SIAM J. COMPUT.
Vol. 47, No. 3, pp. 617–650
© 2018 Society for Industrial and Applied Mathematics

FULLY DYNAMIC MAXIMAL MATCHING IN $O(\log N)$ UPDATE TIME (CORRECTED VERSION)*

SURENDER BASWANA[†], MANOJ GUPTA[‡], AND SANDEEP SEN[§]

Abstract. We present an algorithm for maintaining a maximal matching in a graph under addition and deletion of edges. Our algorithm is randomized and it takes expected amortized $O(\log n)$ time for each edge update, where n is the number of vertices in the graph. Moreover, for any sequence of t edge updates, the total time taken by the algorithm is $O(t \log n + n \log^2 n)$ with high probability.

Key words. matching, dynamic graph algorithm

AMS subject classifications. 05C70, 05C85, 68W05, 68W20, 68W40

DOI. 10.1137/16M1106158

1. Introduction.

Let $G = (V, E)$ be an undirected graph on $n = |V|$ vertices and $m = |E|$ edges. A matching in G is a set of edges $M \subseteq E$ such that no two edges in M share any vertex. The study of matchings satisfying various properties has remained the focus of graph theory for decades [23]. It is due to the elegant structure of matchings that they also appear in various combinatorial optimization problems [12, 22]. A few well studied matching problems are maximum cardinality matching [11, 19, 24, 25], maximum weight matching [14, 20], minimum cost matching [21, Chapter 7], stable matching [15], popular matching [2]. Among these problems, the maximum cardinality matching problem has been studied most extensively. A matching M is of maximum cardinality if the number of edges in M is maximum among all possible matchings. A maximum cardinality matching is also referred to as a *maximum* matching. A matching is said to be a *maximal* matching if it cannot be strictly contained in any other matching. It is well known that a maximal matching guarantees a 2-approximation of the maximum matching. Though it is quite easy to compute a maximal matching in $O(m + n)$ time, designing an efficient algorithm for maximum matching has remained a very challenging problem for researchers [11, 24]. The fastest algorithm, till date, for maximum matching runs in $O(m\sqrt{n})$ time and is due to Micali and Vazirani [24]. In this article, we address the problem of maintaining a maximal matching in a dynamic graph.

Most of the graph applications in the real world deal with graphs that are not static, viz., the graph changes over time. These changes are deletion and insertion of edges. This has motivated researchers to design efficient algorithms for various graph problems in a dynamic environment. An algorithmic graph problem is modeled in a dynamic environment as follows. There is an online sequence of insertion and deletion of edges, and the goal is to update the solution of the graph problem after each edge update. A trivial and inefficient approach to achieve this goal is to run the

*Received by the editors December 1, 2016; accepted for publication (in revised form) December 28, 2017; published electronically May 3, 2018. The previous version of this result appeared in *SIAM J. Comput.*, 44 (2015), pp. 88–113, where the analysis contained some error. This version rectifies this deficiency without any change in the algorithm while preserving the original performance bounds. For convenience of the reader, we present the complete version of the result although the error affects only some parts of the original article.
http://www.siam.org/journals/sicomp/47-3/M110615.html
[†]Department of CSE, I.I.T. Kanpur, India (sbaswana@cse.iitk.ac.in).
[‡]Department of CSE, I.I.T. Gandhinagar, India (gmanoj@iitgn.ac.in).
[§]Department of CSE, I.I.T. Delhi, India (ssen@cse.iitd.ac.in).

best static algorithm for the problem after each edge update; clearly this approach is quite wasteful. The aim of a dynamic graph algorithm is to maintain some clever data structure for the underlying problem such that the time taken to update the solution is much smaller than that of the best static algorithm. There exist many efficient dynamic algorithms for various fundamental problems in graphs [5, 7, 18, 29, 30, 32].

Baswana, Gupta, and Sen [6] had presented a fully dynamic algorithm for maximal matching which achieved $O(\log n)$ expected amortized time per edge insertion or deletion. Moreover, for any sequence of t edge updates, the total update time taken by the algorithm is $O(t \log n + n \log^2 n)$ with high probability. Their algorithm improved an earlier result of Onak and Rubinfeld [27] who presented a randomized algorithm for maintaining a c-approximate (for some unspecified large constant c) matching in a dynamic graph with $O(\log^2 n)$ expected amortized time for each edge update.

Unfortunately, the analysis given in [6] has a flaw, viz., the statement [6, Lemma 4.10] is not true in general. The readers familiar with [6] may refer to section 7.1 in the appendix to see the flaw in the lemma. The bound on the update time was critically dependent on this lemma. In this article, we present an alternate proof of the claimed update time based on an interesting property of the original algorithm that was not reported earlier. This property, stated in Theorem 4.2, is interesting in its own right and may lead to other useful applications of the algorithm.

For convenience of the reader, we present the complete result of the original article [6] with a revised and correct analysis although the error affects only some parts of it. In other words, this article can be read on its own without any knowledge of the original article [6].

1.1. Organization of the result. In the following section, we describe notations and terminologies used in the rest of the article along with a high level overview of the main idea underlying the algorithm. For a gentle exposition of the ideas and techniques, we describe a simple but less efficient fully dynamic algorithm for maximal matching in section 3. We present our fully dynamic algorithm which achieves expected amortized $O(\log n)$ time per update in section 4. In section 5, we illustrate an example of a dynamic graph that establishes the tightness of the approximation factor guaranteed by our algorithm. We conclude with recent interesting results on fully dynamic maximal matching obtained by other researchers and open problems in section 6.

2. An overview.

Let M denote a matching of the given graph at any moment. Every edge of M is called a *matched* edge and an edge in $E \backslash M$ is called an *unmatched* edge. For an edge $(u, v) \in M$, we define u to be the *mate* of v and vice versa. For a vertex x, if there is an edge incident on it in the matching M, then x is a matched vertex; otherwise it is *free* or unmatched.

In order to maintain a maximal matching, it suffices to ensure that there is no edge (u, v) in the graph such that both u and v are free with respect to the matching M. From this observation, an obvious approach will be to maintain the information for each vertex whether it is matched or free at any stage. When an edge (u, v) is inserted, add (u, v) to the matching if u and v are free. When an unmatched edge (u, v) is deleted, no action is required. Otherwise, for both u and v we search their neighborhoods for any free vertex and update the matching accordingly. It follows that each update takes $O(1)$ computation time except when it involves deletion of a matched edge; in this case the computation time is of the order of the sum of the degrees of the two vertices. So this trivial algorithm is quite efficient for *small* degree vertices, but could be expensive for *large* degree vertices. An alternative approach to handle the deletion of a matched edge is to use a simple randomized technique—a vertex u is matched with a randomly chosen neighbor v. Following the standard adversarial model, it can be observed that an expected $\deg(u)/2$ edges incident on u will be deleted before deleting the matched edge (u, v). So the expected amortized cost per edge deletion for u is roughly $O(\frac{\deg(u)+\deg(v)}{\deg(u)/2})$. If $\deg(v) < \deg(u)$, this cost is $O(1)$; but if $\deg(v) \gg \deg(u)$, then it can be as bad as the trivial algorithm. We combine the idea of choosing a random mate and the trivial algorithm suitably as follows.

We introduce a notion of *ownership* of edges in which we assign an edge to that endpoint which has *higher* degree. We maintain a partition of the set of vertices into two levels : 0 and 1. Level 0 consists of vertices which own *fewer* edges than an appropriate threshold and we handle the updates at level 0 using the trivial algorithm. Level 1 consists of vertices (and their mates) which own a *larger* number of edges and we use the idea of a random mate to handle their updates. In particular, a vertex chooses a random mate from its set of owned edges which ensures that it selects a neighbor having a lower degree. This is an overview of a 2-level algorithm that achieves expected amortized $O(\sqrt{n})$ time per update for maintaining a maximal matching.

A careful analysis of our 2-level algorithm suggests that a *finer* partition of vertices into a greater number of levels may help in achieving a better update time. This leads to our final algorithm which achieves expected amortized $O(\log n)$ time per update. This algorithm maintains an invariant that can be informally summarized as follows.

Each vertex tries to rise to a level higher than its current level, if, upon reaching that level, there are sufficiently many edges incident on it from lower levels. Once a vertex reaches a new level, it selects a random edge from this set and makes it matched.

Note that we say that *"a vertex rises"* to indicate that the vertex moves to a higher level and *"a vertex falls"* to indicate that the vertex moves to a lower level. A vertex may also fall to a lower level after deletion of its matched edge if the number of edges owned by it falls below the threshold for the current level. Overall, the vertices use just their neighborhood information to reach a global equilibrium state after each update, where each vertex is at the *right* level having no incentive to either move above or below its current level.

We shall use $\mathcal{M}$ to denote the matching maintained by our algorithm at any stage. Our algorithm maintains a partition of the set of vertices among various levels. We shall use $\text{level}(u)$ to denote the level of a vertex u. We define $\text{level}(u, v)$ for an edge (u, v) as $\max(\text{level}(u), \text{level}(v))$.

3. Fully dynamic algorithm with expected amortized $O(\sqrt{n})$ time per update. The algorithm maintains a partition of the set of vertices into two levels—0 and 1. We now introduce the concept of *ownership* of the edges. Each edge present in the graph will be owned by one or both of its endpoints as follows. If both the endpoints of an edge are at level 0, then it is owned by both of them. Otherwise it will be owned by exactly that endpoint which lies at a higher level. If both the endpoints are at level 1, the tie will be broken suitably by the algorithm. As the algorithm proceeds, the vertices will make a transition from one level to another and the ownership of edges will also change accordingly. Let $\mathcal{O}_u$ denote the set of edges owned by u at any moment of time. Each vertex $u \in V$ keeps the set $\mathcal{O}_u$ in a dynamic hash table [28] so that each search or deletion operation on $\mathcal{O}_u$ can be performed in worst case $O(1)$ time and each insertion operation can be performed in expected $O(1)$ time. This hash table is suitably augmented with a linked list storing $\mathcal{O}_u$ so that we can retrieve all edges of set $\mathcal{O}_u$ in $O(|\mathcal{O}_u|)$ time.

The algorithm maintains the following three invariants after each update.

1. Every vertex at level 1 is matched. Every free vertex at level 0 has all its neighbors matched.
2. Every vertex at level 0 owns less than $\sqrt{n}$ edges.
3. Both the endpoints of every matched edge are at the same level.

Invariant 1 implies that the matching $\mathcal{M}$ maintained is maximal at each stage. A vertex u is said to be a *dirty* vertex at a moment if it violates at least one of the invariants mentioned above. In order to restore the invariants, each dirty vertex might make a transition to another level and do some processing. This processing involves owning or disowning some edges depending upon whether the level of the vertex has risen or fallen. Thereafter, the vertex will execute RANDOM-SETTLE or NAIVE-SETTLE to *settle down* at its new level. The pseudocode of our algorithm for handling insertion and deletion of an edge is given in Figures 1 and 2.

Handling insertion of an edge. Let (u, v) be the edge being inserted. If either u or v are at level 1, there is no violation of any invariant. So the only processing that needs to be done is to assign (u, v) to $\mathcal{O}_u$ if level$(u) = 1$, and to $\mathcal{O}_v$ otherwise. This takes $O(1)$ time. However, if both u and v are at level 0, then we execute the HANDLING-INSERTION procedure which does the following (see Figure 1).

Both u and v become the owner of the edge (u, v). If u and v are free, then the insertion of (u, v) has violated invariant 1 for u as well as v. We restore it by adding (u, v) to $\mathcal{M}$. Note that the insertion of (u, v) also leads to increase of $|\mathcal{O}_u|$ and $|\mathcal{O}_v|$ by one. We process that vertex from $\{u, v\}$ which owns larger number of edges; let u be that vertex. If $|\mathcal{O}_u| = \sqrt{n}$, then invariant 2 has become violated. We execute RANDOM-SETTLE(u); as a result, u moves to level 1 and gets matched to some vertex, say y, selected randomly uniformly from $\mathcal{O}_u$. Vertex y moves to level 1 to satisfy invariant 3. If w and x were, respectively, the earlier mates of u and y at level 0, then the matching of u with y has rendered w and x free. So to restore invariant 1, we execute NAIVE-SETTLE(w) and NAIVE-SETTLE(x). This finishes the processing of insertion of (u, v). Note that when u rises to level 1, $|\mathcal{O}_v|$ remains unchanged. Since all the invariants for v were valid before the current edge update, it follows that invariant 2 for v continues to remain valid after the update as well.

Handling deletion of an edge. Let (u, v) be an edge that is deleted. If $(u, v) \notin \mathcal{M}$, all the invariants are still valid. So let us consider the nontrivial case when $(u, v) \in \mathcal{M}$. In this case, the deletion of (u, v) has made u and v free. Therefore, potentially, invariant 1 might have become violated for u and v, making them dirty. We do the following processing in this case.

If edge (u, v) was at level 0, then following the deletion of (u, v), vertex u executes NAIVE-SETTLE(u), and then vertex v executes NAIVE-SETTLE(v). This restores invariant 1 and the vertices u and v are *clean* again. If edge (u, v) was at level 1, then u is processed using the procedure shown in Figure 2 which does the following (v is processed similarly).

First, u disowns all its edges whose other endpoint is at level 1. If $|\mathcal{O}_u|$ is still greater than or equal to $\sqrt{n}$, then u stays at level 1 and executes RANDOM-SETTLE(u). If $|\mathcal{O}_u|$ is less than $\sqrt{n}$, u moves to level 0 and executes NAIVE-SETTLE(u). Note that the transition of u from level 1 to 0 leads to an increase in the number of edges owned by each of its neighbors at level 0. Invariant 2 for each such neighbor, say w, may get violated if $|\mathcal{O}_w| = \sqrt{n}$, making w dirty. So we scan each neighbor of u sequentially and for each dirty neighbor w (that is, $|\mathcal{O}_w| = \sqrt{n}$), we execute RANDOM-SETTLE(w) to restore invariant 2. This finishes the processing of the deletion of (u, v).

Procedure HANDLING-INSERTION(u, v)

$\mathcal{O}_u \leftarrow \mathcal{O}_u \cup \{(u, v)\}$;
$\mathcal{O}_v \leftarrow \mathcal{O}_v \cup \{(u, v)\}$;
if *u and v are free* **then** $\mathcal{M} \leftarrow \mathcal{M} \cup \{(u, v)\}$;
if $|\mathcal{O}_v| > |\mathcal{O}_u|$ **then** swap(u, v);
if $|\mathcal{O}_u| = \sqrt{n}$ **then**
    foreach $(u, w) \in \mathcal{O}_u$ **do**
        delete (u, w) from $\mathcal{O}_w$;
    $x \leftarrow$ RANDOM-SETTLE(u);
    if $x \neq$ NULL **then** NAIVE-SETTLE(x);
    if *w was previous mate of u* **then** NAIVE-SETTLE(w);

Procedure RANDOM-SETTLE(u): Finds a random edge (u, y) from the edges owned by u and returns the previous mate of y

Let (u, y) be a uniformly randomly selected edge from $\mathcal{O}_u$;
foreach $(y, w) \in \mathcal{O}_y$ **do**
    delete (y, w) from $\mathcal{O}_w$;
if *y is matched* **then**
    $x \leftarrow$ MATE(y);
    $\mathcal{M} \leftarrow \mathcal{M} \backslash \{(x, y)\}$
else
    $x \leftarrow$ NULL;
$\mathcal{M} \leftarrow \mathcal{M} \cup \{(u, y)\}$;
level$(u) \leftarrow 1$; level$(y) \leftarrow 1$;
return x;

Procedure NAIVE-SETTLE(u) : Finds a free vertex adjacent to u deterministically

for *each* $(u, x) \in \mathcal{O}_u$ **do**
    if *x is free* **then**
        $\mathcal{M} \leftarrow \mathcal{M} \cup \{(u, x)\}$;
        Break;

FIG. 1. *Procedure for handling insertion of an edge (u, v), where* level$(u) =$ level$(v) = 0$.

It can be observed that, unlike insertion of an edge, the deletion of an edge may lead to creation of a large number of dirty vertices. This may happen if the deleted edge is a matched edge at level 1 and at least one of its endpoints moves to level 0 increasing the number of owned edges of each of its neighbors by 1.

Remark 3.1. The only criterion for a vertex to move to level 1 from level 0 is that there should be at least $\sqrt{n}$ edges incident on it from the vertices at level 0. Therefore, it is easy to observe that the updates in matching at level 1 depend only on the subgraph induced by vertices at level 0 and not on the exact matching present at level 0. This observation seems obvious and useless in our 2-level algorithm. However, this observation gets carried over to our main algorithm in the following section, and will play a crucial role in its time complexity analysis.

Procedure HANDLING-DELETION(u,v)

```
foreach (u,w) ∈ O_u  and  level(w) = 1 do
    move (u,w) from O_u to O_w;
if |O_u| ≥ √n then
    x ← RANDOM-SETTLE(u);
    if x ≠ NULL then NAIVE-SETTLE(x);
else
    level(u) ← 0;
    foreach (u,w) ∈ O_u  and  level(w) = 0 do
        add (u,w) to O_w;
    NAIVE-SETTLE(u);
    foreach (u,w) ∈ O_u do
        if |O_w| = √n then
            x ← RANDOM-SETTLE(w);
            if x ≠ NULL then NAIVE-SETTLE(x);
```

FIG. 2. *Procedure for processing u when $(u,v) \in \mathcal{M}$ is deleted and* $\text{level}(u) = \text{level}(v) = 1$.

3.1. Analysis of the algorithm. While processing a sequence of insertion and deletion of edges, an edge may become matched or unmatched multiple times. We analyze the algorithm using the concept of *epochs*, which we explain as follows.

DEFINITION 3.1. *At any time t, let (u,v) be any edge in $\mathcal{M}$. Then the epoch defined by (u,v) at time t is the maximal continuous time period containing t during which (u,v) remains in $\mathcal{M}$. An epoch is said to belong to level* 0 *or* 1 *depending upon the level of the matched edge that defines the epoch.*

The entire life span of an edge (u,v) consists of a sequence of epochs of (u,v) separated by the continuous periods when (u,v) is unmatched. It follows from the algorithm that any edge update that does not change the matching is processed in $O(1)$ time. An edge update that changes the matching results in the creation of one or more fresh epochs or the termination of some existing epoch(s). For the sake of analysis, we will distribute the computation carried out while processing an edge update among the corresponding epochs that are created or terminated. More specifically, suppose epochs of $(u_1,v_1),(u_2,v_2),\ldots,(u_\ell,v_\ell)$ get created and epochs of $(w_1,x_1),(w_2,x_2),\ldots,(w_k,x_k)$ get terminated during an edge update. Then each computation performed while processing this update will either be associated with the start of an epoch (u_i,v_i) for some $1 \le i \le \ell$ or to the termination of an epoch (w_j,x_j) for some $1 \le j \le k$. A crucial advantage of this distribution will be the following: For analyzing time complexity of the algorithm, it will suffice to count the number of epochs created or terminated during a sequence of updates. To show how this distribution is carried out, let us analyze the computation involved in each procedure of our algorithm and distribute it suitably among the corresponding epochs.

1. NAIVE-SETTLE(u):
   Observe that whenever the procedure NAIVE-SETTLE(u) is carried out, u is present at level 0 and, hence, $|\mathcal{O}_u| < \sqrt{n}$. The procedure NAIVE-SETTLE(u) searches for a free neighbor of u by scanning $\mathcal{O}_u$. Hence, the time complexity of NAIVE-SETTLE(u) is $O(|\mathcal{O}_u|) = O(\sqrt{n})$. Furthermore, this procedure is

called whenever u loses its mate, say v. So we can associate the computation of NAIVE-SETTLE(u) with the termination of the previous epoch of (u, v).

2. RANDOM-SETTLE(u):
Observe that whenever RANDOM-SETTLE(u) is invoked, u owns at least $\sqrt{n}$ edges incident from its neighbors at level 0 and, hence, $|\mathcal{O}_u| \geq \sqrt{n}$. During RANDOM-SETTLE($u$), u finds a random mate from level 0. This is done by selecting a random number $r \in [1, |\mathcal{O}_u|]$, and then picking the rth edge, say (u, y), from the linked list storing $\mathcal{O}_u$. This takes $O(|\mathcal{O}_u|)$ time. Vertex u then *pulls* y to level 1 to satisfy invariant 3. In this process, y becomes the sole owner of all those edges whose other endpoint is at level 0 (lines 2,3). Since y was the owner of at most $\sqrt{n}$ edges, the total computation time involved in performing this step is $O(\sqrt{n})$. Other steps in RANDOM-SETTLE can be executed in $O(1)$ time. Hence the total computation time of RANDOM-SETTLE(u) is $O(|\mathcal{O}_u| + \sqrt{n})$ which is $O(|\mathcal{O}_u|)$ since $|\mathcal{O}_u| \geq \sqrt{n}$. We associate this computation time with the start of the epoch of (u, y) that gets created at level 1.
3. HANDLING-INSERTION(u, v):
This procedure takes $O(1)$ time unless one of the endpoints of (u, v) starts owning $\sqrt{n}$ edges. In that case, the procedure invokes RANDOM-SETTLE (line 8) and NAIVE-SETTLE (lines 9,10). We have already distributed the time taken in these procedures to the respective epochs that get created. Excluding these tasks, the only computation performed in this procedure is in the FOR loop (lines 6,7). The purpose of this loop is to make u the sole owner of all its edges incident from level 0. Since u owns $\sqrt{n}$ edges from level 0, the total computation time involved in performing this step is $O(\sqrt{n})$. We associate this computation time with the start of the epoch created by u at level 1.
4. HANDLING-DELETION(u, v):
Procedure HANDLING-DELETION(u, v) is carried out when the matched edge (u, v) at level 1 gets deleted. In addition to invoking RANDOM-SETTLE and NAIVE-SETTLE procedures whose computation is already assigned to respective epochs, this procedure scans the list $\mathcal{O}_u$ at most twice. Notice that $|\mathcal{O}_u|$ can be $\Theta(n)$. We associate this computation time of $O(n)$ with the termination of the epoch of (u, v).

Excluding the updates that cause the start and termination of an epoch of (u, v), every other edge update on u and v during the epoch is handled in just $O(1)$ time. Therefore, we shall focus only on the amount of computation associated with the start and termination of an epoch. Let us now analyze the computation time associated with an epoch at level 0 and an epoch at level 1.

- Epoch at level 0:
As discussed above, it is only the procedure NAIVE-SETTLE whose computation time is associated with an epoch at level 0. This procedure takes $O(\sqrt{n})$ time. Hence the computation time associated with an epoch at level 0 is $O(\sqrt{n})$.
- Epoch at level 1:
Consider an epoch at level 1. There are two ways in which this epoch gets created at level 1:
  - In HANDLING-INSERTION:
  An epoch of (u, v) can be created during the procedure HANDLING-INSERTION(u, v). In this case, the computation time associated with the start of the epoch of (u, v) is the computation time incurred in executing the procedure HANDLING-INSERTION and the procedure RANDOM-

SETTLE which it invokes. It follows from the above discussion that the computation time associated with the epoch (u, v) is $O(\sqrt{n}+|\mathcal{O}_u|)$, which is $O(\sqrt{n})$ since $|\mathcal{O}_u| = \sqrt{n}$ when we invoke HANDLING-INSERTION(u, v).

– In HANDLING-DELETION:
Procedure HANDLING-DELETION(u, v) invokes RANDOM-SETTLE at lines 4 and 12 to create new epochs at level 1. The execution of RANDOM-SETTLE at line 4 creates a new epoch for u and its computation time $O(|\mathcal{O}_u|)$, which can be $\Theta(n)$, gets associated with the start of the new epoch created by u. The execution of RANDOM-SETTLE at line 12 creates a new epoch for some vertex w which is some neighbor of u. Note that $|\mathcal{O}_w| = \sqrt{n}$. Its computation time, which is $O(\sqrt{n})$, is associated with the start of the epoch at level 1 created by w.

Now let us calculate the computation time associated with an epoch, say of an edge (u, v), at level 1 when it terminates. It follows from the discussion above that the only computation time associated with the termination of epoch (u, v) is the computation time of HANDLING-DELETION (excluding the time spent in procedures RANDOM-SETTLE and NAIVE-SETTLE that are already associated with the start of their respective epochs). This time is at most $O(n)$.

From our analysis given above, it follows that the amount of computation time associated with an epoch at level 0 is $O(\sqrt{n})$ and the computation time associated with an epoch at level 1 is $O(n)$.

An epoch of (u, v), once created, may either terminate during the processing of some future update or remain *alive*, i.e., (u, v) remains in the matching even at the end of all the updates. It can be observed that an epoch of (u, v) may terminate because of exactly one of the following causes:

(i) (u, v) gets deleted from the graph.
(ii) u (or v) gets matched to some other vertex leaving its current mate free.

An epoch will be called a *natural* epoch if it terminates due to cause (i); otherwise, it will be called an *induced* epoch. An induced epoch terminates prematurely since, unlike a natural epoch, the matched edge is not actually deleted from the graph when an induced epoch terminates.

It follows from the algorithm described above that every epoch at level 1 is a natural epoch whereas an epoch at level 0 can be natural or induced depending on the cause of its termination. Furthermore, each induced epoch at level 0 can be associated with a natural epoch at level 1 whose creation led to the termination of the former. In fact, there can be at most two induced epochs at level 0 which can be associated with an epoch at level 1. It can be explained as follows (see Figure 3).

Consider an epoch at level 1 associated with an edge, say (u, v). Suppose it was created by vertex u. If u was already matched at level 0, let $w \neq v$ be its mate. Similarly, if v was also matched already, let $x \neq u$ be its current mate at level 0. So matching u to v terminates the epoch of (u, w) as well as the epoch of edge (v, x) at level 0. We *charge* the computation time associated with these two epochs to the epoch of (u, v). We have seen that the computation time associated with an epoch at level 0 is $O(\sqrt{n})$. So the overall computation time *charged* to an epoch of (u, v) at level 1 is $O(n + 2\sqrt{n})$ which is $O(n)$. Hence we can state the following lemma.

LEMMA 3.1. *The computation time charged to a natural epoch at level* 1 *is* $O(n)$ *and the computation time charged to a natural epoch at level* 0 *is* $O(\sqrt{n})$.

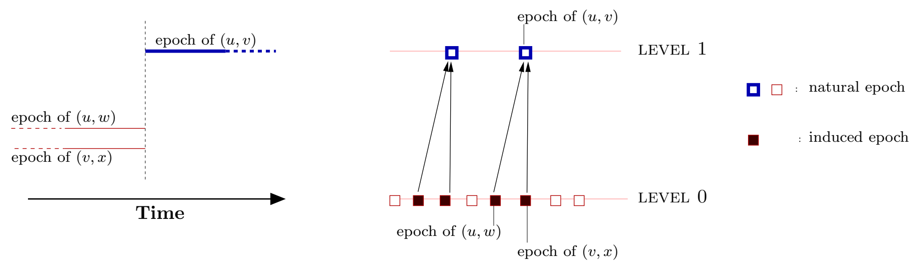

FIG. 3. *Epochs at level* 0 *and* 1*; the creation of an epoch at level* 1 *can terminate at most two epochs at level* 0*.*

In order to analyze time complexity of our algorithm, we shall now get a bound on the computation time charged to all natural epochs that get terminated during a sequence of updates or are alive at the end of all the updates. Let t be the total number of updates. Let us first analyze the computation time charged to all those epochs which are alive at the end of t updates. Consider an epoch of edge (u, v) that is alive at the end of t updates. If this epoch is at level 0, the computation time associated with the start of this epoch is $O(1)$. If this epoch is at level 1, then the computation time associated with the start of this epoch is $O(|\mathcal{O}_u|)$ and notice that $|\mathcal{O}_u| \geq \sqrt{n}$. Note that there can be at most two induced epochs at level 0 whose computation time, which is $O(\sqrt{n})$, is also charged to the epoch of (u, v). Hence the computation charged to the live epoch of (u, v) is $O(|\mathcal{O}_u|)$. Observe that, at any given moment of time, $\mathcal{O}_u \cap \mathcal{O}_w = \emptyset$ for any two vertices u, w present at level 1. Hence the computation time charged to all live epochs at the end of t updates is of the order of $\sum_u |\mathcal{O}_u| \leq 2t = O(t)$. So all we need to do is to analyze the computation time charged to all natural epochs that get terminated during the sequence of updates.

Each natural epoch at level 0 which gets terminated can be assigned uniquely to the deletion of its matched edge. Hence it follows from Lemma 3.1 that the computation time charged to all natural epochs terminated at level 0 during t updates is $O(t\sqrt{n})$. We shall now analyze the number of epochs terminated at level 1. Our analysis will crucially exploit the following lemma. The lemma holds since a vertex picks its matched edge uniformly from its owned edges while creating an epoch at level 1.

LEMMA 3.2. *Suppose vertex v creates an epoch at level* 1 *when the algorithm processes an update in the graph. Let O_v^{init} be the set of edges that v owned at the time of the creation of this epoch. Then, for any arbitrary sequence D of edge deletions of O_v^{init} and for any $(v, w) \in O_v^{init}$,*

$$\mathbf{Pr}[\text{MATE}(v) = w \mid D] = \frac{1}{|O_v^{init}|}.$$

We first carry out the analysis for the high probability bound on the total update time taken by our algorithm. Thereafter we carry out the analysis for the expected value of the total update time.

Note that an edge e may be deleted and inserted multiple times during the algorithm. However, each of them is considered *distinct* from the perspective of the algorithm as well as its analysis.

3.2. High probability bound on the total update time. The key idea of randomization underlying our algorithm is the following: Once a vertex v creates an epoch at level 1, probabilistically we would expect deletion of *many* edges of set $\mathcal{O}_v^{init}$ from the graph before the matched edge of v is deleted. In order to quantify this key idea, we introduce the following definition that categorizes an epoch as good or bad.

DEFINITION 3.2. *An epoch created by a vertex v is said to be bad if it gets terminated naturally before the deletion of $1/3$ edges of set $\mathcal{O}_v^{init}$ from the graph. An epoch is said to be good if it is not bad.*

It follows from Definition 3.2 that a good epoch undergoes many edge deletions before getting terminated. So only the bad epochs are problematic for the efficiency of our algorithm. Now using Lemma 3.2, we establish an upper bound on the probability of an epoch to be bad.

LEMMA 3.3. *Suppose vertex v creates an epoch at level 1 when the algorithm processes an update in the graph. Then this epoch is going to be bad with probability $1/3$ for any sequence of future updates in the graph and the random bits picked during their processing.*

Proof. Let $\mathcal{O}_v^{init}$ be the set of edges owned by v at the time of the creation of the epoch. Recall that the mate picked by v, say w, is brought to level 1 by v when the epoch is created. As a result, none of the neighbors of w can pick it as a mate to terminate the epoch. Hence, the epoch will terminate only when the matched edge (v, w) is deleted. Consider any sequence of updates in the graph following the creation of this epoch. This sequence uniquely defines the sequence D of edge deletions of $\mathcal{O}_v^{init}$. It can be observed that the time when the epoch will be terminated is fully determined by this sequence D and the mate that v picks. Obviously the random bits picked during the processing of future updates has no influence on the duration of this epoch. The epoch will be bad if the mate of v is among the endpoints of the first $1/3$ edges in the sequence D. Now it follows from Lemma 3.2 that the mate of v is distributed uniformly among the endpoints of D. So the probability of the epoch to be bad is $1/3$.[1] ∎

For the time complexity analysis at level 1, we will now show that the number of bad epochs may exceed the number of good epochs by at most $O(\log n)$ with very high probability. For this we shall use the analogy of asymmetric random walk on a line.

Consider a particle performing a discrete random walk on a line. In each unit of time, it takes a unit step to the right with probability p or a unit step to the left with probability $q = 1 - p$. Each step is independent of the steps taken in the past. The following lemma states a well known result about such a random walk.

LEMMA 3.4 (see [13]). *Suppose the random walk starts at a location $z > 0$. If $p > q$, the probability of it ever reaching the origin equals $(q/p)^z$.*

Consider the sequence of updates in the graph in the chronological order. Each update may create zero or more epochs at level 1. Thus the number of epochs created at level 1 is a random variable. However, as shown by Lemma 3.3, each newly created epoch at level 1 will be bad with probability $1/3$ irrespective of the future sequence of updates in the graph and the random bits picked during their processing. The

[1] The analysis assumes that each edge from $\mathcal{O}_v^{init}$ is going to be deleted sometime in the future. If not, place all such edges of $\mathcal{O}_v^{init}$, that are not deleted, arbitrarily at the end of the sequence D. In this case, as the reader may also see, there are chances for the epoch to remain alive instead of getting terminated at the end of the algorithm. So the probability of the epoch to be bad in this case will be even less than $1/3$.

sequence of epochs terminated at level 1 can thus be seen as an instance of the asymmetric random walk as follows. A good epoch corresponds to a rightward step and a bad epoch corresponds to a leftward step. Each step is rightward with probability 2/3 or leftward with probability 1/3 independent of other steps. Therefore, the event that the number of bad epochs ever exceeds the number of good epochs by at least $2\log_2 n$ during the algorithm is stochastically identical to the event that the random walk starting from location $2\log_2 n$ ever reaches the origin within any finite number of steps. It follows from Lemma 3.4 that the probability of the latter event is less than $1/n^2$. So the following lemma holds.

LEMMA 3.5. *During any sequence of t updates, the number of bad epochs at level 1 can exceed the number of good epochs by at least $2\log_2 n$ with probability $< 1/n^2$.*

As stated in Lemma 3.1, each epoch at level 1 has an $O(n)$ computation time charged to it. Let t be the total number of updates in the graph. For each epoch at level 1, the number of owned edges at the time of its creation is at least $\sqrt{n}$. As a result the number of good epochs during t updates is bounded by $3t/\sqrt{n}$ deterministically. So the computation time of all good epochs at level 1 is bounded by $O(t\sqrt{n})$. Lemma 3.5 implies that the computation time of all bad epochs at level 1 can exceed the computation time of all good epochs by at most a $cn\log n$ amount for some constant c with probability $\geq 1 - 1/n^2$. So overall the computation time of all epochs at level 1 is bounded by $O((t\sqrt{n} + n\log n)$ with high probability. The computation time of all epochs at level 0 is bounded deterministically by $O(t\sqrt{n})$. Hence the total computation time taken by our algorithm for any sequence of t updates is $O(t\sqrt{n} + n\log n)$ with high probability.

3.3. Expected value of the total update time. Let $X_{v,i,k}$ be a random variable which is 1 if v creates an epoch at level i during the processing of the kth update, otherwise it is 0. We denote the corresponding epoch by EPOCH(v,i,k). Let $Z_{v,i,k}$ denote the number of edges from $\mathcal{O}_v^{init}$ that are deleted during the epoch. (If EPOCH(v,i,k) is not created, $Z_{v,i,k}$ is defined as 0.) Since each edge deletion at level 1 is uniquely associated with the epoch that owned it, therefore, $\sum_{v,k} Z_{v,1,k} \leq t$. Hence,

$$\sum_{v,k} \mathbf{E}[Z_{v,1,k}] \leq t. \tag{1}$$

We shall now derive a bound on the expected value of $Z_{v,1,k}$ in an alternate way.

LEMMA 3.6. $\mathbf{E}[Z_{v,1,k}] \geq \sqrt{n}/2 \cdot \mathbf{Pr}[X_{v,1,k} = 1]$.

Proof. We shall first find the expectation of $Z_{v,1,k}$ conditioned on the event that v creates an epoch at level 1 during the kth update. That is, we shall find $\mathbf{E}[Z_{v,1,k}|X_{v,1,k} = 1]$. Let $\mathcal{O}_v^{init}$ be the set of edges owned by v at the moment of creation of EPOCH$(v,1,k)$, and let D be the deletion sequence associated with $\mathcal{O}_v^{init}$. It follows from Lemma 3.2 that the matched edge of v is distributed uniformly over D. So $\mathbf{E}[Z_{v,1,k}|X_{v,1,k} = 1] = |\mathcal{O}_v^{init}|/2 \geq \sqrt{n}/2$ since $|\mathcal{O}_v^{init}|$ for an epoch at level 1 is at least $\sqrt{n}$. Using conditional expectation, we get

$$\mathbf{E}[Z_{v,1,k}] = \mathbf{E}[Z_{v,1,k}|X_{v,1,k} = 1] \cdot \mathbf{Pr}[X_{v,1,k} = 1] \geq \sqrt{n}/2 \cdot \mathbf{Pr}[X_{v,1,k} = 1]. \qquad \square$$

Notice that the computation time of an epoch at level 1 is at most cn for some constant c. So the expected value of the computation time associated with all natural epochs that get terminated at level 1 during t updates is

$$\begin{aligned}\sum_{v,k} cn \cdot \mathbf{Pr}[X_{v,1,k} = 1] &= 2c\sqrt{n}\sum_{v,k} \sqrt{n}/2 \cdot \mathbf{Pr}[X_{v,1,k} = 1] \\ &\le 2c\sqrt{n}\sum_{v,k} \mathbf{E}[Z_{v,1,k}] \qquad \text{using Lemma 3.6} \\ &\le 2c\sqrt{n}t \qquad \text{using (1).}\end{aligned}$$

We can thus conclude with the following theorem.

THEOREM 3.1. *Starting with a graph on n vertices and no edges, we can maintain maximal matching for any sequence of t updates in $O(t\sqrt{n})$ time in expectation and $O(t\sqrt{n} + n\log n)$ with high probability.*

3.4. On improving the update time beyond $O(\sqrt{n})$. In order to extend our 2-level algorithm for getting a better update time, it is worth exploring the reason underlying $O(\sqrt{n})$ update time guaranteed by our 2-level algorithm. For this purpose, let us examine invariant 2 more carefully. Let $\alpha(n)$ be the threshold for the maximum number of edges that a vertex at level 0 can own. Consider an epoch at level 1 associated with some edge, say (u, v). The computation associated with this epoch is of the order of the number of edges u and v own which can be $\Theta(n)$ in the worst case. However, the expected duration of the epoch is of the order of the minimum number of edges u can own at the time of its creation, i.e., $\Theta(\alpha(n))$. Therefore, the expected amortized computation time per edge deletion for an epoch at level 1 is $O(n/\alpha(n))$. Balancing this with the $\alpha(n)$ computation time to handle an edge deletion at level 0 yields $\alpha(n) = \sqrt{n}$.

In order to improve the running time of our algorithm, we need to decrease the ratio between the maximum and the minimum number of edges a vertex can own during an epoch at any level. This insight motivates us to have a finer partition of vertices—the number of levels should be increased to $O(\log n)$ instead of just 2. When a vertex creates an epoch at level i, it will own at least 4^i edges, and during the epoch it will be allowed to own at most $4^{i+1} - 1$ edges. As soon as it owns 4^{i+1} edges, it should migrate to the level $i+1$. Notice that the ratio of maximum to minimum edges owned by a vertex during an epoch gets reduced from $\sqrt{n}$ to a constant.

We pursue the approach sketched above combined with some additional techniques in the following section. This leads to a fully dynamic algorithm for maximal matching which achieves expected amortized $O(\log n)$ update time per edge insertion or deletion.

4. Fully dynamic algorithm with expected amortized $O(\log n)$ time per update. This algorithm maintains a partition of vertices among various levels. First we describe the difference in this partition vis-à-vis the 2-level algorithm.

1. The fully dynamic algorithm maintains a partition of vertices among $\lfloor \log_4 n \rfloor +$ 2 levels. The levels are numbered from -1 to $\mathrm{L}_0 = \lfloor \log_4 n \rfloor$. During the algorithm, when a vertex moves to level i, it owns at least 4^i edges. So a vantage point is needed for a vertex that does not own any edges. As a result, we introduce a level -1 that contains all the vertices that do not own any edge.
2. We use the notion of ownership of edges which is slightly different from the one used in the 2-level algorithm. In the 2-level algorithm, at level 0, both the endpoints of the edge are the owner of the edge. Here, at every level, each edge is owned by exactly one of its endpoints. If the endpoints of the

edge are at different levels, the edge is owned by the endpoint that lies at the higher level. If the two endpoints are at the same level, then the tie is broken appropriately by the algorithm.

Like the 2-level algorithm, each vertex u maintains a dynamic hash table storing the edges $\mathcal{O}_u$ owned by it. In addition, the generalized fully dynamic algorithm will maintain the following data structure for each vertex u. For each $i \geq \text{level}(u)$, let $\mathcal{E}_u^i$ be the set of all those edges incident on u from vertices at level i that are not owned by u. The set $\mathcal{E}_u^i$ will be maintained in a dynamic hash table. However, the onus of maintaining $\mathcal{E}_u^i$ will not be on u. For any edge $(u, v) \in \mathcal{E}_u^i$, v will be responsible for the maintenance of (u, v) in $\mathcal{E}_u^i$ since $(u, v) \in \mathcal{O}_v$. For example, suppose vertex v moves from level i to level j. If $j > \text{level}(u)$, then v will remove (u, v) from $\mathcal{E}_u^i$ and insert it into $\mathcal{E}_u^j$. Otherwise $((j \leq \text{level}(u))$, v will remove (u, v) from $\mathcal{E}_u^i$ and insert it into $\mathcal{O}_u$.

4.1. Invariants and a basic subroutine used by the algorithm. As can be seen from the 2-level algorithm, it is advantageous for a vertex to get settled at a higher level once it owns a large number of edges. Pushing this idea still further, our fully dynamic algorithm will allow a vertex to rise to a higher level if it can own a *sufficiently large* number of edges after moving there. In order to formally define this approach, we introduce an important notation here.

For a vertex v with $\text{level}(v) = i$,

$$\phi_v(j) = \begin{cases} |\mathcal{O}_v| + \sum_{i \leq k < j} |\mathcal{E}_v^k| & \text{if } j > i, \\ 0 & \text{otherwise.} \end{cases}$$

In other words, for any vertex v at level i and any $j > i$, $\phi_v(j)$ denotes the number of edges which v can own if v rises to level j. Our algorithm will be based on the following key idea. If a vertex v has $\phi_v(j) \geq 4^j$, then v would rise to the level j. In the case there are multiple levels to which v can rise, v will rise to the highest such level. With this key idea, we now describe the three invariants which our algorithm will maintain.

1. Every vertex at level ≥ 0 is matched and every vertex at level -1 is free.
2. For each vertex v and for all $j > \text{level}(v)$, $\phi_v(j) < 4^j$ holds true.
3. Both the endpoints of a matched edge are at the same level.

It follows that the free vertices, if any, will be present at level -1 only. Any vertex v present at level -1 cannot have any neighbor at level -1. Otherwise, it would imply that $\phi_v(0) \geq 1 = 4^0$, violating the second invariant. Hence, every neighbor of a free vertex must be matched. This implies that the algorithm will always maintain a maximal matching. Furthermore, the key idea of our algorithm is captured by the second invariant—after processing every update there is no vertex which fulfils the criterion of rising. Figure 4 depicts a snapshot of the algorithm.

An edge update may lead to the violation of the invariants mentioned above and the algorithm basically restores these invariants. This may involve the rise or fall of vertices to appropriate levels. Notice that the second invariant of a vertex is influenced by the rise and fall of its neighbors. We now state and prove two lemmas which quantify this influence more precisely.

LEMMA 4.1. *The rise of a vertex v does not violate the second invariant for any of its neighbors.*

Proof. Consider any neighbor u of v. Let $\text{level}(u) = k$. Since the second invariant holds true for u before the rise of v, so $\phi_u(i) < 4^i$ for all $i > k$. It suffices if we can

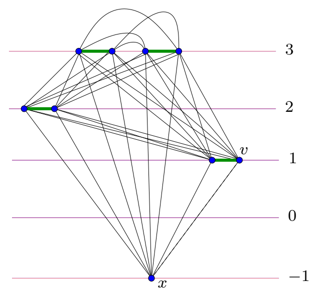

FIG. 4. *A snapshot of the algorithm from perspective of a vertex v. For the sake of brevity, only the subgraph induced by the neighbors of v is shown. All neighbors of v are matched (thick edges) except vertex x at level -1. Vertex v is the owner of just the edge (v, x). $\phi_v(2) = 2 < 4^2$ and $\phi_v(3) = 4 < 4^3$, so v cannot rise to a higher level.*

show that $\phi_u(i)$ does not increase for any i due to the rise of v. We show this as follows.

Let vertex v rise from level j to ℓ. If $\ell \leq k$, the edge (u, v) continues to be an element of $\mathcal{O}_u$, and so there is no change in $\phi_u(i)$ for any i. Let us consider the case when $\ell > k$. The rise of v from j to ℓ causes removal of (u, v) from $\mathcal{O}_u$ (or $\mathcal{E}_u^j$ if $j \geq k$) and insertion to $\mathcal{E}_u^\ell$. As a result $\phi_u(i)$ decreases by one for each i in $[\max(j, k) + 1, \ell]$, and remains unchanged for all other values of i. ∎

LEMMA 4.2. *Suppose a vertex v falls from level j to $j - 1$. As a result, for any neighbor u of v, $\phi_u(i)$ increases by at most 1 for $i = j$ and remains unchanged for all other values of i.*

Proof. Let level$(u) = k$. In the case $k \geq j$, there is no change in $\phi_u(i)$ for any i due to the fall of v. So let us consider the case $j > k$. In this case, the fall of v from level j to $j - 1$ leads to the insertion of (u, v) in $\mathcal{E}_u^{j-1}$ and deletion from $\mathcal{E}_u^j$. Consequently, $\phi_u(i)$ increases by one only for $i = j$ and remains unchanged for all other values of i. ∎

In order to detect any violation of the second invariant for a vertex v due to the rise or fall of its neighbors, we shall maintain $\{\phi_v(i) | i \leq \text{L}_0\}$ in an array $\phi_v[]$ of size $\text{L}_0 + 2$. The updates on this data structure during the algorithm will involve the following two types of operations.

- DECREMENT-$\phi(v, I)$: This operation decrements $\phi_v(i)$ by one for all i in interval I. This operation will be executed when some neighbor of v rises. For example, suppose some neighbor of v rises from level j to ℓ, then $\phi_v(i)$ decreases by one for all i in interval $I = [\max(j, \text{level}(v)) + 1, \ell]$.
- INCREMENT-$\phi(v, i)$: this operation increases $\phi_v(i)$ by one. This operation will be executed when some neighbor of v falls from i to $i - 1$.

It can be seen that a single DECREMENT-$\phi(v, I)$ operation takes $O(|I|)$ time which is $O(\log n)$ in the worst case. On the other hand any single INCREMENT-$\phi(v, i)$ operation takes $O(1)$ time. However, since $\phi_v(i)$ is 0 initially and is nonnegative always, we can conclude the following.

LEMMA 4.3. *The computation cost of all* DECREMENT-$\phi()$ *operations over all vertices is upper bounded by the computation cost of all* INCREMENT-$\phi()$ *operations over all vertices during the algorithm.*

Procedure GENERIC-RANDOM-SETTLE(u, i)

```
if level(u) < i then                                          // u rises to level i
    for each (u,w) ∈ O_u do                                   // u informs w about its rise
        transfer (u,w) from E_w^{level(u)} to E_w^i;
        DECREMENT-φ(w, [level(u)+1, i]);
    for each j = level(u) to i−1 do                           // u gains ownership of some more edges
        for each (u,w) ∈ E_u^j do
            transfer (u,w) from E_u^j to E_w^i;
            transfer (u,w) from O_w to O_u;
            DECREMENT-φ(w, [j+1, i]);
    foreach j = level(u)+1 to i do φ_u(j) ← 0;
    level(u) ← i;
Let (u,v) be a uniformly randomly selected edge from O_u;
if v is matched then
    x ← MATE(v);
    M ← M\{(v,x)};
else
    x ← NULL;
for each (v,w) ∈ O_v do                                       // v informs w about its rise
    transfer (v,w) from E_w^{level(v)} to E_w^i;
    DECREMENT-φ(w, [level(v)+1, i]);
for each j = level(v) to i−1 do                               // v gains ownership of some more edges
    for each (v,w) ∈ E_v^j do
        transfer (v,w) from E_v^j to E_w^i;
        transfer (v,w) from O_w to O_v;
        DECREMENT-φ(w, [j+1, i]);
M ← M ∪ {(u,v)};
foreach j = level(v)+1 to i do φ_v(j) ← 0;
level(v) ← i ;                                                /* v rises to level i */
return x;
```

FIG. 5. *Procedure used by a free vertex* u *to* settle *at* level i.

Observation 4.1. It follows from Lemma 4.3 that we just need to analyze the computation involving all INCREMENT-$\phi()$ operations since the computation involved in DECREMENT-$\phi()$ operations is subsumed by the former.

If any invariant of a vertex, say u, gets violated, it might rise or fall, though in some cases, it may still remain at the same level. However, in all these cases, eventually the vertex u will execute the procedure, GENERIC-RANDOM-SETTLE, shown in Figure 5. This procedure is essentially a generalized version of RANDOM-SETTLE(u) which we used in the 2-level algorithm. GENERIC-RANDOM-SETTLE(u, i) starts with moving u from its current level (level(u)) to level i. If level i is higher than the previous level of u, then u performs the following tasks. For each edge (u, w) already owned by it, u informs w about its rise to level i by updating $\mathcal{E}_w^i$. In addition u acquires the ownership of all the edges whose other endpoint lies at a level $\in$ [level$(u), i-1$]. For each such edge (u, w) that is now owned by u, we perform

DECREMENT-$\phi(w, [\text{level}(w) + 1, i])$ to reflect that the edge is now owned by vertex u which has moved to level i. Henceforth, the procedure resembles RANDOM-SETTLE. It finds a random edge (u, v) from $\mathcal{O}_u$ and moves v to level i. The procedure returns the previous mate of v, if v was matched. We can thus state the following lemma.

LEMMA 4.4. *Consider a vertex u that executes* GENERIC-RANDOM-SETTLE(u, i) *and selects a mate v. Excluding the time spent in* DECREMENT-ϕ *operations, the computation time of this procedure is of the order of $|\mathcal{O}_u| + |\mathcal{O}_v|$, where $\mathcal{O}_u$ and $\mathcal{O}_v$ are the sets of edges owned by u and v just at the end of the procedure.*

4.2. Handling edge updates by the fully dynamic algorithm. Our fully dynamic algorithm will employ a generic procedure called PROCESS-FREE-VERTICES(). The input to this procedure is a sequence S consisting of ordered pairs of the form (x, k), where x is a free vertex at level $k \geq 0$. Observe that the presence of free vertices at a level ≥ 0 implies that matching $\mathcal{M}$ is not necessarily maximal. In order to preserve maximality of matching, the procedure PROCESS-FREE-VERTICES restores the invariants of each such free vertex till S becomes empty. We now describe our fully dynamic algorithm.

Handling deletion of an edge. Consider deletion of an edge, say (u, v). For each $j > \max(\text{level}(u), \text{level}(v))$, we decrement $\phi_u(j)$ and $\phi_v(j)$ by one. If (u, v) is an unmatched edge, no invariant gets violated. So we only delete the edge (u, v) from the data structures of u and v. Otherwise, let $k = \text{level}(u) = \text{level}(v)$. We execute the Procedure PROCESS-FREE-VERTICES$(\langle(u, k), (v, k)\rangle)$.

Handling insertion of an edge. Consider insertion of an edge, say (u, v). Without loss of generality, assume that initially u was at the same level as v or a higher level than v. So we add (u, v) to $\mathcal{O}_u$ and $\mathcal{E}_v^{\text{level}(u)}$. We increment $\phi_u(j)$ and $\phi_v(j)$ by one for each $j > \max(\text{level}(u), \text{level}(v))$. We check if the second invariant has become violated for either u or v. This invariant may get violated for u (likewise for v) if there is any integer $i > \max(\text{level}(u), \text{level}(v))$, such that $\phi_u(i)$ has become 4^i just after the insertion of edge (u, v). In the case there are multiple such integers, let $i_{\max}$ be the largest such integer. To restore the invariant, u leaves its current mate, say w, and rises to level $i_{\max}$. We execute GENERIC-RANDOM-SETTLE$(u, i_{\max})$, and let x be the vertex returned by this procedure. Let j and k be, respectively, the levels of w and x. Note that x and w are two free vertices now. We execute PROCESS-FREE-VERTICES$(\langle(x, k), (w, j)\rangle)$.

If the insertion of edge (u, v) violates the second invariant for both u and v, we proceed as follows. Let j be the highest level to which u can rise after the insertion of (u, v), that is, $\phi_u(j) = 4^j$. Similarly, let ℓ be the highest level to which v may rise, that is, $\phi_v(\ell) = 4^\ell$. If $j \geq \ell$, we allow only u to rise to level j; otherwise we allow only v to rise to ℓ. Note that after u moves to level j, edge (u, v) becomes an element of $\mathcal{E}_v^j$. So $\sum_{\text{level}(v) \leq k < \ell} |\mathcal{E}_v^k|$ decreases by 1. As a result, $\phi_v(\ell) = |\mathcal{O}_v| + \sum_{\text{level}(v) \leq k < \ell} |\mathcal{E}_v^k|$ also decreases by 1 and is now strictly less than 4^ℓ; thus the second invariant for v is also restored.

4.2.1. Description of procedure process-free-vertices. The procedure receives a sequence S of ordered pairs (x, i) such that x is a free vertex at level i. It processes the free vertices in decreasing order of their levels starting from L_0. We give an overview of this processing at level i. For a free vertex at level i, if it owns a *sufficiently* large number of edges, then it settles at level i and gets matched by selecting a random edge from the edges owned by it. Otherwise the vertex falls down

Procedure PROCESS-FREE-VERTICES(S)

```
for each $(x,i) \in S$ do ENQUEUE($Q[i], x$);
for $i = L_0$ to 0 do
    while ($Q[i]$ is not EMPTY) do
        $v \leftarrow$ DEQUEUE($Q[i]$);
        if FALLING($v$) then                                  // $v$ falls to level $i-1$
            level($v$) $\leftarrow i - 1$;
            ENQUEUE($Q[i-1], v$);
            for each $(u,v) \in \mathcal{O}_v$ do
                transfer $(u,v)$ from $\mathcal{E}_u^i$ to $\mathcal{E}_u^{i-1}$;
                INCREMENT-$\phi(u,i)$;
                INCREMENT-$\phi(v,i)$;
                if $\phi_u(i) \geq 4^i$ then                  // $u$ rises to level $i$
                    $x \leftarrow$ GENERIC-RANDOM-SETTLE($u,i$);
                    if $x \neq$ NULL then
                        $j \leftarrow$ level($x$);
                        ENQUEUE($Q[j], x$);

        else                                                  // $v$ settles at level $i$
            $x \leftarrow$ GENERIC-RANDOM-SETTLE($v,i$);
            if $x \neq$ NULL then
                $j \leftarrow$ level($x$);
                ENQUEUE($Q[j], x$);
```

Function FALLING(v)

```
$i \leftarrow$ level($v$);
for each $(u,v) \in \mathcal{O}_v$ with level($u$) $= i$ do    // $v$ disowns all edges from level $i$
    transfer $(u,v)$ from $\mathcal{O}_v$ to $\mathcal{O}_u$;
    transfer $(u,v)$ from $\mathcal{E}_u^i$ to $\mathcal{E}_v^i$;
if $|\mathcal{O}_v| < 4^i$ then return TRUE else return FALSE;
```

FIG. 6. *Procedure for processing free vertices given as a sequence S of ordered pairs (x,i) where x is a free vertex at level i.*

by one level. Notice that the fall of a vertex, say v, from level i to $i-1$ may lead to the rise of some of its neighbors lying at level $< i$. However, as follows from Lemma 4.2, for each such neighbor, say u, only $\phi_u(i)$ increases by one and the $\phi_u()$ value for all other levels remains unchanged. So the second invariant may get violated only for level i. This implies that u will rise only to level i. After these rising vertices move to level i (by executing GENERIC-RANDOM-SETTLE), we move onto level $i-1$ and proceed similarly. Overall, the entire process can be seen as a wave of free vertices falling level by level. Eventually this wave of free vertices reaches level -1 and fades away ensuring maximal matching. With this overview, we now describe the procedure in more details and its complete pseudocode is given in Figure 6.

The procedure uses an array Q of size $L_0 + 2$, where $Q[i]$ is a pointer to a queue (initially empty) storing free vertices at level i. For each ordered pair $(x,k) \in S$, the procedure inserts x into queue $Q[k]$. The procedure executes a FOR loop from L_0 down to 0 where the ith iteration extracts and processes the vertices of queue $Q[i]$ one by one as follows. Let v be a vertex extracted from $Q[i]$. First we execute the function FALLING(v) which does the following. v disowns all its edges whose other endpoint lies at level i. If v owns less than 4^i edges then v falls to level $i-1$, otherwise v will continue to stay at level i. The processing of the free vertex v for each of these two cases is done as follows.

1. *v has to stay at level i:*
   v executes GENERIC-RANDOM-SETTLE and selects a random mate, say w, from level $j < i$ (if w is present in $Q[j]$, then it is removed from it and is raised to level i). If x was the previous mate of w, then x has become a free vertex now. So x gets added to $Q[j]$. This finishes the processing of v.
2. *v has to fall:*
   In this case, v falls to level $i-1$ and is inserted to $Q[i-1]$. At this stage, $\mathcal{O}_v$ consists of neighbors of v from level $i-1$ or below. It follows from Lemma 4.2 that the fall of v from i to $i-1$ leads to an increase in $\phi_u(i)$ by one for each neighbor u of v which is present at a level lower than i. Moreover, $\phi_v(i)$, that was 0 initially, has to be set to $|\mathcal{O}_v|$. So all the vertices of $\mathcal{O}_v$ are scanned, and for each $(u, v) \in \mathcal{O}_v$, we increment $\phi_u(i)$ and $\phi_v(i)$ by 1. In case $\phi_u(i)$ has become 4^i, u has to rise to level i and is processed as follows. u executes GENERIC-RANDOM-SETTLE(u, i) to select a random mate, say w, from level $j < i$. If w was in $Q[j]$, then it is removed from it. Otherwise, if x was the previous mate of w, then x has become a free vertex now, and so it gets added to queue $Q[j]$.

Remark 4.1. Notice a stark similarity between the above procedure for handling a free vertex and the procedure for handling a free vertex at level 1 in the 2-level algorithm.

In case 1, v remains at level i and w moves to the level i from some level $j < i$. This renders vertex x (earlier mate of w) free and the first invariant of x is violated. So x is added to the queue at level j. The processing of v does not change $\phi_u()$ for any neighbor u of v. Furthermore, the rise of w to level i does not lead to violation of any invariant due to Lemma 4.1. In case 2, v falls to level $i-1$ and as a result some vertices may rise to level i. Each such rising vertex executes GENERIC-RANDOM-SETTLE. As in case 1, the processing of these rising vertices may create some free vertices only at level $< i$. We can thus state the following lemma.

LEMMA 4.5. *After the ith iteration of the for loop of* PROCESS-FREE-VERTICES*, the free vertices are present only in the queues at level $< i$, and for all vertices not belonging to these queues the three invariants holds.*

Lemma 4.5 establishes that after the termination of procedure PROCESS-FREE-VERTICES, there are no free vertices at level ≥ 0 and all the invariants get restored globally.

4.3. Analysis of the algorithm. Processing the deletion or insertion of an edge (u, v) begins with decrementing or incrementing $\phi_u(i)$ and $\phi_v(i)$ for each level $j > \max(\text{level}(u), \text{level}(v))$. Since there are $O(\log n)$ levels, the computation associated with this task over any sequence of t updates will be $O(t \log n)$. This task may be followed by executing the procedure PROCESS-FREE-VERTICES that restores the invariants and updates the matching accordingly. The updates in the matching can be seen as creation of new epochs and termination of some of the existing epochs. Like the 2-level algorithm, for the purpose of our analysis, we visualize the entire algorithm as a sequence of creation and termination of various epochs. Excluding the $O(t \log n)$ time for maintaining ϕ, the total computation performed by the algorithm can be associated with all the epochs that get terminated and those that remain alive at the end of the sequence of updates. Along exactly similar lines as in 2-level algorithm, the computation associated with all the epochs that are alive at the end of t updates is $O(t)$ only. So we just need to focus on the epochs that get terminated and the computation associated with each of them.

Let us first analyze the computation associated with an epoch of a matched edge (u, v). Suppose this epoch got created by vertex v at level j. So v would have executed GENERIC-RANDOM-SETTLE and selected u as a random mate from level $< j$. Note that v would be owning less than 4^{j+1} edges and u would be owning at most 4^j edges at that moment. This observation and Lemma 4.4 imply that the computation involved in the creation of the epoch is $O(4^j)$. Once the epoch is created, any update pertaining to u or v will be performed in just $O(1)$ time until the epoch gets terminated. Let us analyze the computation performed when the epoch gets terminated. At this moment either one or both of u and v become free vertices. If v becomes free, v executes the following task (see procedure PROCESS-FREE-VERTICES in Figure 6): v scans all edges owned by it, which is less than 4^{j+1}, and disowns those edges incident from vertices of level j. Thereafter, if v still owns at least 4^j edges, it settles at level j and creates a new epoch at level j. Otherwise, v keeps falling one level at a time. For a single fall of v from level i to $i-1$, the computation performed involves the following tasks: scanning the edges owned by v, disowning those incident from vertices at level i, incrementing ϕ_w values for each neighbor w of v lying at level less than i, and updating $\phi_v(i)$ to $|\mathcal{O}_v|$. All this computation is of the order of the number of edges v owns at level i which is less than 4^{i+1}. Eventually either v settles at some level $k \geq 0$ and becomes part of a new epoch or it reaches level -1. Therefore, the total computation performed by v is of the order of $\sum_{i=k}^{j} 4^{i+1} = O(4^j)$. This entire computation involving v (and u) in this process is associated with the the epoch of (u, v). Hence we can state the following lemma.

LEMMA 4.6. *For any $i \geq 0$, the computation time associated with an epoch at level i is $O(4^i)$.*

An epoch corresponding to edge (u, v) at level i could be terminated if the matched edge (u, v) gets deleted. Such an epoch is called a natural epoch. However, this epoch could also be terminated due to one of the following reasons.

- u (or v) gets selected as a random mate by one of their neighbors present at level $> i$.
- u (or v) becomes eligible to rise to some level $j > i$ because of an increase in $\phi_u(j)$ to 4^j or more.

Each of the above factors renders the epoch to be an induced epoch. For any level $i > 0$, the creation of an epoch causes termination of at most two epochs at levels $< i$. It can be explained as follows: Consider an epoch at level i associated with an edge, say (u, v). Suppose it was created by vertex u. If u was already matched at some level $j < i$, let $w \neq v$ be its mate. Similarly, if v was also matched already at some level $k < i$, let $x \neq u$ be its mate. So matching u to v terminates the epoch of (u, w) and the epoch of (v, x) at level j and k, respectively. We can thus state the following lemma.

LEMMA 4.7. *Creation of an epoch at a level i may cause termination of at most 2 epochs at level $< i$.*

4.3.1. Analyzing an epoch. Consider an epoch created by a vertex v at level i when our algorithm processes the kth update in the graph for some $k \geq 1$. At the time of the creation of the epoch, let $\mathcal{O}_v^{init}$ be the set of edges owned by v, and let $w = \textsc{mate}(v)$. The mate w is brought to level i by v. Therefore, none of the neighbors of w from level $\leq i$ can pick w as a mate so long as the neighbor stays at level $\leq i$. The epoch may terminate when the matched edge (v, w) is deleted. However, as discussed above, it may terminate even before as well if v or w moves to some level $> i$ before the deletion of (v, w). Therefore, in order to analyze the termination of an epoch, we associate an update sequence with it as follows. For each edge $(v, x) \in \mathcal{O}_v^{init}$, we consider the first time in the future that x moves to some higher level.[2] The *update label* associated with edge (v, x) is defined as follows: If x moves to a level $> i$ before the deletion of (v, x) then it is classified as *upward* else it is classified as *deletion*.

Likewise, we also consider the first time in the future that v moves to a level $> i$. If v never moves to any level $> i$ in the future, we just append v at the end of all the updates associated with $\mathcal{O}_v^{init}$. The update sequence U for the epoch is the sequence of these updates on the edges of $\mathcal{O}_v^{init}$ along with vertex v arranged in chronological order. Consider the following example. Suppose $\mathcal{O}_v^{init}$ has 10 edges and let the corresponding neighbors of v be $\{w_1, \ldots, w_{10}\}$. Let the updates in the chronological order be the following: the deletion of (v, w_4), upward movement of w_1, upward movement of w_9, the deletion of (v, w_5), and so on. The corresponding update sequence will be

$$U \;:\; \left\langle \;\overset{\bullet}{w}_4,\;\; \overset{\uparrow}{w}_1,\;\; \overset{\uparrow}{w}_9,\;\; \overset{\bullet}{w}_5,\;\; \overset{\bullet}{w}_8,\;\; \overset{\bullet}{w}_3,\;\; \overset{\uparrow}{w}_2,\;\; \overset{\uparrow}{v},\;\; \overset{\bullet}{w}_7,\;\; \overset{\uparrow}{w}_{10},\;\; \overset{\bullet}{w}_8 \;\right\rangle.$$

Observation 4.2. If the update associated with (the owner) v appears at the ℓth location in U, then the epoch will terminate on or before the ℓth update in U. Therefore, the updates at location $> \ell$ in U will have no influence on the termination of the epoch.

Unlike the 2-level algorithm, the update sequence associated with an epoch is not uniquely defined by the sequence of updates in the graph after the creation of the epoch. Rather, it also depends on the current matching as well as the random bits chosen by the algorithm while processing the future updates. So there is a probability distribution defined over all possible update sequences that depends on these two factors. Consequently, the analysis of an epoch in our final algorithm is more complex compared to the 2-level algorithm. In particular, it is not obvious whether there is any dependence between the random mate picked by a vertex while creating an epoch and the update sequence associated with the epoch. However, using an interesting and nontrivial property of our algorithm stated in Theorem 4.2, we will establish that there is no dependence between the two.

LEMMA 4.8. *Suppose vertex v creates an epoch at some level i when our algorithm processes the kth update in the graph. Let $\mathcal{O}_v^{init}$ be the set of its owned edges at the time of the creation of this epoch. Then, for any update sequence U and for each $(v, w) \in \mathcal{O}_v^{init}$,*

$$\mathbf{Pr}[\textsc{mate}(v) = w \mid U] \;=\; \mathbf{Pr}[\textsc{mate}(v) = w] \;=\; \frac{1}{|\mathcal{O}_v^{init}|}.$$

[2] Vertex x may move to a level $> i$ and down multiple times while the algorithm processes a sequence of updates. However, it is only the first time (after the creation of the epoch) when x moves to a level $> i$ that is relevant as far as the possibility of the termination of the epoch by the upward movement of x is concerned.

Remark 4.2. The sample space considered in Lemma 4.8 is defined by the random bits picked for choosing the mate of v and all the random bits picked thereafter by the algorithm for processing the current as well as all the future updates.

Lemma 4.8 can be seen as a generalization of Lemma 3.2 that we stated for our 2-level algorithm. Its proof is given in section 4.4 where we actually establish that the probability distribution of all matchings at levels $> i$ during the future updates is the same, irrespective of which mate v picks while creating an epoch at level i. This observation has the following implication. Let U be any possible update sequence for an epoch of v. After picking a mate, say w, if U occurs with probability p, then after picking any other mate, say w', the update sequence U is going to occur with the same probability p. In other words, the update sequence associated with an epoch of v is independent of the mate picked by v while creating the epoch. It is easy to notice that Lemma 4.8 is a reformulation of this fact.

The analysis of our algorithm will be critically dependent on Lemma 4.8. Using this lemma, we shall first establish a high probability bound on the total update time of the algorithm to process a sequence of updates in the graph.

4.3.2. High probability bound on the total update time. For getting a high probability bound on the total update time, we shall again use the concept of bad epoch (Definition 3.2). For the convenience of readers, we reproduce it here: *An epoch created by a vertex v is said to be bad if it gets terminated naturally before the deletion of $1/3$ edges of set $\mathcal{O}_v^{init}$ from the graph. An epoch is said to be good if it is not bad.*

It can be observed from this definition that an induced epoch is always a good epoch. Using Lemma 4.8, the lemma for the probability of a bad epoch extends seamlessly from the 2-level algorithm to our final algorithm as follows.

LEMMA 4.9. *Suppose vertex v creates an epoch at level i when the algorithm processes the kth update in the graph. This epoch will be bad with probability at most $1/3$ for any sequence of future updates in the graph and the random bits picked during their processing.*

Proof. Let $\mathcal{O}_v^{init}$ be the set of edges owned by v at the time of the creation of the epoch during the kth update. Let $(\mathcal{U}, P)$ be the probability space of all update sequences associated with the epoch for any sequence of future updates in the graph. It can be observed that the time when the epoch will be terminated is completely determined by the mate that v picks and the update sequence associated with the epoch. This update sequence, in turn, is determined by the random bits picked subsequent to the creation of the epoch and the future updates in the graph. However, as stated in Lemma 4.8, conditioned on each update sequence $U \in \mathcal{U}$, the mate of v will be uniformly distributed among the endpoints of $\mathcal{O}_v^{init}$. We shall use this fact to bound the probability of the epoch to be bad. Recall from Observation 4.2 that for the epoch to be terminated naturally, the mate of v must be among the endpoints of the deleted edges that precede v in U. So we distinguish between the following two cases.

Case 1. There are less than $|\mathcal{O}_v^{init}|/3$ edge deletions preceding v in U.
The epoch will be bad only if the matched edge of v is one of these edge deletions preceding v in U. Since the number of these edge deletions is less than $|\mathcal{O}_v^{init}|/3$, so using Lemma 4.8 the probability of the epoch to be bad in this case is less than $1/3$.

Case 2. There are at least $|\mathcal{O}_v^{init}|/3$ edge deletions preceding v in U.

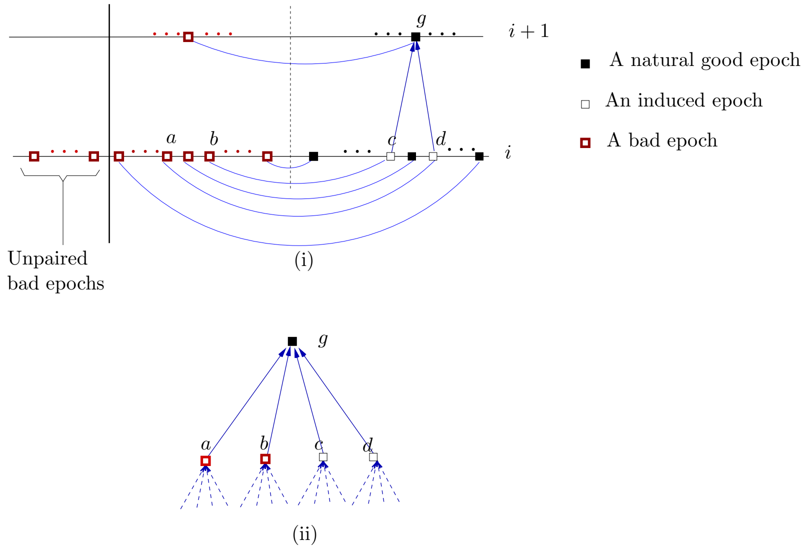

FIG. 7. (i) *Pairing bad and good epochs at level i; at most $2 \log n$ bad epochs remain unpaired with high probability.* (ii) *A tree of epochs: assigning at most 4 epochs from lower levels to an epoch.*

The epoch will be bad only if the matched edge of v is one of the first $|\mathcal{O}_v^{init}|/3$ edge deletions in U. So it follows from Lemma 4.8 that the probability of the epoch to be bad in this case is exactly $1/3$.

It follows that in both cases, the epoch is going to be bad with probability $\leq 1/3$. Hence the epoch is going to be bad with probability $\leq 1/3$ for any update sequence U associated with the epoch, that is, for any sequence of future updates in the graph and the random bits picked during their processing. $\square$

We will now show that the number of bad epochs at a level i could exceed the number of good epochs at level i by at most $O(\log n)$ with very high probability.

Consider the sequence of updates in the graph in the chronological order. Each update may create zero or more epochs at level i. Thus the number of epochs created is a random variable. However, as shown by Lemma 4.9, each epoch created at level i will be bad with probability $\leq 1/3$ irrespective of the future updates and the random bits picked by the algorithm to process them. Hence, the sequence of good and bad epochs at level i can be seen as a sequence of right and left moves during the asymmetric random walk as established in the analysis of the 2-level algorithm. So at level i, the number of bad epochs may exceed the number of good epochs by more than $2 \log_2 n$ with probability less than $1/n^2$. There are $O(\log n)$ levels in the hierarchy. Hence we get the following lemma using the *union* bound.

LEMMA 4.10. *For every level $i \leq \mathrm{L}_0$, the number of bad epochs will not exceed the number of good epochs by more than $2 \log_2 n$ with probability at least $1 - (\log n)/n^2 > 1 - 1/n$.*

We do the pairing of epochs at each level as follows—we try to pair each bad epoch at level i with a good epoch at level i in any arbitrary but unique manner. It follows from Lemma 4.10 that with high probability only $O(\log n)$ bad epochs will remain unpaired at level i. See Figure 7(i). Also the creation of each epoch at a level $i + 1$ can terminate at most two (induced) epochs at lower levels as stated in Lemma 4.7. Using this fact and the pairing between the good and bad epochs at a level, we can construct a forest whose nodes will be the epochs terminated across all levels during the algorithm. The intuition for defining this forest is that eventually the computation cost of a bad epoch or an induced epoch can be charged to a good natural epoch at some higher level. Since a good natural epoch has a sufficiently large number of edge deletions associated with it, these edge deletions can be charged to pay for all the computation carried out by our algorithm.

With this intuition, we now provide the details of the construction by defining the parent of each epoch using the following rules.

1. The parent of each induced epoch is the epoch at the higher level whose creation led to its termination.
2. The parent of a good natural epoch is itself (hence it is the root of its tree).
3. If a bad epoch is paired with an induced epoch, then its parent is the same as the parent of the induced epoch. Otherwise, it is the parent of itself (hence it is the root of its tree).

It follows from Lemma 4.7, rule 1, and rule 3 (the if part) that at most 4 epochs from lower levels can be associated with an epoch at a level. Hence each node in the forest of epochs will have at most four children. See Figure 7(ii). Moreover, the root of each tree in the forest of epochs is either a bad epoch or a good natural epoch. We analyze the total update time of the algorithm by analyzing the computation cost associated with each such tree. Using Lemma 4.6, the computation cost $C(i)$ associated with a tree of epochs whose root is at level i satisfies the following recurrence for some constant a:

$$C(i) \leq a4^i + 4C(i-1).$$

The solution of this recurrence is $C(i) = O(i4^i)$. Now there are at least $4^i/3$ edge deletions associated uniquely with a good natural epoch at a level i. This natural epoch will be charged for the computation cost $C(i) = O(i4^i)$ associated with the corresponding tree rooted at the epoch. So if t is the total number of updates in the graph, then the computation cost associated with all trees rooted at the good natural epochs in the forest is $O(t \log n)$. It follows from Lemma 4.10 and rule 3 (the otherwise part) that, after excluding the unpaired bad epochs, the number of trees rooted at good natural epochs at any level i is at least the number of trees rooted at bad epochs at level i. Hence, after excluding the trees rooted at unpaired bad epochs, the computation cost associated with all the trees in the forest is $O(t \log n)$. Since there will be $O(\log n)$ unpaired bad epochs at any level i with high probability, the computation cost associated with the trees rooted at these unpaired bad epochs at all levels is $O(\sum_{i=1}^{\log_4 n} i4^i \log n) = O(n \log^2 n)$. Hence with high probability, the computation time for processing t edge updates by the algorithm is $O(t \log n + n \log^2 n)$. This also implies that the total expected update time is $O(t \log n)$ for $t = \Omega(n \log n)$. In the following subsection, we will establish an $O(t \log n)$ bound on the expected update time for all values of t.

4.3.3. Expected value of the total update time. During a sequence of t updates in the graph, various epochs get created by various vertices at various levels. Let $X_{v,i,k}$ be a random variable which is 1 if v creates an epoch at level i during the kth update, otherwise it is 0. We denote this epoch as $\textsc{epoch}(v,i,k)$. Let $\mathcal{O}_v^{init}$ denote the set of edges that v owns at the time of the creation of the epoch. We now introduce a random variable $Z_{v,i,k}$. The creation of $\textsc{epoch}(v,i,k)$ will be followed by the updates (deletion or upward movements) on the edges of $\mathcal{O}_v^{init}$. $Z_{v,i,k}$ is the number of updates of type "edge deletion" in the corresponding update sequence associated with this epoch that occur before the epoch gets terminated. If $\textsc{epoch}(v,i,k)$ is not created, $Z_{v,i,k}$ is defined as 0. The key role in bounding the expected running time is played by a random variable $B_{v,i,k}$ defined as follows:

$$B_{v,i,k} = \begin{cases} (8Z_{v,i,k} - 2 \cdot 4^i)X_{v,i,k} & \text{if } \textsc{epoch}(v,i,k) \text{ is natural,} \\ (4^{i+1} - 2 \cdot 4^i)X_{v,i,k} & \text{if } \textsc{epoch}(v,i,k) \text{ is induced.} \end{cases} \tag{2}$$

First observe that $B_{v,i,k} = 0$ if $X_{v,i,k} = 0$, else the random variable $B_{v,i,k}$ can be seen as the *credits* associated with $\textsc{epoch}(v,i,k)$ to be used for paying its computation cost. For a natural epoch, the credits are defined in terms of the updates of type edge deletion in the corresponding update sequence that occur before the epoch gets terminated. So we define $B_{v,i,k}$ to be $8Z_{v,i,k}$. However, we need to discount for the two epochs at lower levels that may be destroyed due to $\textsc{epoch}(v,i,k)$. To this end, from the term, we deduct $2 \cdot 4^i$. Similarly, if $\textsc{epoch}(v,i,k)$ is an induced epoch, then it gets 4^{i+1} credits from the epoch that destroyed it. But here again we need to discount for the two epochs at lower levels that might be destroyed by it. To this end, we again deduct $2 \cdot 4^i$ from 4^{i+1}. The following lemma gives a bound on $\sum_{v,i,k} B_{v,i,k}$.

Lemma 4.11. $\sum_{v,i,k} B_{v,i,k} \le 8 \sum_{v,i,k} Z_{v,i,k} \le 8t$, *where t is the total number of updates in the graph.*

Proof. We need to analyze the sum of $B_{v,i,k}$'s for all those (v,i,k) tuples for which the $\textsc{epoch}(v,i,k)$ got created. If this epoch is an induced epoch, it can be associated with an epoch, say $\textsc{epoch}(v',i',k')$, at a higher level $i' > i$ whose creation destroyed it. Notice that the negative $4^{i'}$ term in $B_{v',i',k'}$ cancels out the positive 4^{i+1} term in $B_{v,i,t}$. Hence, the contribution of induced epochs in $\sum_{v,i,k} B_{v,i,k}$ is nullified and all that remains is the sum of terms $8Z_{v,i,k}$ for each natural epoch. Hence $\sum_{v,i,k} B_{v,i,k} \le 8 \sum_{v,i,k} Z_{v,i,k}$. Notice that the deletion of an edge can appear in the update sequence of at most one of its endpoints, so an edge deletion will contribute to at most one random variable $Z_{v,i,k}$. Therefore, $\sum_{v,i,k} Z_{v,i,k}$ is upper bounded by the total number of edge updates in the graph. ∎

Corollary 4.1. $\sum_{v,i,k} \mathbf{E}[B_{v,i,k}] \le 8t$.

Lemma 4.12. *For each possible (v,i,k) tuple,* $\mathbf{E}[B_{v,i,k}] \ge \mathbf{Pr}[X_{v,i,k} = 1] \cdot 4^i$.

Proof. Since $X_{v,i,k}$ is an indicator random variable,

$$\mathbf{E}[B_{v,i,k}] = \mathbf{Pr}[X_{v,i,k} = 1] \cdot \mathbf{E}[B_{v,i,k} \mid X_{v,i,k} = 1].$$

So we will estimate $\mathbf{E}[B_{v,i,k} \mid X_{v,i,k} = 1]$, that is, the expected value of $B_{v,i,k}$ given that $\textsc{epoch}(v,i,k)$ got created.

Let $(\mathcal{U}, P)$ be the probability space of all the update sequences associated with this epoch and let $U \in \mathcal{U}$ be any update sequence. Suppose among the updates in U that precede the update associated with v, only d are edge deletions. It follows from Lemma 4.8 that the matched edge of v is distributed uniformly over $\mathcal{O}_v^{init}$. So $\textsc{epoch}(v,i,k)$ will be an induced epoch with probability $(|\mathcal{O}_v^{init}| - d)/|\mathcal{O}_v^{init}|$, and in that case $B(v,i,k)$ will be $4^{i+1} - 2 \cdot 4^i$. If the epoch is natural, it could be due to any one of the d edge deletions present in U. In that case the expected value of $B_{v,i,k}$ will be $1/d \sum_{j=1}^{d} (8j - 2 \cdot 4^i) \ge 4d - 2 \cdot 4^i$. Considering the cases of induced and natural epochs together,

$$\begin{aligned} \mathbf{E}[B_{v,i,k} \mid U] &= \frac{|\mathcal{O}_v^{init}| - d}{|\mathcal{O}_v^{init}|}(4^{i+1} - 2 \cdot 4^i) + \frac{d}{|\mathcal{O}_v^{init}|}(4d - 2 \cdot 4^i) \\ &= 2 \cdot 4^i - \frac{4^{i+1}d - 4d^2}{|\mathcal{O}_v^{init}|} \ge 2 \cdot 4^i - \frac{4^i \cdot 4^i}{|\mathcal{O}_v^{init}|} \quad \text{(for all values of } d\text{).} \end{aligned}$$

Therefore,

$$\begin{aligned}\mathbf{E}[B_{v,i,k} \mid X_{v,i,k} = 1] &= \sum_{U \in \mathcal{U}} \mathbf{E}[B_{v,i,k} \mid U] \;\cdot\; \mathbf{Pr}[U] \\ &\geq \left(2 \cdot 4^i - \frac{4^i \cdot 4^i}{|\mathcal{O}_v^{init}|}\right) \cdot \sum_{U \in \mathcal{U}} \mathbf{Pr}[U] = 2 \cdot 4^i - \frac{4^i \cdot 4^i}{|\mathcal{O}_v^{init}|}.\end{aligned}$$

Since $|\mathcal{O}_v^{init}| \geq 4^i$ for level i, the result follows. ∎

Let $W_{v,i,k}$ be a random variable with value equal to the computation time associated with EPOCH(v, i, k) if the epoch is created, and 0 otherwise. It follows from Lemma 4.6 that the computation time associated with an epoch at level i is $c4^{i+1}$ for some constant c. So, $\mathbf{E}[W_{v,i,k}] = \mathbf{Pr}[X_{v,i,k} = 1]c4^{i+1}$. Therefore, using Lemma 4.12,

$$\mathbf{E}[W_{v,i,k}] \leq 4c\mathbf{E}[B_{v,i,k}]. \tag{3}$$

Using the above equation and Corollary 4.1, the total expected computation cost associated with all epochs that get destroyed during the algorithm can be bounded by $O(t)$ as follows:

$$\sum_{v,i,k} \mathbf{E}[W_{v,i,k}] \leq \sum_{v,i,k} 4c\mathbf{E}[B_{v,i,k}] \leq 32ct = O(t).$$

Since for each update in the graph, we incur $O(\log n)$ time to update ϕ at various levels, there is an $O(t \log n)$ overhead for t updates. We thus conclude with the following result.

THEOREM 4.1. *Starting with a graph on n vertices and no edges, we can maintain a maximal matching for any sequence of t updates in $O(t \log n)$ time in expectation and $O(t \log n + n \log^2 n)$ with high probability.*

Having gone through the correct analysis of the algorithm, the reader may refer to section 7.1 in the appendix for a succinct description of the error in the analysis presented in [6].

4.4. Proof of Lemma 4.8. The matching $\mathcal{M}$ maintained by our algorithm also has a level associated with each matched edge. So, for the sake of the proof of Lemma 4.8, it is useful to view the matching $\mathcal{M}$ as a set of tuples:

$$\mathcal{M} = \{(u, v, \ell) \mid u \text{ is matched to } v \text{ at level } \ell\}.$$

Our algorithm uses randomization to maintain a maximal matching. After any given sequence of updates, there is a set of possible maximal matchings that the algorithm may be maintaining and there is a probability distribution associated with these maximal matchings. So it is useful to think about the probability space of these matchings as the algorithm proceeds while processing a sequence of updates.

First, we introduce some notations. For any matching $\mathcal{M}$ maintained at any stage by our algorithm, let $\mathcal{M}_i$ denote the matching at level i. Let $\mathcal{M}_{>i} = \cup_{j>i}\mathcal{M}_j$ denote the matching at all levels $> i$. Let V_i denote the set of all the vertices belonging to levels in the range $\in [-1, i]$. We now extend the notations to incorporate the updates in the graph. For any $k \geq 1$, let $G(k)$ denote the graph after a given sequence of k updates and let $\mathcal{M}(k)$ denote the maximal matching of $G(k)$ as maintained by our algorithm. Let $\mathcal{M}_{>i}(k)$ denote the matching at all levels $> i$ after the given sequence of k updates.

After processing a certain number of updates by the algorithm, suppose M and M' are any two possible matchings such that $M_{>i} = M'_{>i}$ though the two matchings could differ at levels $\leq i$. Consider any single update in the graph at this stage. In order to process it, suppose we carry out two executions I and I' of our algorithm with the initial matching being M and M', respectively. That is, $\mathcal{M}(0) = M$ in the execution I and $\mathcal{M}(0) = M'$ in the execution I'. Our claim is that the probability distribution of matchings at levels $> i$ will be identical at the end of both executions. More precisely, for any maximal matching $\mu(1)$ on a subset of vertices in graph $G(1)$,

$$\mathbf{Pr}[\mathcal{M}_{>i}(1) = \mu(1) | \mathcal{M}(0) = M] = \mathbf{Pr}[\mathcal{M}_{>i}(1) = \mu(1) | \mathcal{M}(0) = M']. \tag{4}$$

In order to establish our claim, we shall crucially exploit the following lemma.

LEMMA 4.13. *For both the matchings M and M', $\phi_v(j)$ is the same for each $v \in V$ and each $j > i$.*

Proof. It is given that $M_{>i} = M'_{>i}$. This implies that for each level $j > i$, the set of vertices present at level j is identical in M and M'. Hence the set V_i of all the vertices present at levels $\in [-1, i]$ is identical in M and M'. Hence for any vertex v, and any level $j > i$, the set of all the neighbors of v at levels $< j$ is identical; notice that $\phi_v(j)$ is just the cardinality of this set. So it follows that $\phi_v(j)$ is the same for each vertex v and each $j > i$. ∎

We shall now establish the validity of (4) for the deletion of an edge $e = (u, v)$. Establishing its validity for the insertion of an edge is similar and is sketched in the appendix. Notice that our algorithm does not alter the matching if e is not a matched edge. If e is a matched edge, a wave of free vertices originates from level(e) and propagates downward. The following facts follow from the description of our algorithm (see section 4.2.1).

- $F1$. The algorithm won't alter the matching at levels $>$ level(e) while processing the deletion of e.
- $F2$. The matching is updated by our algorithm in the decreasing order of levels, and once the updating of the matching at a level is complete, the matching at that level will remain unchanged during the updates of the matching at lower levels.

It follows from the description of M and M' that either level(e) is less than or equal to i in both the matchings or level(e) is the same in M and M'. Let us first consider the (easier) case when level(e) $\leq i$ in M as well as in M'. It follows from fact $F1$ stated above that the only changes in matchings M and M' will be at levels $\leq i$. Hence the matching $\mathcal{M}_{>i}(1)$ will be identical at the end of both the executions I and I'. Let us now consider the more interesting case of level(e) $> i$. Let level(e) $= \ell$. Both the executions I and I' invoke the procedure PROCESS-FREE-VERTICES($\langle (u, \ell), (v, \ell) \rangle$) in this case. For a better understanding, the reader is recommended to revisit this procedure from section 4.2.1 before proceeding further.

In order to establish our claim about I and I', we shall establish the following assertion. While the matching at levels $> i$ is being updated, for each step executed in I, the same step will be executed in I'. Moreover, if the step in I is executed with some probability, the step will be executed with the same probability in I' as well. In order to establish this assertion, let us analyze the first iteration of the procedure PROCESS-FREE-VERTICES. Both I and I' will process u first. After disowning its edges from its present level, u owns the same set of edges in both the executions. Thereafter, u will either stay at level ℓ or fall by one level. If u stays at level ℓ, it chooses a random edge to get matched. The probability that any specific random edge is picked by u is the same in both the executions. Let us consider the case that u falls by one level. It follows from Lemma 4.13 that $\phi_x(j)$ is the same in I as well as I' for each vertex x and each level $j > i$ just before the fall of u. When u falls by one level, for each neighbor z of u at any level $< \ell$, $\phi_z(\ell)$ increases exactly by one and remains unchanged for all other levels (see Lemma 4.2). Hence the set of neighbors of u rising to level ℓ are the same in both the executions I and I'. In addition, the set of edges that each such vertex owns on rising to level ℓ is also the same, hence, the probability that any specific random mate is picked is the same in both the executions. So each update in M and M' is equally likely during the processing of u. The reader may note that after each such identical update in M and M', the matchings are identical at each level $> i$. Hence, Lemma 4.13 holds again for the updated matchings. Unlike the first iteration, a generic iteration of the procedure PROCESS-FREE-VERTICES may have free vertices at levels $\leq$ LEVEL(e) that are kept in respective queues at these levels. Suppose in the beginning of any such iteration of the procedure PROCESS-FREE-VERTICES there are two possible configurations such that the matching as well as the queue storing the free vertices are identical at each level $> i$ but differ at levels $\leq i$. Notice that Lemma 4.13 will hold for these configurations as well. Therefore, along exactly the same lines as the first iteration analyzed above, it can be shown that every update in the matching at level $> i$ will be carried out with the same probability during any generic iteration for any two configurations that match at all levels $> i$.

Therefore, each sequence of updates in the matching is equally likely in both the executions I and I' till the last free vertex at level $i + 1$ is processed. Henceforth, the two executions may differ. But as follows from fact $F2$, it will affect only the matching at levels $\leq i$ and there won't be any change in the matching at higher levels. This establishes our claim, i.e., (4), for any single update in the graph. This claim can be invoked appropriately for a sequence of updates giving us the following theorem.

THEOREM 4.2. *Let M and M' be any two possible matchings by our algorithm at any time such that $M_{>i} = M'_{>i}$. For any sequence of t updates in the graph, suppose we carry out two executions I and I' of our algorithm with the initial matchings being M and M', respectively. The probability distribution of matching at every level $> i$ will be identical after each update in both the executions. That is,*

$$\begin{aligned}\mathbf{Pr}[\mathcal{M}_{>i}(t) = \mu(t), \ldots, \mathcal{M}_{>i}(1) = \mu(1) | \mathcal{M}(0) = M] \\ = \mathbf{Pr}[\mathcal{M}_{>i}(t) = \mu(t), \ldots, \mathcal{M}_{>i}(1) = \mu(1) | \mathcal{M}(0) = M'],\end{aligned}$$

where $\mu(j)$, for $1 \leq j \leq t$, is any maximal matching on a subset of vertices in the graph $G(j)$.

For the proof of Theorem 4.2, we shall apply the argument for a single update inductively and use the following lemma from elementary probability theory.

LEMMA 4.14. *Suppose A, B, C are three events defined over a probability space (Ω, P). Then,*

$$\mathbf{Pr}[A \cap B \mid C] = \mathbf{Pr}[A \mid B \cap C] \cdot \mathbf{Pr}[B \mid C].$$

Let us define events C as $\mathcal{M}(0) = M$ and C' as $\mathcal{M}(0) = M'$. Let us define event B as $\mathcal{M}_{>i}(1) = \mu(1)$. We have shown that

$$\mathbf{Pr}[\mathcal{M}_{>i}(1) = \mu(1) \mid C] = \mathbf{Pr}[\mathcal{M}_{>i}(1) = \mu(1) \mid C'].$$

That is, $\mathbf{Pr}[B \mid C] = \mathbf{Pr}[B \mid C']$. By another application of the arguments that we used for a single update, we get

$$\mathbf{Pr}[\mathcal{M}_{>i}(2) = \mu(2) \mid B, C] = \mathbf{Pr}[\mathcal{M}_{>i}(2) = \mu(2) \mid B, C'].$$

Applying Lemma 4.14, we get

$$\mathbf{Pr}[\mathcal{M}_{>i}(2) = \mu(2), B \mid C] = \mathbf{Pr}[\mathcal{M}_{>i}(2) = \mu(2) \mid B, C] \cdot \mathbf{Pr}[B \mid C].$$

Since $\mathbf{Pr}[B \mid C] = \mathbf{Pr}[B \mid C']$, it follows that

$$\mathbf{Pr}[\mathcal{M}_{>i}(2) = \mu(2), B \mid C] = \mathbf{Pr}[\mathcal{M}_{>i}(2) = \mu(2), B \mid C'].$$

The above argument can be inductively applied for every subsequent update. This completes the proof of Theorem 4.2.

4.4.1. Connection to the analysis. We first state two lemmas from elementary probability theory that deal with the independence of events. For the sake of completeness, the proof of these lemmas is given in the appendix.

The first lemma deals with conditional probability.

LEMMA 4.15. *Let A be an event and $B_1, \ldots, B_k$ be k mutually exclusive events defined over a probability space (Ω, P). If $\mathbf{Pr}[A \mid B_j] = \rho$ for each $1 \le j \le k$, then $\mathbf{Pr}[A \mid C] = \rho$, where event $C = \cup_j B_j$.*

The second lemma deals with independence of events. Let A and B be two events defined over a probability space (Ω, P). A is said to be independent of B if $\mathbf{Pr}[A \mid B] = \mathbf{Pr}[A \mid \bar{B}] = \mathbf{Pr}[A]$. Alternatively, $\mathbf{Pr}[A \cap B] = \mathbf{Pr}[A] \cdot \mathbf{Pr}[B]$. The notion of independence gets carried over from events to random variables in a natural manner as follows.

DEFINITION 4.1. *An event A is said to be independent of a random variable X if for each $x \in X$, $\mathbf{Pr}[A \mid X = x] = \mathbf{Pr}[A]$.*

LEMMA 4.16. *Suppose A is an event and X is a random variable defined over probability space (Ω, P). If A is independent of X, then for each $x \in X$,*

$$\mathbf{Pr}[X = x \mid A] = \mathbf{Pr}[X = x].$$

Now we shall establish the connection of Theorem 4.2 to the analysis of our algorithm. In particular, we shall use this theorem to prove Lemma 4.8. Suppose a vertex v creates an epoch at level i while the algorithm processes the kth update in the graph for any $k < t$. We shall analyze the probability space of the future matchings starting from the time just before the creation of this epoch.

While creating its epoch, v chooses its mate randomly uniformly out of $\mathcal{O}_v^{init}$. Clearly, the change in the matching at levels $\le i$ will depend on the mate that v picks. Let $\mathbf{M}$ be the set of all possible matchings once the algorithm completes the processing of the kth update. Notice that all matchings from the set $\mathbf{M}$ are identical at each level $> i$ due to fact $F2$. So it follows from Theorem 4.2 that for any two matchings $M, M' \in \mathbf{M}$,

$$\begin{aligned}\mathbf{Pr}[\mathcal{M}_{>i}(t) = \mu(t), \ldots, = \mathcal{M}_{>i}(k+1) = \mu(k+1) \mid \mathcal{M}(k) = M] \\ = \mathbf{Pr}[\mathcal{M}_{>i}(t) = \mu(t), \ldots, \mathcal{M}_{>i}(k+1) = \mu(k+1) \mid \mathcal{M}(k) = M'].\end{aligned}$$

Let this conditional probability be ρ. For each $(v, w) \in \mathcal{O}_v^{init}$, there may be multiple matchings in $\mathbf{M}$ in which v is matched to w. By applying Lemma 4.15, the following equation holds for every $(v, w) \in \mathcal{O}_v^{init}$:

$$\mathbf{Pr}[\mathcal{M}_{>i}(t) = \mu(t), \ldots, \mathcal{M}_{>i}(k+1) = \mu(k+1) \mid \text{MATE}(v) = w] = \rho.$$

Notice that this probability is the same for each $(v, w) \in \mathcal{O}_v^{init}$. So using Definition 4.1, it follows that the matchings at levels $> i$ during any sequence of updates is independent of the mate that v picks during the creation of its epoch. Now applying Lemma 4.16 we get the following lemma.

LEMMA 4.17. *Suppose a vertex v creates an epoch at level i while the algorithm processes the kth update in the graph. Consider any sequence of subsequent updates in the graph. The mate picked by v while creating the epoch is independent of the sequence of matchings at levels $> i$ computed by the algorithm while processing these updates. That is, for any $t > k$, and any $(v, w) \in \mathcal{O}_v^{init}$,*

$$\begin{aligned}\mathbf{Pr}[\text{MATE}(v) = w | \mathcal{M}_{>i}(t) = \mu(t), \ldots, \mathcal{M}_{>i}(k+1) = \mu(k+1)] &= \mathbf{Pr}[\text{MATE}(v) = w] \\ &= \frac{1}{|\mathcal{O}_v^{init}|}.\end{aligned}$$

Consider any given sequence of t updates in the graph. Subsequent to the time when v creates an epoch at level i during (the processing of) the kth update, let $\mu = \langle \mu(k+1), \ldots, \mu(t) \rangle$ be the sequence of matchings at levels $> i$ as computed by our algorithm after each of the subsequent updates in the graph. Notice that the upward movement of v and each $(v, w) \in \mathcal{O}_v^{init}$ after the creation of the epoch is captured precisely by the corresponding update in the matching at levels $> i$. Therefore, using μ, we can define the update sequence associated with the epoch as follows. Consider an edge $(v, w) \in \mathcal{O}_v^{init}$, and let the ℓth update in the graph be the deletion of (v, w). Let $j < \ell$ be the smallest integer such that $w \in \mu(j)$, that is, w appears in the matching at a level $> i$ while processing of the jth update in the graph, then the update associated with (v, w) is the upward movement. If no such j exists, the update associated with (v, w) is its deletion. Likewise, we define the update associated with v. The update sequence U for the epoch is the sequence of these updates on the edges of $\mathcal{O}_v^{init}$ along with vertex v arranged in chronological order.

For an update sequence U associated with an epoch, there may exist many sequences $\{\mu_1, \ldots, \mu_q\}$ of matchings at levels $> i$ such that for each of them, the update sequence associated with the epoch is U. It follows from Lemma 4.17 that the mate picked by v while creating its epoch at level i is independent of each such sequence $\mu_r, 1 \le r \le q$. Therefore, using Lemma 4.15, the mate picked by v while creating its epoch at level i is independent of U as well. Thus we have established the validity of Lemma 4.8.

5. A tight example. We tested our algorithm on random graphs of various densities and found that the matching maintained is very close to the maximum matching. This suggests that our algorithm might be able to maintain *nearly* maximum matching for dynamic graphs appearing in various practical applications. However, it is not hard to come up with an update sequence such that at the end of the sequence, the matching obtained by our algorithm is strictly half the size of maximum matching. In other words, the approximation factor 2 for the matching maintained by our algorithm is indeed tight. We present one such example as follows (see Figure 8).

Let $G(V \cup W, E)$ be a graph such that $V = \{v_1, \ldots, v_n\}$ and $W = \{w_1, \ldots, w_n\}$ for some even number n. Consider the following update sequence. In the first phase, add edges between every pair of vertices present in V. This results in a complete subgraph on vertices of V. The size of any maximal matching on a complete graph of size n is $n/2$. After the first phase of updates ends, the size of matching obtained by our algorithm is $n/2$. In the second phase, add edge (v_i, w_i) for all i. Note that the degree of each w_i is one at the end of the updates. Let us now find the matching

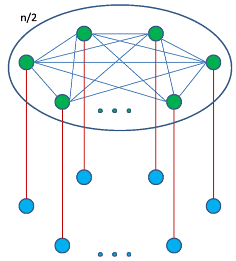

FIG. 8. *An example where our algorithm gives a 2-approximation. The vertices on top are in V and form a complete graph. The vertices at the bottom of the figure are in W.*

which our algorithm maintains. Let (v_i, v_j) be an edge in the matching after phase 1. Note that both these endpoints are at a level greater than -1. A vertex in W is at level -1 as it does not have any adjacent edges after phase 1. When an edge (w_i, v_i) is added, since v_i is at a higher level than w_i, v_i becomes the owner of this edge. The second invariant of v_i is not violated after this edge insertion and nothing happens at this update step and w_i still remains at level -1. Using the same reasoning, we can show that w_j also remains at level -1 after the addition of edge (v_j, w_j). So matching maintained by the algorithm remains unchanged. It is easy to observe that the maximum matching of the graph G now has size n which is twice the size of the matching maintained by our algorithm.

6. Conclusion. We presented a fully dynamic randomized algorithm for maximal matching which achieves expected amortized $O(\log n)$ time per edge insertion or deletion. An interesting question is to explore how crucial randomization is for dynamic maximal matching.

Recently Bhattacharya, Henzinger, and Nanongkai [10] almost answered this question in the affirmative by designing a determinstic algorithm that maintains $(2+\epsilon)$-approximate matching in amortized $O(poly(\log n, 1/\epsilon)$ update time. Another interesting question is to explore whether we can achieve $O(1)$ amortized update time. Very recently Solomon [31] answered this question in the affirmative as well by designing a randomized algorithm that takes $O(t + n\log n)$ update time with high probability to process any sequence of t edge updates. Though the basic building blocks of his algorithm are the same as ours, the two algorithms are inherently different and so is their analysis.

In our algorithm, a vertex may rise to a higher level and create a new epoch even when its matched edge is intact. But the algorithm of Solomon [31] takes a lazy approach to maintain the hierarchy of vertices wherein a vertex is processed only when it becomes absolutely necessary. Another crucial difference is the following. Our algorithm maintains a function $\phi_v(j)$ for each vertex v and each level j. This function is used to ensure an invariant that each vertex v is at the highest possible level ℓ such that the number of edges incident from lower levels is at least 4^ℓ. An important property guaranteed by this invariant is that the mate of a vertex while creating an epoch is independent of the update sequence associated with the epoch. The analysis of our algorithm crucially exploits this property. However, the explicit maintenance of $\phi_v(j)$ imposes an overhead of $\Theta(\log n)$ in the update time. In order to achieve $O(1)$ update time, Solomon [31] gets rid of the maintenance of $\phi_v(j)$ by taking a lazy approach and a couple of new ideas. As a result, unfortunately, the property of our algorithm no longer holds for the algorithm of Solomon [31]—indeed there is dependence between the update sequence associated with an epoch created by a vertex and the random mate picked by it. Solomon [31] makes use of a new concept called *uninterrupted duration* of an epoch that bypasses the need of our property for the analysis. His analysis can be adapted to our algorithm as well and can be viewed as the correct counterpart of [6, Lemma 4.10]. However, our new analysis presented in this article has its own merits since it is based on an insightful property of our algorithm which we believe is of its own independent interest and importance.

Subsequent to the publication of [6] there has been interesting progress in the area of dynamic matching with approximation less than 2 [8, 9, 16, 26], and dynamic weighted matching [3, 4, 16].

One of the technical challenges in theoretical computer science is to prove lower bounds for algorithmic problems. Recently there has been some progress on proving conditional lower bounds for dynamic graph algorithms [1, 17]. In the light of the lower bound presented by Abboud and Williams [1] based on $\Omega(n^2)$ hardness of the 3SUM problem, it would be an interesting and challenging problem to see if c-approximate maximum matching for $c < 2$ can be maintained in $o(n)$ update time.

7. Appendix.

7.1. The error in the analysis of the main algorithm in [6].

In order to establish the high probability bound on the update time of our algorithm, we took the following approach in [6]. We introduced the following terminology. An edge e is said to be deleted at level i if at the time of the deletion, one of the endpoints of e is present at level i. So an edge deletion is associated with at most 2 levels. We made an attempt to bound the number of natural epochs terminated at a level in terms of the number of edge deletions occuring at that level. We made the following claim (stated as [6, Lemma 4.10]). If t_i is the number of edge deletions occuring at level i, then the number of natural epochs terminated at a level i is $O(t_i/2^i + \log n)$ with high probability. However, the following example shows that this claim does not hold for some situations.

Consider an epoch created by a vertex u. Let the set of edges $\mathcal{O}_u^{init}$ owned by u at the time of creation of the epoch be incident from vertices $v_1, \ldots, v_q$. Without loss of generality, suppose the sequence of edge deletions of the owned edges is $\langle (u, v_1), \ldots, (u, v_q) \rangle$. Duration of the epoch is defined as the number of deletions of the edges from $\mathcal{O}_u^{init}$ before the epoch gets terminated. Since u picks its mate randomly, if all the updates associated with the epoch are deletions only, the duration of the epoch will be a random variable distributed uniformly in the range $[1, q]$. This property was rightly used to establish that the number of epochs terminated at level 1 in our 2-level algorithm is bounded by $O(t/\sqrt{n} + \log n)$ with high probability. However, this property fails in the case where the updates associated with the edges of $\mathcal{O}_u^{init}$ can be upward movement as well. As an illustration of this, suppose the update sequence associated with the epoch is

$$U \;:\; \left\langle \;\overset{\uparrow}{v_2},\;\; \overset{\uparrow}{v_3},\;\; \cdots, \overset{\uparrow}{v_{q-1}},\;\; \overset{\uparrow}{v_q},\;\; \overset{\bullet}{v_1},\; \right\rangle.$$

Basically before the first edge, that is (u, v_1), is deleted, all other edges from the set $\mathcal{O}_u^{init}$ move upward. If the mate picked by u is v_1, then the epoch will terminate naturally, and only one edge deletion, that is (u, v_1), will be associated with the termination of this epoch. If the mate of u is any other vertex, the epoch will be induced epoch only and no edge deletion will be associated with the epoch. Now suppose a series of epochs appear at level i such that the update sequence for each of them looks identical to the update sequence shown above. It follows that if t_i edge deletions take place at level i, the number of natural epochs terminated at level i will be t_i and not $O(t_i/2^i)$. Hence [6, Lemma 4.10] is incorrect. However, in the current article, we presented an alternate way to establish that the total update time for maintaining a maximal matching by our algorithm for any sequence of t updates is $O(t + n \log n)$ with high probability. The new analysis is based on an insightful property of the algorithm, stated in Theorem 4.2, and it makes use of a more careful amortization argument spanning multiple levels.

7.2. Proof of (4) for insertion of an edge. Consider insertion of an edge (u, v). It follows from the initial matchings M and M' that just after this insertion, the edges that u (likewise v) will own upon rising to any level $j > i$ will be the same though it may differ for $j \leq i$. Lemma 4.13 indeed holds. So if u rises to a level $j > i$ in I, u will also rise to level j in I as well. Otherwise, that is, if u rises to a level $\leq i$ in I, then u may rise, if at all, to a level $\leq i$. In the latter case, the changes in M and M' will be only in levels $\leq i$ due to fact $F2$ and we are done. In the former case, where u rises to the same level $j > i$ in I and I', henceforth, the proof becomes identical to the case of deletion of an edge.

7.3. Proof of Lemma 4.15.

Proof.

$$\begin{aligned}
\mathbf{Pr}[A \cap C] &= \mathbf{Pr}[A \cap (\cup_i B_i)] \\
&= \sum_i \mathbf{Pr}[A \cap B_i] \qquad \text{since } B_i\text{'s are mutually exclusive} \\
&= \sum_i \mathbf{Pr}[A \mid B_i] \cdot \mathbf{Pr}[B_i] \qquad \text{using the definition of conditional probability} \\
&= \rho \cdot \sum_i \mathbf{Pr}[B_i] \\
&= \rho \cdot \mathbf{Pr}[\cup_i B_i] = \rho \cdot \mathbf{Pr}[C] \qquad \text{since } B_i\text{'s are mutually exclusive}
\end{aligned}$$

Hence $\mathbf{Pr}[A \mid C] = \mathbf{Pr}[A \cap C]/\mathbf{Pr}[C] = \rho$. ∎

7.4. Proof of Lemma 4.16.

Proof. Since A is independent of X, so for each $x \in X$,

$$\mathbf{Pr}[A \cap X = x] = \mathbf{Pr}[A] \cdot \mathbf{Pr}[X = x]. \tag{5}$$

Hence

$$\begin{aligned}
\mathbf{Pr}[X = x|A] &= \frac{\mathbf{Pr}[A \cap X = x]}{\mathbf{Pr}[A]} \\
&= \frac{\mathbf{Pr}[A] \cdot \mathbf{Pr}[X = x]}{\mathbf{Pr}[A]} \qquad \text{using (5)} \\
&= \mathbf{Pr}[X = x].
\end{aligned}$$

∎

Acknowledgments. We are very thankful to Sayan Bhattacharya and Divyarthi Mohan for pointing out the error in [6, Lemma 4.10]. The second author would also like to thank them for the discussions on the proof of the expectation bound and the definition of $B(v, i, k)$. In [6], we used 2^i as the threshold for raising a vertex to level i. The possibility of increasing this threshold from 2^i to b^i for any constant b without any impact on the time complexity of the algorithm was observed by Shay Solomon [31]. We are very thankful to him for this useful observation. Last but not least we are very grateful to the anonymous referee for a very meticulous and thorough review of this article.

REFERENCES

[1] A. Abboud and V. V. Williams, *Popular conjectures imply strong lower bounds for dynamic problems*, in 55th IEEE Annual Symposium on Foundations of Computer Science, FOCS 2014, Philadelphia, IEEE, Piscataway, NJ, 2014, pp. 434–443, https://doi.org/10.1109/FOCS.2014.53.

[2] D. J. Abraham, R. W. Irving, T. Kavitha, and K. Mehlhorn, *Popular matchings*, SIAM J. Comput., 37 (2007), pp. 1030–1045, https://doi.org/10.1137/06067328X.

[3] A. Anand, S. Baswana, M. Gupta, and S. Sen, *Maintaining approximate maximum weighted matching in fully dynamic graphs*, in IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science, FSTTCS 2012, Hyderabad, India, Schloss Dagstuhl, Wadern, Germany, 2012, pp. 257–266, https://doi.org/10.4230/LIPIcs.FSTTCS.2012.257.

[4] A. Anand, S. Baswana, M. Gupta, and S. Sen, *Maintaining approximate maximum weighted matching in fully dynamic graphs*, preprint, arXiv:1207.3976, 2012.

[5] S. Baswana, S. R. Chaudhury, K. Choudhary, and S. Khan, *Dynamic DFS in undirected graphs: breaking the $O(\mathrm{m})$ barrier*, in Proceedings of the Twenty-Seventh Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2016, Arlington, VA, SIAM, Philadelphia, 2016, pp. 730–739, https://doi.org/10.1137/1.9781611974331.ch52.

[6] S. Baswana, M. Gupta, and S. Sen, *Fully dynamic maximal matching in $O(\log n)$ update time*, (Preliminary version appeared in FOCS 2011) SIAM J. Comput., 44 (2015), pp. 88–113, https://doi.org/10.1137/130914140.

[7] S. Baswana, S. Khurana, and S. Sarkar, *Fully dynamic randomized algorithms for graph spanners*, ACM Trans. Algorithms, 8 (2012), 35, http://doi.acm.org/10.1145/2344422.2344425.

[8] A. Bernstein and C. Stein, *Fully dynamic matching in bipartite graphs*, in Proceedings of Automata, Languages, and Programming - 42nd International Colloquium, ICALP 2015, Part I, Springer, Berlin, 2015, pp. 167–179, http://doi.org/10.1007/978-3-662-47672-7_14.

[9] A. Bernstein and C. Stein, *Faster fully dynamic matchings with small approximation ratios*, in Proceedings of the Twenty-Seventh Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2016, Arlington, VA, SIAM, Philadelphia, 2016, pp. 692–711, http://doi.org/10.1137/1.9781611974331.ch50.

[10] S. Bhattacharya, M. Henzinger, and D. Nanongkai, *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, ACM, New York, 2016, pp. 398–411, http://doi.acm.org/10.1145/2897518.2897568.

[11] J. Edmonds, *Paths, trees, and flowers*, Canad. J. Math., 17 (1965), pp. 449–467.

[12] J. Edmonds and E. L. Johnson, *Matching, Euler tours, and the Chinese postman*, Math. Program., 5 (1973), pp. 88–124.

[13] W. Feller, *Random walk and ruin problems*, in An Introduction to Probability Theory and Its Applications, Vol. 1, 3rd ed., Wiley, New York, 2000, pp. 344–348.

[14] H. N. Gabow and R. E. Tarjan, *Faster scaling algorithms for general graph-matching problems*, J. ACM, 38 (1991), pp. 815–853, http://doi.acm.org/10.1145/115234.115366.

[15] D. Gale and L. S. Shapley, *College admissions and the stability of marriage*, Amer. Math. Monthly, 69 (1962), pp. 9–14.

[16] M. Gupta and R. Peng, *Fully dynamic $(1+e)$-approximate matchings*, in 54th Annual IEEE Symposium on Foundations of Computer Science, FOCS 2013, 2013, Berkeley, CA, IEEE, Piscataway, NJ, 2013, pp. 548–557, https://doi.org/10.1109/FOCS.2013.65.

[17] M. Henzinger, S. Krinninger, D. Nanongkai, and T. Saranurak, *Unifying and strengthening hardness for dynamic problems via the online matrix-vector multiplication conjecture*, in Proceedings of the Forty-Seventh Annual ACM Symposium on Theory of Computing, STOC 2015, Portland, OR, ACM, New York, 2015, pp. 21–30, http://doi.acm.org/10.1145/2746539.2746609.

[18] J. Holm, K. de Lichtenberg, and M. Thorup, *Poly-logarithmic deterministic fully-dynamic algorithms for connectivity, minimum spanning tree, 2-edge, and biconnectivity*, J. ACM, 48 (2001), pp. 723–760, http://doi.acm.org/10.1145/502090.502095.

[19] J. E. Hopcroft and R. M. Karp, *An $n^{5/2}$ algorithm for maximum matchings in bipartite graphs*, SIAM J. Comput., 2 (1973), pp. 225–231, https://doi.org/10.1137/0202019.
[20] C.-C. Huang and T. Kavitha, *Efficient algorithms for maximum weight matchings in general graphs with small edge weights*, in Proceedings of the Twenty-Third Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2012, Kyoto, Japan, SIAM, Philadelphia, 2012, 2012, pp. 1400–1412, http://portal.acm.org/citation.cfm?id=2095226&CFID=63838676&CFTOKEN=79617016.
[21] J. Kleinberg and E. Tardos, *Algorithm Design*, Addison Wesley, Boston, 2005.
[22] E. Lawler, *Combinatorial Optimization: Networks and Matroids*, Holt, Rinehart & Winston, New York, 1976.
[23] L. Lovasz and M. Plummer, *Matching Theory*, North-Holland, Amsterdam, 1986.
[24] S. Micali and V. V. Vazirani, *An O(sqrt($|v|$) $|e|$) algorithm for finding maximum matching in general graphs*, in 21st Annual Symposium on Foundations of Computer Science, Syracuse, New York, Long Beach, CA, 1980, pp. 17–27, https://doi.org/10.1109/SFCS.1980.12.
[25] M. Mucha and P. Sankowski, *Maximum matchings via Gaussian elimination*, in Proceedings of the 45th Symposium on Foundations of Computer Science (FOCS 2004), Rome, Italy, IEEE Computer Society, Los Alamitos, CA, 2004, pp. 248–255, https://doi.org/10.1109/FOCS.2004.40.
[26] O. Neiman and S. Solomon, *Simple deterministic algorithms for fully dynamic maximal matching*, ACM Trans. Algorithms, 12 (2016), 7, http://doi.acm.org/10.1145/2700206.
[27] K. Onak and R. Rubinfeld, *Maintaining a large matching and a small vertex cover*, in Proceedings of the 42nd ACM Symposium on Theory of Computing, STOC 2010, Cambridge, Massachusetts, ACM, New York, pp. 457–464, http://doi.acm.org/10.1145/1806689.1806753.
[28] R. Pagh and F. F. Rodler, *Cuckoo hashing*, J. Algorithms, 51 (2004), pp. 122–144.
[29] L. Roditty and U. Zwick, *Improved dynamic reachability algorithms for directed graphs*, SIAM J. Comput., 37 (2008), pp. 1455–1471, https://doi.org/10.1137/060650271.
[30] L. Roditty and U. Zwick, *Dynamic approximate all-pairs shortest paths in undirected graphs*, SIAM J. Comput., 41 (2012), pp. 670–683, https://doi.org/10.1137/090776573.
[31] S. Solomon, *Fully dynamic maximal matching in constant update time*, in IEEE 57th Annual Symposium on Foundations of Computer Science, FOCS 2016, New Brunswick, NJ, IEEE, Piscataway, NJ, 2016, pp. 325–334, https://doi.org/10.1109/FOCS.2016.43.
[32] M. Thorup, *Fully-dynamic min-cut*, Combinatorica, 27 (2007), pp. 91–127.