2016 IEEE 57th Annual Symposium on Foundations of Computer Science

Fully Dynamic Maximal Matching in Constant Update Time

Shay Solomon
Stanford University
Email: shayso@cs.stanford.edu

***Abstract*—Baswana, Gupta and Sen [FOCS'11] showed that fully dynamic maximal matching can be maintained in general graphs with logarithmic amortized update time. More specifically, starting from an empty graph on n fixed vertices, they devised a randomized algorithm for maintaining maximal matching over any sequence of t edge insertions and deletions with a total runtime of $O(t \log n)$ in expectation and $O(t \log n + n \log^2 n)$ with high probability. Whether or not this runtime bound can be improved towards $O(t)$ has remained an important open problem. Despite significant research efforts, this question has resisted numerous attempts at resolution even for basic graph families such as forests.**

In this paper, we resolve the question in the affirmative, by presenting a randomized algorithm for maintaining maximal matching in general graphs with *constant* amortized update time. The optimal runtime bound $O(t)$ of our algorithm holds both in expectation and with high probability.

As an immediate corollary, we can maintain 2-approximate vertex cover with constant amortized update time. This result is essentially the best one can hope for (under the unique games conjecture) in the context of dynamic approximate vertex cover, culminating a long line of research.

Our algorithm builds on Baswana et al.'s algorithm, but is inherently different and arguably simpler. As an implication of our simplified approach, the space usage of our algorithm is linear in the (dynamic) graph size, while the space usage of Baswana et al.'s algorithm is always at least $\Omega(n \log n)$.

Finally, we present applications to approximate weighted matchings and to distributed networks.

***Keywords*-dynamic algorithm; dynamic matching; maximal matching; vertex cover;**

I. Introduction

1.1 Dynamic maximal matching. For any graph $G = (V, E)$, with $n = |V|, m = |E|$, an (inclusion-wise) *maximal matching* $\mathcal{M} = \mathcal{M}(G)$ can be computed in time $O(n + m)$ via a naïve greedy algorithm.[1] A fundamental challenge is to efficiently maintain a maximal matching in a *fully dynamic setting*. Starting from an empty graph G_0 on n fixed vertices, at each time step i a single edge (u, v) is either inserted to the graph G_{i-1} or deleted from it, resulting in graph G_i, and the dynamic algorithm should update the maintained matching $\mathcal{M} = \mathcal{M}(G_{i-1})$ to preserve maximality w.r.t. the new graph G_i. The holy grail is that the total runtime of the algorithm would be linear in the total number t of update steps. Put in other words, if the total runtime is T, the *amortized* update time T/t should be constant.

There is a naïve deterministic algorithm for maintaining a maximal matching with an update time of $O(n)$.[2] In 1993, Ivković and Lloyd [13] devised a deterministic algorithm with update time $O((n+m)^{\frac{\sqrt{2}}{2}})$, which, despite being quite involved, is inferior to the naïve algorithm in the regime $m = \omega(n^{\sqrt{2}})$. The state-of-the-art deterministic update time, by Neiman and this author [21], is $O(\sqrt{m})$.

Baswana, Gupta and Sen [3] used randomization to obtain an exponential improvement in the update time, under the oblivious adversarial model.[3] Specifically, they devised a randomized algorithm for maintaining a maximal matching over any sequence of t edge insertions and deletions with a total runtime of $O(t \log n)$ in expectation and $O(t \log n + n \log^2 n)$ with high probability (w.h.p.). In other words, the (amortized) update time of their algorithm is $O(\log n)$ in expectation and $O(\log n + \frac{n \log^2 n}{t})$ w.h.p. (For $t = o(n \log n)$, the high probability bound becomes super-logarithmic; e.g., for $t = \Theta(n)$, it is $O(\log^2 n)$.)

Whether or not the update time of [3] can be improved towards constant has remained an important open problem. Some progress towards its resolution was made for uniformly sparse graphs, such as forests, planar graphs and graphs excluding fixed minors [21], [11], [16]. However, the state-of-the-art update time in such graphs is $O(\sqrt{\log n})$ [11], even in forests, which is still far from constant.

Our contribution. We resolve this basic question in the affirmative, by presenting a randomized algorithm for maintaining maximal matching in *general graphs* with *constant* update time. The optimal runtime bound $O(t)$ of our algorithm holds both in expectation and w.h.p.

Our algorithm builds on Baswana et al.'s algorithm [3], but is *inherently different*. Also, it is arguably much simpler (though quite tricky), both conceptually and technically. This simplification is, in our opinion, an important contribution

[1] Any edge of $\mathcal{M}$ (resp., $E \setminus \mathcal{M}$) is called *matched* (resp., *unmatched*). A vertex incident on a matched edge is called *matched*, and the other endpoint is its *mate*; a vertex that is not matched is called *free*.

[2] The naïve update time bound holds also in the *worst case*. In this paper we focus on amortized time bounds, so in some of the literature survey that follows we do not distinguish between amortized and worst-case bounds.

[3] The *oblivious adversarial model* is a standard model, which has been used for analyzing randomized data-structures such as universal hashing [7] and dynamic connectivity [14]. The model allows the adversary to know all the edges in the graph and their arrival order, as well as the algorithm to be used. However, the adversary is not aware of the random bits used by the algorithm, and so cannot choose updates adaptively in response to the randomly guided choices of the algorithm.

0272-5428/16 $31.00 © 2016 IEEE
DOI 10.1109/FOCS.2016.43

by itself. As a practical implication of our simplified approach, we implement the algorithm using *optimal* space $O(n+m)$, where m stands for the dynamic number of edges in the graph. This should be contrasted to the space usage $O(n \log n + m)$ of [3], which is suboptimal when $m = o(n \log n)$. In particular, whenever $m = O(n)$ (which is always the case in uniformly sparse graphs, e.g., planar graphs), this leads to a logarithmic space improvement.

1.2 Dynamic approximate matching and vertex cover. On static graphs, the classic maximum cardinality matching (MCM) algorithms [12], [20], [27] run in $O(m\sqrt{n})$ time. For dynamic MCMs, Sankowski [25] devised a randomized algorithm with update time $O(n^{1.495})$.[4] On the other hand, finding a minimum cardinality vertex cover (MCVC) is NP-hard. Moreover, under the unique games conjecture (UGC), the MCVC cannot be efficiently approximated within any factor better than 2 [15]. Thus, while the ultimate goal in the context of dynamic MCMs is to efficiently maintain an exact MCM (or maybe a $(1+\epsilon)$-MCM), the analogous goal for dynamic MCVCs is to maintain a 2-MCVC, where t-MCM (resp., t-MCVC) is a shortcut for t-approximate MCM (resp., t-approximate MCVC), for any $t \geq 1$. Despite the inherent difference between these problems, they remain closely related as their LP-relaxations are duals of each other. Note also that (1) the MCM and MCVC are of the same size up to a factor of 2; (2) the set of matched vertices in a maximal matching is a 2-MCVC; (3) a maximal matching is a 2-MCM.

In view of the difficulty of dynamically maintaining *exact* MCMs and MCVCs, there has been a growing interest in dynamic approximate matching and vertex cover since the pioneering work of Onak and Rubinfeld [22], who presented a randomized algorithm for maintaining c-MCM and c-MCVC, where c is a large unspecified constant. The update time of the algorithm of [22] is w.h.p. $O(\log^2 n)$.

The dynamic maximal matching algorithms discussed in Section 1.1 provide 2-MCMs. Also, they imply dynamic algorithms for maintaining 2-MCVCs with the same (up to constants) update time. In particular, Baswana et al.'s algorithm [3] can be used to maintain 2-MCM and 2-MCVC with update time $O(\log n)$ in expectation and $O(\log n + \frac{n \log^2 n}{t})$ w.h.p., significantly improving the result of [22].

There are many other works on dynamic approximate MCMs and MCVCs; we briefly mention the state-of-the-art: [6] presented a dynamic algorithm for $(2+\epsilon)$-MCVCs with update time $O(\log n \cdot \epsilon^{-2})$; [10] gave an algorithm for $(1+\epsilon)$-MCMs with update time $O(\sqrt{m} \cdot \epsilon^{-2})$; [4], [5] devised dynamic algorithms for $(3/2+\epsilon)$-MCMs in general graphs with update time $O(m^{1/4} \cdot \epsilon^{-2.5})$. For uniformly sparse graphs, [24] presented algorithms for maintaining $(1+\epsilon)$-MCMs and $(2+\epsilon)$-MCVCs with update times that depend on the density (or *arboricity*) of the graph and ϵ.

Our contribution. Our dynamic maximal matching algorithm can be used to maintain 2-MCM and 2-MCVC in general graphs with update time that is bounded in expectation and w.h.p. by constant. Under the UGC, a $(2-\epsilon)$-MCVC cannot be efficiently maintained, for any $\epsilon > 0$. Hence, our dynamic 2-MCVC algorithm is essentially the best one can hope for, culminating a long line of research.

1.3 Additional applications. Static and dynamic algorithms for exact and approximate maximum weighted matchings have been extensively studied, both in centralized and distributed networks (see, e.g., [18], [19], [2], [10], [8], [9]). In particular, Anand et al. [2] gave a randomized algorithm for maintaining an 8-approximate maximum weight matching (8-MWM) in general n-vertex weighted graphs with expected update time $O(\log n \log \Delta)$, where Δ is the ratio between the maximum and minimum edge weights in the graph. The algorithm of [2] employs the dynamic maximal matching algorithm of [3] as a black-box. By plugging our improved algorithm, we shave a factor of $\log n$ from the update time, obtaining an algorithm for maintaining an 8-MWM in general weighted graphs with expected update time $O(\log \Delta)$.

Another application of our dynamic maximal matching algorithm is to *distributed networks*. In a static distributed setting all processors wake up simultaneously, and computation proceeds in fault-free synchronous rounds during which every processor exchanges messages of size $O(\log n)$.[5] In a dynamic distributed setting, however, upon the insertion or deletion of an edge $e = (u, v)$, only the affected vertices (u and v) are woken up. In this setting, the goal is to optimize (1) the number of communication rounds, and (2) the number of messages, needed for repairing the solution per update operation, over a worst-case sequence of update operations. Following an edge update, $O(1)$ communication rounds trivially suffice for updating a maximal matching (and thus 2-MCM and 2-MCVC). However, the number of messages sent per update may be as high as $O(n)$. Our dynamic maximal matching algorithm can be easily distributed, so that the average number of messages sent per update is a small constant.

For a detailed discussion on the applications of our main result, see the appendix of the full version of this paper [26].

1.4 Our and Previous Techniques. At the core of Baswana et al. (hereafter, *BGS*) algorithm [3] is a complex bucketing scheme, where each vertex is assigned a unique *level* among $\{-1, 0, \ldots, \log n\}$ according to some stringent criteria. (See

[4] [1] proved conditional lower bounds for dynamic MCMs, showing that the update time must be $\Omega(m^{\epsilon})$ under conjectures concerning the complexity of triangle detection, combinatorial boolean matrix multiplication and 3SUM, where ϵ is a constant depending on the specific conjecture. For example, under the 3SUM conjecture, it was shown in [1] that $\epsilon \geq 1/3$; moreover, a stronger bound of $\epsilon \geq 1/2 - o(1)$ was given in [17].

[5] We consider the standard $\mathcal{CONGEST}$ model (cf. [23]), which captures the essence of spatial locality and congestion.

Section 2 in the full version [26] for more details.) The BGS algorithm constantly and persistently changes the level of vertices. Changing the level of a single vertex may trigger a sequence of changes to multiple lower level vertices, referred to in [3] as a *fading wave*. The cost of a fading wave may be linear in the level of the vertex initiated it, thus the update time is $O(\log n)$. This is not the only logarithmic-time bottleneck incurred by the fading wave mechanism. Indeed, a central ingredient of this mechanism determines the "right" level to which a vertex should move. To this end, an explicit counter $\phi_v[j]$ in maintained for each vertex v and any level $j \in \{-1, 0, \ldots, \log n\}$. Following a single update on vertices u and v, the BGS algorithm has to update a logarithmic number of counters for u and v. Also, finding the right level to which a vertex should move using these counters cannot be done in sublogarithmic time (via a binary search or other search methods) due to the special nature of these counters. On top of these logarithmic-time bottlenecks, the fading wave mechanism of BGS is highly intricate, in terms of both the implementation and the analysis.

To break the logarithmic-time barrier, one has to take a significantly different approach, which circumvents the fading wave mechanism, and more importantly, the usage of the counters $\phi_v[j]$. Although our approach builds on the BGS scheme, it borrows mainly from its simpler ideas, successfully avoiding the use of the more complicated ones. Thus, despite the resemblance between the two approaches, our algorithm deviates significantly from the BGS scheme. We too use a bucketing scheme with $O(\log n)$ levels. However, to remove the dependency on the number of levels from the update time, our level-maintenance scheme cannot follow stringent criteria as in BGS. Instead, we use a "lazy" approach, which changes the level of a vertex only when necessary. When this happens, it essentially means that the vertex will "rise" to a higher level. The BGS algorithm also lets vertices rise to higher levels. However, the "right" level computed by our algorithm is inherently different than that computed by BGS. Roughly speaking, while in the BGS algorithm it is crucial that a vertex v would rise to the *highest possible* level satisfying a certain condition, in our algorithm v may rise to *any* such level. To speed up the runtime, our algorithm will let v rise to the *lowest* such level. Consequently, as opposed to the BGS algorithm that uses the $\phi_v[j]$ counters to determine the right level for a vertex v, which may require $\Omega(\log n)$ time, our algorithm determines the right level using a single counter that gradually increases "on the fly", via a novel "level-rising mechanism". This level-rising mechanism is a central ingredient in our algorithm, and is obtained by combining ideas from BGS with numerous new ideas. We anticipate that our level-rising mechanism will be applicable to additional dynamic graph problems under the oblivious adversarial model, where some subgraph structure has to be maintained.

II. The Update Algorithm

The update algorithm is applied following edge insertions and deletions to and from the graph. In this section we provide a complete description of this algorithm; refer to the full version [26] for the pseudo-code of the algorithm.

2.1 Invariants and data structures. Our algorithm maintains for each vertex v a *level* ℓ_v, with $-1 \leq \ell_v \leq \log_5(n-1)$. We use logarithms in base 5, whereas [3] use logarithms in base 2. More generally, we use the constant 5 rather than 2 throughout. Following the BGS algorithm, our algorithm will maintain the following two invariants.

Invariant 1: An edge (u, v) with $\ell_u > \ell_v$ is *oriented* by the algorithm as $u \to v$. (If $\ell_u = \ell_v$, the orientation of (u, v) will be determined suitably by the algorithm.)

Invariant 2: (a) Any free vertex has level -1 and out-degree 0. (So the maintained matching is maximal.) (b) Any matched vertex has level at least 0. (c) The endpoints of any matched edge are of the same level, and this level remains unchanged until the edge is deleted from the matching. (We henceforth define the level of a matched edge, which is at least 0 by item (b), as the level of its endpoints.)

For each vertex v, we maintain linked lists $\mathcal{N}_v$ and $\mathcal{O}_v$ of its neighbors and outgoing neighbors, respectively. The information about v's incoming neighbors will be maintained via a more detailed data structure $\mathcal{I}_v$: A hash table, where each element corresponds to a distinct level $\ell \in \{-1, 0, \ldots, \log_5(n-1)\}$. Specifically, an element $\mathcal{I}_v[\ell]$ of $\mathcal{I}_v$ corresponding to level ℓ holds a *pointer* to the head of a non-empty linked list that contains all incoming neighbors of v with level ℓ. If that list is empty, then the corresponding pointer is not stored in the hash table. While the number of pointers stored in the dynamic hash table $\mathcal{I}_v$ is bounded by $\log_5(n-1)+2$, it may be much smaller than that. In particular, the total space over all hash tables is linear in the dynamic number of edges in the graph. We can use a static array of size $\log_5(n-1)+2$ instead of a dynamic hash table, but the total space usage over all these arrays will be $\Omega(n \log n)$. If the dynamic graph is usually dense (i.e., having $\Omega(n \log n)$ edges), it is advantageous to use arrays, as it is easier to implement all the basic operations (delete, insert, search), and the time bounds become deterministic.

Note that no information whatsoever on the levels of v's outgoing neighbors is provided by the data structure $\mathcal{O}_v$. In particular, to determine if v has an outgoing neighbor at a certain level (most importantly at level -1, i.e., a free neighbor), we need to scan the entire list $\mathcal{O}_v$. On the other hand, v has an incoming neighbor at a certain level ℓ iff the corresponding list $\mathcal{I}_v[\ell]$ is non-empty. It will not be in v's *responsibility* to maintain the data structure $\mathcal{I}_v$, but rather within the responsibility of v's incoming neighbors.

We keep mutual pointers between the elements in the various data structures: For any vertex v and any outgoing neighbor u of v, we have mutual pointers between all

elements $u \in \mathcal{O}_v, v \in \mathcal{I}_u[\ell_v], v \in N_u, u \in N_v$. For example, when an edge (u, v) oriented as $v \to u$ is deleted from the graph, we get a pointer to either $v \in N_u$ or $u \in N_v$, and through this pointer we delete all elements $u \in \mathcal{O}_v, v \in \mathcal{I}_u[\ell_v], v \in N_u, u \in N_v$. As another example, when the orientation of edge (u, v) is flipped from $u \to v$ to $v \to u$, then assuming we have a pointer to $v \in \mathcal{O}_u$ (which is the case in our algorithm), we can reach element $u \in \mathcal{I}_v[\ell_u]$ through the pointer, delete them both from the respective lists, and then create elements $u \in \mathcal{O}_v$ and $v \in \mathcal{I}_u[\ell_v]$ with mutual pointers. We also keep mutual pointers between a matched edge and its endpoints. (We do not provide a complete description of the trivial maintenance of these pointers for the sake of brevity.)

Following [3], we define $\phi_v(\ell)$ to be the number of neighbors of v with level strictly lower than ℓ.

2.2 Procedure `set-level`(v, ℓ). Whenever the update algorithm examines a vertex v, it may need to re-evaluate its level. After the new level ℓ is determined, the algorithm calls Procedure `set-level`(v, ℓ). The procedure starts by updating the outgoing neighbors of v about v's new level. Specifically, we scan the entire list $\mathcal{O}_v$, and for each vertex $w \in \mathcal{O}_v$, we move v from $\mathcal{I}_w[\ell_v]$ to $\mathcal{I}_w[\ell]$.

Suppose first that $\ell < \ell_v$. In this case the level of v is decreased by at least one. As a result, we need to flip the outgoing edges of v towards vertices of level between $\ell + 1$ and ℓ_v to be incoming to v. Specifically, we scan the entire list $\mathcal{O}_v$, and for each vertex $w \in \mathcal{O}_v$ such that $\ell + 1 \leq \ell_w \leq \ell_v$, we perform the following operations: Delete w from $\mathcal{O}_v$, add w to $\mathcal{I}_v[\ell_w]$, delete v from $\mathcal{I}_w[\ell]$, and add v to $\mathcal{O}_w$.

If $\ell > \ell_v$, the level of v is increased by at least one. As a result, we flip v's incoming edges from vertices of level between ℓ_v and $\ell - 1$ to be outgoing of v. Specifically, for each non-empty list $\mathcal{I}_v[i]$, with $\ell_v \leq i \leq \ell - 1$, and for each vertex $w \in \mathcal{I}_v[i]$, we perform the following operations: Delete w from $\mathcal{I}_v[i]$, add w to $\mathcal{O}_v$, delete v from $\mathcal{O}_w$, and add v to $\mathcal{I}_w[\ell]$. Note, however, that we do not know for which levels i the corresponding list is non-empty; the time overhead needed to verify this information is $O(\ell)$.

Only after the data structures have been updated, we set $\ell_v = \ell$. The next observation is immediate from the description of the procedure, assuming Invariants 1 and 2 hold. In particular, it shows that the runtime of this procedure is at most $O(d^{old}_{\texttt{out}}(v) + d^{new}_{\texttt{out}}(v) + \ell)$, where $d^{old}_{\texttt{out}}(v)$ and $d^{new}_{\texttt{out}}(v)$ denote v's out-degree before and after the execution of this procedure, respectively.

Observation 2.1: Let ℓ_v denote the out-degree of v before the execution of this procedure.

(1) If $\ell = \ell_v$, the procedure does nothing, and the runtime is constant.

(2) If $\ell < \ell_v$, then $d^{new}_{\texttt{out}}(v) \leq d^{old}_{\texttt{out}}(v)$ and the procedure's runtime is $O(d^{old}_{\texttt{out}}(v))$.

(3) If $\ell > \ell_v$, then $d^{new}_{\texttt{out}}(v) \geq d^{old}_{\texttt{out}}(v)$ and the procedure's runtime is $O(d^{new}_{\texttt{out}}(v) + \ell)$. Moreover, after the execution of the procedure, all outgoing neighbors of v are of level at most $\ell - 1$. In the particular case of $\ell_v = -1$ and $\ell = 0$, which occurs when a free vertex v becomes matched at level 0, we have $d^{new}_{\texttt{out}}(v) = 0$; hence the procedure's runtime in this case is constant.

2.3 Procedures `handle-insertion`(u, v) and `handle-deletion`(u, v). Following an edge insertion (u, v), we apply Procedure `handle-insertion`(u, v). Besides updating the relevant data structures in the obvious way, this procedure matches between u and v if they are both free, otherwise it leaves them unchanged. Matching u and v involves setting their level to 0 by making the calls `set-level`$(u, 0)$ and `set-level`$(v, 0)$, whose runtime is $O(1)$ by Observation 2.1(3).

Following an edge deletion (u, v), we apply Procedure `handle-deletion`(u, v). If edge (u, v) does not belong to the matching, we only need to update the relevant data structures. In the case that edge (u, v) is matched, both u and v become *temporarily free*, meaning that they are not matched to any vertex yet, but their level remains temporarily as before. (We handle them next, one after another, but until each of them is handled, its level will exceed -1.) We handle vertices u and v via Procedure `handle-free`, i.e., by calling `handle-free`(u) and later `handle-free`(v); Procedure `handle-free` is the main ingredient of the update algorithm, and is described next.

2.4 Procedure `handle-free`(v). The execution of this procedure splits into two cases.
Case 1: $d_{\texttt{out}}(v) < 5^{\ell_v+1}$. In other words, the first case is when the out-degree of v is not much greater than 5^{ℓ_v}, and we run Procedure `deterministic-settle`(v) described in Section 2.4.1.
Case 2: $d_{\texttt{out}}(v) \geq 5^{\ell_v+1}$. We run Procedure `random-settle`$(v)$ described in Section 2.4.2.

2.4.1 Procedure `deterministic-settle`(v). The procedure starts by scanning the list $\mathcal{O}_v$ for a free vertex. By Invariant 2(a), if no free vertex is found in $\mathcal{O}_v$, then v does not have any free neighbor. (Note that we ignore temporarily free neighbors of v. See Section 3.1 for more details.) If no free vertex is found in $\mathcal{O}_v$, then v becomes free and we set its level ℓ_v to -1 by calling to `set-level`$(v, -1)$. Otherwise a free vertex is found in $\mathcal{O}_v$. In this case we match v to an arbitrary such vertex w, and set the levels of v and w to 0 by calling to `set-level`$(v, 0)$ and `set-level`$(w, 0)$.

Next, we show that the procedure's runtime is $O(5^{\ell_v})$. Let $d^{old}_{\texttt{out}}(v)$ denote v's out-degree before the execution of the procedure, and note that this procedure is invoked only when $d^{old}_{\texttt{out}}(v) < 5^{\ell_v+1}$. Moreover, since v is a temporarily free vertex, its level at this stage is at least 0. The procedure starts by scanning the list $\mathcal{O}_v$ for a free vertex, which takes

$O(d^{old}_{\texttt{out}}(v)) = O(5^{\ell_v})$ time. Next, the procedure calls to either `set-level`$(v,-1)$ or `set-level`$(v,0)$. Since the level of v prior to either one of these calls is at least 0, v's level may only decrease, hence the runtime is bounded by $O(d^{old}_{\texttt{out}}(v)) = O(5^{\ell_v})$ by Observations 2.1(1) and 2.1(2). Finally, there is a potential call to `set-level`$(w,0)$, which increases the level of w from -1 to 0; the runtime of this call is constant by Observation 2.1(3).

We remark that the out-degree of v after the potential call to `set-level`$(v,0)$ may be large, as it may have many outgoing neighbors of level 0. However, by Observations 2.1(1) and 2.1(2), v's new out-degree is bounded by $d^{old}_{\texttt{out}}(v) < 5^{\ell_v+1}$. Although we can afford to flip all edges leading to those vertices (this would require at most $O(5^{\ell_v})$ time, which we spend anyway), there is no need for it.

2.4.2 Procedure `random-settle`(v). The procedure employs what we refer to as a *level-rising mechanism*. Roughly speaking, the procedure matches v at some level ℓ^* higher than ℓ_v, with a random (possibly matched) neighbor of level *strictly lower* than ℓ^*. More accurately, it *attempts* to create such a matched edge (v,w) at level ℓ^* within time $O(5^{\ell^*})$; upon failure, it calls itself recursively to match w at yet a higher level, in which case v becomes free and handled via Procedure `handle-free`(v).

Next, we describe this procedure, and the underlying level-rising mechanism, in detail. We find it instructive to provide this description in two stages. First, we outline the two main challenges that this procedure has to cope with, and the specific manner in which it copes with these challenges. Only after the appropriate intuition is established, we turn to the formal description of the procedure.

Challenge 1. Recall that $\phi_v(\ell)$ is the number of v's neighbors of level *strictly lower* than ℓ. The reason we restrict our attention to neighbors of v of level *strictly lower* than a certain threshold is fundamental. When choosing a random mate w for v, we cannot guarantee that w would be free. Assuming w is matched to w', matching v with w triggers the deletion of edge (w,w') from the matching. This edge deletion is *induced* by the algorithm itself rather than the adversary, i.e., the edge remains in the graph but deleted from the matching. Alas, a naive *adversarial argument*, which bounds the expected number of edges that are deleted from the graph until the matched edge is deleted from the matching, does not hold for induced deletions. Coping with induced deletions is the crux of the problem. As in [3], when choosing a mate w for v, we restrict our attention to v's neighbors of level strictly lower than that of v, or more accurately, strictly lower than the *new* level to which v rises in order to be matched. Restricting our attention to the lower level neighbors is what allows us to employ a natural (yet intricate) charging argument. On the other hand, this restriction poses a nontrivial challenge, as we discuss next.

For an adversarial argument to work, it is crucial that v's new level would depend on the probability with which the new matched edge is chosen: If v's new level is ℓ^*, the probability that w is chosen as v's mate should be $O(1/5^{\ell^*})$. To guarantee that this condition holds, however, v must have $\Omega(5^{\ell^*})$ neighbors of level strictly lower than ℓ^*.

Since $d_{\texttt{out}}(v) \geq 5^{\ell_v+1}$, it is easy to verify that such a level ℓ^* exists (see Lemma 2.2(1)). It is much less clear, however, how to compute it efficiently. The challenge that we face is actually more intricate: After setting the level ℓ_v of v to ℓ^*, we need to update the data structures accordingly. For this scheme to work, the entire runtime should be $O(5^{\ell^*})$. To see where the difficulty lies, suppose we take ℓ^* to be ℓ_v. Recalling that $d_{\texttt{out}}(v) \geq 5^{\ell_v+1}$, v has many neighbors of level at most ℓ^*. However, we need that v would have many neighbors of level *strictly lower* than ℓ^*, and it is possible that most (or all) neighbors of v have level ℓ^*. One may try taking ℓ^* to be ℓ_v+1. This would indeed guarantee that v has sufficiently many neighbors of level strictly lower than ℓ^*. However, now there in another, somewhat contradictory, problem. After setting ℓ^* to ℓ_v+1, we need to update the data structures. In particular, we must flip all incoming edges of v from neighbors of level ℓ_v to be outgoing of v. This may be prohibitively expensive.

In general, the "right" level ℓ^* should balance two contradictory requirements: While v should have sufficiently many neighbors of level strictly lower than ℓ^*, it should not have too many of them. Any level balancing these requirements will do the job, but we also need to be able to compute it efficiently. We next show that, somewhat surprisingly, a sequential scan for the right level works smoothly.

Computing the "right" level for v: Set $\ell = \ell_v$, and gradually increase ℓ as long as $\phi_v(\ell+1) \geq 5^{\ell+1}$. Let ℓ^* be the level in which we stop the process, i.e., the *minimum* level such that $\ell^* \geq \ell_v$ and $\phi_v(\ell^*+1) < 5^{\ell^*+1}$.

We set v's level to ℓ^* by calling `set-level`(v,ℓ^*), thus updating the data structures, which involves flipping all incoming edges of v from neighbors of level between ℓ_v and ℓ^*-1 to be outgoing of v.

The following lemma, whose proof is deferred to the full version [26], is crucial for the correctness of the level-rising mechanism. We stress that, to be able to efficiently compute the level ℓ^*, we make critical use of the fact that we have complete information of the incoming neighbors of v. This fact is also crucial for the validity of Observation 2.1(3), which, in turn, is what guarantees that the runtime of the call to `set-level`(v,ℓ^*) would also be in check.

Lemma 2.2: Let $d^{old}_{\texttt{out}}(v)$ and $d^{new}_{\texttt{out}}(v)$ denote v's out-degree before and after the call to `set-level`(v,ℓ^*), respectively. Here ℓ_v denotes the level of v before the call, whereas ℓ^* is its level afterwards.

(1) $\ell_v < \ell^* \leq \log_5(n-1)$. In particular, a level ℓ^* as required exists. Moreover, the call to `set-level`(v,ℓ^*)

increases v's level by at least one.

(2) After this call, all outgoing neighbors of v are of level at most $\ell^* - 1$, i.e., $\phi_v(\ell^*) = d^{new}_{\texttt{out}}(v)$.

(3) $d^{old}_{\texttt{out}}(v) \leq d^{new}_{\texttt{out}}(v)$ and $5^{\ell^*} \leq d^{new}_{\texttt{out}}(v) = \phi_v(\ell^*) \leq \phi_v(\ell^* + 1) < 5^{\ell^*+1}$. (In particular, $d^{old}_{\texttt{out}}(v) < 5^{\ell^*+1}$.)

(4) The runtime of computing ℓ^* and calling to $\texttt{set-level}(v, \ell^*)$ is bounded by $O(5^{\ell^*})$.

Challenge 2. Lemma 2.2 implies that $O(5^{\ell^*})$ time suffices for computing the "right" level ℓ^* and letting v rise to that level by making the call $\texttt{set-level}(v, \ell^*)$. Moreover, v has at least 5^{ℓ^*} outgoing neighbors after this call, all having level at most $\ell^* - 1$. By picking uniformly at random an outgoing neighbor w of v to match with, the matched edge (v, w) will be chosen with probability at most $1/5^{\ell^*}$. Using an adversarial argument, this matched edge can cover the entire cost $O(5^{\ell^*})$ spent thus far in the amortized sense.

In order to add edge (v, w) to the matching, however, we need to update the data structures accordingly. In particular, if w is matched, say to w', we must delete edge (w, w') from the matching; this is an *induced* edge deletion. As mentioned, to cope with induced deletions, our charging argument makes critical use of the fact that w is of level strictly lower than ℓ^*. However, this requirement by itself would suffice only if we were guaranteed that edge (w, w') was chosen to the matching by w. (If the vertex initiating the match is of level ℓ, then the cost of creating the matched edge is $O(5^{\ell})$ by Lemma 2.2.) In general, this edge might have been chosen to the matching by w' rather than w, and so it is critical that both endpoints w and w' would be of levels strictly lower than ℓ^* for the charging argument to work. To this end, as in [3], we maintain the stronger invariant that the endpoints of any matched edge are of the same level; see Invariant 2(c). Consequently, to match v with w, the invariant requires that we let w rise to the new level ℓ^* of v. This requirement, however, poses another nontrivial challenge.

We set the level ℓ_w of w to ℓ^* by calling $\texttt{set-level}(w, \ell^*)$. This call updates the data structures accordingly, which involves flipping all incoming edges of w from neighbors of level between ℓ_w and $\ell^* - 1$ to be outgoing of w. As before, this may be prohibitively expensive. Specifically, by Observation 2.1(3), the runtime of the call to $\texttt{set-level}(w, \ell^*)$ is $O(d_{\texttt{out}}(w) + \ell^*)$, where $d_{\texttt{out}}(w)$ is the new out-degree of w.

By setting the level of w to ℓ^*, we have created a matched edge (v, w) at level ℓ^*. If and when this matched edge is deleted from the graph, we will be able to cover an expected cost of $O(5^{\ell^*})$ in the amortized sense. If $d_{\texttt{out}}(w) < 5^{\ell^*+1}$, the runtime of the call to $\texttt{set-level}(w, \ell^*)$, and thus of the entire procedure, is $O(5^{\ell^*})$, which can be covered in the amortized sense by the creation of the new matched edge (v, w). However, the complementary case $d_{\texttt{out}}(w) \geq 5^{\ell^*+1}$ is where the difficulty lies. Indeed, in this case the runtime of the call to $\texttt{set-level}(w, \ell^*)$ may be significantly higher than 5^{ℓ^*}. To cover it, we create a matched edge at level higher than ℓ^*. To this end we first delete the new matched edge (v, w) from the matching, and then invoke Procedure `random-settle` recursively, but on w this time.

The recursive call $\texttt{random-settle}(w)$ creates a matched edge at level higher than ℓ^*, which, in the amortized sense, can cover the cost of the call to $\texttt{set-level}(w, \ell^*)$. Thus, in each recursive call we rise to yet a higher level, attempting to charge the yet-uncharged costs of the procedure to the most recently created matched edge. We stress that the level-rising mechanism is not a single computation of a matched edge at some "right" level, but rather a recursive attempt at doing so: Try, rise to a higher level upon failure, and then try again. Note that the maximum level is $\log_5(n-1)$. Since $d_{\texttt{out}}(w) \leq \deg(w) \leq n - 1 < 5^{\log_5(n-1)+1}$, for any vertex w, this recursive attempt eventually succeeds.

The procedure. Procedure $\texttt{random-settle}(v)$ starts by computing the level ℓ^* as described above (i.e., the first level after ℓ_v such that $\phi_v(\ell^* + 1) < 5^{\ell^*+1}$) and setting v's level accordingly by calling $\texttt{set-level}(v, \ell^*)$. We then pick uniformly at random an outgoing neighbor w of v, to match them. (Lemma 2.2(3) implies that the matched edge is chosen with probability at most $1/5^{\ell^*}$, whereas $\ell_w \leq \ell^* - 1$ follows from Lemma 2.2(2).) If w is matched, say to w', then we delete edge (w, w') from the matching. This renders w' *temporarily free*, meaning that it is not matched to any vertex, but its level remains temporarily as before; w' will be handled soon, but in the interim, its level will exceed -1.

We set the level of w to ℓ^* by calling $\texttt{set-level}(w, \ell^*)$, which increases its level by at least one, and add edge (v, w) to the matching, thus creating a matched edge of level ℓ^*. (If and when this matched edge is deleted from the graph, we will be able to cover an expected cost of $O(5^{\ell^*})$ in the amortized sense.)

The runtime of the call to $\texttt{set-level}(w, \ell^*)$ is $O(d_{\texttt{out}}(w) + \ell^*)$, where $d_{\texttt{out}}(w)$ is the new out-degree of w. If $d_{\texttt{out}}(w) < 5^{\ell^*+1}$, this runtime is $O(5^{\ell^*})$, and it can be covered in the amortized sense.

However, in the complementary case $d_{\texttt{out}}(w) \geq 5^{\ell^*+1}$, the runtime may be significantly higher than 5^{ℓ^*}. To cover this runtime, we create a matched edge at level higher than ℓ^*. To this end we first delete the new matched edge (v, w) from the matching, thus rendering v and w temporarily free, and then invoke Procedure `random-settle` recursively, by calling $\texttt{random-settle}(w)$. (This recursive call creates a matched edge at level higher than ℓ^*, which, in the amortized sense, can cover the cost of the call to $\texttt{set-level}(w, \ell^*)$.) It is possible that v will become matched as a result of the recursive call to $\texttt{random-settle}(w)$; if v is not matched to any vertex, we invoke Procedure $\texttt{handle-free}(v)$.

Finally, if w' is not matched to any vertex, we invoke Procedure `handle-free`(w').

III. Analysis

3.1 Invariants. It is easy to verify that our update algorithm satisfies Invariants 1 and 2. The only (technical) exception to Invariant 2 is with temporarily free vertices, which are unmatched, yet their level exceeds -1. A vertex becomes temporarily free after its matched edge is deleted, either by the adversary (via procedure `handle-deletion`) or via Procedure `random-settle` of the update algorithm. When a (temporarily free) vertex v is handled via Procedure `handle-free`(v), it may *ignore* its temporarily free neighbors, as the corresponding edges may be incoming to v. In particular, Procedure `deterministic-settle`(v) *deliberately ignores* the temporarily free neighbors of v, and as a result, v may become free although it may have temporarily free neighbors. (Obviously, this is a matter of choice; we can change the procedure to consider temporarily free neighbors of v that belong to $\mathcal{O}_v$, but there is no need.) For this reason, our update algorithm makes sure to handle all vertices that become temporarily free *later*, via appropriate calls to Procedure `handle-free`. Hence, if any temporarily free neighbor w of v is ignored by v and v becomes free, the subsequent call to `handle-free`(w) will match w, either with v or with another neighbor of w.

3.2 Epochs. Given any sequence of edge updates, an edge (u, v) may become matched or unmatched by the algorithm at different update steps. The entire lifespan of an edge (u, v) consists of a sequence of *epochs*, which refer to the maximal time intervals in which the edge is matched, separated by the maximal time intervals in which the edge is unmatched. (This notion of epoch was introduced in [3].) Formally, let $e = (u, v)$ be any matched edge at some time step l. The *epoch* $\mathcal{E}(e, l)$ corresponding to edge e at time l refers to the maximal time interval containing l during which the edge (u, v) is matched. By Invariant 2(c), the endpoints of a matched edge are of the same level, and this level remains unchanged. We henceforth define the level of an epoch to be the level of the corresponding edge.

Any edge update that does not change the matching is processed by our algorithm in constant time. However, an edge update that changes the matching may trigger the creation of some epochs and the termination of some epochs. The computation cost of creating or terminating an epoch by the algorithm may be large. Moreover, the number of epochs created and terminated due to a single edge update may be large by itself. Hence an amortized analysis is required. Following the amortization scheme of [3], we re-distribute the total computation performed at any step l among the epochs created or terminated at step l. Specifically, let $\mathcal{E}_1 = \mathcal{E}(e_1, l), \ldots, \mathcal{E}_j = \mathcal{E}(e_j, l)$ (resp., $\mathcal{E}'_1 = (e'_1, l), \ldots, \mathcal{E}'_k = (e'_k, l)$) be the epochs created (resp., terminated) at update step l, and let $C_{create}(\mathcal{E}_1), \ldots, C_{create}(\mathcal{E}_j)$ (resp., $C_{term}(\mathcal{E}'_1), \ldots, C_{term}(\mathcal{E}'_k)$) be the respective computation costs charged to the creation (resp., termination) of these epochs. Then we re-distribute the total computation cost performed at update step l, denoted by $C^{(l)}$, to the respective epochs so that $C^{(l)} = \sum_{i=1}^{j} C_{create}(\mathcal{E}_i) + \sum_{i=1}^{k} C_{term}(\mathcal{E}'_i)$.

Notice that any epoch created by Procedures `handle-insertion` or `deterministic-settle` is of level 0. On the other hand, we argue that any epoch created by Procedure `random-settle` is of level at least 1. Indeed, consider a vertex v that is handled via Procedure `random-settle`(v). First note that $d_{\mathtt{out}}(v) \geq 5^{\ell_v+1} \geq 1$. By Invariant 2, $\ell_v \geq 0$. By the description of Procedure `random-settle`(v) and Lemma 2.2(1), we conclude that any epoch created by this procedure is of level at least $\ell^* > \ell_v$. Applying now Lemma 2.2, we derive the following corollary.

Corollary 3.1: For any epoch at level $\ell > 0$, initiated by vertex v and corresponding to edge (v, w):

(1) The out-degree of v at the time the epoch is created is at least 5^ℓ and less than $5^{\ell+1}$, though the out-degree of w at that time may be significantly smaller or larger than 5^ℓ.

(2) w is chosen as v's mate uniformly at random among all outgoing neighbors of v at that time, so the corresponding edge (v, w) becomes matched with probability at most $1/5^\ell$.

3.3 Re-distributing the computation costs to epochs. By re-distributing the total computation cost of the update algorithm to the various epochs, we can *visualize* the entire update algorithm as a sequence of creation and termination of these epochs. The computation cost associated with an epoch at level ℓ (hereafter, *level-ℓ epoch*) includes both its creation cost and its termination cost.

The following lemma plays a central role in our analysis. Although a similar lemma was proved in [3], our proof is inherently different than the corresponding proof of [3].

Lemma 3.2: The total computation cost of the update algorithm can be re-distributed to various epochs so that the computation cost associated with any level-ℓ epoch is bounded by $O(5^\ell)$, for any $\ell \geq 0$.

Proof: The update algorithm is triggered following edge insertions and edge deletions. Following an edge insertion, we apply Procedure `handle-insertion`(u, v). This procedure is called at most once per update step, and its runtime is constant. We may disregard this constant cost (formally, we charge this cost to the corresponding update step). Hence, the lemma holds vacuously for edge insertions.

Following an edge deletion, we apply Procedure `handle-deletion`(u, v). This procedure is called at most once per update step; disregarding the calls to `handle-free`(u) and `handle-free`(v), this procedure's runtime is constant. We henceforth disregard this constant cost (charging it to the corresponding update step), and demonstrate how to re-distribute the costs of the calls to

handle-free(u) and handle-free(v) to appropriate epochs in a manner satisfying the condition of the lemma.

Let $z \in \{u, v\}$. The execution of Procedure handle-free(z) splits into two cases. In the first case $d_{\texttt{out}}(z) < 5^{\ell_z+1}$, and Procedure handle-free(z) invokes Procedure deterministic-settle(z), whose runtime is $O(5^{\ell_z})$. Even though Procedure deterministic-settle(z) may create a new level-0 epoch, there is no need to charge this epoch with any costs. The entire cost $O(5^{\ell_z})$ of Procedure deterministic-settle(z) is charged to the termination cost of epoch $\mathcal{E}((u, v), l)$ (triggered by the deletion of edge (u, v) from the graph). which is of level ℓ_z, thus satisfying the condition of the lemma.

Otherwise $d_{\texttt{out}}(z) \geq 5^{\ell_z+1}$, and Procedure handle-free(z) calls random-settle(z).

The runtime of Procedure random-settle(z) may be much larger than 5^{ℓ_z} and even than $d_{\texttt{out}}(z)$. To cover the costs of this procedure, we create a new epoch at a sufficiently high level, attempting to charge the costs of the procedure to the creation cost of the new epoch. However, this charging attempt may sometimes fail, in which case we call the procedure recursively. Each recursive call will create a new epoch at yet a higher level, attempting to charge the yet-uncharged costs of the procedure to the creation cost of the most recently created epoch. Since the levels of epochs created during this process grow by at least one with each recursion level, this recursive process will terminate. Specifically, the depth of this recursive process is bounded by $\log_5(n-1)$. (Formally, one should also take into account the calls to Procedure handle-free from within Procedure random-settle, which may, in turn, invoke Procedure random-settle.[6]) Next, we describe the charging argument in detail.

For each recursion level of Procedure random-settle, our charging argument may (slightly) *over-charge* the current level's costs, so as to cover some of the yet-uncharged costs incurred by previous recursion levels. Denote by random-settle($z^{(i)}$) the ith recursive call, with $z^{(i)}$ being the examined vertex. The initial call to Procedure random-settle may be viewed as the 0th recursion level. As argued below (see Claim 3.3), it may leave a "debt" to the 1st recursion level, but this debt must be bounded by $O(d_{\texttt{out}}(z^{(1)}))$ units of cost. In general, for each $i \geq 1$, the *debt* at recursion level i (left by previous recursion levels) must be bounded by $O(d_{\texttt{out}}(z^{(i)}))$; in what follows we may view this debt as part of the costs incurred at level i. (Note that the initial call to the procedure has no debt.)

Observe that the ith recursive call random-settle($z^{(i)}$) may also invoke Procedure handle-free. Our charging argument views the call to handle-free as part of the execution of random-settle($z^{(i)}$). More concretely, the costs incurred by the call to handle-free are charged to epochs that are created or terminated throughout the execution of the ith recursive call random-settle($z^{(i)}$). Note, however, that Procedure handle-free may, in turn, invoke Procedure random-settle. Our charging argument does *not* view the new call to random-settle as part of the execution of random-settle($z^{(i)}$), but rather as an *independent* call. More concretely, the costs of the new call to Procedure random-settle are charged only to epochs that are created or terminated throughout the execution of this new call to random-settle.

To conclude the proof of Lemma 3.2, we use the following claim, whose proof is deferred to the full version [26].

Claim 3.3: For any $i \geq 0$, we can re-distribute the costs of the ith recursive call random-settle($z^{(i)}$) of Procedure random-settle (including the debt from previous recursion levels, if any) to various epochs, allowing a potential debt to the subsequent recursive call random-settle($z^{(i+1)}$) (if the procedure proceeds to recursion level $i+1$), so that:

(1) The cost charged to any level-ℓ epoch is bounded by $O(5^{\ell})$, for any ℓ.

(2) The debt left for recursion level $i+1$ is bounded by $O(d_{\texttt{out}}(z^{(i+1)}))$. If the procedure terminates at recursion level i, then no debt is allowed, i.e., the entire cost in this case is re-distributed to the epochs.

By Claim 3.3, whenever Procedure random-settle terminates, no debt whatsoever is left. Consequently, the entire cost of this procedure (over all recursion levels) can be re-distributed to various epochs in a manner satisfying the conditions of Lemma 3.2. Lemma 3.2 follows. ■

3.4 Natural versus induced epochs. An epoch corresponding to edge (u, v) is terminated either because edge (u, v) is deleted from the graph, and then it is called a *natural epoch*, or because the update algorithm deleted edge (u, v) from the matching, and then it is called an *induced epoch*. Following [3], we will charge the computation cost of each induced epoch $\mathcal{E}$ of level $\ell \geq 0$ to the cost of the unique epoch $\mathcal{E}'$ whose creation "triggered" the termination of $\mathcal{E}$.

An induced epoch is terminated only by Procedure random-settle. Consider the first call to Procedure random-settle(z), which we also view as the 0th recursion level. First, some vertex w of level ℓ_w lower than ℓ^* is chosen as a random mate for vertex z, and a level-ℓ^* epoch $\mathcal{E}((z, w), l)$ is created. If w is matched to w', the edge (w, w') is deleted from the matching, and we view the creation of the level-ℓ^* epoch $\mathcal{E}((z, w), l)$ as terminating the level-$\ell_{w'}$ epoch $\mathcal{E}((w, w'), l)$. Next, suppose that $d_{\texttt{out}}(w) \geq 5^{\ell^*+1}$. In this case the newly created epoch $\mathcal{E}((z, w), l)$ is terminated, and immediately afterwards

[6] Since the assertion that the recursive process terminates is a corollary of our ultimate bound on the update time, an additional proof is not required. Nevertheless, a simpler proof that is independent of our proof of the update time bound is sketched in the full version [26].

we make the 1st recursive call to random-settle(w). By Lemma 2.2(1), the call to random-settle(w) is guaranteed to create a new epoch $\mathcal{E}((w,x),l)$ at level ℓ' higher than ℓ^*, where x is a random outgoing neighbor of w, which is of level ℓ_x lower than ℓ' by Lemma 2.2(2). We view the creation of the level-ℓ' epoch $\mathcal{E}((w,x),l)$ as terminating the level-ℓ^* epoch $\mathcal{E}((z,w),l)$. Furthermore, if x is matched to x', the edge (x,x') is deleted from the matching, and we view the creation of the level-ℓ' epoch $\mathcal{E}((w,x),l)$ as terminating the level-$\ell_{x'}$ epoch $\mathcal{E}((x,x'),l)$ as well. We have shown that the creation of epoch $\mathcal{E}((w,x),l)$ at the 1st recursion level terminates (at most) two epochs of lower levels. Exactly the same reasoning applies to an arbitrary recursion level i by induction. To summarize, the creation of a new epoch terminates at most two epochs, both of which are of levels strictly lower than that of the created epoch. That is, if the level of the created epoch is ℓ and the levels of the terminated epochs are ℓ_1 and ℓ_2, then $\ell_1, \ell_2 < \ell$.

We say that a level ℓ is *induced* if the number of induced level-ℓ epochs exceeds the number of natural level-ℓ epochs; otherwise it is *natural*. Next, we show that the computation costs incurred by any induced level can be charged to the computation costs at higher levels. Specifically, in any induced level ℓ we can define a one-to-one mapping from the natural to the induced epochs. For each induced epoch, at most one natural epoch (at the same level) is mapped to it; any natural epoch that is mapped to an induced epoch is called *semi-natural*. By definition, for any induced level ℓ, all natural ℓ-level epochs are semi-natural. For the natural levels, however, we do not define such a mapping. Thus for any natural level, all natural epochs terminated at that level remain as before; these epochs are called *fully-natural*.

We define the *recursive cost* of an epoch as the sum of its actual cost and the recursive costs of the (at most) two induced epochs terminated by it as well as the (at most) two semi-natural epochs mapped to them; the recursive cost of a level-0 epoch is its actual cost. Denote the highest recursive cost of any level-ℓ epoch by $\hat{C}_\ell$, for any $\ell \geq 0$. (Obviously $\hat{C}_\ell$ is monotone non-decreasing with ℓ.) By Lemma 3.2, we get the recurrence $\hat{C}_\ell \leq \hat{2}C_{\ell_1} + 2\hat{C}_{\ell_2} + O(5^\ell) \leq 4\hat{C}_{\ell-1} + O(5^\ell)$, where $\ell_1, \ell_2 < \ell$, with the base condition $\hat{C}_0 = O(1)$. This recurrence resolves to $\hat{C}_\ell = O(5^\ell)$.

Corollary 3.4: For any $\ell \geq 0$, the recursive cost of any level-ℓ epoch is bounded by $O(5^\ell)$.

By definition, the sum of recursive costs over all fully-natural epochs is equal to the sum of actual costs over all epochs (fully-natural, semi-natural and induced) that have been *terminated* throughout the update sequence.

3.5 Bounding the algorithm's runtime. During any sequence of t updates, the total number of epochs created equals the number of epochs terminated and the number of epochs that *remain alive* at the end of the t updates. To bound the computation cost charged to all epochs that remain alive at the end of the update sequence, one may use an argument similar to [3]. However, instead of distinguishing between terminated epochs and ones that remain alive, we find it cleaner to get rid of those epochs that remain alive by deleting all edges of the final graph, one after another. That is, we append additional edge deletions at the end of the original update sequence, so as to finish with an empty graph. The order in which these edges are deleted from the graph may be random, but it may also be deterministic, as long as it is oblivious to the maintained matching.

This tweak guarantees that no epoch remains alive at the end of the update sequence. As a result, the sum of recursive costs over all fully-natural epochs will bound the sum of actual costs over all epochs (fully-natural, semi-natural and induced) that have been *created* (and also terminated) throughout the update sequence, or in other words, it will bound the total runtime of the algorithm. Note that the total runtime of the algorithm may only increase as a result of this tweak. Since we increase the length of the update sequence by at most a factor of 2, an amortized runtime bound of the algorithm with respect to the new update sequence will imply the same (up to a factor of 2) amortized bound with respect to the original sequence.

Let Y be the r.v. for the sum of recursive costs over all fully-natural epochs terminated during the entire sequence. Note that Y stands for the total runtime of the algorithm.

Lemma 3.5: $Y = O(t + n \log n)$ w.h.p.

The proof of Lemma 3.5 is deferred to the full version [26]. While it follows similar lines as in [3], the bound it provides shaves a logarithmic factor from the bound of [3]. (This is because our algorithm is inherently different than the BGS algorithm and since other parts in the analysis are different.) Moreover, there is a significant difference between our proof and that of BGS. In particular, we extend the epoch definition as follows. Consider an arbitrary epoch initiated by some vertex u at some update step l. An epoch is terminated when its matched edge (u,v) becomes unmatched (i.e., deleted from the matching); the *(interrupted) duration of the epoch* is defined as the number of outgoing edges of u at time l that get deleted from the graph between time step l and the epoch's termination. This definition of duration is given in [3], and it coincides with the epoch definition of Section 3.2. We define the *uninterrupted duration* of the epoch as the number of outgoing edges of u at time l that get deleted from the graph between time step l and the time that the random matched edge (u,v) is deleted *from the graph*.

Roughly speaking, to prove Lemma 3.5, one should establish independence between the durations of the various epochs, and then employ it to argue that sufficiently many epochs at each level have w.h.p. long durations. To this end, we find it crucial to analyze the uninterrupted durations of epochs rather than their (interrupted) durations. If the epoch is natural, its uninterrupted duration equals its duration. For an induced epoch, however, its uninterrupted duration

may be significantly larger than its duration. Consequently, proving that many epochs at each level have w.h.p. long *uninterrupted* durations is rather useless, if it turns out that most of them are induced. For this reason, we need to be able to restrict our attention to levels where there are sufficiently many natural epochs, and this is exactly why we distinguished between natural and induced levels, or analogously, between fully-natural and semi-natural epochs.

It is not difficult to extend the high probability proof to obtain $\mathbf{E}(Y) = O(t + n \log n)$. Shaving the $O(n \log n)$ term from the high probability bound, and consequently from the expected bound, requires additional new ideas; see the full version [26] for the details. We remark that there are alternative proofs for obtaining an expected bound of $O(t)$, without deriving it from the high probability bound.

Finally, the space usage of our algorithm is linear in the dynamic number of edges in the graph.

Theorem 3.6: Starting from an empty graph on n fixed vertices, a maximal matching (and thus 2-MCM and also 2-MCVC) can be maintained over any sequence of t edge insertions and deletions in $O(t)$ time in expectation and w.h.p., and using $O(n + m)$ space, where m denotes the dynamic number of edges.

Acknowledgements. The author is grateful to Amir Abboud, Surender Baswana, Sayan Bhattacharya, Manoj Gupta, David Peleg, Sandeep Sen, Noam Solomon and Virginia Vassilevska Williams for helpful discussions.

REFERENCES

[1] A. Abboud and V. V. Williams. Popular conjectures imply strong lower bounds for dynamic problems. In *Proc. 55th FOCS*, pages 434–443, 2014.

[2] A. Anand, S. Baswana, M. Gupta, and S. Sen. Maintaining approximate maximum weighted matching in fully dynamic graphs. In *Proc. 32nd FSTTCS*, pages 257–266, 2012.

[3] S. Baswana, M. Gupta, and S. Sen. Fully dynamic maximal matching in $O(\log n)$ update time. In *Proc. of 52nd FOCS*, pages 383–392, 2011.

[4] A. Bernstein and C. Stein. Fully dynamic matching in bipartite graphs. In *42nd ICALP*, pages 167–179, 2015.

[5] A. Bernstein and C. Stein. Faster fully dynamic matchings with small approximation ratios. In *Proc. of 26th SODA*, pages 692–711, 2016.

[6] S. Bhattacharya, M. Henzinger, and G. F. Italiano. Deterministic fully dynamic data structures for vertex cover and matching. In *Proc. 26th SODA*, pages 785–804, 2015.

[7] L. Carter and M. N. Wegman. Universal classes of hash functions. In *Proc. 9th STOC*, pages 106–112, 1977.

[8] R. Duan and S. Pettie. Linear-time approximation for maximum weight matching. *J. ACM*, 61(1):1, 2014.

[9] M. Gupta. Maintaining approximate maximum matching in an incremental bipartite graph in polylogarithmic update time. In *Proc. 34th FSTTCS*, pages 227–239, 2014.

[10] M. Gupta and R. Peng. Fully dynamic $(1 + \epsilon)$-approximate matchings. In *54th FOCS*, pages 548–557, 2013.

[11] M. He, G. Tang, and N. Zeh. Orienting dynamic graphs, with applications to maximal matchings and adjacency queries. In *Proc. 25th ISAAC*, pages 128–140, 2014.

[12] J. E. Hopcroft and R. M. Karp. An $n^{5/2}$ algorithm for maximum matchings in bipartite graphs. *SIAM J. Comput.*, 2(4):225–231, 1973.

[13] Z. Ivković and E. L. Lloyd. Fully dynamic maintenance of vertex cover. In *19th WG*, pages 99–111, 1993.

[14] B. M. Kapron, V. King, and B. Mountjoy. Dynamic graph connectivity in polylogarithmic worst case time. In *Proc. of 24th SODA*, pages 1131–1142, 2013.

[15] S. Khot and O. Regev. Vertex cover might be hard to approximate to within 2-epsilon. *J. Comput. Syst. Sci.*, 74(3):335–349, 2008.

[16] T. Kopelowitz, R. Krauthgamer, E. Porat, and S. Solomon. Orienting fully dynamic graphs with worst-case time bounds. In *Proc. 41st ICALP*, pages 532–543, 2014.

[17] T. Kopelowitz, S. Pettie, and E. Porat. Higher lower bounds from the 3SUM conjecture. In *Proc. of 26th SODA*, pages 1272–1287, 2016.

[18] Z. Lotker, B. Patt-Shamir, and S. Pettie. Improved distributed approximate matching. In *Proc. SPAA*, pages 129–136, 2008.

[19] Z. Lotker, B. Patt-Shamir, and A. Rosén. Distributed approximate matching. *SIAM J. Comput.*, 39(2):445–460, 2009.

[20] S. Micali and V. V. Vazirani. An $O(\sqrt{|V||E|})$ algorithm for finding maximum matching in general graphs. In *Proc. 21st FOCS*, pages 17–27, 1980.

[21] O. Neiman and S. Solomon. Simple deterministic algorithms for fully dynamic maximal matching. In *Proc. 45th STOC*, pages 745–754, 2013.

[22] K. Onak and R. Rubinfeld. Maintaining a large matching and a small vertex cover. In *Proc. STOC*, pages 457–464, 2010.

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

[24] D. Peleg and S. Solomon. Dynamic $(1 + \epsilon)$-approximate matchings: A density-sensitive approach. In *Proc. of 26th SODA*, pages 712–729, 2016.

[25] P. Sankowski. Faster dynamic matchings and vertex connectivity. In *Proc. 18th SODA*, pages 118–126, 2007.

[26] S. Solomon. Fully Dynamic Maximal Matching in Constant Update Time. *CoRR*, abs/1604.08491, 2016.

[27] V. V. Vazirani. An improved definition of blossoms and a simpler proof of the MV matching algorithm. *CoRR*, abs/1210.4594, 2012.