Fully Dynamic Almost-Maximal Matching: Breaking the Polynomial Worst-Case Time Barrier

Moses Charikar[1]
Computer Science Department, Stanford University, Stanford, California, USA

Shay Solomon[2]
IBM Research, T. J. Watson Research Center, Yorktown Heights, New York, USA

Abstract

The state-of-the-art algorithm for maintaining an approximate maximum matching in fully dynamic graphs has a polynomial *worst-case* update time, even for poor approximation guarantees. Bhattacharya, Henzinger and Nanongkai showed how to maintain a constant approximation to the minimum vertex cover, and thus also a constant-factor estimate of the maximum matching size, with polylogarithmic worst-case update time. Later (in SODA'17 Proc.) they improved the approximation to $2+\epsilon$. Nevertheless, the fundamental problem of *maintaining* an approximate matching with sub-polynomial worst-case time bounds remained open.

We present a randomized algorithm for maintaining an *almost-maximal* matching in fully dynamic graphs with polylogarithmic worst-case update time. Such a matching provides $(2+\epsilon)$-approximations for both maximum matching and minimum vertex cover, for any $\epsilon > 0$. The worst-case update time of our algorithm, $O(\mathsf{poly}(\log n, \epsilon^{-1}))$, holds deterministically, while the almost-maximality guarantee holds with high probability. Our result was done independently of the $(2+\epsilon)$-approximation result of Bhattacharya et al., thus settling the aforementioned problem on dynamic matchings and providing essentially the best possible approximation guarantee for dynamic vertex cover (assuming the unique games conjecture).

To prove this result, we exploit a connection between the standard oblivious adversarial model, which can be viewed as inherently "online", and an "offline" model where some (limited) information on the future can be revealed efficiently upon demand. Our randomized algorithm is derived from a deterministic algorithm in this offline model. This approach gives an elegant way to analyze randomized dynamic algorithms, and is of independent interest.

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

Keywords and phrases dynamic graph algorithms, maximum matching, worst-case bounds

Digital Object Identifier 10.4230/LIPIcs.ICALP.2018.33

Related Version A full version of the paper can be found at [9], `https://arxiv.org/abs/1711.06883`.

Acknowledgements The second author is grateful to Danupon Nanongkai and Uri Zwick for their suggestion to study dynamic matchings in the offline model.

[1] Supported by NSF grant CCF-1617577 and a Simons Investigator Award.
[2] Supported by the IBM Herman Goldstine Postdoctoral Fellowship.

45th International Colloquium on Automata, Languages, and Programming (ICALP 2018).
Editors: Ioannis Chatzigiannakis, Christos Kaklamanis, Dániel Marx, and Donald Sannella;
Article No. 33; pp. 33:1–33:14

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

1 Introduction

Consider a *fully dynamic* setting where we start from an initially empty graph on n fixed vertices G_0, and at each time step i a single edge (u, v) is either inserted in the graph G_{i-1} or deleted from it, resulting in graph G_i. The problem of maintaining a large matching or a small vertex cover in such graphs has attracted a lot of research attention in recent years. In general, one would like to devise an algorithm for maintaining a "good" matching and/or vertex cover with $\mathsf{polylog}(n)$ *update time* (via a data structure that answers queries of whether an edge is matched or not in constant time), where "good" means a good approximation to the maximum matching and/or the minimum vertex cover, and the update time is the time required by the algorithm to update the matching/vertex cover at each step.

One may try to optimize the *amortized* (i.e., average) update time of the algorithm or its *worst-case* (i.e., maximum) update time, over a worst-case sequence of graphs. There is a strong separation between the state-of-the-art amortized bounds and the worst-case bounds. A similar separation exists for various other dynamic graph problems, such as spanning tree, minimum spanning tree and two-edge connectivity. Next, we provide a brief literature survey on dynamic matchings. (See [17, 2, 18, 19] for a detailed survey.)

In FOCS'11, [2] devised an algorithm for maintaining a *maximal matching* with an expected amortized update time of $O(\log n)$ under the oblivious adversarial model.[3] Building on [2], [19] devised a different randomized algorithm with constant amortized update time. Note that a maximal matching provides a 2-approximation for both the maximum matching and the minimum vertex cover, while a better-than-2 approximate vertex cover cannot be efficiently computed under the unique games conjecture (UGC) [14]. In SODA'15 [5] (respectively, STOC'16 [6]) devised a deterministic algorithm for maintaining $(2+\epsilon)$-approximate vertex cover (resp., matching) with amortized update time $O(\log n/\epsilon^2)$ (resp., $O(\mathsf{poly}(\log n, \epsilon^{-1}))$). All these time bounds are amortized.

All the known algorithms for maintaining a better-than-2 approximate matching (for general graphs) require polynomial update time. In FOCS'13 [12] devised a deterministic algorithm for maintaining $(1+\epsilon)$-approximate matching with a worst-case update time $O(\sqrt{m}/\epsilon^2)$, improving over the 3/2-approximation result of [16]. [4] maintained $(3/2+\epsilon)$-approximate matching with an *amortized* update time $O(m^{1/4}/\epsilon^{2.5})$, generalizing their earlier work [3] for bipartite graphs, but the time bound in [3] is worst-case not amortized.

There are two main open questions in this area. The 1st is if one can maintain a *better-than-2* approximate matching in *amortized* polylogarithmic update time. The 2nd is:

▶ **Question 1.** *Can one maintain a "good" (close to 2) approximate matching and/or vertex cover with* worst-case *polylogarithmic update time?*

In a recent breakthrough, Bhattacharya, Henzinger and Nanongkai devised a deterministic algorithm that maintains a *constant* approximation to the minimum vertex cover, and thus also a constant-factor estimate of the maximum matching size, with polylogarithmic worst-case update time. While this result makes significant progress towards Question 1, this fundamental question remained open.[4] In particular, no algorithm for maintaining a matching with sub-polynomial worst-case update time was known, even if a polylogarithmic approximation guarantee on the matching size is allowed!

[3] In the standard *oblivious adversarial model* (cf. [8], [13]), the adversary knows all the edges in the graph and their arrival order, but is not aware of the random bits used by the algorithm.

[4] Later (in SODA'17 Proc. [7]) Bhattacharya et al. significantly improved the approximation to $2+\epsilon$. However, our result was done independently to [7]. Moreover, [7] solves Question 1 only for vertex cover. Independently of us, Arar et al. [1] solves Question 1 for matching by building on [7].

In this paper we devise a *randomized* algorithm that maintains an *almost-maximal matching (AMM)* with a polylogarithmic update time. We say that a matching for G is *almost-maximal* w.r.t. some slack parameter ϵ, or $(1-\epsilon)$-*maximal* in short, if it is maximal w.r.t. any graph obtained from G after removing $\epsilon \cdot |\mathcal{M}^*|$ arbitrary vertices, where $\mathcal{M}^*$ is a maximum matching for G. Just as a maximal matching provides a 2-approximation for matching and vertex cover, an AMM provides a $(2+\epsilon)$-approximation. We show:

▶ **Theorem 2.** *For any $\epsilon > 0$, one can maintain an AMM with worst-case update time $O(\mathsf{poly}(\log n, \epsilon^{-1}))$, where the $(1-\epsilon)$-maximality guarantee holds with high probability.*

Our update time is $O(\max\{\log^7 n/\epsilon, \log^5 n/\epsilon^4\})$; reducing this bound towards constant lies outside the scope of this paper; see Sec. 9 in the full version [9] (shortly, "i.t.f.v.").

The algorithm's worst-case guarantee can be strengthened, using [20], to bound the number of changes (replacements) to the matching. Optimizing this measure is important in various practical applications; refer to [20] for a motivation of this measure.

Our result resolves Question 1 in the affirmative, up to the ϵ dependency. In particular, it is essentially the best result possible under the UGC for the dynamic vertex cover problem; it started to circulate in Nov. 2016, and is independent of the $(2+\epsilon)$-vertex cover result of [7].

On the way to this result, we devise a *deterministic* algorithm that maintains an AMM with a polylogarithmic update time in a natural *offline model* that is described next. This deterministic algorithm may be of independent interest, as the offline setting seems important in its own right; see p. 3 i.t.f.v. for further details.

1.1 A Technical Overview

The offline model. Suppose the entire update sequence is known in advance, and is stored in some data structure. Suppose further that for any i, accessing the ith edge update via the data structure is efficient, taking $\mathsf{polylog}(n)$ or even $O(1)$ time. A natural question is whether one can exploit this knowledge of the future to obtain better algorithms for maintaining a good matching and/or vertex cover. Consider in particular the *maximal matching* problem, and a deletion of a matched edge (u, v) from the graph (which is the problematic part). If u has a free neighbor, we need to match them, and similarly for v. The algorithm may naively scan the neighbors of u and v, which may require $O(n)$ time. Surprisingly, this naive $O(n)$ bound is the state-of-the-art for general (dense) graphs, unless one allows both randomization and amortization [2, 19]. Can one do better in the offline setting?

We argue that a dynamic maximal matching can be maintained in the offline setting *deterministically* with constant *amortized* update time. To this end, we make the following observation: The machinery of [2, 19] extends *seamlessly* to the offline setting. More specifically, in contrast to the algorithms [2, 19], which choose the matched edge of v *uniformly at random* among a subset of its adjacent edges E_v that is computed carefully by those algorithms (details below), in the offline setting we choose the matched edge to be the one that will be deleted last among E_v. (We do not choose the edge that will be deleted last among all adjacent edges of v, as this is doomed; see the technical overview i.t.f.v). It is readily verified that the analysis of [2, 19] carries over to the offline setting directly.

The resulting deterministic algorithm for the offline setting is inherently *amortized*, whereas our focus is on *worst-case* bounds. To obtain good worst-case bounds, we build on the machinery of [2, 19]. The price of translating the amortized bounds of [2, 19] into a similar worst-case bound is that the maintained matching is no longer maximal, but rather

almost-maximal.[5] This translation is highly non-trivial, and is carried out in two stages. First, we consider the offline setting, and devise a deterministic algorithm there. Coping with the offline setting is easier than with the standard setting, as it allows us to ignore intricate probabilistic considerations, and to handle them separately. Second, we convert the results for the offline setting to the standard setting. The algorithm itself remains essentially the same. (Instead of choosing the edge that will be deleted last, choose a random edge.) On the other hand, showing that the maintained matching remains almost-maximal requires more work. This two-stage approach thus provides an elegant way to analyze randomized dynamic algorithms, and we believe it would be useful in other dynamic graph problems as well.

The framework of [2, 19]. We next provide a rough description of the *amortized* framework of [2, 19]. (The approach of [19] builds on the framework of [2] and extends it; for clarity, we won't distinguish between [2] and [19].)

Matched edges will be chosen randomly. If an edge $e = (u, v)$ is chosen to the matching uniformly at random among k adjacent edges of either u or v, w.l.o.g. u, we say that its *potential* is k. Under the oblivious adversarial model, the expected number of edges incident on u that are deleted from the graph before deleting edge (u, v) is $k/2$. Thus, following a deletion of a matched edge (u, v) with potential k from the graph, we have time $\tilde{O}(k)$ to handle u and v in the amortized sense.

Each vertex v dynamically maintains a *level* ℓ_v; informally, v's level will be logarithmic in the potential value of the matched edge adjacent to v. Free vertices are at level -1; matched vertices are at levels between 0 and $O(\log n)$. Based on vertices' levels, a dynamic edge orientation is maintained, where each edge is oriented towards the lower level endpoint.

When a vertex u becomes free, the algorithm (usually) chooses a mate for it randomly. If this mate w is already matched, say to w', the algorithm has to delete edge (w, w') from the matching in order to match u with w. However, we will be able to compensate for the loss in potential value (caused by deleting edge (w, w') from the matching) if it is significantly smaller than the potential of the newly created matched edge. Since vertices' levels are logarithmic in their potential, all neighbors of u with lower level should have potential value at most half the potential value of the new matched edge on u. In other words, for each of those neighbors, we can afford to break their old matched edge. Hence, the mate w of u will be chosen uniformly at random among u's neighbors with lower level.

A central obstacle is to distinguish between neighbors of u with level ℓ_u and those with lower level. Indeed, it is possible that most of u's neighbors have level ℓ_u, and none of them can be chosen as a mate for u. Roughly speaking, the execution of the algorithm splits into two cases. If the current out-degree of u is not (much) larger than its out-degree at the time its old matched edge got created, then we should be able to afford to scan all of them, due to sufficiently many adversarial edge deletions that are expected to occur. Notice that in this case the charging argument is based on *past* edge deletions.

The second case is when the out-degree of u is (much) larger than what it was when the old matched edge got created. The time needed for distinguishing u's neighbors at level ℓ_u from those at lower levels could be significantly larger than the "money" we got from

[5] The amortized update time analysis of the algorithm from [2] (both the FOCS'11 and subsequent journal SICOMP'15 versions) was erroneous, but was corrected in a subsequent erratum by the same authors. (The amortized update time analysis of the algorithm from [19] is different than the one used in [2], and does not have that mistake.) Although our algorithm builds on the machinery of [2, 19], the mistake in [2] does not affect the current paper, as we provide an independent analysis for a different algorithm, which bounds the worst-case update time of our algorithm rather than the amortized update time.

past edge deletions. In this case the algorithm *raises* u to a possibly much higher level ℓ^*, where there are not too many neighbors for u at that level as compared to the number of neighbors at lower levels. Having raised u to that level, we can perform the random sampling of its mate among all neighbors of level lower than ℓ^*. Notice that in this case the charging argument is not based on the past, but rather on *future* edge deletions.

Our approach. Note that the framework of [2, 19] is inherently *amortized*: Every once in a while there are "expensive" operations, which are charged to "cheap" operations that occurred in the past or will occur in the future. To obtain a low worst-case update time, we should be *cheap in any time interval*, thus we can rely neither on the past nor the future. Consider a matched edge (u, v) deleted by the adversary. We expect the adversary to make many edge deletions on at least one of these endpoints before deleting this edge. Alas, it is possible that all these edge deletions occurred a long time ago, which is useless for a worst-case algorithm. Consider the offline setting, and let $e_1, \ldots, e_\eta$ be η arbitrary matched edges with potential value k. For each such edge e_i, let $S(e_i)$ be its *sample*, i.e., the set of edges from which e_i was sampled to the matching. In the offline setting e_i will be deleted only after all $k-1$ other edges from its sample have been deleted. However, it is possible that the adversary first deletes the first $k-1$ edges from the samples of each of the matched edges, and only then turn to deleting the matched edges. Assuming k is large, it takes a long time for the adversary to delete the first $\eta(k-1)$ edges from all η samples, but the amortized algorithms of [2, 19] remain *idle* during all this time. An algorithm with a low worst-case update time must be active in this time interval, as immediately afterwards the adversary can remove the η matched edges from the graph, much faster than the algorithm can add edges to the matching in their place, leading to a poor approximation guarantee. Hence, at any point in time, the algorithm needs to be proactive and protect itself from such a situation happening in the future.

Generally, while in an amortized algorithm invariants may sometimes be violated and then restored via expensive operations, an algorithm with a low worst-case update time should persistently "clean" the graph, making sure that it is never close to violating any invariant. Naturally, we will need to maintain additional invariants to those maintained by the amortized algorithms [2, 19]. To this end we employ four different data structures that we call *schedulers*, each for a different purpose. Each of those schedulers consists of $O(\log n)$ sub-schedulers, a single sub-scheduler per level. Next we fix some level $\ell \approx \log k$, where k is the potential of the matched edges on that level, and focus on it.

The scheduler `unmatch-schedule` periodically removes edges from the matching, one after another, by always picking a matched edge whose *remaining sample* (i.e., the set of edges from the sample that have not been deleted yet from the graph) is smallest. As strange as it might seem, this strategy enables us to guarantee that only few matched edges will ever be deleted by the adversary. Note that removing a matched edge from the matching is not a cheap operation, as we need to find new mates for the two endpoints of the edge. Thus, the execution of the scheduler must be *simulated* over sufficiently many adversarial updates, which may include more deletions. But, as we control the rate at which the scheduler is working, we can make sure that it works sufficiently faster than the adversary. Therefore, in this "game" between the scheduler and the adversary, the scheduler will always win.

The role of `unmatch-schedule` is to make sure that all the samples are pretty full. Intuitively, this provides the counter-measure of relying on past adversarial edge deletions, as done in the amortized argument. The next scheduler `rise-schedule` provides the counter-measure of relying on future adversarial edge deletions. Recall that future edge deletions

are used in the amortized argument only in the case that a vertex had to rise to a higher level, which occurs only if its out-degree became too large for its current level. The role of `rise-schedule` is to make sure that vertices' out-degrees are always commensurate with their level. This scheduler periodically raises vertices to the level ℓ of which it is in charge, one after another, by always choosing to raise a vertex with the largest number of neighbors at level lower than ℓ. Although the two schedulers are based on the same principle, the game that we play here is not between the scheduler and the adversary, because here the algorithm itself may change the level of vertices and their out-degree, so `rise-schedule` has to compete against both the adversary and the algorithm. In contrast to the other scheduler, speeding up the rate at which `rise-schedule` works will not help winning the game. Instead, we manage to bound the speed of this scheduler with respect to that of the (adversary + algorithm), which enables us to show that the out-degree of all vertices is always in check.

For the offline model, these two schedulers suffice. However, in the oblivious adversarial model, the adversary will manage to delete some matched edges from time to time. The scheduler `free-schedule` periodically handles all the vertices that become free due to the adversary, one after another. Using the property that all samples are always pretty full, we manage to prove that only an ϵ-fraction of the matched edges get destroyed by the adversary at any time interval. Note that this bound is probabilistic – to make sure that it indeed occurs with high probability, we also use another scheduler `shuffle-schedule`, which periodically removes a random edge from the matching. For technical reasons, it is vital that `shuffle-schedule` would work sufficiently faster than some of the other schedulers.

2 The Update Algorithm

2.1 Invariants and schedulers. Our algorithm builds on the amortized algorithms by [2, 19], which maintain for each vertex v a *level* ℓ_v, with $-1 \leq \ell_v \leq \log_\gamma(n-1)$, where $\gamma = \Theta(\log n)$. (We use logarithms in base $\gamma = \Theta(\log n)$, whereas [2] and [19] use base 2 and 5, respectively.) Based on the levels of vertices, an edge orientation is maintained, with the vertex out-degree serving as an important parameter. The amortized algorithms of [2, 19] maintain the following invariants (Invariants 3(a)-3(d)) *at all times*. i.e., these invariants hold at the *end of the execution* of the update algorithms (and before the next update operation occurs). These invariants may become violated *throughout the execution* of the update algorithms. Also, the runtime of the update algorithms of [2, 19] may be $\Omega(n)$ in the worst case, thus it may take them a lot of time to restore the validity of these invariants, once violated. We added a comment to the right of each of these invariants, either /* maintained */ or /* partially maintained */, to indicate jf the respective invariant is maintained fully or only partially by our new algorithm. Our algorithm will maintain Invariants 3(a) and 3(b) at all times, as in the amortized algorithms [2, 19], where Invariants 3(c) and 3(d) are maintained only partially. Next, we make this statement precise.

▶ **Invariant 3.**

(a) *Any matched vertex has level at least 0. /* maintained */*

(b) *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 (a), as the level of its endpoints.) /* maintained */*

(c) *Any free vertex has level -1 and out-degree 0. (The matching is maximal.) /* partially maintained */*

(d) *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) is determined suitably by the algorithm.) /* partially maintained */*

Once a matched vertex becomes free, its level will exceed -1 until the update algorithm handles it. We say that such a vertex is *temporarily free* (shortly, *TF*), meaning that it is not matched to any vertex yet, but its level and out-degree remain temporarily as before. From now on, we distinguish between *free* and *TF* vertices: Free vertices are unmatched and their level is -1, while TF vertices are unmatched and their level exceeds -1. By making this distinction, Invariant 3(c) holds true as stated. Combining it with Invariant 3(a), we obtain:

▶ **Invariant 4.** *Any vertex of level -1 is unmatched and has out-degree 0.*

Invariants 3(c) and 4 do not apply to TF vertices; thus there may be edges between TF (thus unmatched) vertices, hence the matching is not maximal. The challenge is to guarantee that the number of TF vertices is small w.r.t. the number of matched vertices, yielding an AMM.

TF vertices are handled via data structures that we call *schedulers*. We distinguish between vertices that become TF due to the adversary and those due to the update algorithm itself. For each level ℓ, we maintain a queue Q_ℓ of level-ℓ vertices that become TF due to the adversary, and the vertices in Q_ℓ will be handled, one after another, via appropriate schedulers. We will need to make sure that the total number of vertices over the queues of all levels is in check at all times. The various schedulers need to work together, without conflicting each other; the exact way in which they work is described soon.

A TF vertex that is being handled by some scheduler is called *active*, and the process of handling it may be simulated over multiple update operations. Hence, there might be *inconsistencies* in the data structures throughout this process concerning the active vertices. To account for those inconsistencies, we hold a list of active vertices, denoted *Active*, and we will make sure that this list is of size $O(\log_\gamma n) = O(\log n)$ at any point in time. By bounding the number of active vertices, we can *authenticate* the up-to-date information concerning active vertices efficiently; this *authentication process* is described i.t.f.v in Section 2.3. Our algorithm maintains the following relaxation of Invariant 3(d).

▶ **Invariant 5.** *Any edge (u, v), with $\ell_u > \ell_v$ and $u, v \notin$ Active, is* oriented *as $u \to v$.*

For each vertex v, we maintain its neighbors and outgoing neighbors in linked lists $\mathcal{N}_v$ and $\mathcal{O}_v$, and its incoming neighbors via a more detailed structure $\mathcal{I}_v$; see p. 8 i.t.f.v.

Our algorithm employs four different schedulers, each of which consists of $O(\log_\gamma n) = O(\log n)$ sub-schedulers, a single sub-scheduler per level $\ell = 0, 1, \ldots, \log_\gamma(n-1)$. It is instructive to think of each sub-scheduler as running threads of execution, and of its scheduler as synchronizing $O(\log n)$ threads, one per level. Each thread executed by a level-ℓ sub-scheduler, hereafter *level-ℓ thread*, will run in the same amount of time $T_\ell = \gamma^\ell \cdot \Theta(\log^4 n)$, by "sleeping" if finishing the execution prematurely. To achieve a low worst-case update time, the execution of any such thread is not carried out at once, but is rather carried out (or *simulated*) over multiple update operations, simulating a fixed number of computation steps per update operation; we refer to that number as a *simulation parameter*, and we'll use two of them, $\Delta := \Theta(\log^5 n/\epsilon)$ and $\Delta' = \Delta \cdot \gamma = \Delta \cdot \Theta(\log n)$. The schedulers `free-schedule`, `rise-schedule` and `shuffle-schedule` use a simulation parameter of Δ', whereas `unmatch-schedule` uses a simulation parameter of Δ, and is thus "slower" than the others by a factor of $\gamma = \Theta(\log n)$. The simulation parameters, Δ or Δ', determine the number of update operations required to finish the execution of the thread, T_ℓ/Δ or T_ℓ/Δ', respectively. We refer to this number as the *(level ℓ) simulation time*; unlike the simulation parameters, which do not change with the level, the simulation times grow with each level by a factor of γ.

1st scheduler. The first scheduler `free-schedule` handles all vertices that become TF due to the adversary; for each level $\ell = 0, 1, \ldots, \log_\gamma(n-1)$, the corresponding sub-scheduler `free-schedule`$_\ell$ handles all vertices of Q_ℓ, one after another. The exact procedure for handling a TF vertex v, `handle-free`(v), is described in Section 2.4. Procedure `handle-free`(v) will be executed by a single level-ℓ thread corresponding to v that runs in an overall time of T_ℓ, simulating Δ' steps of this procedure following each update operation. The $\log_\gamma(n-1)+1$ execution threads (over all levels) executed by `free-schedule` following every update operation are handled *sequentially*, by decreasing order of simulation times, and thus by decreasing order of levels, i.e., the $\log_\gamma(n-1)$-level thread is handled first, then the $\log_\gamma(n-1)-1$-level thread, etc., until the 0-level thread. Note that these threads execute different calls of Procedure `handle-free`, which handle vertices at different levels. [[S: moved this sentence]] Following each update operation, the $\log_\gamma(n-1)$-level thread simulates Δ' steps of its own call of Procedure `handle-free`, the $\log_\gamma(n-1)-1$-level thread simulates Δ' steps of its own call, and so on, hence the total time spent by `free-schedule` following a single update operation is $\Delta' \cdot (\log_\gamma(n-1)+1) = O(\log^7 n/\epsilon)$.

By the same principle, the total time spent by `rise-schedule` and `shuffle-schedule` following a single update operation will be bounded by $\Delta' \cdot (\log_\gamma(n-1)+1) = O(\log^7 n/\epsilon)$. On the other hand, `unmatch-schedule` has a simulation parameter of Δ rather than Δ', so the total time spent by this scheduler following a single update will be bounded by $\Delta \cdot (\log_\gamma(n-1)+1) = O(\log^6 n/\epsilon)$. This scheme gives rise to a worst-case update time of $O(\log^7 n/\epsilon)$, and this bound holds *deterministically*.

2nd scheduler. The second scheduler `unmatch-schedule` removes matched edges from the matching; for each ℓ, the corresponding sub-scheduler `unmatch-schedule`$_\ell$ removes level-ℓ edges from the matching, one after another, as follows. Each level-ℓ matched edge $e = (u, v)$ is sampled uniformly at random from between $(1-\epsilon)\cdot\gamma^\ell$ and γ^ℓ edges. (In the offline setting, we choose the edge that will be deleted last among those.) We refer to this edge set, denoted by $S(e)$, as the *sample space* (or *sample*) of e. As time progresses, some edges of $S(e)$ may be deleted from the graph; denote by $S^*(e)$ the *original* sample of e, with $(1-\epsilon)\cdot\gamma^\ell \le |S^*(e)| \le \gamma^\ell$, and by $S_t(e) = S(e)$ its sample *remaining* at time t. The goal of `unmatch-schedule`$_\ell$ is to guarantee that the samples of all level-ℓ matched edges never reach $(1-2\epsilon)\cdot\gamma^\ell$; more accurately, `unmatch-schedule`$_\ell$ maintains the following invariant:

▶ **Invariant 6.** *For any level-ℓ matched edges e with $T_\ell/\Delta \ge 1$ and any t, $|S_t(e)| > (1-2\epsilon)\cdot\gamma^\ell$.*

To maintain this invariant, `unmatch-schedule`$_\ell$ will always remove a matched edge of smallest remaining sample (can be easily carried out in $O(1)$ time). For each level-ℓ matched edge $e = (u, v)$ that is removed by `unmatch-schedule`$_\ell$, its two endpoints u and v become TF, and they are handled by appropriate calls to Procedure `handle-free`. More specifically, we execute Procedure `handle-free`(u) and then `handle-free`(v) by running a level-ℓ thread, which runs in an overall time of T_ℓ, simulating Δ steps of execution following each update operation. The intuition as to why `unmatch-schedule`$_\ell$ can maintain Invariant 6 is the following. (See Section 4.2 i.t.f.v. for the formal argument.) Since $T_\ell = \gamma^\ell \cdot \Theta(\log^4 n)$ and $\Delta = \Theta(\log^5 n/\epsilon)$, the simulation time T_ℓ/Δ of a thread run by `unmatch-schedule`$_\ell$ (which designates the number of update operations needed for simulating its entire execution) is $\Theta(\epsilon(\gamma^\ell/\log n))$. In other words, `unmatch-schedule`$_\ell$ can remove a level-ℓ matched edge within $T_\ell/\Delta = \Theta(\epsilon(\gamma^\ell/\log n))$ adversarial update operations. On the other hand, the expected number of adversarial edge deletions needed to turn a "full" level-ℓ matched edge e (with sample $|S^*(e)| \ge (1-\epsilon)\cdot\gamma^\ell$) into an "under full" edge (with sample $\le (1-2\epsilon)\cdot\gamma^\ell$) is

$\Omega(\epsilon \cdot \gamma^\ell)$. Thus `unmatch-schedule`$_\ell$ is "faster" than the adversary by at least a logarithmic factor, assuming $T_\ell/\Delta \geq 1$ (which holds when $\gamma^\ell = \Omega(\log n/\epsilon)$), a property that suffices for showing that no edge is ever under full, i.e., the samples of all level-ℓ matched edges are always in check. This is the idea behind maintaining the validity of Invariant 6 in levels ℓ for which the simulation time satisfies $T_\ell/\Delta \geq 1$. This invariant, in turn, guarantees that the adversary is unlikely to delete any particular edge from the matching, using which we show (in Section 6 i.t.f.v.) that the maintained matching is always almost-maximal with high probability. The complementary regime of levels, namely, levels ℓ for which $T_\ell/\Delta < 1$, is trivial and does not rely on Invariant 6, as then the adversary does not make any edge deletion within the time required by a level-ℓ thread to complete its entire execution.

3rd scheduler. Let $N_v(\ell)$ denote the set of neighbors of v with level strictly lower than ℓ, and write $\phi_v(\ell) = |N_v(\ell)|$. For each vertex v, we will maintain the $\phi_v(\ell)$ values for all levels ℓ greater than the current level ℓ_v of v. For any level $\ell \leq \ell_v$, the corresponding value $\phi_v(\ell)$ will not be maintained, and the algorithm will have to compute it "on the fly", if needed. The algorithm of [2] maintains the invariant that $\phi_v(\ell) < \gamma^\ell$, for any v and $\ell > \ell_v$. (Recall that γ is taken to be constant in [2], whereas here we take γ to be $\Theta(\log n)$.) The scheduler `rise-schedule` maintains the following relaxation of the invariant from [2], and it does so by raising vertices to higher levels in a specific order, as described next.

▶ **Invariant 7.** *For any vertex v and any level $\ell > \ell_v$, $\phi_v(\ell) \leq \gamma^\ell \cdot O(\log^2 n)$.*

For each level ℓ, the corresponding sub-scheduler `rise-schedule`$_\ell$ is responsible for maintaining the invariant w.r.t. that level. Whenever a new level-ℓ thread is initiated by `rise-schedule`$_\ell$, it starts by *authenticating* the $\phi_v(\ell)$ values over all vertices v using the *Active* list. (The authentication process takes time $O(\log^2 n)$ to guarantee that all $\phi_v(\ell)$ values are up to date, and is described in Section 2.3 i.t.f.v.) Then the thread picks a vertex v whose $\phi_v(\ell)$ value is highest among all vertices with level lower than ℓ (can be easily carried out in $O(1)$ time). These steps take time $O(\log^2 n)$, and are thus carried out by the thread "instantly", i.e., without simulating their execution over subsequent update operations. The same execution thread continues to removing v's old matched edge (v, w) (if exists) from the matching, and raises v to level ℓ by executing Procedure `set-level`(v, ℓ), whose description is in Section 2.2. The runtime of this procedure is high, so its execution is simulated over multiple update operations, simulating Δ' execution steps following each update operation. Then the same execution thread handles the two TF vertices v and w using Procedure `handle-free`, i.e., it continues to executing the call to `handle-free`(v) and then to `handle-free`(w), simulating Δ' execution steps following each update operation.

4th scheduler. The fourth scheduler `shuffle-schedule` removes matched edges from the matching uniformly at random. By working faster than some other schedulers (`unmatch-schedule` in particular), it forms a dominant part of the algorithm, using which we show (Section 6.2 i.t.f.v.) that it provides a near-uniform random shuffling of the matched edges. This random shuffling facilitates the proof of the assertion that the adversary is unlikely to delete any particular edge from the matching. For each ℓ, the sub-scheduler `shuffle-schedule`$_\ell$ always picks a matched edge uniformly at random among all remaining level-ℓ edges, and then removes it from the matching. As with `unmatch-schedule`$_\ell$, for each level-ℓ matched edge $e = (u, v)$ that is removed by `shuffle-schedule`$_\ell$, its two endpoints u and v become TF, and they are handled by calls to Procedure `handle-free`. We execute these calls (to `handle-free`(u) and `handle-free`(v)) by running a level-ℓ thread, which runs

in an overall time of T_ℓ, and we simulate Δ' (rather than Δ as with `unmatch-schedule`$_\ell$) execution steps following each update operation, which ensures that `shuffle-schedule` is faster than `unmatch-schedule` by a logarithmic factor. We only need to apply the shuffling in levels ℓ for which the simulation time satisfies $T_\ell/\Delta \geq 1$, as in the complementary regime ($T_\ell/\Delta < 1$) the adversary does not make any edge deletion within the time required by a level-ℓ thread to complete its entire execution, and then a random shuffling is redundant.

2.2 Procedure `set-level`(v, ℓ). (This procedure is described in detail in Section 3.1 i.t.f.v.) Whenever the update algorithm examines a vertex v, it may need to re-evaluate its level. After the new level ℓ is determined (outside this procedure, details below), the algorithm calls Procedure `set-level`(v, ℓ). Although setting the level of v to ℓ can be done instantly, the task of Procedure `set-level`(v, ℓ) is to update the relevant data structures as a result of this level change. This process involves updating the sets of outgoing and incoming neighbors of v and some of its neighbors (or "flipping" the respective edges) so as to maintain Invariant 5, and also updating the ϕ values of v and its relevant neighbors. We refer to this (possibly long) process as the *falling* (if $\ell < \ell_v$) or *rising* of v (if $\ell > \ell_v$); the thread executing this procedure simulates multiple execution steps following each update operation. We will need to make sure that any call to `set-level`(v, ℓ) is executed by a level-$\hat{\ell}$ thread, where $\hat{\ell} \geq \tilde{\ell} := \max\{\ell_v, \ell\}$. We show (see Lemma 3.2 i.t.f.v.) that the runtime of this procedure is bounded by $O((\phi_v(\tilde{\ell}+1) + \log n) \cdot \log n)$, where $\phi_v(\tilde{\ell}+1)$ is the number of v's neighbors of level $< \tilde{\ell}+1$ at the beginning of this procedure's execution.

2.3 Procedures `handle-insertion`(u, v) and `handle-deletion`(u, v). (These procedures are described in detail in Section 3.2 i.t.f.v.) An edge insertion (u, v) is handled (via `handle-insertion`(u, v)) in the obvious way in time $O(\log^4 n)$, which is within the time reserved for a single update operation. An edge deletion (u, v) is handled (via `handle-deletion`(u, v)) similarly, unless (u, v) is matched, in which case both u and v become TF, and they are inserted to the queue Q_{ℓ_u} (by Invariant 3(b) $\ell_u = \ell_v$). As described above, `free-schedule`$_{\ell_u}$ will handle u and v (by making the calls to `handle-free`(u) and `handle-free`(v)), one after another, after handling all preceding vertices in Q_{ℓ_u}.

2.4 Procedure `handle-free`(v). (This procedure is described in detail in Section 3.3 i.t.f.v.) This procedure handles a TF vertex v, and is first invoked by the schedulers as described above, but then also recursively. It starts by computing the highest level $\ell, 0 \leq \ell \leq \ell_v$, where $\phi_v(\ell) \geq \gamma^\ell$, and the corresponding vertex set $N_\ell(v)$ of v, in order to sample a neighbor w of level $< \ell$ as v's new mate. The sampling is done from the set $N'_\ell(v)$ of non-active vertices in $N_\ell(v)$. To match v with w, we first delete the old matched edge (w, w') on w (if exists), thus rendering w' TF. Second, we let v and w fall and rise to the same level ℓ, respectively, by calling to `set-level`(v, ℓ) and `set-level`(w, ℓ). We then match v with w, thus creating a new level-ℓ matched edge (satisfying Invariant 3(b)). Finally, assuming w was previously matched to w', we handle w' recursively by calling to `handle-free`(w'). (In the degenerate case that no level ℓ as above exists, we have $\phi_v(0) = 0$, i.e., v does not have any neighbor at level -1, thus we call `set-level`$(v, -1)$ and v becomes free.)

Our update algorithm guarantees that this procedure is executed by a level-ℓ_v thread, where ℓ_v is v's level at the outset of the procedure's execution. The same thread is used also for all subsequent recursive calls. We show (Lemma 3.1 i.t.f.v.) that the runtime of Procedure `handle-free`(v) is bounded by $O(\gamma^{\ell_v} \cdot \log^4 n)$.

3 Analysis

3.1 Schedulers. The principle that governs the operation of `unmatch-schedule` and `rise-schedule` can be described via a balls and bins game by [11, 15, 10] between two players. Initially there are N empty bins. In each round Player I removes a bin of largest size, then Player II may add up to $b \geq 1$ balls to bins. The game ends when no bin is left or when the size of any bin reaches some parameter k. Player I wins (respectively, loses) in the former (resp., latter) case. As follows from [11, 15, 10], Player I wins if $b < \frac{k}{(\ln N+1)}$.

To prove that `unmatch-schedule` and `rise-schedule` maintain Invariants 4 and 5, respectively, we carefully build on this principle in several steps; see Sections 4.2 and 4.3 i.t.f.v. The analysis of `rise-schedule` is more intricate than that of `unmatch-schedule`, as our update algorithm affects both players in the underlying balls and bins game; we henceforth focus on `rise-schedule`, highlighting some insights behind our analysis. Consider the variant of the game where the bins are not empty initially, but rather contain at most $k' \ll k$ balls each. Using the same argument, Player 1 wins if $b < \frac{k-k'}{(\ln N+1)}$. We show that Invariant 7 is maintained by translating this variant of the game appropriately.

Fix any level $\ell \geq 0$. Invariant 7 requires that the $\phi_v(\ell)$ values are always $< \gamma^\ell \cdot O(\log^2 n)$, for all v with $\ell_v < \ell$. In the balls and bins game, the bins represent the respective vertex sets $N_v(\ell)$ (of v's neighbors of level $\leq \ell - 1$), for $\ell_v < \ell$. (Our algorithm does not maintain these sets, only the ϕ values.) The sub-scheduler $\texttt{rise-schedule}_\ell$ is Player I in the game; it always picks a vertex v whose $\phi_v(\ell)$ value is highest, and raises it to level ℓ. Following this rise, $\ell_v = \ell$, hence the invariant for v and level ℓ holds vacuously. Thus, the analog of removing a bin by Player I is to raise a vertex to level ℓ.

At the beginning the graph is empty, so all vertex levels are -1 and all vertex sets $N_v(\ell)$ are empty. Thus initially we have an empty bin for every vertex. As time progresses some of these bins are being removed due to vertex rising. When a vertex rises to level ℓ, all its bins up to level ℓ are removed instantly. Bins are also created due to vertices falling, by Procedure `handle-free`. When a vertex v starts falling from level ℓ_v to level ℓ, it is as if the corresponding vertex sets $N_v(j)$ in all levels $j \in \{\ell + 1, \ldots, \ell_v\}$ are created instantly; the level of v is viewed as its destination level ℓ from the moment its falling to level ℓ starts. Although the same vertex set $N_v(j)$ may be removed and created multiple times, we view any such newly created set as a different bin that was there from the game's outset. To comply with the initial bound of $\leq k'$ balls in any bins, we set $k' = k'_\ell$ as γ^ℓ, and prove (Lemma 4.1 i.t.f.v.) that any newly created level-ℓ bin contains $\leq k' = \gamma^\ell$ balls.

The level-ℓ vertex sets $N_v(\ell)$ and values $\phi_v(\ell)$ may grow either due to edge insertions (by adversary) or due to falling vertices (by update algorithm). In other words, Player II in the game is (adversary + update algorithm). Letting Player I ($\texttt{rise-schedule}_\ell$) work faster than the other sub-schedulers is problematic: While this would lead to more vertices rising, which helps Player I win, each vertex rising may trigger the fall of another vertex, which has the opposite effect. Instead, we prove (Lemma 4.2 i.t.f.v.) that the number of balls $b = b_\ell$ added to the bins by Player II while Player I removes a bin is $O(\gamma^\ell \cdot \log n)$. It is easy to verify that the number N of level-ℓ bins is polynomially bounded, so $\ln N = \Theta(\log n)$. Taking $k = O(\gamma^\ell \cdot \log^2 n)$ completes the translation of the balls and bins game. By setting the constant appropriately, we obtain $b < \frac{k-k'}{\ln N}$, hence Player I wins the game. Consequently, we showed that Invariant 7 is maintained.

3.2 Proof of (Almost-)Maximality. To prove almost-maximality, we show that the number of TF vertices is always an ϵ-fraction of the number of matched edges. We only consider here TF vertices due to the adversary of levels ℓ with $T_\ell/\Delta \geq 1$, as the complementary case

is easy. Any matched edge is created by the algorithm by first determining its level, and only then performing the sampling. If the edge is matched at level ℓ, it is chosen uniformly at random from between $(1-\epsilon)\cdot\gamma^\ell$ and γ^ℓ edges. We thus fix some level ℓ with $T_\ell/\Delta \geq 1$, and focus on the matched edges at that level. Invariant 6 holds for level ℓ, thus the samples of all level-ℓ edges always contain with probability (w.p.) 1 at least $(1-2\epsilon)\cdot\gamma^\ell$ edges.

Consider any time step t, and let V_t be the set of vertices of level ℓ at time t. Let $A_t = A'_t \cap A''_t$, where A'_t is the event that $|V_t| = \Omega(\log^4 n/\epsilon^3)$ and A''_t is the event that an $\Omega(\epsilon)$-fraction of the vertices of V_t are TF due to the adversary at time t. We argue that $\mathbb{P}(A_t) = O(n^{-c+2})$, for some (big enough) constant c. This assertion, which is given as Lemma 6.1 i.t.f.v., is central in our proof of the almost-maxiamlity guarantee, and the almost-maximality guarantee is derived as a simple corollary (Theorems 6.5 and 6.6 i.t.f.v.). Next, we give some insights behind the proof.

Any matched edge is sampled uniformly at random from between $(1-\epsilon)\cdot\gamma^\ell$ and γ^ℓ edges. Consider the edges of the sample $S^*(e)$ of e in the order they are deleted by the adversary, *even after the edge is removed from the matching*, either by the adversary or by the algorithm. A matched edge is called *bad* if it is one of the first (at most) $2\epsilon\cdot\gamma^\ell$ edges in this ordering; otherwise it is good. Invariant 6 guarantees that the samples of all level-ℓ edges always contain $\geq (1-2\epsilon)\cdot\gamma^\ell$ edges, so at most $2\epsilon\cdot\gamma^\ell$ edges are deleted from the sample of any matched edge (while it is matched). It follows that a good edge cannot get deleted by the adversary while it is matched (hereafter, get *hit*); a bad edge may get hit.

The probability of an edge to be bad is $\leq \frac{2\epsilon\cdot\gamma^\ell}{(1-\epsilon)\cdot\gamma^\ell}$, which is at most 4ϵ for all $\epsilon < 1/2$. Our argument, alas, is not applied on all matched edges created since the algorithm's outset, but rather on a subset of edges that are matched at a certain time step t', and there are dependencies on previous coin flips of our algorithm, which are the result of edges being removed from the matching by the update algorithm itself (not the adversary). Indeed, given that some edge e is matched at time t', the sample of e may be significantly reduced, which could increase the probability of e being bad. To overcome this hurdle, we use `shuffle-schedule` to show that the fraction of bad edges at any time is $O(\epsilon)$ w.h.p. To this end, we apply a game, hereafter the *shuffling game*, where in each step a single edge is either added or deleted (starting with no edges) by the following players: (1) `Adder`: adds an edge, which is bad w.p. $\leq 4\epsilon$, (2) `Shuffler`: deletes an edge uniformly at random among the existing edges, (3) `Malicious`: deletes a good edge. A newly created matched edge is bad w.p. $\leq 4\epsilon$, thus `Adder` assumes the role of creating matched edges by the algorithm, and so only an $O(\epsilon)$-fraction of the matched edges created by `Adder` are bad w.h.p. `Shuffler` assumes the role of `shuffle-schedule` in the algorithm, deleting matched edges uniformly at random. If the fraction of bad edges during some time interval is $\Theta(\epsilon)$, the fraction of bad edges deleted by `Shuffler` in this interval is $\Theta(\epsilon)$ w.h.p., hence `Shuffler` does not change the fraction of bad edges by too much. The role of `Malicious` is not to model the exact behavior of the other (non-shuffled) parts of the algorithm that remove matched edges, but rather to capture the worst-case scenario that might happen. We show that the affect of `Malicious` to the game is negligible, which implies that even if the other (non-shuffled) parts were to delete only good edges, the fraction of bad edges would be $O(\epsilon)$; formally, we prove that the fraction of bad edges at any step t' is w.h.p. $O(\epsilon)$. This proof, provided in Section 6.2 i.t.f.v, is nontrivial and makes critical use of the property that `Shuffler` is faster than `Malicious` by a logarithmic factor; we then show that the parties corresponding to `Shuffler` and `Malicious` in the algorithm indeed satisfy this property (Lemma 6.2 i.t.f.v.).

Equipped with this bound on the fraction of bad edges at any time step, we consider the last time t' prior to t in which the queue Q_ℓ of TF level-ℓ vertices is empty, i.e., Q_ℓ is non-empty in the entire time interval $[t'+1, t]$, thus `free-schedule`$_\ell$ is never idle during that

time. We need to bound the fraction of bad edges not only among the ones matched at time t', but also among those that get matched between times t' and t. The fraction of bad edges among those matched at time t' is $O(\epsilon)$ w.h.p. by the shuffling game; as for those that get created later on, there is no dependency on coin flips that the algorithm made prior to time t', and so the probability of any of those edges to be bad is $\leq 4\epsilon$, independently of whether previously created matched edges are bad, and by Chernoff we get that the fraction of bad edges among them is $O(\epsilon)$ too. The formal proof for this bound on the fraction of bad edges among those is provided in Section 6.3 i.t.f.v., and it implies that only an $O(\epsilon)$-fraction of all those edges may get hit w.h.p., and thus get into the queue. This bound, however, does not suffice to argue that the number of vertices in Q_ℓ at time t is an $O(\epsilon)$-fraction of the matching size, due to edges that get deleted from the matching by the algorithm itself. Nonetheless, since `free-schedule`$_\ell$ is no slower than the other sub-schedulers (as its simulation parameter is Δ'), we show in Section 6.3 i.t.f.v. that it removes vertices from Q_ℓ in the interval $[t'+1, t]$ at least at the same rate as matched edges get deleted by the algorithm. By formalizing these assertions and carefully combining them, we conclude with the required result, and with the almost-maximality guarantee as a corollary. The deterministic worst-case update time follows from the description of the algorithm (refer to the first paragraph of page 8).

References

1. Moab Arar, Shiri Chechik, Sarel Cohen, Cliff Stein, and David Wajc. Dynamic matching: Reducing integral algorithms to approximately-maximal fractional algorithms. *CoRR*, abs/1711.06625, 2017.
2. Surender Baswana, Manoj Gupta, and Sandeep Sen. Fully dynamic maximal matching in $O(\log n)$ update time. In *Proc. of 52nd FOCS*, pages 383–392, 2011 (see also *SICOMP'15* version, and subsequent erratum).
3. Aaron Bernstein and Cliff Stein. Fully dynamic matching in bipartite graphs. In *Proc. 42nd ICALP*, pages 167–179, 2015.
4. Aaron Bernstein and Cliff Stein. Faster fully dynamic matchings with small approximation ratios. In *Proc. of 26th SODA*, pages 692–711, 2016.
5. Sayan Bhattacharya, Monika Henzinger, and Giuseppe F. Italiano. Deterministic fully dynamic data structures for vertex cover and matching. In *Proc. 26th SODA*, pages 785–804, 2015.
6. Sayan Bhattacharya, Monika Henzinger, and Danupon Nanongkai. New deterministic approximation algorithms for fully dynamic matching. In *Proc. 48th STOC*, pages 398–411, 2016.
7. Sayan Bhattacharya, Monika Henzinger, and Danupon Nanongkai. Fully dynamic approximate maximum matching and minimum vertex cover in $O(\log^3 n)$ worst case update time. In *Proc. of 28th SODA*, pages 470–489, 2017.
8. Larry Carter and Mark N. Wegman. Universal classes of hash functions. In *Proc. 9th STOC*, pages 106–112, 1977.
9. Moses Charikar and Shay Solomon. Fully dynamic almost-maximal matching: Breaking the polynomial barrier for worst-case time bounds. *CoRR*, abs/1711.06883, 2017.
10. Paul F Dietz and Rajeev Raman. Persistence, amortization and randomization. In *Proc. of 2nd SODA*, pages 78–88, 1991.
11. Paul F. Dietz and Daniel Dominic Sleator. Two algorithms for maintaining order in a list. In *Proc. of 19th STOC*, pages 365–372, 1987.
12. Manoj Gupta and Richard Peng. Fully dynamic $(1+\epsilon)$-approximate matchings. In *54th FOCS*, pages 548–557, 2013.

13 Bruce M. Kapron, Valerie King, and Ben Mountjoy. Dynamic graph connectivity in polylogarithmic worst case time. In *Proc. of 24th SODA*, pages 1131–1142, 2013.

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

15 Christos Levcopoulos and Mark H. Overmars. A balanced search tree with O (1) worst-case update time. *Acta Inf.*, 26(3):269–277, 1988.

16 Ofer Neiman and Shay Solomon. Simple deterministic algorithms for fully dynamic maximal matching. In *Proc. 45th STOC*, pages 745–754, 2013.

17 Krzysztof Onak and Ronitt Rubinfeld. Maintaining a large matching and a small vertex cover. In *Proc. of 42nd STOC*, pages 457–464, 2010.

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

19 Shay Solomon. Fully dynamic maximal matching in constant update time. In *Proc. 57th FOCS*, pages 325–334, 2016.

20 Shay Solomon. Dynamic approximate matchings with an optimal recourse bound. *CoRR*, abs/1803.05825, 2018.