Near-Optimal Distributed Implementations of Dynamic Algorithms for Symmetry Breaking Problems

Shiri Antaki
Tel Aviv University, Tel Aviv, Israel

Quanquan C. Liu
Massachusetts Institute of Technology, Cambridge, MA, USA

Shay Solomon
Tel Aviv University, Tel Aviv, Israel

Abstract

The field of dynamic graph algorithms aims at achieving a thorough understanding of real-world networks whose topology evolves with time. Traditionally, the focus has been on the classic sequential, centralized setting where the main quality measure of an algorithm is its update time, i.e. the time needed to restore the solution after each update. While real-life networks are very often distributed across multiple machines, the fundamental question of finding efficient dynamic, distributed graph algorithms received little attention to date. The goal in this setting is to optimize both the round and message complexities incurred per update step, ideally achieving a message complexity that matches the centralized update time in $O(1)$ (perhaps amortized) rounds.

Toward initiating a systematic study of dynamic, distributed algorithms, we study some of the most central symmetry-breaking problems: maximal independent set (MIS), maximal matching/(approx-) maximum cardinality matching (MM/MCM), and $(\Delta + 1)$-vertex coloring. This paper focuses on dynamic, distributed algorithms that are deterministic, and in particular – robust against an adaptive adversary. Most of our focus is on our MIS algorithm, which achieves $O\left(\frac{m^2}{3\log^2 n}\right)$ amortized messages in $O\left(\log^2 n\right)$ amortized rounds in the Congest model. Notably, the amortized message complexity of our algorithm matches the amortized update time of the best-known deterministic centralized MIS algorithm by Gupta and Khan [SOSA’21] up to a polylog $n$ factor. The previous best deterministic distributed MIS algorithm, by Assadi et al. [STOC’18], uses $O\left(\frac{m^{3/4}}{\log^2 n}\right)$ amortized messages in $O(1)$ amortized rounds, i.e., we achieve a polynomial improvement in the message complexity by a polylog $n$ increase to the round complexity; moreover, the algorithm of Assadi et al. makes an implicit assumption that the network is connected at all times, which seems excessively strong when it comes to dynamic networks. Using techniques similar to the ones we developed for our MIS algorithm, we also provide deterministic algorithms for MM, approximate MCM and $(\Delta + 1)$-vertex coloring whose message complexities match or nearly match the update times of the best centralized algorithms, while having either constant or polylog($n$) round complexities.

2012 ACM Subject Classification Theory of computation → Distributed algorithms; Theory of computation → Dynamic graph algorithms

Keywords and phrases dynamic graph algorithms, distributed algorithms, symmetry breaking problems, maximal independent set, matching, coloring

Digital Object Identifier 10.4230/LIPIcs.ITCS.2022.7


Funding Shay Solomon is supported by the Israel Science Foundation (ISF) grant No.1991/1, and by a grant from the United States-Israel Binational Science Foundation (BSF), Israel, and the United States National Science Foundation (NSF).

Acknowledgements We are grateful to Boaz Patt-Shamir for useful comments on an earlier version of this paper that helped improve its presentation.

© Shiri Antaki, Quanquan C. Liu, and Shay Solomon; licensed under Creative Commons License CC-BY 4.0

13th Innovations in Theoretical Computer Science Conference (ITCS 2022).

Editor: Mark Braverman; Article No. 7; pp. 7:1–7:25

Leibniz International Proceedings in Informatics

Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany
Traditional graph algorithms process a static graph on a single (centralized) machine and are sequential; thus, their runtime is at least linear in the graph size. Even linear-time static algorithms, which are traditionally considered extremely fast and optimal, are often inadequate for coping with modern big data, which dynamically changes and evolves at a rapid pace. Often, such big data cannot be stored in one machine and hence distributed methods are required to process it. Efficiently coping with such data is widely recognized as one of the most important challenges of modern computation. A dynamic graph algorithm is one that efficiently deals with rapid changes to the input graph, where a common goal is to maintain a subgraph with some key property while the underlying graph changes over time. A distributed graph algorithm is one that efficiently deals with graph data stored in multiple machines, where the corresponding processors work in parallel in order to achieve a common goal by communicating and coordinating their actions via message passing.

The focus of dynamic graph algorithms, up until the past few years, has almost exclusively been in the classic sequential, centralized setting, where the main quality measure is the algorithm’s update time, i.e., the time needed to update the graph structure of interest per update step. Meanwhile, the focus of distributed algorithms has been mainly on the static setting, primarily on the round complexity of static tasks. Surprisingly, the fundamental question of distributing known sequential dynamic graph algorithms received very little attention to date. Most previous works [3, 5, 6, 9, 10, 20, 17, 26, 28, 23] focused on small amortized round complexity. A few works considered message complexity [3, 28] but only for certain problems.

Minimizing the number of messages is an important goal. A small number of messages implies a small load on the communication links, which in some cases enables running multiple algorithms (or multiple instances of the same algorithm) concurrently and it can allow for pipeline implementation. Moreover, this measure is useful when considering cases where one cares about the total work done (as captured by the number of messages sent), e.g. when the total network bandwidth is limited. We remark that many real-world systems are often bandwidth limited; examples of bandwidth limited systems include systems with poor wireless connections, an over-saturated network (many independent agents are on the same network, as in, e.g., a large company), or a mobile data network in a poorly connected area. On a low bandwidth network, an algorithm which uses less rounds but more messages in principle can actually take longer to finish its execution than an algorithm that uses less messages but more rounds.

We define the message-efficiency of our algorithms to be the ratio between the amortized message complexity per update of our algorithm and the sequential amortized update time of the best-known centralized algorithm. Note that although we consider message-efficiency as a separate property, we still want our algorithms to run in $\polylog n$ or $O(1)$ rounds, as is standard. Our goal is to design distributed algorithms with (nearly) constant message-efficiency, meaning the amortized message complexity asymptotically matches the amortized update time of the best centralized algorithm for the problem. Of course, at the same time, we want to upper bound the amortized number of rounds of such algorithms by (nearly) constant. We allow $\polylog n$ slacks in both the message and round complexities.

In this paper, we aim at initiating a systematic study of dynamic, distributed message-efficient algorithms by considering a single edge update per step, as in the classic, sequential centralized setting. Even for this basic setting, there are many challenges underlying the adaption of centralized algorithms to the distributed setting. Very recently, Censor-
Hillel et al. [9] and Bamberger et al. [5] studied the question of simultaneously handling concurrent updates in a distributed setting. However, their algorithms require $\Omega(m)$ amortized messages. Thus, it seems crucial to first thoroughly understand the base case of a single update and only later extend to more general settings. We focus on classic symmetry-breaking problems: maximal independent set (MIS), maximal matching (MM), $3/2$-maximum cardinality matching (MCM), and $(\Delta + 1)$-vertex coloring. Symmetry-breaking constitutes one of the most important challenges in distributed computing, since in many distributed systems processors might be in the same state, yet one must somehow break the symmetry to perform almost any nontrivial computation.

One challenge in optimizing the message-efficiency is that many centralized algorithms for these problems rely on global variables and data structures, and they often rely on periodic \emph{global restarts}. The idea behind a periodic global restart is for the algorithm to handle some number of updates “lazily”, essentially without changing the data structure, until a sufficient number of updates have been accumulated. Such centralized approaches are problematic in the distributed setting as each individual vertex does not have access to global information. Moreover, many centralized algorithms [3, 13, 15, 19, 25] also assume knowledge of the number of edges in the graph at any time, which does not lose generality in a centralized setting but is a very strong assumption to make in (dynamic) distributed settings – such information can be acquired, but through potentially expensive communication. As a result, distributing centralized dynamic algorithms is a challenging task.

We shall restrict our attention to the standard CONGEST model of communication ([27]), where message size is bounded by $O(\log n)$ bits. In particular, we use the most studied model of the distributed dynamic setting, the \emph{local wakeup} (CONGEST) model (cf. [3, 10, 17, 26, 28]), where following an edge update $(u, v)$, only the updated vertices $u$ and $v$ wake up. The update procedure proceeds in fault-free synchronous rounds during which every processor that has been woken up is allowed to exchange $O(\log n)$-bit messages with its neighbors until finishing its execution, which differs from the standard static setting in a crucial aspect: in the static setting all the vertices are woken up at the outset and engage in the algorithm, whereas in the dynamic setting a vertex has to be woken up as part of the update procedure – by receiving a message that propagated from either $u$ or $v$ – in order to engage in the update procedure. In particular, to achieve good message-efficiency, vertices cannot blindly participate in the update procedure as even the size of the 2-hop neighborhood of an updated vertex can be large, leading to large message complexity; this poses a highly nontrivial challenge for algorithms which need to maintain some global invariant.

The round and message complexities can be viewed as the “runtime” and “total work” of the algorithm, respectively. Our paper focuses on solving the dynamic MIS problem via a deterministic, CONGEST algorithm that minimizes message complexity and number of rounds using new techniques to resolve the issue surrounding an unknown number of edges in the entire graph. We then use similar techniques to those used in our algorithm for solving MIS to solve the other problems discussed in this paper, namely, MM, $3/2$-approximate MCM, and $(\Delta + 1)$-vertex coloring. For MM, $3/2$-approximate MCM, and $(\Delta + 1)$-coloring, there exist $O(\Delta)$-message, $O(1)$-round naive algorithms where each node queries its immediate neighbors for the desired property (their color, whether they are in a matching...etc.); the goal is beat these algorithms in number of messages in the setting where $\Delta = \Omega(\sqrt{m})$, where $\Delta$ is a fixed upper bound on the maximum degree in the graph. For MM, $(3/2)$-approximate MCM, and $(\Delta + 1)$-coloring, we provide the first non-trivial $O(1)$-message-efficient deterministic, distributed algorithms that match the update times of $O(\sqrt{m})$ of the best-known centralized algorithms [19, 25]. Our algorithm runs in $O(1)$
worst-case rounds for MM and $(\Delta + 1)$-coloring and $O(\log \Delta)$ worst-case rounds for $3/2$-approximate MCM. Furthermore, our algorithms for these problems are simple and, hence, practically implementable.

**Related Work for MIS.** We give a more comprehensive survey of the related work on MIS since this problem has received more attention in the dynamic, distributed setting. The MIS problem has been intensively studied over the years since the celebrated works of [1, 22, 24] from the mid 1980s. Most of the literature on the problem arguably revolves around parallel and distributed settings, perhaps due to the practical applications of MIS in these settings.

In recent years there has been a growing body of work on the problem of maintaining an MIS in dynamic networks [3, 4, 7, 10, 12, 13, 15, 21]. Although most of this work has focused on the standard centralized, sequential setting, Censor-Hillel et al. [10] gave a randomized algorithm for maintaining an MIS against an oblivious adversary in distributed dynamic networks. The distributed algorithm of [10] achieves an (amortized) message complexity of $\Omega(\Delta)$ and an expected round complexity of $O(1)$. König and Wattenhofer [21] gave an algorithm for maintaining an MIS which requires a constant number of rounds, but as opposed to [10], makes the assumptions that a node gracefully leaves the network, and messages may have unbounded size. Also, the number of broadcasts that are done may be large. Assadi et al. [3] showed that their main centralized algorithm for MIS can be naturally adapted into a deterministic distributed algorithm with amortized message complexity $O(\text{min}\{m^{3/4}, \Delta\})$ and an amortized round complexity of $O(1)$. However, this algorithm operates under the assumption that knowledge of up-to-date estimates of $m$ is provided to all vertices; this leads to an implicit assumption that the graph remains connected throughout the course of the algorithm, which is a strong assumption to make, especially in dynamic networks.

In the centralized, sequential setting, the deterministic $O(m^{3/4})$ bound of Assadi et al. [3] for general graphs was improved to $O(m^{2/3})$ by Gupta and Khan [15] and independently to $O(m^{2/3} \sqrt{\log m})$ by Du and Zhang [13]. Allowing randomization against an oblivious adversary, the update time in general graphs was reduced to $O(\sqrt{m})$ [13], further to $\tilde{O}(\text{min}\{m^{1/3}, \sqrt{n}\})$ [4], and ultimately to poly log $n$ [7, 12]. To the best of our knowledge, the only distributed algorithms for the problem are the two mentioned above [3, 10]. That said, it seems rather straightforward to distribute the randomized algorithms with amortized update time $\text{polylog}(n)$ [7, 12], to obtain a distributed algorithm with both message and round complexities bounded by $O(\text{polylog}(n))$. However, the disadvantage of these randomized algorithms is that they crucially require the oblivious adversary assumption. While such an assumption might be fine in the centralized setting, in the distributed setting it is easier for adversaries to corrupt links in the network as well as to corrupt and/or eavesdrop on the messages sent through such links. Thus, in such settings the oblivious adversary assumption seems excessively strong and impractical. In this paper, we shall restrict our attention to deterministic algorithms, which are, in particular, robust against an adaptive adversary. Prior to this work there was no deterministic (or even randomized against a non-oblivious adversary) distributed algorithm for MIS with a message complexity of $o(m^{3/4})$, and moreover, it seemed highly unclear if the deterministic algorithms of [13, 15], with amortized update time $\tilde{O}(m^{2/3})$ could be distributed efficiently.

---

1 In the oblivious adversarial model (see, e.g., [8, 18]), the adversary has complete knowledge of all the edges in the graph and their arrival order, as well as of the algorithm, but does not have access to the random bits used by the algorithm, and so cannot adapt its updates in response to the random choices of the algorithm.

2 We did not make any effort to verify this claim, as our goal was to achieve an efficient deterministic algorithm, or at least one that does not assume an oblivious adversary.
1.1 Our Contributions

Maximal independent set. We present a deterministic distributed algorithm that achieves amortized message and round complexities of $O(m^{2/3} \log^2 n)$ and $O(\log^3 n)$, respectively. To this end, we reduce the problem of dynamically maintaining an MIS to that of statically computing it. Our reduction builds on the aforementioned (centralized, sequential) algorithm of Gupta and Khan [15] in a nontrivial way.

▶ Theorem 1. Equipped with a black-box static deterministic algorithm for computing an MIS within $T(n)$ rounds for any $n$-vertex distributed network, an MIS can be maintained deterministically (in the local wakeup model) over any sequence of edge insertions and deletions that start from an empty distributed network on $n$ vertices, within $O(T(n))$ amortized round complexity and $O(m^{2/3} \cdot T(n))$ amortized message complexity, where $m$ denotes the dynamic number of edges.

Using the MIS algorithm of [14, 29], which runs in $O(\log^5 n)$ rounds, on top of the transformation of Theorem 1, the amortized bounds on the round and message complexities of the resulting distributed algorithm are $O(\log^5 n)$ and $O(m^{2/3} \log^5 n)$, respectively.

While the black-box static MIS algorithm used in the transformation of Theorem 1 applies to arbitrary graphs, bounded diameter graphs admit faster MIS algorithms. Next, we strengthen the transformation of Theorem 1, to achieve a diameter-sensitive transformation. We stress that the algorithm returned as output of this transformation applies to any dynamic graph; the restriction on the diameter is only for the black-box static MIS algorithm. The black-box MIS algorithm should also satisfy another property; given as input an independent set $M' \subseteq V$ of the graph, the output MIS should be a superset of the input set $M'$; we shall call this an input-respecting MIS.

▶ Theorem 2. Equipped with a black-box static deterministic algorithm for computing an input-respecting MIS within $T'(n)$ rounds for any $n$-vertex distributed network with diameter at most 6, an MIS can be maintained deterministically over any sequence of edge insertions and deletions that start from an empty network on $n$ vertices, within $O(T'(n))$ amortized round complexity and $O(m^{2/3} \cdot T'(n))$ amortized message complexity, where $m$ denotes the dynamic number of edges. 3

The MIS algorithm of [11] runs in $O(D \log^2 n)$ rounds in distributed graphs of diameter $D$. We adapt the algorithm of [11] to return an input-respecting MIS. Plugging the resulting MIS algorithm into the transformation of Theorem 2 yields:

▶ Corollary 3. Starting from an empty distributed network on $n$ vertices, an MIS can be maintained deterministically over any sequence of edge insertions and deletions with $O(\log^2 n)$ amortized round complexity and $O(m^{2/3} \log^2 n)$ amortized message complexity.

Our algorithm uses unicast rather than broadcast messages, which allows each processor to communicate differently with each of its neighbors, and, more concretely, to communicate with a subset of its neighbors – otherwise there is no hope to achieve a message complexity of $o(\Delta)$.

Prior to this work, the distributed algorithm of [3] was the only one providing a deterministic algorithm with $o(m)$ amortized message complexity. By allowing the amortized round complexity to grow from constant to a small polylogarithmic factor, we obtain a polynomial

3 For our purposes it suffices to take a constant of 6, but any fixed constant $c \geq 6$ works.
improvement in the message complexity. Perhaps more important than this improvement, in contrast to the work of [3] that relies on the network being connected at all times, our algorithm does not make any assumptions on the network’s connectivity. Assuming that the network is always connected seems to be too much to ask for – particularly for dynamic networks. To cope efficiently with disconnected graphs, our algorithm has to bypass several nontrivial challenges, which are discussed in more detail in the full version of our paper [2].

Other symmetry-breaking problems. We also show the following results for MM, (3/2)-approximate MCM, and (Δ + 1)-vertex coloring, which were not known prior to this paper. All of our results for these problems are $O(1)$-message-efficient, matching the sequential running time of the best centralized deterministic algorithms for these problems [19, 25].

- **Theorem 4.** Starting from an empty distributed network on $n$ vertices, a maximal matching (MM) and a (3/2)-approximate maximum matching can be maintained deterministically over any sequence of edge insertions and deletions with $O(1)$ and $O(\log \Delta)$ rounds, worst-case, respectively, and $O(\sqrt{m})$ worst-case and amortized message complexity, respectively.

- **Theorem 5.** Starting from an empty distributed network on $n$ vertices, a (Δ + 1)-vertex coloring can be maintained deterministically over any sequence of edge insertions and deletions with $O(1)$ rounds, worst-case, and $O(\sqrt{m})$ worst-case message complexity.

### 1.2 Paper Organization

We provide a technical overview of our algorithms in the full version of our paper [2]. Section 2 is devoted to preliminaries. Then, in Section 3 we review the edge orientation technique and present our algorithms for MM and (Δ + 1)-coloring. In Section 4, we present our algorithm for (3/2)-approximate MCM. Finally, in Section 5, we provide our algorithm for MIS.

## 2 Preliminaries

Given an undirected graph $G = (V, E)$, which represents a distributed network, each vertex $v \in V$ has access to an adjacency list of its current neighbors. We let $\deg(v)$ denote the degree of $v$ or the size of $v$’s adjacency list. As commonly assumed, each vertex can distinguish its neighbors via unique IDs assigned to each vertex.

Following standard convention in dynamic graph algorithms, the graph is empty at the outset, and there is a single edge update per step. As mentioned, we consider the local wakeup model where, following an edge update (insertion or deletion), only the two endpoints incident to that edge wake up and may initiate an update procedure. Any other vertex remains asleep until receiving a message from a neighbor, at which stage it can start participating in the update procedure (by exchanging messages with its neighbors and performing other actions). Computation proceeds in synchronous rounds; thus all vertices (those which are awake) know when a round of communication starts and ends, and a vertex cannot respond to messages it received in the same round - it can do so in the next round. Our algorithms operate in the CONGEST model so messages have size $O(\log n)$ bits.

In addition to considering worst-case rounds and messages per update, in this paper, we also consider amortized round complexity (or amortized update time) and amortized message complexity, which bound the average number of communication rounds and $O(\log n)$-bit messages sent, respectively, needed to update the solution per edge update, over a worst-case sequence of updates. In contrast to the static setting, the amortized message complexity of distributed tasks could be sublinear in the graph size, which makes it natural to optimize both measures of amortized round and message complexities rather than just the former.
We consider three different notions of number of edges in the graph: \( m_{\text{max}} \), \( m_{\text{avg}} \) and \( m_{\text{cur}} \). \( m_{\text{max}} \) is the maximum number of edges in the graph over all updates in the update sequence. \( m_{\text{avg}} \) is the average number of updates in the graph over the update sequence. Finally, \( m_{\text{cur}} \) is the current number of edges in the graph after the most recent update. When the particular definition being used is obvious from context, we simply use \( m \).

3 Dynamic MM and \((\Delta + 1)\)-Coloring via Dynamic Edge Orientations

We begin our discussion of our dynamic, distributed algorithms for symmetry-breaking problems with our simple warm-up algorithms for MM and \((\Delta + 1)\)-coloring. We first present the edge orientation technique which is one way of ensuring that vertices only need to send messages to a small enough subset of neighbors upon an update. This technique is based on the idea that for each vertex \( v \), if the edge \((u, v)\) is oriented towards \( v \), then \( v \) has all the relevant information about \( u \). For every edge insertion or deletion that causes \( u \) to change its properties, \( u \) notifies all its outgoing neighbors. In such a case, when \( v \) needs to make a decision using all its neighbors’ properties, it only needs to ask its outgoing neighbors about their properties to have a full picture of all its neighbors. By using the edge orientation technique we can solve problems like dynamic, distributed MM and \((\Delta + 1)\)-coloring. All round and message complexities in this section are in terms of \( m_{\text{cur}} \), the current number of edges in the graph. For convenience, we define \( m = m_{\text{cur}} \) in this section.

3.1 Edge Orientation Algorithm

We orient the edges in the following way: provided an edge insertion \((u, v)\), the edge is oriented from the vertex with lower degree to the vertex with higher degree. If the degrees of the two vertices are equal, then the edge is oriented arbitrarily. Each vertex \( v \) also maintains a number \( p_v \) indicating its degree during the last time it checked the orientation of all its adjacent edges and reoriented any of the checked edges. Initially, we set \( p_v = 1 \) for all vertices. If an edge insertion or edge deletion \((u, v)\) causes \( v \)’s degree to fall outside the range \([p_v/2, 2p_v]\), \( v \) asks its neighbors for their degree. For each neighbor \( w \) of \( v \), \( v \) directs the edge \((w, v)\) from \( w \) to \( v \) if \( \deg(w) \leq \deg(v) \). Otherwise, \( v \) directs the edge \((v, w)\) from \( v \) to \( w \). After performing the reorientation of edges, \( v \) sets \( p_v = \deg(v) \). The reorientation of the edges is done gradually, 20 edges per update step, which means that this reorientation process would be carried over during the course of the next \( \deg(v)/10 \) updates incident to \( v \). This means that we finish updating the changes before the next set of updates causes \( \deg(v) \) to fall outside the range \([p_v/2, 2p_v]\) again.

The pseudocode for this algorithm is given in Algorithm 1 and Algorithm 2. Our edge orientation algorithm satisfies the following crucial invariant and also requires \( O(1) \) message and round complexity, both worst-case.

\begin{itemize}
  \item \textbf{Invariant 1.} For any edge \((u, v)\), if the edge is oriented from \( u \) to \( v \), then \( \deg(u) \leq 4\deg(v) \).
\end{itemize}

We prove in the full version [2] that the invariant is always maintained under the above update algorithm.

\begin{itemize}
  \item \textbf{Observation 6.} An edge \((u, v)\) that was oriented from \( u \) to \( v \) is reoriented only if \( \deg(v) > \deg(u) \).
\end{itemize}

We prove the following properties of the orientation maintained in this manner using Observation 6 in [2]. These properties allow us to obtain our efficient dynamic \((\Delta + 1)\)-coloring and maximal matching results.
Algorithm 1 Orient\_edges(u, v).

Result: Orient the edges properly on an edge insertion or deletion.

1 Input: An edge insertion or deletion (u, v).
2 if \(\text{deg}(v) > \text{deg}(u)\) then
3 Orient the edge from u to v.
4 end
5 else
6 Orient the edge from v to u.
7 end
8 for \(z \in (u, v)\) do
9 if \(\text{deg}(z) > 2p_z\) or \(\text{deg}(z) < p_z/2\) then
10 \(z\) updates \(p_z = \text{deg}(z)\).
11 end
12 if \(z\) is marked as in process of checking reorientation then
13 Reorient\_edges(z)
14 end
15 end

Algorithm 2 Reorient\_edges(z).

Result: Reorient the edges adjacent to \(z\) gradually

1 Input: A vertex \(z\) that needs to reorient its adjacent edges.
2 if all the edges adjacent to \(z\) are marked as checked then
3 Mark \(z\) as done.
4 return
5 end
6 for 20 adjacent edges (z, x) of \(z\) that still might need reorientation do
7 if \(\text{deg}(z) > \text{deg}(x)\) then
8 Orient the edge from \(x\) to \(z\).
9 end
10 else
11 Orient the edge from \(z\) to \(x\).
12 end
13 Mark (z, x) as checked
14 end

Lemma 7. Each vertex \(v\) has at most \(4\sqrt{m}\) outgoing neighbors.

Lemma 8. Each vertex \(u\) has a complete information on the states of all its incoming neighbors after each update.

3.2 Distributed \((\Delta + 1)\)-Coloring

In this section, we maintain a \((\Delta + 1)\)-coloring in the graph using the edge orientation technique we presented in Section 3.1. First, given an edge update, we run our edge orientation algorithm. For each edge \((u, v)\) that was oriented from \(u\) to \(v\) and is reoriented (such that it is now oriented from \(v\) to \(u\)), \(v\) sends its color to \(u\). Therefore, the following property is maintained: for every edge \((u, v)\) that is oriented from \(u\) towards \(v\), \(v\) knows the current color of \(u\). The algorithm works as follows: given an edge insertion \((u, v)\) between two vertices \(u\) and \(v\) with the same color, pick one of the two vertices arbitrarily, w.l.o.g. \(u\), to recolor. \(u\) queries all its outgoing neighbors for their colors. Then, \(u\) picks a color that does not conflict with any of its neighbors. This can be done since \(u\) already has complete information about all its incoming neighbors, so by asking its outgoing neighbors it receives information about all its neighbors’ colors. Finally, \(u\) informs all its outgoing neighbors of its new color. All of \(u\)'s outgoing neighbors store \(u\)'s color.
Algorithm 3 \((\Delta + 1)-\text{Coloring}\).

**Result:** Maintains a \((\Delta + 1)-\text{coloring}\) in the graph upon edge insertion or deletion

1. **Input:** An edge insertion or deletion \((u, v)\).
2. Call our edge orientation algorithm on \((u, v)\).

3. **if** there were edge flips **then**
   4. **for** an edge \((z, x)\) that was flipped, where \(z \in \{u, v\}\) **do**
   5. **if** the edge is oriented towards \(x\) **then**
      6. \(z\) updates \(x\) about its color
   7. **end**
   8. **else**
      9. \(x\) updates \(z\) about its color
   10. **end**
   11. **end**

12. **if** \((u, v)\) is an inserted edge **then**
   13. **if** \(u\) and \(v\) have the same color **then**
       14. pick a vertex from \(\{u, v\}\) arbitrarily to recolor (w.l.o.g. \(u\) is picked)
       15. **for** each outgoing neighbor \(w\) of \(u\) **do**
           16. \(u\) asks \(w\) for its color
           17. \(w\) responds to \(u\) with its color
       18. **end**
       19. recall that \(u\) has complete information of all colors of its incoming neighbors (Lemma 8)
          and now has complete information about all its neighbors
       20. \(u\) picks a different color from all its neighbors
       21. **for** each outgoing neighbor \(w\) of \(u\) **do**
           22. \(u\) informs \(w\) about its new color
       23. **end**
   24. **end**
   25. **end**

Because Algorithm 3 uses the edge orientation technique presented above, Invariant 1 and Lemma 8 are maintained after each edge insertion and deletion and the algorithm has the same message and round complexity.

**Theorem 9.** Starting from an empty graph, there exists a dynamic, distributed CONGEST \((\Delta + 1)-\text{coloring}\) algorithm that maintains a valid coloring of the distributed network in \(O(\sqrt{m})\) message complexity and \(O(1)\) round complexity, both worst-case.

### 3.3 Maximal Matching

We shall follow the above approach for \((\Delta + 1)-\text{coloring}\) to maintain a maximal matching under edge updates in \(O(\sqrt{m})\) messages and \(O(1)\) rounds, both worst case, per update. For any edge \((u, v)\) that is oriented from \(u\) to \(v\), \(v\) knows whether \(u\) is currently matched or free. Then, given any update we first perform the edge orientation algorithm. (Any edge reorientation \((v, u)\) causes \(v\) to send a message to \(u\) indicating whether it is matched or free.) For an edge insertion, no additional updates need to be made. For an edge deletion \((u, v)\), if the deleted edge was an edge in the matching, then we do the following for \(u\) and after that for \(v\); we next describe how to handle \(u\) but \(v\) should be handled in the same way. \(u\) first checks whether any of its incoming neighbors are free. If any are free, \(u\) arbitrarily picks such an incoming neighbor to match with. If no incoming neighbors of \(u\) are free, \(u\) asks its outgoing neighbors whether they are free (and its outgoing neighbors sends back their answer). If any are free, \(u\) arbitrarily picks a neighbor to be matched with. This algorithm allows us to achieve the same message and round complexity as our \((\Delta + 1)-\text{coloring}\) algorithm. The pseudocode for our algorithm is provided in Algorithm 4.
Algorithm 4 Maximal Matching.

**Result:** Maintains a maximal matching in the graph following any edge update

1. **Input:** An edge insertion or deletion \((u, v)\).
2. Call our edge orientation algorithm on \((u, v)\).
3. **if** there were edge flips **then**
   4. **for** an edge \((z, x)\) that was flipped, where \(z \in \{u, v\}\) **do**
      5. **if** the edge is oriented towards \(x\) **then**
         6. \(z\) updates \(x\) about its current status (whether it is matched or free)
      7. **end**
   8. **else**
      9. \(x\) updates \(z\) about its current status
   10. **end**
   11. **end**
4. **end**
5. **if** \((u, v)\) is a deleted edge **then**
6. **if** \(u\) and \(v\) were matched **then**
   7. First do the following algorithm on \(u\) and then on \(v\) (to avoid the situation which both of them choose the same neighbor to be matched with)
   8. **for** \(z \in \{u, v\}\) **do**
      9. \(z\) informs all its outgoing neighbors that it is unmatched
   10. **if** there exists an incoming neighbor \(x\) of \(z\) that is unmatched **then**
       11. match \(z\) with \(x\)
       12. \(x\) informs all its outgoing neighbors that it is matched
       13. \(z\) informs all its outgoing neighbors that it is matched
       14. **end**
   15. **else**
      16. **for** each outgoing neighbor \(w\) of \(z\) **do**
         17. \(z\) asks \(w\) if it is unmatched
         18. \(w\) responds to \(z\) if it’s unmatched
      19. **end**
   20. **if** there exists an unmatched outgoing neighbor \(w\) of \(z\) **then**
       21. match \(z\) with \(w\)
       22. \(w\) informs all its outgoing neighbors that it is matched
       23. \(z\) informs all its outgoing neighbors that it is matched
       24. **end**
   25. **end**
26. **end**
4. **end**

▶ **Theorem 10.** Starting from an empty graph, there exists a dynamic, distributed CONGEST MM algorithm that maintains a maximal matching in the distributed network in \(O(\sqrt{m})\) message complexity and \(O(1)\) round complexity, both worst-case.

### 4 3/2-Approximate Maximum Cardinality Matching

In this section, we provide a distributed algorithm that uses \(O(\sqrt{m})\) messages and \(O(\log \Delta)\) rounds in the CONGEST model to maintain a 3/2-approximate maximum cardinality matching (MCM) in the input graph under edge insertions and deletions. Our algorithm is based on the sequential algorithm of [25]. The main challenge we must surmount in the distributed setting is the fact that vertices do not know \(m_{cur}\), the current number of edges in the graph. Unfortunately, unlike the case with MM and \((\Delta + 1)\)-coloring, it is no longer sufficient to bound the number of outgoing edges of a vertex using the edge orientation algorithm. Thus, in this section, we show a technique to differentiate a low-degree vertex from a high-degree vertex using information obtained from the 2-hop neighborhood of a vertex without knowledge of the global \(m_{cur}\). As in Section 3, we define for convenience \(m = m_{cur}\).
The message complexity is given in terms of $m_{\text{avg}}$ since we use amortized complexity in this case. The round complexity is given in terms of $m_{\text{cur}}$ since it is given in worst case complexity.

Let $G = (V, E)$ be an arbitrary graph and let $M$ be an arbitrary matching for $G$. The edges of $M$ are called matched edges and the unmatched edges are the remaining edges $E \setminus M$. An augmenting path with respect to $M$ is a path whose edges alternate between edges in $M$ and edges in $E \setminus M$ which starts and ends on different free vertices (i.e. vertices which are not matched). It is well-known that if $G$ does not have an augmenting path of length 3, then $M$ is a 3/2-approximate MCM [16]. The natural algorithmic idea is to exclude all augmenting paths of length at most 3 from the graph.

Ideally, we would like to use the edge orientation technique to efficiently maintain a 3/2-approximate MCM as used for MM and $(\Delta + 1)$-coloring. We know how to use the edge orientation technique to efficiently maintain a maximal matching. However, to determine whether or not there is an augmenting path of length 3 in the graph, and if so to find it, the vertices adjacent to the edge update must have updated information not only about their neighbors, but also about their neighbors’ neighbors, which makes the edge orientation technique insufficient for solving this problem. To find an augmenting path, if exists, we first partition (using the procedure explained below) the vertices into low-degree and high-degree vertices based on the threshold of $\Theta(\sqrt{m})$ (high-degree vertices have degree greater than $\sqrt{2m}$ and low-degree vertices have degree less or equal to $\sqrt{2m}$). We ensure that high-degree vertices $v$ would not go through all their neighbors each round to find an augmenting path. We do this by using an algorithm that finds a surrogate for each high-degree vertex $w$ that becomes free. A surrogate is a vertex $v'$ that is matched to a neighbor $v$ of $w$, such that the degree of $v'$ is at most $\sqrt{2m}$. The key observation that was made in [25] is the following: each unmatched high-degree vertex can always find a free neighbor or surrogate in the first $O(\sqrt{m})$ neighbors it queries. Given this fact, we are able to look only at as many neighbors of $w$ as necessary to find a surrogate. However, since we are working in the distributed setting we don’t know the current value of $m$. Therefore, the algorithm works as follows: for a vertex $v$ that became free during an edge update, in each round we “guess” the number of neighbors we would like to check by successively doubling our number of neighbors to check (starting with 1 neighbor), until we find a free neighbor or surrogate or there are no more neighbors to check. Later on we show that this algorithm won’t check more than $O(\sqrt{m})$ neighbors although the algorithm may not have a good estimation on the value of $m$. Using this algorithm we maintain the fact that high degree vertex will always be matched after an update that is incident to it. Furthermore, assuming there were no augmenting paths of length 3 before the update, after the update only vertices that are incident to the edge update or vertices that became free during the update can be part of new augmenting paths of length 3.

For a vertex that incident to an edge update, if it is matched after the update then it cannot be the start or the end of an augmenting path of length 3, this is due to the fact that an augmenting path must start and end with unmatched vertices. This means that a high-degree vertex that is incident to an edge update will not be the start or the end of any augmenting path during this update unless it becomes low-degree. Hence, we only need to look for augmenting paths that start at free low-degree vertices. This leads to a natural procedure: for a free low-degree vertex $u$, we need to look at all its matched neighbors, e.g. $v$, and ask $v$ whether they have a mate $v'$ which has a neighbor, e.g. $w$, that is free. For any such path $(u, v, v', w)$, we remove from the matching the edge $(v, v')$ and add instead the edges $(u, v)$ and $(v', w)$. It is easy to verify (e.g. [25]) that this eliminates all augmenting paths that start or end at $u$. 

ITCS 2022
We describe this algorithm in detail in the full pseudocode provided in the full version of the paper [2]. Here we include the pseudocode for the harder case of an edge deletion (Algorithm 5). For the pseudocode of the insertion case and subprocesses, please refer to our full paper [2]. Our algorithm allows us to obtain our desired result:

**Theorem 11.** Starting from an empty graph, there exists a dynamic, distributed CONGEST \((3/2)\)-approximate MCM algorithm that maintains a \((3/2)\)-approximate MCM in the graph in \(O(\sqrt{m_{avg}})\) amortized message complexity and \(O(\log \Delta)\) worst-case round complexity.

Algorithm 5 \((3/2)\)-approximate MCM: Edge deletion \((u, v)\).

Result: \((3/2)\)-approximate MCM in the graph.

1. **Input:** An edge deletion \((u, v)\).
2. for \(z \in \{u, v\}\) do
3.   if \(u\) and \(v\) were matched to each other then
4.     if \(z\) has at least one free neighbor then
5.       \(z\) chooses a free neighbor \(w\) arbitrarily.
6.       Match\((z, w)\).
7.       Add edge \((z, w)\) to the matching.
8.       \(w\) notifies all its neighbors that it is matched.
9.   else
10.      \(u' = \text{Surrogate}(z)\)
11.      if \(u' \neq \emptyset\) then
12.        //In this case, it is guaranteed that \(z\) is matched and \(u'\) is free.
13.        Match_Surrogate\((u')\)
14.      end
15.    else
16.      \(z\) notifies all its neighbors that it is free.
17.      \(z\) changes its mate: \(\text{mate}_z \leftarrow \emptyset\).
18.      Aug-path\((z)\)
19.    end
20.  end
21.  end
22. else if \(z\) is free
23.   //We want to make \(z\) a matched vertex in case it is high-degree.
24.   then
25.     \(u' = \text{Surrogate}(z)\)
26.     if \(u' \neq \emptyset\) then
27.       Match_Surrogate\((u')\)
28.     end
29.   end
30. end

5 Maximal Independent Set

Building on the techniques used in the previous sections, we now present our main algorithm for dynamic, distributed MIS. Similar to our algorithm for MCM, our algorithm for MIS also needs to partition vertices into low-degree and high-degree vertices without knowledge of \(m\). However, instead of looking at the two-hop neighborhood as in our algorithm for MCM, our algorithm for MIS instead looks at a subset of vertices in the \(6\)-hop neighborhood to determine whether each updated vertex is low-degree or high-degree. Since the procedure for determining this information is much more complicated, we dedicate an entire section to its explanation (Section 5.5).
Our MIS algorithm is first analyzed with respect to the maximum number of edges, denoted by $m_{\text{max}}$, that exist in the graph at any point in time, and later analyzed with respect to the average number of edges, $m_{\text{avg}}$. The latter analysis is much more challenging, since we do not assume that the graph is connected, hence vertices do not have up-to-date estimates of the current number $m$ of edges. To carry out this analysis, we assume that edge updates are tagged with a global timestamp; only the two vertices adjacent to the edge update can read the timestamp. We assume that the timestamps are given by a global running number, where each timestamp designates the number of the current update step. Of course, since the update sequence can be arbitrarily long, the update steps (and timestamps) could become prohibitively large, so the system is allowed to periodically reset the timestamps in order to make sure that each timestamp can be represented via $O(\log(n))$ bits. We do not address here the technical details behind such periodic resets of timestamps, as this may change from one system to another; this optimization lies within the responsibility of the designers of such systems. (The same assumption was made also in the work of [3], even though their work also made an additional connectivity assumption.)

In this section, we present our main deterministic algorithm that maintains an MIS under edge insertions and deletions. Our deterministic distributed algorithm is inspired at a high-level by the sequential algorithm given in [15] that partitions the vertices into high-degree and low-degree vertices, while giving priority to the low-degree vertices to be included in the MIS. Because each vertex does not have access to the precise value of $m$ (neither $m_{\text{max}}$ nor $m_{\text{avg}}$), instead we partition the vertices based on information from the local neighborhood of each vertex. We update the degree designations of each vertex as its local neighborhood changes. We provide parts of our full algorithm in the next few sections; the full algorithm is presented in our full paper [2].

5.1 High-Degree/Low-Degree Partitioning

First, we provide some necessary characteristics and invariants maintained by vertices in our algorithm.

High-Degree and Low-Degree Vertices

Our algorithm ensures that all vertices in the input graph are always partitioned into high-degree and low-degree vertices, denoted by $V_H$ and $V_L$, respectively. We first provide an intuitive definition of these two concepts and then provide the formal definition as it relates to our algorithm.

Intuitive definition based on [15]

In the sequential algorithm of [15], each vertex $v \in V$ that has degree $\text{deg}(v) > m^{2/3}$ in $G$ (where $m$ is the current number of edges in the graph) is labeled as a high-degree vertex; otherwise, it is labeled as a low-degree vertex. Since, in the centralized setting, the algorithm has access to the current number of edges in the graph, such a definition suffices for this setting.

However, in the distributed setting, a vertex does not know the current number of edges in the graph. Thus, we must perform the partition differently in the distributed model.
High-degree/low-degree partitioning in the distributed setting

We provide the partitioning algorithm in terms of $m_{\text{max}}$ in this section and update it accordingly for $m_{\text{avg}}$ in Section 5.6.

Each vertex $v$ stores a counter $\text{deg}(v)$ indicating the degree it thinks is the movement bound for moving from $V_L$ to $V_H$ or vice versa. If a vertex $v$ has degree $\text{deg}(v) > \text{deg}'(v)$ then it labels itself high-degree; otherwise, it labels itself low-degree. We initialize all $\text{deg}'(v)$ values to be $\text{deg}'(v) = 2$ for all $v \in V$ in the beginning when $\text{deg}(v) = 0$ and the graph is empty. We describe how to update $\text{deg}'(v)$ in Section 5.4. Intuitively, we show that $\text{deg}'(v)$ approximates $m_{\text{max}}^{2/3}$. When we consider $m_{\text{avg}}$ in Section 5.6, $\text{deg}'(v)$ will not always approximate $m_{\text{avg}}^{2/3}$ and therefore we instead use the timestamp of the current update to determine when we need to update a vertex’s degree designation (see Section 5.6 for more details).

Let $G_H = (V_H, E_H)$ be the subgraph induced by the high-degree vertices in $G$ and $G_L = (V_L, E_L)$ be the subgraph induced by the low-degree vertices in $G$. In our distributed setting $G_H$ and $G_L$ are composed of vertices which currently (in the present round) consider themselves to be high-degree and low-degree, respectively.

5.2 Algorithm Overview

We consider low-degree vertices and high-degree vertices separately. The general theme of our algorithm is that low-degree vertices are given priority to be in the MIS. They do not care about whether a high-degree neighbor is in the MIS. In other words, a low-degree vertex adds itself to the MIS if and only if it has no low-degree neighbors in the MIS. On the other hand, a high-degree vertex removes itself from the MIS whenever a low-degree neighbor adds itself to the MIS. Thus, low-degree vertices and high-degree vertices follow different algorithms for adding themselves to the MIS. We provide such algorithms in detail in Section 5.3 and Section 5.4. However, with insertions and deletions of edges, $m_{\text{max}}$ may change. We provide a restart procedure that allows vertices to reassign their degree designations when $m_{\text{max}}$ changes by enough. The restart procedure for each vertex checks after every update incident to it to determine whether it is a low-degree or a high-degree vertex. Finally, each vertex $v$ in the graph maintains a counter $c_v$ that indicates the number of $v$’s low-degree neighbors that are in the MIS.

More specifically, our algorithm contains the following procedures (described briefly here and expanded upon in [2]):

1. Edge updates between two vertices, such that one of which is low-degree, are processed following the algorithms in Section 5.3. There are two possible scenarios:
   a. If an edge update causes a low-degree vertex $v$’s counter to become 0, $c_v = 0$, then $v$ must add itself to the MIS. Then, it must inform its high-degree neighbors that it was added to the MIS. All high-degree neighbors are processed according to Section 5.3.1.
   b. If an edge insertion occurs between two low-degree vertices both in the MIS, then one of them, $v$, removes itself from the MIS and informs all neighbors. Then, $v$’s low-degree neighbors are processed using Section 5.3 and $v$’s high-degree neighbors are processed using Section 5.3.2.

2. Edge updates between two vertices, such that both of them are high-degree, are processed in Section 5.4. There are two scenarios:
   a. If any update causes a high-degree vertex to have no neighbors in the MIS, it adds itself to the MIS.
b. If any such update causes any high-degree vertex \( v \) to leave the MIS, then it must call Algorithm 6 to determine which of its neighbors should enter the MIS (with \( v \) as the leader).

3. A restart procedure given by Algorithm 9 is called before handling high-degree vertices. This restart procedure ensures that the number of high-degree vertices remains bounded by \( m^{1/3}_{\text{max}} \).

4. If a low-degree vertex \( v \)'s degree, \( \text{deg}(v) \), exceeds some internally maintained threshold, \( \text{deg}'(v) \), then \( v \) changes its degree designation to high-degree. (Note that \( v \) can become low-degree again via Item 3.)

As defined above, a vertex \( v \) is high-degree if \( \text{deg}(v) > \text{deg}'(v) \).

Our algorithm maintains the following invariants:

- **Invariant 2.** If a high-degree vertex \( u \) is in the MIS then none of its low-degree neighbors \( w \in N_{\text{low}}(u) \) have \( c_w = 0 \).

- **Invariant 3.** Throughout the execution, \( \text{deg}'(v) < 4m^{2/3}_{\text{max}} \) for all \( v \in V \).

Invariant 2 ensures that a high-degree vertex is in the MIS if and only if none of its low-degree vertices can enter the MIS. Invariant 3 helps us maintain that vertices designated as low-degree have degree \( O \left( m^{2/3}_{\text{max}} \right) \).

5.3 Updates on Low-Degree Vertices

We first describe our algorithm on low-degree vertices. Specifically, we describe what happens when an edge insertion or deletion occurs between two vertices where at least one of these two vertices is a low-degree vertex.

Since the graph is initially empty (contains no edges), all vertices are initially low-degree. When a vertex which is high-degree becomes low-degree and vice versa, it immediately notifies all its neighbors of its new designation. When a low-degree vertex becomes high-degree, it further needs to ensure that each of its low-degree neighbors are in the MIS if they can be. In other words, it needs to ensure that none of its low-degree neighbors are depending on it to be in the MIS. We describe this procedure in detail and prove in the full version [2] that this procedure is not too costly.

The full detailed procedure for low-degree vertices, including pseudocode, can be found in the full version of our paper [2]. For the sake of space, we describe the hard case of when a low-degree neighbor enters the MIS. This requires its high-degree neighbors to leave the MIS which could result in other high-degree vertices to enter the MIS.

### 5.3.1 Low-Degree Neighbor Enters the MIS

We assume an edge insertion between a low-degree vertex \( v \) and a high-degree vertex \( u \) falls under this category if \( v \) is in the MIS. A low-degree vertex can also enter the MIS if it moves from \( V_H \) to \( V_L \), or if one of its low-degree neighbors left the MIS due to an update.

Suppose \( u \) is a high-degree vertex that satisfies Invariant 2 and is in the MIS. If a low-degree neighbor \( v \in N_{\text{low}}(u) \) enters the MIS, then \( u \) must leave the MIS and inform all its high-degree neighbors. We use one of two procedures to determine the set of high-degree vertices that must enter the MIS due to \( u \) leaving the MIS. For convenience, we denote the set of high-degree vertices which want to enter the MIS by \( V_H' \).

There are several situations which may cause a low-degree vertex to enter the MIS. Furthermore, our algorithms described below rely on several characteristics of the sets of vertices that are affected by the change. Specifically we define these roles:
1. **Leader** $v$: The leader $v$ coordinates the addition into the MIS of high-degree vertices that want to enter the MIS, so that no collisions occur.

2. **Set $U$**: The set of high-degree vertices that want to enter the MIS.

3. **Set $U'$**: The set of high-degree vertices that want to leave the MIS.

4. **Set $L$**: The set of low-degree neighbors of $v$ that want to enter the MIS after the current update.

We set the above roles as follows for each of the below situations. We consider the roles after performing the restart procedure. This means that no vertices switch from $V_L$ to $V_H$ or vice versa after being assigned to the below roles.

1. **Edge insertion** $(u, v)$ where $u$ is low-degree and $v$ is high-degree and $u, v$ are in the MIS: $u$ is the leader, $U'$ is the set of high-degree neighbors in $N_{\text{high}}(v)$ which have no neighbors in the MIS, $L = \emptyset$.

2. $v$ leaves $V_H$ and moves to $V_L$, then enters the MIS: $v$ is the leader, $U'$ is the set of high-degree neighbors in $N_{\text{high}}(v)$ which are in the MIS, $L = \emptyset$.

3. **Edge insertion** $(u, v)$ where both $u$ and $v$ are low-degree and $u, v$ are in the MIS: w.l.o.g. $v$ leaves the MIS. $v$ is the leader, $U$ consists of the high-degree neighbors in $N_{\text{high}}(v)$ and the set of high-degree neighbors $N_{\text{high}}(w)$ of vertices $w \in U'$ that have no neighbors in the MIS, $L = \emptyset$.

4. $v$ leaves $V_L$ and moves to $V_H$, $v$ was originally in the MIS: Suppose $v$ leaves the MIS when it moves into $V_H$. $v$ is the leader, $U$ consists of $w \in N_{\text{high}}(v)$ and the set of high-degree neighbors $N_{\text{high}}(w)$ of vertices $w \in U'$ that have no neighbors in the MIS, $L = \emptyset$.

5. **Edge deletion between** $(u, v)$ where $u, v$ are low-degree and $u$ is in the MIS, $c_v = 1$: $v$ enters the MIS in this case. $v$ is the leader, $U'$ consists of $w \in N_{\text{high}}(v)$ where $w$ is in the MIS, $U$ is the set of high-degree neighbors $N_{\text{high}}(w)$ of vertices $w \in U'$ that have no neighbors in the MIS, $L = \emptyset$.

### 5.3.1.1 Finding an MIS in the Subgraph Induced by $V_H'$

Suppose we are given some subset of high-degree vertices $V_H' \subseteq V_H$. Our first procedure uses the result of [14] as a black box on the subgraph induced by $V_H'$. To use the result of [14], we must first determine $V_H'$ such that all vertices in $V_H'$ know to participate in running the static MIS procedure to determine the set of vertices from $V_H'$ which would enter the MIS. We first perform the following procedure provided in Algorithm 6 so that all vertices in $V_H'$ receive knowledge of their participation in the MIS algorithm. Then, each vertex in $V_H'$ runs [14], given in Theorem 12, as a black box to obtain the set of vertices which need to enter the MIS.

Our procedures also uses a restart algorithm which we describe in Algorithm 9 of Section 5.5.

**Theorem 12 (Corollary 1.2 [14]).** There exists a deterministic distributed algorithm that computes a maximal independent set in $O(\log^3 n)$ rounds in the CONGEST model.

Since the algorithm of [14] operates in the distributed model where each vertex initially does not know the topology of the graph (except for its set of adjacent neighbors), we can directly apply this algorithm to our induced subgraph $V_H'$ as initially all vertices in $V_H'$ do not know the entire topology of $V_H'$ but do know their set of neighbors (each of which has a unique ID).
Algorithm 6  Find MIS within $V'_H$.

Result: A set of high-degree vertices which enter the MIS.

1. Input: Leader $v$, sets $U$, $U'$ and $L$ as defined above.
2. Suppose vertex $v$ is the designated leader of this procedure.
3. $v$ informs all neighbors it entered/leaves the MIS using $O(1)$-bit messages.
4. If $v$ entered the MIS then
5. Let $B$ be the set of all high-degree neighbors of $v$.
6. end
7. else
8. ▷ If $v$ exited the MIS
9. Recall $L$ is the set of all low-degree neighbors $w$ of $v$ that want to enter the MIS because their $c_w = 0$.
10. Determine vertices in $L$ that enter the MIS.
11. Let $J$ be the set of vertices of $L$ picked to enter the MIS.
12. for each $a \in J$ do
13. Every high-degree neighbor of $a$ becomes part of the set $B$.
14. end
15. for each $u \in U'$ do
16. $u$ leaves the MIS.
17. $u$ informs all its high-degree neighbors that it left the MIS using $O(1)$-bit messages.
18. end
19. end
20. for each $w \in J$ do
21. $u$ enters the MIS.
22. $u$ informs all its neighbors that it entered the MIS using $O(1)$-bit messages.
23. for each $x \in N(w)$ do
24. $x$ increments $c_x = c_x + 1$.
25. end
26. for each $u \in U$ do
27. $u$ informs all its high-degree neighbors that it is part of $V'_H$.
28. All vertices which are in $V'_H$ know which of its neighbors are in $V'_H$.
29. Each vertex $w \in V'_H$ runs the algorithm given by Theorem 12 to find an MIS in the induced subgraph defined by $V'_H$.
30. end
31. end}

5.3.1.2 Finding an MIS from a Partial MIS

Our second approach relies on an input-respecting subroutine that obtains an MIS from a partial MIS; although this approach is technically more complex, it obtains a round complexity that is better than the one obtained via our first approach. We use the following algorithm which incorporates the deterministic algorithm of [11] to accomplish this goal. We prove an extension of [11] in Lemma 13 to handle the case when the input is a graph with some number of vertices which are already in the MIS and use this procedure as a black box in our detailed procedure provided in Algorithm 6.

We provide a self-contained description of the algorithm of [11] in the appendix of the full version of our paper [2].

Lemma 13 (Modified Theorem 1.5 [11]). \textit{Given a connected graph }$G = (V, E)$, a leader $v$ which is distance $D$ away from all vertices in the graph and a set of vertices $S \subseteq V$ already in the MIS, there exists a deterministic MIS algorithm that finds an MIS among all vertices $V \setminus S$ in $G$ that completes in $O(D \log^2 |V|)$ rounds and sends at most $O(D|E| \log^2 |V|)$ messages in the Congest model.
5.3.2 Low-Degree Neighbor Leaves MIS

We assume an edge deletion between a low-degree vertex $u$ and a high-degree vertex $v$ falls under this category if $u$ is in the MIS. If a low-degree neighbor $u$ leaves the MIS and its high-degree neighbor $v$ does not have any neighbors in the MIS, $v$ enters the MIS and informs all its high-degree neighbors. All high-degree neighbors $N_{high}(v) \cup \{v\}$ perform the restart procedure provided in Algorithm 9.

The remaining scenarios are described in the full version of our paper [2]: edge insertion between two low-degree vertices, edge deletion between two low-degree vertices, edge insertion between one low-degree and one high-degree vertex, edge deletion between one low-degree and one high-degree vertex, and several high-degree neighbors leave the MIS.

5.4 Edge Updates Between High-Degree Vertices

In this section, we describe our algorithms for edge updates between two high-degree vertices.

**Edge Insertion Between Two High-Degree Neighbors**

If at most one of the endpoints is in the MIS, both do nothing except for changing their counter. If both high-degree neighbors are in the MIS, one leaves (e.g. $u$) and performs Algorithm 6 with $u$ as the leader.

**Edge Deletion Between Two High-Degree Neighbors**

If w.l.o.g. the degree of one of the vertices (say $v$), $\text{deg}(v)$, becomes less than $\text{deg}'(v)$, then $v$ becomes low-degree and informs all its neighbors. If $v$ has only high-degree neighbors which are in the MIS, $v$ enters the MIS and all its high-degree neighbors perform Algorithm 6 with $v$ as the leader. After that, the update continues as an edge deletion between one high-degree vertex and one low-degree vertex. If the degrees of both of the vertices $u$ and $v$, $\text{deg}(u)$ and $\text{deg}(v)$, become less than $\text{deg}'(v)$ and $\text{deg}'(u)$, respectively, then both of them become low-degree, inform all their neighbors and enter the MIS if needed; the update continues as edge deletion between two low-degree neighbors.

Otherwise, if no degree assignment changes, and neither of the endpoints is in the MIS, nothing happens. If exactly one e.g. $u$ is in the MIS, the other, e.g. $v$, adds itself to the MIS if it has no neighbors in the MIS and informs all its neighbors. Algorithm 9 is called on all neighbors $N_{high}(v) \cup \{v\}$.

5.5 Restart Procedures

We describe here the first of two restart procedures called whenever Algorithm 6 runs or when a high-degree vertex informs its neighbors it entered or left the MIS. This procedure ensures that all high-degree neighbors of such vertices become low-degree if necessary. This might occur if the number of edges in the graph increases over time. We first provide intuition for our algorithm and then a detailed description of it.

5.5.1 Maintaining Degree Bounds under $m_{max}$

5.5.1.1 Intuition

Since the vertices in the graph do not know the number of the edges in the graph, the partition to high-degree vertices and low-degree vertices can be meaningless. Therefore, we guess for each vertex the maximum number of edges to ever exist in the graph, which we
will tighten during the process. We present a restart procedure that maintains Invariant 3 and proves Lemma 15 with $O(1)$ amortized number of rounds and $O(m_{\max}^{2/3})$ amortized communication complexity, where $m_{\max}$ is the maximum number of edges that in the graph throughout the updates. For the remainder of this section, we let $m$ denote $m_{\max}$.

5.5.1.2 The Algorithm

Let $\deg'(v)$ be the approximate movement degree; when a vertex $v$ has degree $\deg(v) \geq \deg'(v)$ it considers itself a high-degree vertex and if $\deg(v) < \deg'(v)$ it considers itself a low-degree vertex. Whenever its degree crosses this threshold it informs all its neighbors of the change. Since the algorithm starts from an empty graph (no edges), for each $v \in V$, $\deg(v) = 0$ and we initialize $\deg'(v) = 2$.

It might be the case that for some vertices $\deg'(v) < m^{2/3}$ and therefore more than $m^{1/3}$ vertices will be in $V_H$. To be able to perform Algorithm 6 or Algorithm 6 and allow high-degree vertices to inform their high-degree neighbors of whether they entered or left the MIS, we must first “clean” $V_H$ and move the unnecessary vertices to $V_L$. This procedure is described below in Algorithm 9, which uses Algorithm 7, and Algorithm 8 as subroutines. Namely, the crux of the subroutines is to determine the total degree of all the vertices in the subgraph and move vertices which have degrees that are too small into $V_L$. Denote by $G' = (V', E')$ the subgraph induced by the vertices that participate in Algorithm 6 or Algorithm 6. Let this subgraph consist of vertices $\{v\} \cup L \cup B \cup W$ such that $v$ is the vertex that started the restart (the leader).

We call Algorithm 7 on $G'$.

Algorithm 7  Construct BFS tree.

---

Result: A BFS tree $T'$ in a carefully chosen subgraph $G'$ of $G$

1: for each $u \in N(v) \cap V'$ (neighbor of $v$ in $G'$) do
2: \hspace{1cm} $v$ sends $O(1)$-bit message to $u$ indicating that $v$ is $u$'s parent in $T'$.
3: end

4: while there exists at least one high-degree vertex in $V'$ that does not yet have a parent do
5: \hspace{1cm} for each $u$ that received a message from its parent in the previous round do
6: \hspace{2cm} $u$ sends $O(1)$-bit message to $w \in N(u) \cap V'$ if $w$ does not have a parent yet, indicating that $u$ is $w$'s parent in $T'$.
7: \hspace{2cm} If $u$ receives multiple parent messages, it arbitrarily chooses one to be its parent.
8: \hspace{2cm} $w$ stores $u$ as its parent.
9: \hspace{2cm} $w$ informs all its neighbors in $G'$ it has a parent.
10: \hspace{1cm} end
11: end
12: The process ends when all the vertices in $G'$ have a parent (except $v$ which is the root).

In Algorithm 8, since each vertex has only one parent (because $T'$ is a BFS tree), the degree of each vertex is added to the sum only once, and therefore the sum is given by $S = \sum_{v \in V'} \deg(v)$.

Note that after performing Algorithm 9, a vertex that had only high-degree neighbors in the MIS might move to $V_L$ and enter the MIS. After that, all its high-degree neighbors must leave the MIS and perform another restart (by calling Algorithm 6 or Algorithm 6). We show in our analysis that this “chain” of restarts is constant on average.

5.5.1.3 Analysis of the Restart Procedure for $m_{\max}$

We provide the full analysis of the restart procedure described above in the full version of our paper [2]. Our analysis uses the following observation and lemmas to prove Invariant 3 holds.
Algorithm 8

**Estimate a lower bound of** $m$ **in** $G'$.

**Result:** A lower bound estimation of $m$.

1. Construct a BFS tree $T'$ according to Algorithm 7 and mark $v$ as the root.
2. For each $u$ which is a child of $v$ in $T'$ do
   1. $v$ sends $O(1)$-bit message to $u$ asking what is the sum of the degrees of all vertices in $u$’s subtree.
3. While exists a vertex $u$ that received a message from its parent in the previous round do
   1. If $u$ has children in $T'$ then
      1. $u$ sends the same message to all its children and waits for a response.
      2. $u$ sums the values it received from its children and adds $\deg(u)$ to the sum.
      3. $u$ sends the sum to its parent.
   2. If $u$ has no children in $T'$ then
      1. $u$ sends $\deg(u)$ to its parent.
4. End
5. If $v$ receives a non-zero sum of the degrees of all its children then
   1. $v$ computes a final sum of the values received from its children and adds $\deg(v)$ to the sum.
      1. Let this sum be $S$.
   2. For each child $u$ of $v$ in $T'$ do
      1. $v$ sends a $O(\log(n))$-bit message to $u$ containing $S$.
   3. For each $u$ that received $S$ from its parent do
      1. $u$ sends $S$ to its children.
4. End
5. End
6. The process ends when all the vertices in $G'$ know $S$.

Algorithm 9

**Restart procedure.**

**Result:** Some vertices $v$ with degree $\deg(v) < 4m^{2/3}$ move to $V_L$ and the number of vertices in remaining $G'$ (to be used in Algorithm 6 or Algorithm 6) is at most $2m^{1/3}$.

1. Let $S$ be the sum calculated in Algorithm 8.
2. Assume as in the previous algorithms $v$ is the leader and root of the tree.
3. $v$ broadcasts using $T'$ and tells all its descendants to become low-degree (i.e. move to $V_L$) if necessary.
4. For each $u$ which is a child of $v$ in $T'$ do
   1. $v$ sends $O(\log(n))$-bit message to $u$ telling it to become low-degree (i.e. move to $V_L$) if $\deg(u) < S^{2/3}$.
5. End
6. For a descendant $u$ of $v$ that got the message do
   1. $u$ propagates the message to its children in $T'$ (if any).
   2. If $\deg(u) < S^{2/3}$ then
      1. $u$ considers itself as a low-degree vertex (i.e. moves to $V_L$).
      2. If $u$ has no low-degree neighbors in the MIS it enters the MIS.
      3. $u$ notifies all its neighbors in $G$ that it is low-degree and if it is in the MIS or not.
   3. $u$ updates $\deg'(u) = 2\deg(u)$.
7. End
8. End

Observation 14. Throughout the execution of Algorithm 9, a vertex $u$ in $G'$ moves from $V_H$ to $V_L$ if and only if its degree $\deg(u)$ is smaller than $S^{2/3}$.

Lemma 15. After performing Algorithm 9, the number of vertices that remain in $G'$ is at most $2m^{1/3}$.

Lemma 16. If a vertex $v \in V'$ moves from $V_H$ to $V_L$ during Algorithm 9, then $\deg(v) < 2m^{2/3}$. 
The following proves Invariant 3 holds.

**Lemma 17.** Throughout all the updates, $\deg'(v) < 4m^{2/3}$ for all $v \in V$ when $m \geq 1$.

### 5.5.2 Runtime Analysis of the Restart Procedure for $m_{\text{max}}$

We first provide a worst case analysis and afterwards provide an amortized analysis. In Algorithm 7, since the diameter of $G'$ is at most 6, building $T'$ takes $O(\text{diameter}) = O(1)$ rounds. In each round each vertex $v$ sends a constant number of messages to all of its neighbors in $G'$, so the total number of messages would be $O(|E'|)$ messages. The same analysis as Algorithm 7 holds for Algorithm 8. As we described in each of these procedures, the messages have size $O(\log n)$.

In Algorithm 9, the broadcast costs $O(\text{diameter}) = O(1)$ rounds and $O(|E'|)$ messages as explained for Algorithm 7. However, each vertex $u$ that moves from $V_H$ to $V_L$ during this algorithm, must check if it needs to enter the MIS and notify all its neighbors about this change.

Let $V_{\text{low}}$ be all the vertices from $V'$ that moved from $V_H$ to $V_L$ during the restart, and let $S' = \sum_{v \in V_{\text{low}}} \deg(v)$ be the sum of their degrees in $G$. All the vertices that move to $V_L$ need to notify their neighbors about the movement and might enter the MIS, however, this should be done sequentially. Therefore, notifying all the neighbors about the change costs at most $O(|V_{\text{low}}|)$ rounds and $O(S')$ messages. Overall we pay for one restart $O(|V_{\text{low}}|)$ rounds and $O(|E'| + S')$ messages.

#### 5.5.2.1 Amortized analysis

We obtain in the worst case that one restart costs $O(|V_{\text{low}}|)$ rounds and $O(|E'| + S')$ messages. We present now the amortized analysis of one restart.

**Lemma 18.** Algorithm 7, Algorithm 8, and Algorithm 9 take $O(1)$ amortized rounds and $O(m^{2/3})$ amortized messages over the set of updates in our graph.

We now analyze how we can pay for a chain of restarts.

**Lemma 19.** All restarts that happen during an update costs $O(1)$ amortized rounds and $O(m^{2/3})$ amortized messages.

#### 5.5.2.2 Analysis of the Main Algorithm under $m_{\text{max}}$

We provide the full analysis of our main algorithm under $m_{\text{max}}$ in [2]. The analysis of our main algorithm follows straightforwardly from Invariant 2 and Invariant 3 and the analysis of our restart procedures above. We obtain the following theorem with respect to $m_{\text{max}}$.

**Theorem 20.** There exists a deterministic algorithm in the CONGEST model that maintains an MIS in a graph $G = (V, E)$ under edge insertions/deletions in $O(\log^2 n)$ amortized rounds and $O(m_{\text{max}}^{2/3} \log^2 n)$ amortized messages per update.

### 5.6 Extending from $m_{\text{max}}$ to the Average Number of Edges $m_{\text{avg}}$

Thus far, our message and round complexity have been in terms of the maximum number of edges present in the graph at any time. In this section, we extend our results to the case of the average number of edges in the graph $m_{\text{avg}}$, such that $m_{\text{avg}}$ is the average number of edges throughout all of the update sequence. We do this by dividing the update sequence
into *phases*, such that if the number of edges in the beginning of phase $i$ is denoted by $m_i$ then the length of phase $i$ is $m_i^{2/3}$ updates (if $m_i^{2/3} < 1$ we assume we have one update at this phase). In each phase, we obtain $\tilde{O}(1)$ amortized rounds and $\tilde{O}(m_i^{2/3})$ amortized messages. This result, using Hölder’s inequality, will imply that the amortized message complexity is $\tilde{O}(m_{\text{avg}}^{2/3})$ (see our full paper [2] for more details).

To obtain this bound, we assume that each edge insertion/deletion is tagged with a *timestamp* indicating the number of edge updates (including itself) that have occurred since the beginning. We assume that when an edge update $(u, v)$ occurs, both $u$ and $v$ receive the timestamp $t_{(u,v)}$ associated with the update. Using this information, we modify our algorithms above for handling updates in the $m_{\text{avg}}$ case.

Each vertex $v$ stores two additional pieces of information, $S_v$ indicating the number of edges it thinks are in the graph and $t_v$ indicating the timestamp of the last time $S_v$ was updated. Initially $S_v$ is set to 0 when the graph is empty. If an edge insertion adjacent to $v$ occurs, and after the update $\text{deg}(v) > S_v$, $v$ updates $S_v$ such that $S_v \geq \text{deg}(v)$ is always satisfied; $t_v$ is updated with the timestamp of the edge insertion that caused $S_v$ to be updated. $S_v$ is *not* updated when an edge deletion that is adjacent to $v$ occurs (if no restart algorithm is called). $S_v$ is also updated with a new value when Algorithm 9 runs on $v$. When Algorithm 9 runs on $v$, $v$ stores $S_v = S$ to be its new estimate of the number of edges in its sub-component and $t_v$ to be the timestamp of the update that caused Algorithm 9 to be called.

**Updated main algorithm (based on Section 5.1 with some minor changes)**

Consider an edge update $(u, v)$ (insertion or deletion) such that at least one vertex from $u$ and $v$ is low-degree, w.l.o.g. say $u$ is low-degree. In this case, if $u$ needs to send a message to its neighbors, $u$ first checks whether $t_{(u,v)} - t_u > S_u/4$ where $t_{(u,v)}$ is the update timestamp it received. If $t_{(u,v)} - t_u > S_u/4$ then $u$ henceforth considers itself a high-degree vertex. Therefore, $u$ notifies all its neighbors about its movement to $V_H$. If $u$ was in the MIS, every neighbor of $u$ that wakes up (say $w$) and needs to enter the MIS, receives $t_{(u,v)}$ from $u$ and first checks whether $t_{(u,v)} - t_w > S_w/4$ (notifying all of $w$’s neighbors the timestamp requires $O(\log n)$-bit messages assuming the timestamps resets to 0 after sufficiently many updates). If so, it follows the same process as $u$. Note that more than one vertex can move from $V_L$ to $V_H$ in a single update. However, we will prove later that we can afford to pay for those movements. Once a vertex moves to $V_H$, it can return to $V_L$ only by a restart, which means that even if $w$ moved to $V_H$ but $\text{deg}(w) \leq \text{deg}'(w)$, $w$ will not return to $V_L$ until a restart. The rest of the process occurs as we described in the main algorithm.

**5.6.1 Updated restart procedure**

Our restart procedure works like the restart procedure given in Section 5.5 except for the follow change: each vertex that moves from $V_H$ to $V_L$, say $w$, not only updates $\text{deg}'(w)$ but also saves $S_w \leftarrow S$ and updates $t_w \leftarrow t_j$, where $t_j$ is the current update’s timestamp. Since we changed the definition of $m$, we also need to update Lemma 15 and Lemma 16 to show the following invariants hold.

- **Invariant 4.** Any vertex adjacent to an edge update that happens during phase $i$ and is low-degree after our updated restart has degree $O(m_i^{2/3})$.
- **Invariant 5.** Any vertex adjacent to an edge update that happens during phase $i$ and is high-degree after our updated restart has at most $O(m_i^{1/3})$ high-degree vertices in its neighborhood when participating in Algorithm 6.
Note that Lemma 17 is not true anymore because $m$ can decrease significantly throughout phases but $\deg'(v)$ can stay the same for more than one phase, and in that case $\deg'(v)$ could be more than $m_i$. Hence, the timestamp that a vertex $v$ receives during its movement from $V_H$ to $V_L$ is critical. Using these invariants, we prove the following lemma.

▶ **Lemma 21.** The average number of low-degree vertices that enter or leave the MIS during an update is constant.

### 5.6.2 Updated analysis of the Restart Procedure for $m_{\text{avg}}$

All proofs can be found in the full version of our paper [2].

▶ **Theorem 22.** During phase $i$ for some $i \geq 0$, the amortized round complexity is $\tilde{O}(1)$ and the amortized message complexity is $\tilde{O}(m_i^{2/3})$ for each update.

To prove Theorem 22, we divide the restart procedure into two kinds of restarts:

- **Heavy restart:** a restart that happens during phase $i$ in which the estimated number of edges in the graph $S > 4m_i^{2/3}$.
- **Light restart:** a restart in which the estimated number of edges in the graph $S \leq 4m_i^{2/3}$.

▶ **Lemma 23.** A light restart takes $O(1)$ amortized rounds and $O(m_i^{2/3})$ messages.

The difficulty with our update procedure for $m_{\text{avg}}$ is that trivially heavy restarts can cost more than $O(m_i^{2/3})$ worst-case, and since we cannot bound the number of heavy restarts during a phase, we might need to pay a lot for heavy restarts during phase $i$. In the next lemma, we prove that a vertex $v$ would participate in a heavy restart only if it could “pay” for it.

▶ **Lemma 24.** If a vertex $v$ participated in a heavy restart $r_a$ during phase $i$ and moved from $V_H$ to $V_L$ during $r_a$, it can move back to $V_H$ and participate in another restart $r_b$ during phase $i$ only if its degree $\deg(v)$ was doubled after $r_a$.

▶ **Lemma 25.** During phase $i$, the cost of a heavy restart is $O(1)$ amortized rounds and $O(m_i^{2/3})$ amortized messages.

By Lemma 23, during phase $i$, we know that a light restart costs $O(1)$ amortized rounds and $O(m_i^{2/3})$ messages and by Lemma 25 we know that a heavy restart also costs $O(1)$ amortized rounds and $O(m_i^{2/3})$ amortized messages.

The last difficulty we encounter is that one restart can cause a chain of restarts. However, recall that after a restart only a vertex that moved from $V_H$ to $V_L$ and entered the MIS can cause another restart. Since the average number of low-degree vertices that enter the MIS during an update is constant (by Lemma 21), we get that the average number of restarts during a phase is also constant and by our lemmas above obtains our desired costs.

Overall, the payment for all the restarts during a phase is $O(1)$ amortized rounds and $O(m_i^{2/3})$ amortized messages.

▶ **Lemma 26.** The total cost of moving vertices from $V_L$ to $V_H$ is amortized $O(1)$ rounds and $O(m^{2/3})$ messages.

By Lemma 26 and since a vertex can from $V_H$ to $V_L$ only through a restart, we get that overall the running time of the movements in phase $i$ is $O(1)$ amortized rounds and $O(m_i^{2/3})$ amortized messages.
5.6.2.1 Analysis of the Main Algorithm under $m_{avg}$

**Theorem 27.** There exists a deterministic algorithm in the CONGEST model that maintains an MIS in a graph $G = (V, E)$ under edge insertions/deletions in $O(\log^2 n)$ amortized rounds and $O(m_{avg}^2 \log^2 n)$ amortized messages per update.

References


