel *et al.* [17] also extend their result to this more general setting and achieved the same guarantees except for message complexity of abrupt deletion of a node which is now $O(\min\{\log n, \Delta\})$ broadcasts as opposed to $O(1)$. We do not consider this model explicitly. However, it is straightforward to verify that our amortized $O(\Delta)$-message complexity algorithm (distributed implementation of Lemma 3.1) works in this more general setting with virtually no change and even still achieves amortized $O(1)$ broadcast per abrupt deletion of a vertex as well. We believe that our main algorithm in Theorem 2 should also work in this more general setting with proper modifications but we did not prove this formally.

Synchronous vs Asynchronous Communication. We focused only on the synchronous communication in this paper. Censor-Hillel [17] also considered the asynchronous model of communication and showed that their algorithm holds in this model as well, albeit with a weaker guarantee on its message complexity. Our algorithms can be modified to work in an asynchronous model as well, as at each stage of the algorithm we can identify a (different) local "coordinator" that can be used to synchronize the operations with an added overhead that is within a constant multiplicative of the synchronous complexity (as per each update only vertices within two-hop neighborhood of a vertex need to communicate with each other in our algorithm); we omit the details but refer the reader to Section 5.2.2 for more information on the use of a local coordinator in our algorithms.

We now turn to proving Theorem 2, using the following lemma the same way we used Lemma 4.1 in the proof of Theorem 1.

Lemma 5.1. *Starting with any arbitrary graph on n vertices and m edges, a maximal independent set* $\mathcal{M}$ *can be maintained* deterministically *in a distributed fashion (under the* $\mathcal{CONGEST}$ *communication model) over any sequence of* $K = \Omega(m)$ *vertex/edge insertions and (graceful) deletions as long as the number of edges in the graph remains within a factor* 2 *of m. The algorithm:*

(i) makes $O(K)$ *adjustment to* $\mathcal{M}$ *in total, i.e., has* $O(1)$ *amortized adjustment complexity,*
(ii) requires $O(K)$ *rounds in total, i.e., has* $O(1)$ *amortized round complexity, and*
(iii) communicates $O(K \cdot m^{3/4})$ *messages in total, i.e., has* $O(m^{3/4})$ *amortized message complexity.*

The algorithm in Lemma 5.1 is a simple implementation of our sequential dynamic algorithm in Lemma 4.1. In the following, we first adapt the data structures introduced in Section 4.1 to the distributed setting. We then show that with proper adjustments, the (sequential) update algorithm in Section 4.2 can also be used in the $\mathcal{CONGEST}$ model and prove Theorem 2.

5.1 The Data Structure

We store the same exact information in Section 4.1 per each vertex here as well and maintain Invariants 1 and 2. We first prove that the two procedures UpdateNeighbors and UpdateTwoHopNeighbors can both be implemented in constant rounds and $O(m^{3/4})$ messages in total. In particular,

Lemma 5.2. *For any vertex* $u \in V$,

(i) UpdateNeighbors(u) *operation requires spending* 1 *round and* $m^{3/4}$ *messages in total, and*

(ii) UpdateTwoHopNeighbors(u) *operation requires* 2 *rounds and* $2 \cdot m^{3/4}$ *messages in total.*

Proof. *Part* (i). If $u \in V_{\mathsf{High}}$, it only needs to send a message to its neighbors in $V \setminus V_{\mathsf{Low}}$ and inform them on the status of u (whether it is inserted to or deleted from $\mathcal{M}$), which requires only 1 round (as they are all neighbors to u) and $m^{3/4}$ messages as $|V \setminus V_{\mathsf{Low}}| \leq m/m^{1/4} = m^{3/4}$. If $u \notin V_{\mathsf{High}}$, it would update all its neighbors again in 1 round and $m^{3/4}$ messages as the latter is an upper bound on number of its neighbors.

Part (ii). If $u \notin V_{\mathsf{Low}} \cup V_{\mathsf{Med\text{-}Low}}$ there is nothing to do. Otherwise, u needs to send a message to all its (at most $m^{1/2}$) neighbors that belong to V_{Low} and ask them to relay this information to their neighbors. These vertices can then spend another round to inform all their (at most $m^{1/4}$) neighbors about the status of u. This takes 2 rounds and $m^{1/2} + m^{1/2} \cdot m^{1/4} < 2 \cdot m^{3/4}$ messages in total. □

Lemma 5.2 ensures that Invariants 1 and 2 (the only MIS-related information stored for u beside `MIS-flag`$[u]$ that can be trivially updated) are preserved after any change in $\mathcal{M}$ within a constant number of rounds and $O(m^{3/4})$ messages.

In the following, we briefly describe how to update the information stored for vertices per each topology change in the graph.

Vertex Updates. Let u be the updated vertex. In case of vertex insertion, we simply initialize the data structures at u and we are almost done as the neighbors of u are already informed about u being inserted to the graph and hence can update their information locally. We only need to send `degree`$[u]$ to all the neighbors (the time needed for this can be charged to the initialization cost of this algorithm). Now suppose u is being deleted. We can update `neighbors`$[v]$ and `neighbors-degree`$[v]$ for any neighbor v of u without any communication as they are informed that u is deleted. We can also run UpdateNeighbors(u) (virtually) with no communication as this procedure only informs the neighbors of u that this vertex is being deleted from $\mathcal{M}$ and by knowing that u has left the graph, any vertex v in the neighborhood of u can update `MIS-neighbors`$[v]$ accordingly. Finally, we can also run UpdateTwoHopNeighbors(u) with only 1 round of communication and $m^{3/4}$ messages (see Lemma 5.2) by relaying the information from the neighbors of u (which are informed about u leaving the graph) to their neighbors.

Edge Updates. These updates are handled in the same way as in our sequential algorithm. Let (u, v) be the updated edge. Vertices u and v can update all information except for updating the list `MIS-2hop-neighbors`$[\cdot]$ (in particular `neighbors-degree`$[\cdot]$ can be updated by the procedure described in Section 4.1 with $O(1)$ worst-case round complexity and $O(1)$ amortized message complexity). To do the latter task, vertex u (resp. v) can simulate UpdateTwoHopNeighbors(v) (resp. UpdateTwoHopNeighbors(u)) as described above, which takes $m^{3/4}$ messages and 1 round.

We hence showed that after each change in the topology, all the information stored for vertices can be updated in $O(1)$ rounds and $O(m^{3/4})$ amortized messages.

5.2 The Distributed Algorithm

We design a distributed algorithm for updating $\mathcal{M}$ in the network in the spirit of our update algorithm in Section 4.2. The algorithm is a simple adaption of our sequential algorithm to this dynamic model. For every update, we first perform the steps in the previous section to update the information on every vertex in the graph and then make the network stable again by adjusting $\mathcal{M}$.

Throughout, we aim to maintain the following invariant which is the direct analogue of Invariant 3 in the dynamic setting.

Invariant 4. *Following every vertex/edge update, the set* $\mathcal{M}$ *maintained by the algorithm is an MIS of the input graph. Moreover,*

(i) if only a single vertex leaves $\mathcal{M}$ *then there is no restriction on the number of vertices joining* $\mathcal{M}$ *(which could be zero).*

(ii) if at least two vertices leave $\mathcal{M}$*, then at least twice as many vertices are added to* $\mathcal{M}$*.*

In either case, the worst case number of rounds and messages spent by the algorithm for any update is within, respectively, an $O(1)$ *and an* $O(m^{3/4})$ *factor of the total number of vertices leaving and joining* $\mathcal{M}$*.*

Using the same exact argument as in the proof of Lemma 4.1, maintaining Invariant 4 ensures that the amortized adjustment complexity and amortized round complexity of the algorithm is $O(1)$ and its amortized message complexity is $O(m^{3/4})$. Hence, to prove Lemma 5.1, it suffices to prove that Invariant 4 is preserved after every update. We consider different cases based on insertion and deletion of edges and vertices.

5.2.1 Edge Deletions. Suppose we delete the edge (u, v). We only consider the case that u belongs to $\mathcal{M}$ and v is not; the remaining cases are either symmetric to this one or need no update in $\mathcal{M}$ (see Section 4.2.1). If v is not in V_{Low}, by Invariant 1, it knows all its neighbors in $\mathcal{M}$ and can decide whether to join $\mathcal{M}$ or not to locally; if it enters $\mathcal{M}$, it can update the network in $O(1)$ rounds and $O(m^{3/4})$ messages by Lemma 5.2. If v is in V_{Low}, it first sends a message to all its $O(m^{1/4})$ neighbors and ask for their status to which its neighbors reply whether they belong to $\mathcal{M}$ or not. This only takes 2 rounds and $O(m^{1/4})$ communication and then v can decide again whether to join $\mathcal{M}$ or not to. Note that this part of the result holds even with abrupt deletions.

5.2.2 Edge Insertions. Suppose we insert the edge (u, v). We only consider the case when both u and v belong to $\mathcal{M}$; the remaining cases need no update in $\mathcal{M}$ (see Section 4.2.2). Remember that in Section 4.2.2, we needed to handle these updates in three separate cases. While the algorithm and analysis in each case is different, the procedures needed to carry the information around the network are essentially the same among these cases and hence in the following, for simplicity, we only consider one of the main cases, namely case *1-b* (see Section 4.2.2 for definition of this case). The algorithm in the remaining cases can be adapted to this setting in the same exact way.

Recall that in this case, the vertex u is deleted from $\mathcal{M}$ and moreover u knows the set $L_{\text{1-hop}}$ entirely, which is of size $O(m^{3/4})$.

The general approach is to make u a "coordinator" for running the update algorithm in Section 4.2.2 by communicating with its two-hop neighborhood and gather the necessary information to run the sequential update algorithm.

Vertex u first sends a message to all its neighbors in $L_{\text{1-hop}}$ and asks for their status to which they respond whether or not they belong to $\mathcal{M}$. This takes 2 rounds and $O(m^{3/4})$ messages. Next, u informs one of its neighbors that it can join $\mathcal{M}$ and this new vertex updates its status and the information in the graph which takes $O(1)$ rounds and $O(m^{3/4})$ messages by Lemma 5.2. After this, u again sends a message to all its neighbors in $L_{\text{1-hop}}$ and asks for their status in $\mathcal{M}$ to which they respond whether they belong to $\mathcal{M}$ or whether one of their neighbors in $L_{\text{1-hop}}$ has been added to $\mathcal{M}$ in this step. Then, again, u informs one of its neighbors (if such exists) that it can join $\mathcal{M}$, and continues. This way, we only spend $O(1)$ rounds and $O(m^{3/4})$ communication per each vertex entering $\mathcal{M}$ in addition to $O(1)$ rounds and $O(m^{3/4})$ communication for communicating with neighbors of u that would not join $\mathcal{M}$ eventually.

After processing the list $L_{\text{1-hop}}$, we also need to delete from $\mathcal{M}$, the set of vertices in V_{High} that are now incident to vertices in $L_{\text{1-hop}}$ that just joined $\mathcal{M}$. Note that such vertices are necessarily in the two-hop neighborhood of u and hence u can communicate with them (which are only $O(m^{1/4})$ many) in $O(1)$ rounds and use the above idea to implement the same update algorithm in Section 4.2.2 in this model. This allows us to preserve Invariant 4 by the same exact analysis in Section 4.2.2.

5.2.3 Vertex Deletions. This case is essentially equivalent to the edge insertion case discussed above. Since we have a graceful deletion, we can treat the deleted vertex the same way as in Section 5.2.2 by deleting it from $\mathcal{M}$ (if it belonged to it) and using it as the "coordinator" to implement the process described in Section 5.2.2.

5.2.4 Vertex Insertions. The only thing we need to do in this case is to check whether we need to add this new vertex to $\mathcal{M}$ or not. If this vertex is not in V_{Low}, it already knows this information and hence can decide whether or not to join $\mathcal{M}$; after that we are done. Otherwise, if the vertex belongs to V_{Low}, it sends a message to all its $O(m^{1/4})$ neighbors and ask for their status in $\mathcal{M}$, and use that to decide about joining $\mathcal{M}$. In either case, we only need $O(1)$ rounds and $O(m^{1/4})$ total communication. After this, we update the neighbors using first part of Lemma 5.2 in $O(m^{3/4})$ communication and $O(1)$ rounds.

To conclude, we showed that Invariant 4 is preserved after any edge or vertex insertion or deletion by the distributed algorithm, hence proving Lemma 5.1. We are now ready to prove Theorem 2.

Proof of Theorem 2. The proof is identical to the proof of Theorem 1. The only difference is that in this distributed setting, we are not able to maintain the exact number of edges in the graph in a distributed fashion across all vertices. However, recall that we assumed vertices affected by an update in the topology know the index of this update, i.e., how many updates have happened before this one. Hence, whenever the number of updates reaches $\Omega(m)$, any vertex that knows this information sends a message to all its neighbors to terminate the process which would then be broadcast across the whole graph. This takes $O(m)$ rounds and $O(m)$ communication and can be charged to the total number of updates, i.e., $\Omega(m)$ in this step. Hence, the vertices can initialize their data structure using the new choice of m and continue the distributed algorithm in Lemma 5.1. □

REFERENCES

[1] Ittai Abraham, Shiri Chechik, and Sebastian Krinninger. 2017. Fully dynamic all-pairs shortest paths with worst-case update-time revisited. In *Proceedings of the 28th Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2017, Barcelona, Spain, January 16-19, 2017*. 440–452.

[2] Noga Alon, László Babai, and Alon Itai. 1986. A Fast and Simple Randomized Parallel Algorithm for the Maximal Independent Set Problem. *J. Algorithms* 7, 4 (1986), 567–583.

[3] Luis Barba, Jean Cardinal, Matias Korman, Stefan Langerman, André van Renssen, Marcel Roeloffzen, and Sander Verdonschot. 2017. Dynamic Graph Coloring. In *Proceedings of the 15th International Symposium on Algorithms and Data Structures, WADS 2017, St. John's, NL, Canada, July 31 - August 2, 2017*. 97–108.

[4] Leonid Barenboim, Michael Elkin, and Fabian Kuhn. 2014. Distributed $(\Delta + 1)$-Coloring in Linear (in Δ) Time. *SIAM J. Comput.* 43, 1 (2014), 72–95.

[5] Leonid Barenboim, Michael Elkin, Seth Pettie, and Johannes Schneider. 2016. The Locality of Distributed Symmetry Breaking. *J. ACM* 63, 3 (2016), 20:1–20:45.

[6] Leonid Barenboim and Tzalik Maimon. 2017. Fully-Dynamic Graph Algorithms with Sublinear Time Inspired by Distributed Computing. In *Proceedings of the International Conference on Computational Science, ICCS 2017, Zurich, Switzerland, June 12-14, 2017*. 89–98.

[7] Surender Baswana, Manoj Gupta, and Sandeep Sen. 2011. 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*. 383–392 (see also *SICOMP'15* version, and subsequent erratum).

[8] Aaron Bernstein and Shiri Chechik. 2016. Deterministic decremental single source shortest paths: beyond the $O(mn)$ bound. In *Proceedings of the 48th Annual ACM SIGACT Symposium on Theory of Computing, STOC 2016, Cambridge, MA, USA, June 18-21, 2016*. 389–397.

[9] Aaron Bernstein and Liam Roditty. 2011. Improved Dynamic Algorithms for Maintaining Approximate Shortest Paths Under Deletions. In *Proceedings of the 22nd Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2011, San Francisco, CA, USA, January 23-25, 2011*. 1355–1365.

[10] Aaron Bernstein and Cliff Stein. 2015. Fully Dynamic Matching in Bipartite Graphs. In *Proceedings of the 42nd International Colloquium on Automata, Languages, and Programming, ICALP 2015, Kyoto, Japan, July 6-10, 2015, Part I*. 167–179.

[11] Aaron Bernstein and Cliff Stein. 2016. Faster Fully Dynamic Matchings with Small Approximation Ratios. In *Proceedings of the 27th Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2016, Arlington, VA, USA, January 10-12, 2016*. 692–711.

[12] Sayan Bhattacharya, Deeparnab Chakrabarty, and Monika Henzinger. 2017. 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*. 86–98.

[13] Sayan Bhattacharya, Deeparnab Chakrabarty, Monika Henzinger, and Danupon Nanongkai. 2018. Dynamic Algorithms for Graph Coloring. In *Proceedings of the 29th Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2018, New Orleans, LA, USA, January 7-10, 2018*. 1–20.

[14] Sayan Bhattacharya, Monika Henzinger, and Giuseppe F. Italiano. 2015. 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*. 785–804.

[15] Sayan Bhattacharya, Monika Henzinger, and Danupon Nanongkai. 2016. New deterministic approximation algorithms for fully dynamic matching. In *Proceedings of the 48th Annual ACM SIGACT Symposium on Theory of Computing, STOC 2016, Cambridge, MA, USA, June 18-21, 2016*. 398–411.

[16] Sayan Bhattacharya, Monika Henzinger, and Danupon Nanongkai. 2017. 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*. 470–489.

[17] Keren Censor-Hillel, Elad Haramaty, and Zohar S. Karnin. 2016. 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*. 217–226.

[18] Stephen A. Cook. 1983. An Overview of Computational Complexity. *Commun. ACM* 26, 6 (1983), 400–408.

[19] Camil Demetrescu and Giuseppe F. Italiano. 2004. A new approach to dynamic all pairs shortest paths. *J. ACM* 51, 6 (2004), 968–992.

[20] David Eppstein, Zvi Galil, Giuseppe F. Italiano, and Amnon Nissenzweig. 1997. Sparsification - a technique for speeding up dynamic graph algorithms. *J. ACM* 44, 5 (1997), 669–696.

[21] Shimon Even and Yossi Shiloach. 1981. An On-Line Edge-Deletion Problem. *J. ACM* 28, 1 (1981), 1–4.

[22] Greg N. Frederickson. 1985. Data Structures for On-Line Updating of Minimum Spanning Trees, with Applications. *SIAM J. Comput.* 14, 4 (1985), 781–798.

[23] Mohsen Ghaffari. 2016. An Improved Distributed Algorithm for Maximal Independent Set. In *Proceedings of the 27th Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2016, Arlington, VA, USA, January 10-12, 2016.* 270–277.

[24] Anupam Gupta, Ravishankar Krishnaswamy, Amit Kumar, and Debmalya Panigrahi. 2017. 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.* 537–550.

[25] Manoj Gupta and Richard Peng. 2013. Fully Dynamic $(1 + \epsilon)$-Approximate Matchings. In *Proceedings of the 54th IEEE Annual Symposium on Foundations of Computer Science, FOCS 2013, Berkeley, CA, USA, October 26-29, 2013.* 548–557.

[26] Monika Henzinger, Sebastian Krinninger, and Danupon Nanongkai. 2014. Decremental Single-Source Shortest Paths on Undirected Graphs in Near-Linear Total Update Time. In *Proceedings of the 55th IEEE Annual Symposium on Foundations of Computer Science, FOCS 2014, Philadelphia, PA, USA, October 18-21, 2014.* 146–155.

[27] Monika Henzinger, Sebastian Krinninger, and Danupon Nanongkai. 2014. Sublinear-time decremental algorithms for single-source reachability and shortest paths on directed graphs. In *Proceedings of the 46th Annual ACM on Symposium on Theory of Computing, STOC 2014, New York, NY, USA, May 31 - June 03, 2014.* 674–683.

[28] Monika Henzinger, Sebastian Krinninger, Danupon Nanongkai, and Thatchaphol Saranurak. 2015. Unifying and Strengthening Hardness for Dynamic Problems via the Online Matrix-Vector Multiplication Conjecture. In *Proceedings of the 47th Annual ACM on Symposium on Theory of Computing, STOC 2015, Portland, OR, USA, June 14-17, 2015.* 21–30.

[29] Monika Rauch Henzinger and Valerie King. 1997. Maintaining Minimum Spanning Trees in Dynamic Graphs. In *Proceedings of the 24th International Colloquium on Automata, Languages, and Programming, ICALP 1997, Bologna, Italy, July 7-11, 1997.* 594–604.

[30] Monika Rauch Henzinger and Valerie King. 1999. Randomized Fully Dynamic Graph Algorithms with Polylogarithmic Time per Operation. *J. ACM* 46, 4 (1999), 502–516.

[31] Jacob Holm, Kristian de Lichtenberg, and Mikkel Thorup. 2001. Poly-logarithmic deterministic fully-dynamic algorithms for connectivity, minimum spanning tree, 2-edge, and biconnectivity. *J. ACM* 48, 4 (2001), 723–760.

[32] Zoran Ivković and Errol L. Lloyd. 1993. Fully Dynamic Maintenance of Vertex Cover. In *Proceedings of the 19th International Workshop on Graph-Theoretic Concepts in Computer Science, WG 1993, Utrecht, The Netherlands, June 16-18, 1993.* 99–111.

[33] Richard M. Karp and Avi Wigderson. 1985. A Fast Parallel Algorithm for the Maximal Independent Set Problem. *J. ACM* 32, 4 (1985), 762–773.

[34] Valerie King. 1999. Fully Dynamic Algorithms for Maintaining All-Pairs Shortest Paths and Transitive Closure in Digraphs. In *Proceedings of the 40th Annual Symposium on Foundations of Computer Science, FOCS 1999, New York, NY, USA, October 17-18, 1999.* 81–91.

[35] Nathan Linial. 1987. Distributive Graph Algorithms-Global Solutions from Local Data. In *Proceedings of the 28th IEEE Annual Symposium on Foundations of Computer Science, FOCS 1987, Los Angeles, CA, USA, October 27-29, 1987.* 331–335.

[36] Michael Luby. 1986. A Simple Parallel Algorithm for the Maximal Independent Set Problem. *SIAM J. Comput.* 15, 4 (1986), 1036–1053.

[37] Ofer Neiman and Shay Solomon. 2013. 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.* 745–754.

[38] Krzysztof Onak and Ronitt Rubinfeld. 2010. Maintaining a large matching and a small vertex cover. In *Proceedings of the 42nd Annual ACM SIGACT Symposium on Theory of Computing, STOC 2010, Cambridge, MA, USA, June 6-8, 2010.* 457–464.

[39] Alessandro Panconesi and Aravind Srinivasan. 1996. On the Complexity of Distributed Network Decomposition. *J. Algorithms* 20, 2 (1996), 356–374.

[40] Merav Parter, David Peleg, and Shay Solomon. 2016. 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.* 220–239.

[41] David Peleg. 2000. *Distributed Computing: A Locality-Sensitive Approach.* Society for Industrial and Applied Mathematics (SIAM).

[42] Liam Roditty and Uri Zwick. 2011. On Dynamic Shortest Paths Problems. *Algorithmica* 61, 2 (2011), 389–401.

[43] Shay Solomon. 2016. 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.* 325–334.

[44] Mikkel Thorup. 2005. Worst-case update times for fully-dynamic all-pairs shortest paths. In *Proceedings of the 37th Annual ACM SIGACT Symposium on Theory of Computing, STOC 2005, Baltimore, MD, USA, May 21-24, 2005.* 112–119.

[45] Christian Wulff-Nilsen. 2017. Fully-dynamic minimum spanning forest with improved worst-case update time. In *Proceedings of the 49th Annual ACM SIGACT Symposium on Theory of Computing, STOC 2017, Montreal, QC, Canada, June 19-23, 2017.* 1130–1143.