A Faster Deterministic Algorithm for Fully Dynamic Maximal Matching*

Julia Chuzhoy
Toyota Technological Institute at Chicago
Chicago, USA
cjulia@ttic.edu

Sanjeev Khanna
New York University
New York, USA
sanjeev.khanna@nyu.edu

Junkai Song
New York University
New York, USA
junkaisong@nyu.edu

Abstract

In the fully dynamic maximal matching problem, the goal is to maintain a maximal matching in a graph undergoing an online sequence of edge insertions and deletions, while minimizing the update time. The problem has been studied extensively in the *oblivious-adversary* setting, where randomized algorithms with polylogarithmic worst-case and constant amortized update time have been known for some time. A major challenge in this area has been designing an algorithm with non-trivial update time against an *adaptive* adversary, who may explicitly tailor the update sequence to the algorithm's choices. In a recent breakthrough, Bernstein, Bhattacharya, Kiss, and Saranurak (STOC 2025; hereafter, BBKS25) obtained the first algorithms with sublinear in n update time for this setting: namely, a randomized algorithm with $\tilde{O}(n^{3/4})$ amortized update time, and a deterministic algorithm with $\tilde{O}(n^{8/9})$ amortized update time. Our main result is a deterministic algorithm for fully dynamic maximal matching with amortized update time $n^{1/2+o(1)}$.

A powerful tool in dynamic matching is the use of matching sparsifiers: sparse subgraphs that preserve enough information to recover matchings with desired properties. Sparsifiers have been successfully used for approximate maximum matching, yielding sublinear update-time algorithms even against adaptive adversaries. For maximal matching, however, this paradigm is not as natural, since maximality must hold with respect to the entire graph, and so the algorithm must be able to detect and repair violations across all edges. Nevertheless, BBKS25 showed that the EDCS data structure can be ingeniously repurposed as a verification-and-repair mechanism for fully dynamic maximal matching against adaptive adversaries.

We introduce a new deterministic framework, referred to as the *subgraph system*, which, in contrast to the EDCS data structure used by BBKS25, is purpose-built for verification and maintenance of maximality. The structure of the subgraph system is also carefully designed to allow efficient recursive refinements leading to stronger and stronger parameters. This recursive approach yields our deterministic algorithm with $n^{1/2+o(1)}$ amortized update time, and provides a new deterministic framework for one of the central graph optimization problems in the dynamic setting.

CCS Concepts

• **Theory of computation** → **Dynamic graph algorithms**.

Keywords

Dynamic Algorithm, Graph Algorithm, Maximal Matching

ACM Reference Format:

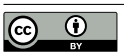
Julia Chuzhoy, Sanjeev Khanna, and Junkai Song. 2026. A Faster Deterministic Algorithm for Fully Dynamic Maximal Matching. In *Proceedings of the 58th Annual ACM Symposium on Theory of Computing (STOC '26), June 22–26, 2026, Salt Lake City, UT, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3798129.3800868

1 Introduction

We study the fully dynamic maximal matching problem: given an n-vertex graph G that undergoes an online sequence of edge insertions and deletions, the goal is to maintain a matching M, that is a maximal matching with respect to the current graph G at all times. Maximal matching is a fundamental graph theoretic notion, and algorithms for computing it serve as a key subroutine in dynamic algorithms for coloring, scheduling, and approximate maximum matching. As such, the problem has been studied extensively in various settings, including that of dynamic algorithms.

In the *oblivious* adversary scenario, where the update sequence is fixed in advance, fully dynamic maximal matching is by now a well-understood problem. Baswana, Gupta, and Sen [BGS11, BGS18] designed the first algorithm with $O(\text{poly}\log n)$ amortized update time, that was later improved to an $O(1)$ amortized update time by Solomon [Sol16], and to $O(\text{poly}\log n)$ worst-case update time by Bernstein, Forster, and Henzinger [BFH21]. In contrast, in the setting of the *adaptive adversary*, where the update sequence may depend on the algorithm's decisions, progress was long stalled. Until very recently, the best known algorithm in this setting, due to Neiman and Solomon [NS15], achieved an $O(\sqrt{m})$ worst-case update time, improving upon the earlier $O((n+m)^{1/\sqrt{2}})$ result of

*Julia Chuzhoy is supported in part by NSF award CCF-2402283 and NSF HDR TRIPODS award 2216899. Sanjeev Khanna is supported in part by NSF award CCF-2402284 and AFOSR award FA9550-25-1-0107.



STOC '26, Salt Lake City, UT, USA

ACM ISBN 979-8-4007-2536-4/2026/06
https://doi.org/10.1145/3798129.3800868

Ivković and Lloyd [IL93]; both these algorithms are deterministic. Thus in dense graphs, until very recently, even with randomization, no algorithm was known to achieve a better than $\Theta(n)$ amortized update time against an adaptive adversary—the same update time bound as the trivial approach of restoring maximality by exhaustively searching the neighborhood of a vertex affected by a deletion. To summarize, while algorithms with $O(1)$ amortized update time have been known for the fully dynamic maximal matching problem in the oblivious adversary setting for some time, in the adaptive adversary setting, even the problem of achieving a sublinear in n amortized update time was open until very recently.

To gain intuition for this stark discrepancy, consider a vertex v of degree $\Omega(n)$. The most challenging updates are deletions of edges that lie in the current matching M maintained by the algorithm, since such deletions require that the affected vertices are rematched. If an edge that is incident to v in the matching M that the algorithm maintains is selected uniformly at random among the $\Omega(n)$ incident edges of v in G, then in *expectation*, an oblivious adversary must perform $\Omega(n)$ deletions before removing the matching edge incident on v. Thus, even if the algorithm spends $\Theta(n)$ time to rematch v, it still achieves an $O(1)$ amortized update time. In contrast, an adaptive adversary can deliberately target matching edges incident to high-degree vertices, forcing updates that require $\Omega(n)$ time if a naive rematching strategy is used.

As every maximal matching is a 1/2-approximation to the maximum matching, and since the past decade has seen major progress on algorithms for *approximate maximum matching* in dynamic graphs (e.g., [OR10, GP13, BS15, BS16, BHN16, BCH17, BHI18, ACC+18, Waj20, BDL21, Kis22, RSW22, BK22, Beh23, ABKL23, BKS23, BKSW24a, BKSW24b, ABR24, Liu24, BG24, AKK25]), it is natural to ask whether techniques employed in these algorithms can be extended to the maximal matching setting. A common tool underlying most these algorithms is the notion of a *matching sparsifier*—a sparse subgraph that compresses the input fully dynamic graph G, while preserving enough information, so that an approximate matching can be recovered using only the edges of the sparsifier. Prominent examples of this approach include the use of *kernel* in algorithms for $(1/2-\epsilon)$-approximate matchings [BHI18], the use of the *Edge-Degree Constrained Subgraph (EDCS)* in algorithms for $(2/3-\epsilon)$-approximate matchings [BS15, BS16], and algorithms based on Szemerédi's regularity lemma and Ruzsa–Szemerédi graphs, that achieve $(1-o(1))$-approximation [ABKL23, BG24, AKK25]. These structures lead to a sublinear in n update time, because a matching of the desired quality can be found while restricting the attention to the sparsifier graph, which is much sparser than G. Moreover, these sparsifier structures have proven robust enough to yield sublinear update times even against an adaptive adversary. In the context of maintaining a maximal matching, however, this sparsification paradigm is less natural, since maximality must hold with respect to the *entire* graph, and hence requires the ability to detect and repair violations that may involve edges outside of the sparsifier. The core algorithmic challenge thus becomes one of *efficient verification and repair* of maximality. An additional impediment is that, unlike the setting of approximate matchings, where, for example, a valid $(1-\Theta(\epsilon))$-approximate matching remains so even if one ignores the next $\Theta(\epsilon \cdot n)$ updates [GP13, AKL19, Kis22], maximality is fragile: it can be violated by the insertion or the deletion of even a single edge.

A major breakthrough was achieved very recently by Bernstein, Bhattacharya, Kiss, and Saranurak [BBKS25], who obtained a randomized algorithm with $\tilde{O}(n^{3/4})$ amortized update time and a deterministic algorithm with $\tilde{O}(n^{8/9})$ amortized update time; both algorithms work against an adaptive adversary. In their algorithm, [BBKS25] ingeniously repurpose the EDCS data structure as a verification-and-repair mechanism, using it as the algorithmic backbone for maintaining a dynamic maximal matching. While [BBKS25] demonstrate that the EDCS framework can be adapted to the maximal matching setting, their approach inherits some structural limitations, since EDCS was originally designed for approximate matchings rather than for enforcing maximality. In particular, an EDCS acts as a *sparse subgraph* of the underlying graph, relying on degree constraints that ensure approximate matchings can be maintained efficiently. Some of the challenges of using EDCS as a backbone include a high update time for maintaining the EDCS data structure in the fully dynamic setting [GSSU22], and difficulty of finding short augmenting paths efficiently in a dynamically changing graph.

Our Results. Our main result is a *deterministic* algorithm for maintaining a maximal matching in a fully dynamic graph with $n^{1/2+o(1)}$ amortized update time. Our approach is different from that of [BBKS25]: we introduce a new deterministic framework referred to as the *subgraph system*, designed specifically to allow efficient *verification and repair of maximality* directly, rather than to sparsify the graph while preserving an approximate matching. Our subgraph system design serves two key purposes. First, it allows for efficient deterministic construction (amortized over a batch of updates) and efficient maintenance of some key data structures as the underlying graph evolves. In particular, the subgraph system is designed to allow a recursive composition that gradually yields stronger and stronger properties. Second, it offers a structured way to detect and restore violations of maximality through bounded local operations. In this respect, the subgraph system provides for maximal matching what matching sparsifiers provide for approximate matching: an explicit, combinatorial framework that retains all information necessary to maintain maximality.

Adaptive vs. Oblivious Adversary. It is worth noting that maximal matching is not unique in suffering a large performance gap between the oblivious and the adaptive adversary settings. A similar dichotomy arises in other dynamic problems, such as maximal independent set and $(\Delta+1)$-coloring. In the oblivious adversary setting, algorithms with polylogarithmic or even constant amortized update times are known for both problems [BDH+19, CZ19, BGK+22, HP22], while in the adaptive adversary setting, the best known update times are polynomial in n [GK18, DZ18, BRW25, FH25]. More generally, such gaps are a recurrent phenomenon in dynamic graph problems. A very recent work by Bernstein, Bhattacharya, Fischer, Kiss, and Saranurak [BBF+26] establishes conditional update-time separations between adaptive and oblivious adversaries for multiple dynamic symmetry-breaking problems, including maximal independent set (MIS) and maximal clique. Specifically, they show that, under the combinatorial Boolean Matrix Multiplication (BMM)

hypothesis, every algorithm for the *incremental MIS* problem that works against an adaptive adversary must incur $n^{1-o(1)}$ amortized update time, whereas, in the oblivious adversary setting, polylogarithmic update time algorithms are known. Similarly, [BBF+26] show that, under the 3SUM or the APSP hypotheses, any algorithm for *decremental maximal clique* that works against an adaptive adversary requires $\Omega(\Delta/n^{o(1)})$ amortized update time when the initial maximum degree is $\Delta \leq \sqrt{n}$, while in the oblivious adversary setting, a poly $\log(n)$ update time is achievable. Together, these results show that the performance gap between oblivious and adaptive adversaries is a pervasive barrier across dynamic symmetry-breaking and related graph problems.

1.1 Our Results and Techniques

Our main result is summarized in the following theorem.

Theorem 1.1. *There is a deterministic algorithm that, given an n-vertex graph G initially containing no edges and undergoing an online sequence of edge insertions and deletions, maintains a maximal matching in G with amortized update time $n^{1/2+o(1)}$.*

We now provide a high-level overview of the proof of Theorem 1.1 and compare it with the approach of [BBKS25]. For clarity of exposition, we consider a slightly different setting, where the input n-vertex graph G initially may contain edges, but the only updates that it undergoes are edge deletions, so G is decremental and not fully dynamic. We also assume that the update sequence is sufficiently long, so, for example it contains at least $n^{4/3}$ updates. This setting seems to capture the main challenges of the problem, and extending the algorithm to the fully dynamic setting with an arbitrary number of updates is not difficult. We start by describing an algorithm with $\tilde{O}(n^{2/3})$ amortized update time, since this algorithm allows us to convey some of our main ideas, and to motivate the more involved data structures used in the algorithm with $n^{1/2+o(1)}$ amortized time.

1.2 Algorithm with $\tilde{O}\left(n^{2/3}\right)$ Amortized Update Time

The main new combinatorial object that we introduce is the *z-subgraph system*, that we describe next; for clarity of exposition, we omit some technical details that are not essential.

The z-subgraph system. Let G be a graph and let $1 \leq z \leq n$ be an integral parameter. A z-subgraph system for G consists of a subset $M \subseteq E(G)$ of edges, and a partition (A, B, U) of the vertices of G; we denote by $S = A \cup B$. We require that every vertex in S is incident to exactly z edges of M, and every vertex in U is incident to at most z edges of M. We also require that, for every vertex $u \in U$, at most z neighbors of u in G may lie in U, so $|N_G(u) \cap U| \leq z$. Additionally, we require that the following two crucial properties hold:

(P1) For every vertex $u \in U$, $|N_G(u) \cap B| \leq 2z$; and
(P2) For every vertex $a \in A$, for every edge $(a, v) \in M$ incident to a, $v \in S$ must hold.

Lastly, the subgraph system must contain, for every vertex $u \in U$, the list $\Lambda(u) = N_G(u) \cap (B \cup U)$ of vertices, whose length must be $O(z)$, and, for every vertex $a \in A$, the list $L(a) = N_G(a) \cap U$ of vertices, whose length may be arbitrary. We denote by $\Lambda = \{\Lambda(u)\}_{u \in U}$ and $L = \{L(v)\}_{v \in A}$, and we denote the entire z-subgraph system by (S, B, U, M, Λ, L).

We note that the notion of the z-subgraph system is somewhat similar to that of the *Edge-Degree Constrained Subgraph* (EDCS) data structure, that was used in the algorithm of [BBKS25] for dynamic maximal matching. The EDCS data structure was introduced by Bernstein and Stein [BS15, BS16] in the context of computing approximate maximum matchings, and it was used in numerous algorithms since (see [BBKS25] for futher references). Given a pair $0 < B^- < B$ of parameters, a (B, B^-)-EDCS of a graph G is a subgraph $G' \subseteq G$, that has the following two properties. First, for every edge $(u, v) \in E(G')$, $\deg_{G'}(u) + \deg_{G'}(v) \leq B$ must hold. Second, for every edge $(u, v) \in E(G) \setminus E(G')$, $\deg_{G'}(u) + \deg_{G'}(v) \geq B^-$. Note that the maximum vertex degree in G' is at most B, ensuring that G' is a sparse graph, regardless of the density of G, provided that B is chosen to be sufficiently small. The algorithm of [BBKS25] for fully dynamic maximal matching maintains a $(B, (1-\epsilon)B)$-EDCS G' of the input fully dynamic graph G, for some parameters $0 < \epsilon < 1$ and $B > 0$, by using the deterministic algorithm of [GSSU22], whose worst-case update time is $O\left(\frac{n}{\epsilon \cdot B}\right)$. In the algorithm of [BBKS25], the vertices of G are partitioned into three subsets, based on their degree in G': set V^{high} contains all vertices v with $\deg_{G'}(v) \geq \left(\frac{1}{2} + \Theta(\epsilon)\right) B$; set V^{low} contains all vertices v with $\deg_{G'}(v) \leq \left(\frac{1}{2} - \Theta(\epsilon)\right) B$; and set V^{med} contains all remaining vertices. The properties of the EDCS data structure are then exploited by [BBKS25] in order to maintain a matching M', in which every vertex of V^{high}, and the vast majority of the vertices of V^{med} are matched. Additionally, they maintain a maximal matching M'' in G, that is obtained by extending the matching M' to the vertices of V^{low}. In order to do so, they exploit the fact that each such vertex may only have relatively few neighbors in V^{low}. One can think of the parameter B in their EDCS data structure as playing a similar role to the parameter z in the z-subgraph system, with vertex sets V^{high}, V^{med} and V^{low} corresponding to sets A, B and U, respectively, in the z-subgraph system. However, there are several crucial differences between the two data structures. The first difference is the sharper degree bounds that are required for the vertices of S in the subgraph system. For example, this allows us to decompose the set M of edges into $(z+1)$ disjoint matchings $M_1, \ldots, M_{z+1}$ so that, for every vertex $v \in S$, all but at most one of the matchings M_i contain an edge incident to v. Additionally, Properties P1 and P2 of the z-subgraph system are specifically designed so as to make the task of maintaining a maximal matching in G easier, and it is not clear how to ensure these properties in the EDCS data structure. We also believe that the z-subgraph system is a simpler data structure, in that, it is both easier to construct and manipulate. In fact our final algorithm for maximal matching with $n^{1/2+o(1)}$ amortized update time uses a *multi-level* generalization of this data structure, that is constructed from the basic data structure using a recursive algorithm. The multi-level subgraph system is specifically designed

to allow an efficient recursive refinement of the parameters of the subgraph system.

Our algorithm for dynamic maximal matching uses two main algorithmic tools. The first tool allows us to construct a z-subgraph system of a graph G in time $\tilde{O}(|V(G)| + |E(G)|)$. The second tool allows us to maintain a maximal matching in a graph G undergoing a bounded number of edge deletions, given a z-subgraph system for the initial graph G. We now describe each of these tools in turn.

Constructing a z-subgraph system. We provide a simple algorithm, that, given an n-vertex and m-edge graph G, constructs a z-subgraph system (S, B, U, M, Λ, L) for G. The algorithm consists of two steps. In the first step, we construct a z-subgraph system for G that has all required properties except for, possibly, Property P1. Then in the second step we "fix" this subgraph system, to ensure that it has all required properties.

The algorithm for the first step is quite straightforward. We construct a maximal set $M \subseteq E(G)$ of edges with the following property: every vertex of G has at most z edges incident to it in M. Such an edge set can be constructed in $O(n+m)$ time using a straightforward greedy algorithm. For every vertex $v \in V(G)$, let $m(v)$ denote the number of edges of M incident to v. We then let S contain all vertices v with $m(v) = z$ and U contain all remaining vertices of G. From the maximality of M, it is immediate to verify that, for every vertex $u \in U$, $|N_G(u) \cap U| \leq z$. We also partition the vertex set S into two subsets: set A containing every vertex $v \in S$ such that every edge (v, v') of M incident to v connects v to a vertex in S; and set B containing all remaining vertices of S. Finally, we initialize the lists $\{\Lambda(u)\}_{u \in U}$ and $\{L(v)\}_{v \in A}$, in time $O(n+m)$. It is easy to verify that this data structure has all properties required from the z-subgraph system, except for, possibly, Property P1.

In the second step, we modify the subgraph system computed in the first step, in order to ensure that it has Property P1. In order to do so, we process every vertex of U one by one. When a vertex $u \in U$ is processed, we check whether $|N_G(u) \cap B| \geq z$ holds. If so, we insert edges connecting u to vertices of B into M, until $m(u) = z$ holds, and u can be moved from u to S, where it joins the set B. Note that, if an edge (u, v) connecting u to a vertex $v \in B$ is inserted into M, then, in order to ensure that $m(v) = z$ continues to hold, we need to delete another edge connecting v to a vertex of U from M. Whenever a vertex $v \in B$ loses all edges of M connecting it to vertices of U (that is, Property P2 holds for v), we move it from B to A. This ensures that, as long as a vertex v remains in B, there is an edge of M connecting it to a vertex of U; this, in turn, allows us to perform "edge switching" described above, when an edge connecting v to some new vertex $u \in U$ is inserted into M, and an edge connecting v to another vertex of U is deleted from M. It is not difficult to efficiently implement this step, and to update the lists $\{\Lambda(u)\}_{u \in U}$ and $\{L(v)\}_{v \in A}$ at the end of this step, in time $\tilde{O}(n+m)$. At the end of this procedure we obtain a valid z-subgraph system for G.

Our algorithm uses the parameter $z = \left\lfloor n^{2/3} \right\rfloor$. Unlike the algorithm of [BBKS25], that maintains the EDCS data structure for the input dynamic graph G over the course of the entire algorithm, we only compute the z-subgraph system for the graph G from time to time. Specifically, we partition our algorithm into *phases*, each of which spans exactly $r = \left\lfloor n^{4/3} \right\rfloor$ updates by the adversary (except for the last phase that may be shorter). We compute the z-subgraph system for G from scratch at the beginning of every phase, in time $\tilde{O}(|E(G)| + |V(G)|) \leq \tilde{O}(n^2)$. Since this computation is amortized over r updates to G that occur during the phase, we get that computing the z-subgraph system at the beginning of every phase can be executed in amortized $\tilde{O}(n^{2/3})$ update time. Our second main tool is an algorithm that exploits the z-subgraph system in the initial graph G (at the start of a phase) in order to maintain a maximal matching in G over the course of a phase spanning r edge deletions. We describe this algorithm next.

Maintaining a maximal matching via the z-subgraph system. We now describe our algorithm for implementing a single phase. We assume that we are given an n-vertex graph G (corresponding to the input dynamic graph G at the beginning of the current phase), that undergoes an online sequence of $r \geq n$ edge deletions. We also assume that we are given a z-subgraph system $\mathcal{S} = (S, B, U, M, \Lambda, L)$ for the initial graph G. We now describe an algorithm that exploits this subgraph system in order to maintain a maximal matching in G. As our first step, we use the almost linear-time *deterministic* algorithm of [ABB+26] for edge coloring in order to compute, in time $O\left(n^{1+o(1)} \cdot z\right)$, a partition $(M_1, \ldots, M_{z+1})$ of the set M of edges from the subgraph system into $z+1$ disjoint matchings (here, each color class defines a matching). Recall that, from the properties of the subgraph system, every vertex $v \in S$ has z edges incident to it in M; so there is at most one index $1 \leq i \leq z+1$, such that v is not matched in M_i. Therefore, in a typical matching M_i, all but at most $\frac{2|S|}{z}$ vertices of S are matched. We assume without loss of generality that matching M_1 has this property. Over the course of the algorithm, we may modify the matching M_1 by deleting some edges from it, and by inserting edges of M into it, so $M_1 \subseteq M$ will always hold. The remaining matchings $M_2, \ldots, M_{z+1}$ only undergo the deletions of edges that the adversary deletes from the graph G. Since G undergoes a sequence of r edge deletions, for most indices $2 \leq i \leq z+1$, the number of edges of M_i that are deleted by the adversary over the course of the algorithm is bounded by $\frac{2r}{z}$. It is then not hard to see that, at all times, there is an index $2 \leq i \leq z+1$, such that the number of vertices of S that are not matched by M_i is bounded by $\frac{2|S|}{z} + \frac{4r}{z} \leq \frac{32r}{z}$; denote this bound by τ. We partition each phase of the algorithm into *subphases*, each of which spans $\left\lfloor \frac{r}{z} \right\rfloor \approx n^{2/3}$ adversarial updates to graph G, so that the number of subphases is bounded by $2z$. We will ensure that, at the beginning of every subphase, all but at most $\tau = \frac{32r}{z}$ vertices of S are matched by M_1. Over the course of each subphase, our algorithm may delete edges from M_1, but it ensures that every edge deletion by the adversary may lead to at most two edge deletions from M_1. Since at most $\frac{r}{z}$ adversarial edge deletions may occur during a subphase, this ensures that, overall, at most $\frac{3r}{z}$ edges may be deleted from M_1 over the course of a subphase, and so at most $\frac{6r}{z}$ additional vertices of S may become unmatched during a subphase. Therefore, the following crucial invariant holds over the course of the subphase.

(I1) Throughout the subphase, all but at most $2\tau = \frac{64r}{z}$ vertices of S are matched by M_1.

Over the course of a subphase, our algorithm maintains a matching M^* that contains all edges of M_1, such that, at all times, M^* is a maximal matching in G. Recall that our algorithm ensures that $M_1 \subseteq M$ holds at all times. Since, from Invariant P2 of the subgraph system, for every vertex $a \in A$, every edge of M incident to a connects it to a vertex of S, from Invariant I1, we get that all but at most 2τ vertices of A are matched to vertices of S by M_1. Recall that $M_1 \subseteq M^*$, so the only way that a vertex $a \in A$ is matched to a vertex of U by M^* is if a is not matched by M_1. Therefore, the following property is also guaranteed to hold over the course of the subphase:

(I2) Throughout the subphase, the number of vertices of A that are matched by M^* to vertices of U is bounded by $2\tau = \frac{64r}{z}$.

Recall that, at the beginning of a subphase, we are given a matching M_1 in G, such that all but at most τ vertices of S are matched by M_1. Throughout the algorithm, we denote by $\hat{S}$ the collection of all vertices $v \in S$ that are currently not matched by M^*. From Property I1, $|\hat{S}| \leq 2\tau$ always holds. We start by setting $M^* = M_1$, and then we extend M^* to be a maximal matching in a relatively straightrforward manner, by inspecting all vertices of U and checking whether we can insert an edge connecting a vertex of U to vertices of $U \cup B \cup \hat{S}$ into M^*. We also iteratively check whether an edge connecting a pair of vertices in $\hat{S}$ can be inserted into M^*. Since $|\hat{S}| \leq O(r/z)$ and, for every vertex $u \in U$, $|N_G(u) \cap U| \leq |\Lambda(u)| \leq O(z)$, this can be done in time $\tilde{O}(nz + nr/z)$.

Throughout the algorithm, we say that a vertex $v \in V(G)$ is *settled* if either (i) some edge of M^* is incident to v; or (ii) for every vertex $y \in N_G(v)$, some edge of M^* is incident to y. It is easy to verify that matching M^* is maximal if and only if every vertex of G is settled. Whenever an edge $e = (u, v)$ is deleted from M^*, we execute a procedure that attempts to *rematch* each of its endpoints; in order to rematch a vertex x, we need to either insert an edge incident to x into M^*, or verify that every vertex in $N_G(x)$ is currently matched by M^*. In other words, we need to ensure that x is settled.

Intuitively, it is easy to rematch vertices of U efficiently. Specifically, consider a vertex $u \in U$ that needs to be rematched. Recall that $|N_G(u) \cap U| \leq O(z)$, so we can check, in time $O(z)$, whether some vertex $u' \in N_G(u) \cap U$ is currently not matched by M^*, and, if so, insert the edge (u, u') into M^*. Additionally, we can scan every vertex of $\hat{S}$ (recall that $\hat{S}$ contains all vertices of S that are currently not matched, and that $|\hat{S}| \leq O(\tau) \leq O(r/z)$), to check whether u can be matched to a vertex of S that is currently unmatched. Therefore, the procedure for rematching a vertex of U can be implemented in time $\tilde{O}\left(z + \frac{r}{z}\right)$. Moreover, rematching the vertices of B is also relatively easy, as long as we maintain an appropriate data structure: namely, a directed graph H, whose vertex set is $B \cup U$, and whose edge set contains, for every vertex $u \in U$ that is currently unmatched in M^*, an edge connecting u to every vertex in $N_G(u) \cap (B \cup U) = \Lambda(u)$. Recall that $|\Lambda(u)| \leq O(z)$, so, whenever a vertex u switches its status between matched and unmatched in M^*, we can insert all edges connecting u to vertices of $\Lambda(u)$, or delete them from H, as needed, in time $\tilde{O}(z)$. Whenever a vertex $b \in B$ becomes unmatched, we can then check whether H contains an edge (u, b) entering b in H. If so, then vertex u must be currently unmatched by M^*, and we can insert the edge (u, v) into M^*. If b has no incoming edges in H, then we are guaranteed that every vertex in $N_G(b) \cap U$ is currently matched by M^*. We then scan the vertices of $\hat{S}$ in order to check whether b can be matched to a vertex of S that is currently unmatched by M^*, in time $O(|\hat{S}|) \leq O(r/z)$. Overall, we can execute a rematching procedure for a vertex of $B \cup U$ in time $\tilde{O}(z + r/z)$, and we can maintain the graph H that allows us to rematch the vertices of B efficiently in time $\tilde{O}(z)$ per update.

The main challenge of the algorithm is in rematching the vertices of A. The problem is that a vertex $a \in A$ may be incident to a large number of vertices of U in G (recall that all such vertices are stored in list $L(a)$), and, moreover, every vertex of U may be incident to a large number of vertices of A. Therefore, whenever a vertex $a \in A$ loses an edge of M^* that is incident to it, it is not clear how to check whether a can be matched to a vertex of U efficiently. Our main idea is to give the vertices of A a *priority* in rematching, exploiting the fact that vertices of $B \cup U$ are relatively easy to rematch. Therefore, when attempting to rematch a vertex $a \in A$, we allow ourselves to delete at most one edge from M^* (and from M_1), provided that this edge deletion only affects vertices in $B \cup U$, which we can then rematch efficiently. Specifically, recall that, from Invariant I2, at most $2\tau = \frac{64r}{z}$ vertices of U may be matched to vertices of A at all times. Therefore, either $L(a) = N_G(a) \cap U$ contains at most $\frac{64r}{z}$ vertices, or among the first $\frac{64r}{z} + 1$ vertices of $L(a)$, there must be some vertex u that is not currently matched to a vertex of A by M^*. We then insert the edge (a, u) into M^*, and, if another edge (u, u') incident to u lies in M^*, we delete this edge from M^* (and from M_1, if $(u, u') \in M_1$). Notice that $u' \in B \cup U$ must hold, so we can rematch u' efficiently. Therefore, in order to rematch a vertex $a \in A$, we only need to scan $O(r/z)$ vertices of $L(a)$, and $O(r/z)$ vertices of $\hat{S}$. Overall, as long as Invariant I2 holds, we can maintain a maximal matching in M^* in amortized update time $\tilde{O}\left(z + \frac{r}{z}\right)$.

From our discussion so far, in order to guarantee that Invariant I2 holds, it is enough to ensure that, at the beginning of every subphase, all but at most $\tau = \frac{32r}{z}$ vertices of S are matched by M_1. As shown above, we are guaranteed that, at all times, there is an index $2 \leq i \leq z + 1$, such that all but at most τ vertices of S are matched by M_i. Therefore, the most natural approach would be to simply replace M_1 with a matching M_i that has this property at the beginning of every subphase. The problem is that we would then need to extend this matching to vertices of U and construct the data structure H from scratch, which is too time consuming (we can afford to do so once at the beginning of the algorithm, but not at the beginning of every subphase). Instead, we use the graph $M_i \cup M_1$ in order to gradually augment the matching M_1 via augmenting paths originating at vertices of S that are unmatched by M_1, so as to reduce the number of such vertices. This approach ensures that only $O(r/z)$ vertices of G may switch their status between matched and unmatched in M_1, which, in turn, allows us to repair the matching M^* efficiently. At the end of this "repair" procedure,

we are guaranteed that all but at most τ vertices of S are matched by M_1, and that M^* is a maximal matching for G with $M_1 \subseteq M^*$.

Overall, our algorithm spends $O\left(n^{1+o(1)} \cdot z\right)$ time in order to compute the initial collection $M_1, \ldots, M_{z+1}$ matchings. After that, it spends $\tilde{O}\left(z + \frac{r}{z}\right)$ time per update. Intuitively, we spend $O(z)$ time per update in order to rematch vertices of U (as each such vertex may have up to $\Theta(z)$ neighbors in U); equivalently, every time a vertex of U switches its status between matched and unmatched in M^*, we may need to spend up to $\Theta(z)$ time in order to update its outgoing edges in graph H. Additionally, since the adversary may delete r edges over the course of the algorithm, it is possible that, for every matching M_i, $\Omega(r/z)$ vertices of S become unmatched in it. Therefore, in particular, M_1 may contain as many as $\Theta(r/z)$ vertices of S that are currently unmatched. Whenever any vertex v of G becomes unmatched, we need to inspect all vertices of $\hat{S}$, in order to check whether v can be matched to these vertices, which may require $\Theta(r/z)$ time. (Note a vertex of U can be incident to an arbitrarily large number of vertices of A in G and vice versa, so we do not have a more efficient way to check whether a newly unmatched vertex can be rematched to vertices of A). We note that it is not hard to adapt this algorithm to work with a slightly more relaxed definition of the z-subgraph system, where we only require that, for every vertex $v \in S$, $z-k \leq m(v) \leq z$ holds for some parameter $k < z$; in this case, the amortized update time of the algorithm is bounded by $\tilde{O}\left(z + \frac{rk}{z}\right)$. We will use this relaxed definition of the subgraph system later.

Overall, including the time required to compute the z-subgraph system at the beginning of every phase, our algorithm spends time $\tilde{O}\left(n^2 + n^{1+o(1)} \cdot z + r \cdot \left(z + \frac{r}{z}\right)\right)$ per phase, which leads to $\tilde{O}\left(\frac{n^2+n^{1+o(1)} \cdot z}{r} + z + \frac{r}{z}\right)$ amortized update time, as every phase contains r adversarial updates to G. Substituting $z = \left\lfloor n^{2/3} \right\rfloor$ and $r = \left\lfloor n^{4/3} \right\rfloor$, we get $\tilde{O}\left(n^{2/3}\right)$ update time overall.

1.3 Obtaining a Faster Algorithm

Observe that our algorithm for maintaining a maximal matching from a z-subgrpah system has amortized update time $\tilde{O}\left(\frac{r}{z} + z\right)$; at a very high level, additive factor z accounts for rematching vertices of U, since each such vertex may have up to $\Theta(z)$ neighbors that lie in U, while additive factor $\frac{r}{z}$ accounts for the fact that every matching M_i may have up to $\Theta\left(\frac{r}{z}\right)$ vertices of A that are not matched in it. Both these bounds appear to be fundamental bottlenecks for this approach. Therefore, in order to obtain a faster amortized update time, we need to make sure that both the parameters z and r are significantly smaller; in particular, the length r of every phase must be reduced. However, computing the z-subgraph system from scratch at the beginning of every phase requires $\Omega(|E(G)|)$ time, which may be as high as $\Omega(n^2)$, and decreasing the length of each phase means that this computation needs to be repeated more often, which will lead to a higher amortized update time.

In order to overcome this difficulty, we use a recursive approach. We use a parameter r_1 that is sufficiently high, and partition the timeline into *level-1 phases*, each of which spans r_1 adversarial edge deletions (except for possibly the last phase that may be shorter). At the beginning of every level-1 phase, we construct a z_1-subgraph system $\mathcal{S}$ for G exactly as before; in order to ensure that this subgraph system does not sustain too much damage over the course of the phase due to edge deletions, we need to ensure that the parameter z_1 is sufficiently high. However, we do not use this subgraph system directly in order to maintain a maximal matching in G. Instead, we design an algorithm that, given a z_1-subgraph system for G, constructs a new z_2-subgraph system, for any chosen parameter $z_2 < z_1$, in time rougly $O\left(n^{1+o(1)} \cdot z_1\right)$; note that, if the input graph G is sufficiently dense and the parameter z_1 is not too high, this running time may be significantly lower than $\Theta(|E(G)|)$. We then select an appropriate parameter $r_2 < r_1$ and partition every level-1 phase into *level-2 phases*, each of which spans r_2 adversarial edge deletions. Consider now a level-1 phase Φ, and let $\mathcal{S}$ be the z_1-subgraph system computed at the beginning of Phase Φ. Let $\Phi' \subseteq \Phi$ be a level-2 phase that is contained in Φ. We denote by τ and τ' the times when the phases Φ and Φ' started, respectively, and we denote by $G^{(\tau)}$ and $G^{(\tau')}$ the graph G at time τ and τ', respectively. Let E_D denote the set of all edges that the adversary deleted from G during the time interval $[\tau, \tau')$. We do not delete the edges of E_D from the subgraph system $\mathcal{S}$. Instead, we use $\mathcal{S}$ in order to construct a subset $E'_D \subseteq E_D$ of at most $\frac{z_2}{z_1} \cdot |E_D|$ edges, and a z_2-subgraph system $\mathcal{S}'$ for the graph $G' = G^{(\tau)} \setminus (E_D \setminus E'_D)$ (note that, equivalently, G' is the graph that is obtained from the current graph $G^{(\tau')}$ by inserting the edges of E'_D into it). Then we apply the algorithm for maintaining a maximal matching in G', given a z_2-subgraph system $\mathcal{S}'$ for it. This algorithm first processes the deletions of the edges of E'_D from G' as adversarial updates to ensure that $G' = G^{(\tau')}$, before processing the edge deletions from G that the adversary performs during the level-2 phase Φ'. Since our algorithm for constructing a z_2-subgraph system from a z_1-subgraph system is more efficient than the $\tilde{O}(|E(G)|)$-time algorithm that constructs a subgraph system for G from scratch (if G is sufficiently dense), we can afford to execute it more often. This allows us to ensure that the length r_2 of level-2 phases is shorter, and the corresponding parameter z_2 is smaller, leading to a faster running time of the algorithm that maintains a maximal matching given a subgraph system.

We now briefly describe our algorithm for constructing a z_2-subgraph system from a z_1-subgraph system, in order to motivate the multi-level subgraph system that our algorithm uses. Recall that we are given as input a graph G (which we previously denoted by G'), a z_1-subgraph system $\mathcal{S} = (S, B, U, M, \Lambda, L)$ for G, and a subset $E_D \subseteq E(G)$ of edges (these are the edges that were already deleted from G by the adversary but we do not implement these edge deletions yet, that is, $E_D \subseteq E(G)$ currently holds). Our goal is to construct a subset $E'_D \subseteq E_D$ of at most $\frac{z_2}{z_1} \cdot |E_D|$ edges, and a z_2-subgraph system $\mathcal{S}' = (S', B', U', M', \Lambda', L')$ for the graph $\tilde{G} = G \setminus (E_D \setminus E'_D)$. In order to do so, we use the almost linear-time deterministic algorithm of [ABB+26] for edge coloring in order to compute, in time $O\left(n^{1+o(1)} \cdot z_1\right)$, a partition $(M_1, \ldots, M_{z_1+1})$ of the set M of edges from the subgraph system $\mathcal{S}$ into $z_1 + 1$ disjoint

matchings. For all $1 \leq i \leq z_1+1$, we denote by $w_i = |M_i \cap E_D|$, and we reindex the matchings so that $w_1 \leq w_2 \leq \cdots \leq w_{z_1+1}$ holds. We then let $M' = \bigcup_{i=1}^{z_2} M_i$ and $E'_D = E_D \cap M'$. It is easy to verify that $|E'_D| \leq \frac{z_2}{z_1} \cdot |E_D|$, as required. Let $\tilde{G} = G \setminus (E_D \setminus E'_D)$, and let $\mathcal{S}' = (S', B', U', M', \Lambda', L')$ be a subgraph system for $\tilde{G}$, where M' is the set of edges we have just constructed, $S' = S$, $B' = B$, and $U' = U$; for now we ignore the data structures Λ' and L' but we will revisit them later. For every vertex $v \in V(\tilde{G})$, let $m'(v)$ denote the number of edges of M' incident to v. Recall that, from the properties of the z_1-subgraph system $\mathcal{S}$, for every vertex $v \in S$, $m(v) = z_1$ held, so there is at most one matching M_i that does not contain an edge incident to v. It is then easy to verify that, for every vertex $v \in S'$, $z_2 - 1 \leq m'(v) \leq z_2$ must hold, and for every vertex $v \in V(\tilde{G})$, $m'(v) \leq z_2$. The subgraph system $\mathcal{S}'$ therefore *almost* has all required properties, except for the following: first, it is required that, for every vertex $u \in U$, $|N_{\tilde{G}}(u) \cap U| \leq z_2$ holds, but we are only guaranteed that $|N_{\tilde{G}}(u) \cap U| \leq z_1$ from the properties of the z_1-subgraph system $\mathcal{S}$. Additionally, it is required that, for every vertex $u \in U$, $|N_{\tilde{G}}(u) \cap B| \leq z_2$ holds, but we are only guaranteed that $|N_{\tilde{G}}(u) \cap B| \leq 2z_1$ from the properties of the z_1-subgraph system $\mathcal{S}$. In our next step, we "fix" this problem by processing every vertex $u \in U$ one by one. Recall that, for each such vertex u, we are given the list $\Lambda(u) = N_G(u) \cap (B \cup U)$ of vertices as part of the z_1-subgraph system $\mathcal{S}$. If $\Lambda(u)$ contains at least z_2 vertices of $B' \cup U'$, we insert into M' additional edges connecting u to vertices of $U' \cup B'$, so that $m'(u) = z_2$ holds, and then move u from U' to S', where it joins the vertex set B'. As before, if we insert into M' an edge (u, v) with $v \in B'$, then we may need to delete another edge connecting v to a vertex of U' from M' to ensure that $m'(v) \leq z_2$ continues to hold. As before, every vertex of B' that loses all edges of M' connecting it to vertices of U' is "promoted" to set A'. It is not difficult to implement this step to run in time $\tilde{O}(nz_1)$, by exploiting the lists $\Lambda(u)$ for vertices $u \in U$ that are given as part of the subgraph system $\mathcal{S}$.

Generally, this approach allows us to compute the desired z_2-subgraph system $\mathcal{S}'$ for the graph $\tilde{G}$, except for the following critical issue: at the end of this procedure we need to construct, for every vertex $v \in A'$, a list $L'(v) = N_{\tilde{G}}(v) \cap U'$. In order to do so, we can exploit the data structures $L \cup \Lambda$ that are given as part of the z_1-subgraph system $\mathcal{S}$. The problem is that, for every vertex $a \in A$, the initial list $L(v)$ may be arbitrarily long, and it is possible that every vertex $u \in U$ has a large number of neighbors in $\tilde{G}$ that lie in A. Therefore, if u is promoted from U' to S', we may need to update a large number of the lists $L'(v)$, which we cannot afford. At the same time, these lists are crucially used by our algorithm for maintaining a maximal matching in G, so it is important to maintain them correctly.

In order to overcome this difficulty, we slightly change the definition of the subgraph system, to a *multi-level subgraph system*. The new definition allows us to avoid the difficulty outined above, so we do not need to update the lists $L(v)$ for vertices that lie in the initial set A. We then slightly modify the algorithm for maintaining a maximal matching in G, so that it can work with this modified definition of the subgraph system.

We now provide the description of the 2-level subgraph system. In order to provide intuition and motivation for its construction, consider the initial z_1-subgraph system $\mathcal{S} = (S, B, U, M, \Lambda, L)$, and the (incomplete) new z_2-subgraph sysetm $\mathcal{S}' = (S', B', U', M')$ that we constructed above (recall that we did not construct the required data structures Λ' and L').

The 2-level z-subgraph system for a graph G is defined very similarly to the basic z-subgraph system: it consists of a partition (A, B, U) of $V(G)$, where we denote by $S = A \cup B$, and a set M of edges of G. As before, for every vertex $v \in V(G)$, we denote by $m(v)$ the number of edges of M incident to v. We require that, for every vertex $v \in S$, $z-1 \leq m(v) \leq z$ holds, and for every vertex $u \in U$, the following three properties hold: $m(u) \leq z$; $|N_G(u) \cap U| \leq z$; and $|N_G(u) \cap B| \leq 2z$. For every vertex $u \in U$, the subgraph system also needs to contain the list $\Lambda(u) = N_G(u) \cap (B \cup U)$ of vertices, whose length must be $O(z)$. So far, the definition of the 2-level z-subgraph system is essentially identical to that of the basic z-subgraph system. The main new ingredient is that the 2-level subgraph system must contain a partition (A_1, A_2) of A (for intuition, we can think of A_1 as the set A of vertices in the initial subgraph system $\mathcal{S}$, and of A_2 as the set $A' \setminus A$ of all vertices that were added to A when constructing $\mathcal{S}'$). Additionally, the subgraph system must contain a subset $N_1 \subseteq A_2 \cup B$ of vertices, such that, for every vertex $v \in A_1$, for every edge $(v, v') \in M$ that is incident to v, $v' \in A_1 \cup N_1$ must hold. Intuitively, N_1 corresponds to the set B of vertices in the initial subgraph system $\mathcal{S}$. We then denote by $R_1 = V(G) \setminus (A_1 \cup N_1)$, and require that, for every vertex $v \in A_1$, the list $L(v)$ contains all vertices in $N_G(v) \cap R_1$. Intuitively, R_1 corresponds to the initial set U of vertices in the subgaph system $\mathcal{S}$, and this definition is designed so that that the lists $L(v)$ for vertices $v \in A_1$ do not need to undergo major updates when constructing the z_2-subgraph system $\mathcal{S}'$ from the z_1-subgraph system $\mathcal{S}$. For every vertex $v \in A_2$, we require, as before, that, for every edge $(v, v') \in M'$ incident to v, $v' \in S$ holds, and that $L(v)$ contains all vertices of $N_G(v) \cap U$. For consistency of notation, we denote $N_2 = B$ and $R_2 = U$. This definition ensures that the algorithm we outined above can indeed be used in order to construct a z_2-subgraph system from a z_1-subgraph system in time $O(n^{1+o(1)} \cdot z_1)$, though the resulting z_2-subgraph system will be a 2-level system. We then adapt our algorithm for maintaining a maximal matching from a subgraph system, so that it works with a 2-level subgraph system.

The latter algorithm remains largely unchanged, except that Invariant I1 now implies that, in addition to Invariant I2, the following invariant also holds:

(I3) Throughout the phase, the number of vertices of A_1 that may be matched by M^* to vertices of R_1 is bounded by $2\tau = \frac{64r}{z}$.

The remainder of the algorithm remains unchanged, except in how it handles rematching the vertices of A. The idea here is that it is easy to rematch vertices of $B \cup U$ using the same algorithm as before. Whenever we need to rematch a vertex $a \in A_2$, we also employ the same algorithm as before, that may delete an edge that unmatches a vertex of $B \cup U$, which is then rematched as before. However, rematching vertices of A_1 is done slightly differently. In order to rematch a vertex $a \in A_1$, we scan the first $2\tau + 1$ vertices

of the list $L(a) = N_G(a) \cap R_1$, until we encounter the first vertex v that is not matched to a vertex of A_1. From Invariant I3, some vertex v from among the first $2\tau + 1$ vertices of $L(a)$ must have this property. We then insert the edge (a, v) into M^*, and, if another edge (v, v') incident to v currently lies in M^*, we delete it from M^* (and from M_1 if it lies there). Notice that, from the choice of v, vertex v may not lie in A_1, so we can rematch it using one of the procedures outlined above.

In order to optimize the final running time of the algorithm, we use a natural extension of the above 2-level construction to an arbitrary number k of levels. Our final algorithm employs $k = O(\log n)$ levels, and parameters $z_1 > z_2 > \cdots > z_k$, where $z_1 = n$, z_k is close to $\sqrt{n}$, and for all $1 < i \leq k$, $z_i = z_{i-1}/2$. As before, we partition the update sequence into a number of level-1 phases of an appropriate length, and, at the beginning of every level-1 phase, we construct a basic z_1-subgraph system using the $\tilde{O}(m + n)$ time algorithm outlined above. Generally, for all $1 < i \leq k$, every level-$(i-1)$ phase is partitioned into two level-i phases of roughly identical length, and, at the beginning of every level-i phase, we construct an i-level z_i-subgraph system using the $(i-1)$-level z_{i-1} subgraph system constructed at the beginning of the current level-$(i-1)$ phase. This way we obtain, at the beginning of every level-k phase, a k-level z_k-subgraph system. Our recursive construction ensures that z_k is close to $\sqrt{n}$, and that every k-level phase spans roughly n adversarial updates. We then employ the algorithm we outlined above in order to maintain a maximal matching in G over the course of a level-k phase, using the k-level z_k-subgraph system for the graph G at the beginning of that phase.

2 Preliminaries

All logarithms in this paper are to the base of 2. Given an integer $k \geq 1$, we denote by $[k] = \{1, \ldots, k\}$.

By default, all graphs considered in this paper are undirected, unless stated otherwise. Given a graph $G = (V, E)$, for every vertex $v \in V$, we denote by $\delta_G(v)$ the set of all edges of E that are incident to v, by $N_G(v) = \{u \in V \mid (u, v) \in E\}$ the set of neighbors of v in G, and by $\deg_G(v) = |\delta_G(v)| = |N_G(v)|$ the degree of v in G.

Given a graph $G = (V, E)$, a *matching* in G is a subset $M \subseteq E$ of edges, such that, for every vertex $v \in V$, at most one edge in M is incident to v. We say that a vertex v is *matched* by M, if M contains an edge incident to v, and we say that it is *not matched*, or *unmatched* by M otherwise. A matching M in G is *maximal* if, for every edge $e \in E \setminus M$, the edge set $M \cup \{e\}$ is not a valid matching; equivalently, for every pair $u, v \in V$ of vertices that are not matched by M, $(u, v) \notin E$ must hold.

Throughout the paper, all lists are implemented as binary search trees to support $O(\log n)$ time insertion, deletion and lookup. All dynamic graphs are stored in adjacency list form, with each vertex maintaining its neighbors using a binary search tree.

Fully dynamic graphs. A fully dynamic graph is a graph G that undergoes an online sequence of adversarial updates, where every update either deletes a single edge from G or inserts an edge into G. We usually denote by $G^{(0)}$ the *initial graph* G, before any updates, and, for an integer $\tau > 0$, we denote by $G^{(\tau)}$ the graph G at time τ, that is, after the first τ adversarial updates are applied to it.

Edge coloring. Given a graph G, an *edge coloring of G with k colors* is an assignment of a color $c(e) \in \{1, \ldots, k\}$ to every edge $e \in E(G)$, such that, for every vertex $v \in V(G)$, all its incident edges are assigned distinct colors. Equivalently, for all $1 \leq i \leq k$, the set $\{e \in E(G) \mid c(e) = i\}$ of edges must define a matching.

Vizing's theorem [Viz64] states that any simple n-vertex and m-edge graph with maximum degree Δ can be edge-colored using at most $\Delta + 1$ colors, and the algorithmic proof yields a deterministic $O(mn)$-time algorithm for finding such a coloring. Subsequent work has focused on improving this running time, culminating in a randomized $O(m \log \Delta)$-time algorithm [ABB+25] and a very recent deterministic $O(m^{1+o(1)})$-time algorithm [ABB+26]. We use the latter algorithm, summarized in the following theorem.

Theorem 2.1 (Theorem 1.1 of [ABB+26]). *There is a deterministic algorithm, that, given a simple n-vertex and m-edge graph G with maximum vertex degree Δ, computes an edge coloring of G with $(\Delta + 1)$ colors. The running time of the algorithm is* $O\left(m \cdot 2^{O(\sqrt{\log \Delta})} \cdot \log n\right) \leq O\left(m \cdot n^{o(1)}\right)$.

We remark that the amortized update time of our algorithm for dynamic maximal matching can be improved from $O(n^{1/2+o(1)})$ to $\tilde{O}(n^{1/2})$ (against an adaptive adversary) by using the randomized $(\Delta + 1)$-edge coloring algorithm of [ABB+25] instead.

3 Main Algorithm: Proof of Theorem 1.1

In this section we provide our main result – the proof of Theorem 1.1. We start by describing the main new technical tool that we use – the multi-level subgraph system, and the two key subroutines used in our algorithm. The first subroutine allows us to efficiently compute a multi-level subgraph system with appropriate parameters. The second subroutine allows us to maintain a maximal matching in a fully dynamic graph G, over a limited number of updates, given such a system. We only state the theorems summarizing these two subroutines in this section, and we defer their proofs to the full version of the paper. We then complete the proof of Theorem 1.1 using these tools.

3.1 A Multi-Level Subgraph System

The main new tool that we use is the *multi-level subgraph system*, that we define next.

Definition 3.1 (Multi-Level Subgraph System). *Let G be an n-vertex graph, and let $k > 0$ and $1 \leq z \leq n$ be integral parameters. A k-level z-subgraph system for G consists of the following ingredients:*

- *A subset $M \subseteq E(G)$ of edges. For every vertex $v \in V(G)$, we denote by $m(v)$ the number of edges of M that are incident to v, and we require that the following property holds:*

(P1) *For every vertex $v \in V(G)$, $m(v) \leq z$;*

- *A partition (S, U) of $V(G)$, for which the following properties hold:*

(P2) *No edge of M has both endpoints in U;*

(P3) *For every vertex $v \in S$, $m(v) \geq z - k + 1$; and*

(P4) *For every vertex $u \in U$, $|N_G(u) \cap U| \leq z$.*

- *A partition $(A_1, \ldots, A_k, B)$ of S that satisfies the following property:*

(P5) *For every vertex $u \in U$, $|N_G(u) \cap B| \leq 2z$.*

- *For every vertex $u \in U$, a list $\Lambda(u) = N_G(u) \cap (B \cup U)$, whose length must be $O(z)$;*

  For all $1 \leq i \leq k$, we denote by $A^{\leq i} = A_1 \cup \cdots \cup A_i$ and by $A^{\geq i} = A_i \cup \cdots \cup A_k$. For consistency, we also denote $A^{\leq 0} = A^{\geq k+1} = \emptyset$.

- *For all $1 \leq i \leq k$, a vertex set $N_i \subseteq A^{\geq i+1} \cup B$, and the vertex set $R_i = (A^{\geq i+1} \cup B \cup U) \setminus N_i$ such that the following properties hold:*

(P6) *For every vertex $v \in A_i$, for every edge $(u, v) \in M$ incident to v, $u \in N_i \cup A^{\leq i}$; and*

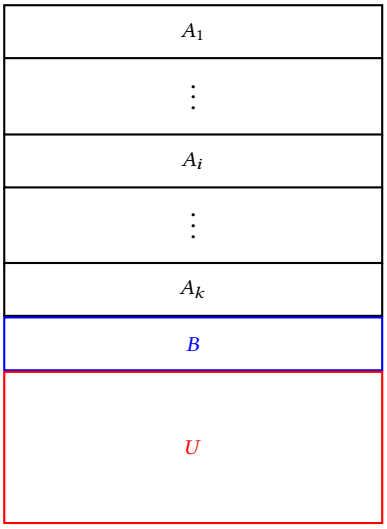
(P7) $U = R_k \subseteq R_{k-1} \subseteq \cdots \subseteq R_1$.

- *For all $1 \leq i \leq k$, for every vertex $v \in A_i$, the list $L(v) = N_G(v) \cap R_i$.*

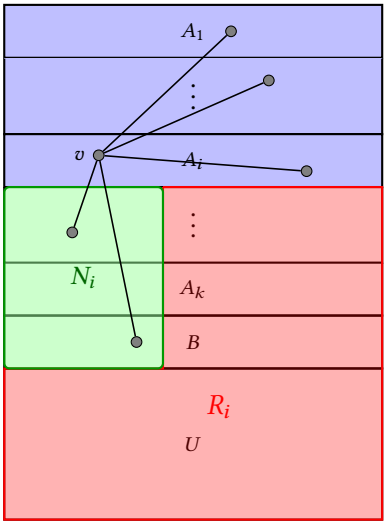
We denote the k-level z-subgraph system by $\mathcal{S} = \left(S, B, U, M, \{A_i, N_i, R_i\}_{i=1}^{k}, \Lambda, L\right)$, where $\Lambda = \{\Lambda(u)\}_{u \in U}$ and $L = \{L(v)\}_{v \in A}$.

Note that, from Property P7, $R_k = U$ must hold, and so $N_k = B$, and $A^{\leq k} \cup B = S$ must hold. Requirement P6 for the vertices of A_k is then equivalent to the requirement that, for every vertex $v \in A_k$, for every edge $(u, v) \in M$ incident to v, $u \in S$ must hold.

For intuition for the definition of the multi-level subgraph system, consider the sets $A_1, \ldots, A_k, B, U$ as being "stacked" on top of each other (see Figure 1a), so for $1 \leq i \leq k$, we view the sets $A_1, \ldots, A_{i-1}$ as lying "above" A_i, and $A_{i+1} \cdots, A_k, B, U$ as lying "below" it. The set of all vertices that lie below A_i is partitioned into two subsets N_i and R_i, with $U \subseteq R_i$. The requirement is that, if an edge $(u, v) \in M$ connects a vertex $u \in A_i$ to a vertex v that lies below A_i, then $v \in N_i$ must hold; so in particular $v \notin U$ (see Figure 1b). Moreover, from Property P7, M may not contain edges connecting vertices that lie above A_i to vertices of R_i. On the other hand, every vertex $v \in A_i$ must store a list $L(v) = N_G(v) \cap R_i$; these lists may be arbitrarily long. Lastly, the set B and U must have the property that, for every vertex $u \in U$, the number of its neighbors in G that lie in $B \cup U$ is small (at most $O(z)$), and all such neighbors must be stored in list $\Lambda(u)$; note that u may have arbitrarily many neighbors in $S \setminus B$. We note that we could have defined, for all $1 \leq i \leq k$, $N_i = A^{\geq i+1} \cup B$ and $R_i = U$. The main problem with this definition is that, since the lists $L(v)$ for $v \in S \setminus U$ may be arbitrarily long, it is challenging to construct them efficiently enough, when building a k-level subgraph system from a $(k-1)$-level subgraph system. With the current definition, we can allow the sets N_i and the lists



(a) Stacking the sets $A_1, \ldots, A_k, B$ and U. Every vertex $u \in U$ satisfies $|N_G(u) \cap (U \cup B)| \leq O(z)$.



(b) N_i is the green region and R_i is the red region. For every edge $e = (u, v) \in M$ with $v \in A_i$, its other endpoint u must lie in the green or the blue region. No edges of M may connect vertices in the blue region to vertices in the red region.

Figure 1: Illustration of the subgraph system

$L(v)$ for $v \in A_i$, for all $1 \leq i \leq k-1$, to remain unchanged when constructing a k-level subgraph system from a $(k-1)$-level one, resulting in a much more efficient algorithm.

3.2 From Subgraph System to Fully Dynamic Maximal Matching

A central ingredient of our approach is an algorithm that, given a fully dynamic graph G and a k-level z-subgraph system $\mathcal{S}$ for an initial graph G (that is **not** assumed to be empty), maintains a maximal matching in G, as it undergoes an online sequence of (a

limited number) of edge insertions and deletions. The algorithm is summarized in the following theorem, whose proof appears in full version.

Theorem 3.2. *There is a deterministic algorithm, whose input consists of an initial n-vertex graph G given in the adjacency-list representation, together with integral parameters $0 < k \leq \log n$ and $1 \leq z \leq n$, and a k-level z-subgraph system $\mathcal{S}$ for G. The algorithm maintains a maximal matching in G, as it undergoes an online sequence of at most n edge insertions and deletions, in total time $O\left(n^{1+o(1)} \cdot \left(z + \frac{n}{z}\right)\right)$.*

3.3 Algorithms for Constructing a Subgraph System

We start with the following theorem, that provides an efficient algorithm for computing a 1-level z-subgraph system for a graph G, for any given integral parameter $1 \leq z \leq n$.

Theorem 3.3. *There is a deterministic algorithm, that, given an n-vertex and m-edge graph G and a parameter $z \in [n]$, constructs a 1-level z-subgraph system $\mathcal{S} = (S, B, U, M, A_1, N_1, R_1, \Lambda, L)$ for G, in time $\tilde{O}(n + m)$.*

Note that the running time of the algorithm from Theorem 3.3 may be as high as $\Omega(n^2)$, if the input graph G is dense. Therefore, we cannot afford to recompute the subgraph system too often. For example, if our goal is to obtain an algorithm for maximal matching whose amortized update time is $n^{1/2+o(1)}$, then we can only afford to recompute it every time the adversary makes roughly $n^{3/2}$ edge updates to the graph G. On the other hand, the running time of the algorithm from Theorem 3.2, that maintains a maximal matching in G given an initial subgraph system, is $O\left(n^{1+o(1)}\left(z + \frac{n}{z}\right)\right)$ for n updates; therefore, if our goal is to achieve amortized update time $n^{1/2+o(1)}$, we should choose z to be roughly $\sqrt{n}$. Lastly, if the parameter z is not too large, then the initial subgraph system may become significantly damaged after a relatively small number of edge deletions by the adversary, so, for example, after $\omega(n)$ edge updates to G, the subgraph system computed for the initial graph G may no longer be possible to reuse by the algorithm from Theorem 3.2 in order to continue and maintain a maximal matching in G.

In order to overcome these difficulties, we select a parameter $k = \Theta(\log n)$, and a sequence $z_1 \geq z_2 \geq \cdots \geq z_k$ of integral parameters, where z_1 is close to n and z_k is close to $\sqrt{n}$, while $\frac{z_{i-1}}{z_i} = 2$ for all $1 < i \leq k$. Our algorithm will construct a 1-level z_1-subgraph system every once in awhile (after a sufficiently large number of edge updates by the adversary). We will not use this subgraph system in order to maintain a maximal matching in G directly; instead, we will use it in order to compute a 2-level z_2-subgraph system. This subgraph system, in turn, will be used in order to compute a 3-level z_3-subgraph system and so on.

The key tool that we use is an algorithm that, given an h-level z-subgraph system for a graph G, efficiently constructs an $(h+1)$-level z'-subgraph system for G, for a given parameter $z' < z$. The main idea is that the h-level z-subgraph system can be exploited in order to speed up the construction of the z'-subgraph system (but unfortunately this speedup comes at the cost of increasing the number of levels in the subgraph system); the running time of the algorithm is bounded by $O\left(n^{1+o(1)} \cdot z\right)$, which may be significantly smaller than $|E(G)|$ if G is sufficiently dense and the parameter z is sufficiently small. This speedup allows us to compute the subgraph systems for higher levels i more often, and this is what allows us to set the parameters z_i to be progressively smaller (recall that the smaller the parameter z of the subgraph system, the faster the adversary may destroy the subgraph system via edge deletions, so a subgraph system with a lower parameter z needs to be recomputed more often; therefore, we need to ensure that its computation is more efficient).

For technical reasons, in our algorithm that computes an $(h+1)$-level z'-subgraph system from an h-level z-subgraph system, we assume that we are given, as part of input, a subset $E_D \subseteq E(G)$ of "deleted" edges, and we would like to ensure that the new $(h+1)$-level subgraph system only contains a limited number of such edges. Intuitively, these are edges that were deleted by the adversary since the h-level subgraph system was constructed, but we temporarily keep them in G because their removal may have a large impact on the h-level subgraph system. Instead, we will ensure that the final k-level subgraph system only contains few such edges. We also assume that we are given a set E_I of edges that do not lie in the input graph; these are edges that the adversary has inserted into G since the h-level subgraph system was constructed but they were not yet incorporated into the h-level subgraph system (so for example there may be many such edges connecting a vertex $u \in U$ to other vertices in U). We require that these edges are incorporated into the new $(h+1)$-level subgraph system. The following theorem, whose proof appears in the full version, provides an algorithm for constructing an $(h+1)$-level z'-subgraph system from an h-level z-subgraph system.

Theorem 3.4. *There is a deterministic algorithm, whose input is:*

- *an n-vertex graph G given in the adjacency-list representation;*
- *an integral parameter $h > 0$ and a parameter $1 \leq z \leq n$ that is an integral power of 2;*
- *an h-level z-subgraph system $\mathcal{S} = \left(S, B, U, M, \{A_i, N_i, R_i\}_{i=1}^{h}, \Lambda, L\right)$ for G;*
- *a subset $E_D \subseteq E(G)$ of edges;*
- *a set $E_I \subseteq V(G) \times V(G)$ of edges that do not lie in G; and*
- *a parameter $1 \leq z' < z$ that is an integral power of 2.*

The algorithm constructs a subset $E'_D \subseteq E_D$ of at most $\frac{|E_D| \cdot z'}{z}$ edges and an $(h+1)$-level z'-subgraph system $\mathcal{S}' = \left(S', B', U', M', \{A'_i, N'_i, R'_i\}_{i=1}^{h+1}, \Lambda', L'\right)$ for the graph $G' = (G \cup E_I) \setminus (E_D \setminus E'_D)$. The running time of the algorithm is $\tilde{O}(n^{1+o(1)} \cdot z + |E_D| + |E_I|)$. The algorithm may modify the input adjacency-list representation of G and the input representation of the h-level z-subgraph system $\mathcal{S}$ for G.

We note that generally, the size of the $(h+1)$-level z'-subgraph system $\mathcal{S}'$ that the algorithm from Theorem 3.4 constructs may be quite large (specifically, the lists in $L'(v)$ for $v \in A'$ may be very long), and so the algorithm may not be able to output it explicitly within its required running time. Instead, it modifies the input h-level z-subgraph system $\mathcal{S}$ in order to transform it into the $(h+1)$-level z'-subgraph system $\mathcal{S}'$ for G'.

3.4 Completing the Proof of Theorem 1.1

We are now ready to complete the proof of Theorem 1.1. We denote by r the total number of adversarial updates to the input graph G, that may not be known to the algorithm in advance. It will be convenient for us to assume that $n = |V(G)|$ is an integral power of 2. If this is not the case, then we add dummy vertices into G until $n = |V(G)|$ becomes an integral power of 2. Note that this may only increase n by at most factor 2. Our algorithm works in phases. We partition the time horizon into phases $\Phi_0, \Phi_1, \Phi_2, \ldots$ as follows. The first phase, Φ_0, starts immediately after the first edge is inserted into the initially empty graph G, and lasts for $r_0 = \min\{n, r\}$ updates, after which Phase Φ_1 starts. For all $i \geq 1$, we denote by m_i the number of edges in graph G at the beginning of Phase Φ_i. If $m_i \leq n^{3/2}$, then the ith phase lasts for exactly $r_i = n$ updates. In this case, we say that Phase Φ_i is of *type 1*. Otherwise, the ith phase lasts for $r_i = m_i$ updates, and we say that it is of *type 2*. For all $i \geq 0$, if the update sequence terminates before r_i updates are completed since the beginning of Phase Φ_i, it becomes the last phase of the algorithm.

We present three algorithms: one for Phase Φ_0 in Lemma 3.5, one for type-1 phases in Theorem 3.6, and one for type-2 phases in Theorem 3.7; their proofs are deferred to the full version of the paper. The algorithm for Phase Φ_0 (Lemma 3.5) is quite simple and is from first principles; it is mainly needed for the special case where the update sequence is short. Since type-1 phases are relatively short, the algorithm from Theorem 3.6 only uses a 1-level z-subgraph system, with $z \approx \sqrt{n}$, that is computed once at the beginning of the phase; after that, it uses the algorithm from Theorem 3.2 in order to maintain a maximal matching from the subgraph system. Lastly, the algorithm from Theorem 3.7 for type-2 phases needs to accommodate a much larger number of updates to G. In order to do so, it uses the multi-level subgraph system combined with the algorithm from Theorem 3.2 for maintaining a maximal matching from the subgraph system.

The following simple lemma provides the implementation of Phase Φ_0.

Lemma 3.5. *There is a deterministic algorithm, that, given as input an n-vertex graph G that initially contains no edges, and undergoes an online sequence of $r > 0$ edge deletions and insertions, together with an integral parameter $t > 0$, maintains a maximal matching in G. The running time of the algorithm is $\tilde{O}\left(r\left(t + \frac{r}{t}\right)\right)$.*

We will employ the algorithm from Lemma 3.5 for Phase Φ_0 with parameter $t = n^{1/2}$. The algorithm for implementing type-1 phases is summarized in the following theorem.

Theorem 3.6. *There is a deterministic algorithm, that, given an n-vertex and m-edge graph G with $m \leq n^{3/2}$, that undergoes an online sequence of $r \leq n$ edge insertions and deletions, maintains a maximal matching in G with total running time $O\left(n^{3/2+o(1)}\right)$.*

The following theorem summarizes our algorithm for implementing type-2 phases.

Theorem 3.7. *There is a deterministic algorithm, that, given an n-vertex and m-edge graph G with $m > n^{3/2}$ and n an integral power of 2, that undergoes an online sequence of $r \leq m$ edge insertions and deletions, maintains a maximal matching in G with total running time $O\left(m \cdot n^{1/2+o(1)}\right)$.*

We are now ready to complete the proof of Theorem 1.1. We start by employing the algorithm from Lemma 3.5 with parameter $t = \lceil \sqrt{n} \rceil$ over the course of Phase Φ_0. Recall that the running time of the algorithm from Lemma 3.5 is bounded by:

$$\tilde{O}\left(r_0 \cdot \left(t + \frac{r_0}{t}\right)\right) \leq \tilde{O}\left(r_0 \cdot \left(n^{1/2} + \frac{n}{n^{1/2}}\right)\right) \leq \tilde{O}\left(r_0 \cdot n^{1/2}\right).$$

If Φ_0 is the only phase (that is, the number of adversarial updates is $r < n$), then the amortized update time of the algorithm is bounded by $\tilde{O}(n^{1/2})$, as required. Assume now that $r > n$. Let q denote the index of the last phase. For all $1 \leq i \leq q$, let τ_i be the time when Phase Φ_i starts, and recall that m_i is the number of edges in graph G at time τ_i. Recall also that r_i is the number of updates that G undergoes during Phase Φ_i. If $m_i \leq n^{3/2}$ (that is, Φ_i is a type-1 phase), then we use the algorithm from Theorem 3.6 with the initial graph $G^{(\tau_i)}$, and the updates that G undergoes during Phase Φ_i, to maintain a maximal matching in G during the phase Φ_i. Recall that, if $i < q$, then $r_i = n$, and the running time of the algorithm from Theorem 3.6 is $O\left(n^{3/2+o(1)}\right) \leq O\left(r_i \cdot n^{1/2+o(1)}\right)$. If $m_i > n^{3/2}$ (so Φ_i is a type-2 phase), then we use the algorithm from Theorem 3.7 with the initial graph $G^{(\tau_i)}$, and the updates that G undergoes during Phase Φ_i, to maintain a maximal matching in G during the phase Φ_i. Recall that, if $i < q$, then $r_i = m_i$, and the running time of the algorithm from Theorem 3.7 is $O\left(m_i \cdot n^{1/2+o(1)}\right) \leq O\left(r_i \cdot n^{1/2+o(1)}\right)$. If the last phase is of type 1, then its running time is $O\left(n^{3/2+o(1)}\right) \leq O\left(r \cdot n^{1/2+o(1)}\right)$, since we assumed that $r \geq n$. Otherwise, the running time of the last phase is $O\left(m_q \cdot n^{1/2+o(1)}\right) \leq O\left(r \cdot n^{1/2+o(1)}\right)$, since $r \geq m_q$ must hold, as the initial graph G contained no edges. Overall, the total running time of the algorithm is bounded by:

$$O\left(r \cdot n^{1/2+o(1)}\right) + \sum_{i=0}^{q-1} O\left(r_i \cdot n^{1/2+o(1)}\right) \leq O\left(r \cdot n^{1/2+o(1)}\right).$$

References

[ABB+25] Sepehr Assadi, Soheil Behnezhad, Sayan Bhattacharya, Martín Costa, Shay Solomon, and Tianyi Zhang. Vizing's theorem in near-linear time. In *STOC 2025: Proceedings of the 57th Annual ACM Symposium on Theory of Computing*, pages 24–35, 2025.

[ABB+26] Sepehr Assadi, Soheil Behnezhad, Sayan Bhattacharya, Martín Costa, Shay Solomon, and Tianyi Zhang. Vizing's theorem in deterministic almost-linear time. In *Proceedings of the 2026 Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2026*, pages 5558–5585. 2026.

[ABKL23] Sepehr Assadi, Soheil Behnezhad, Sanjeev Khanna, and Huan Li. On regularity lemma and barriers in streaming and dynamic matching. In *Proceedings of the 55th Annual ACM Symposium on Theory of Computing*, pages 131–144, 2023.

[ABR24] Amir Azarmehr, Soheil Behnezhad, and Mohammad Roghani. Fully dynamic matching: (2-$\sqrt{2}$)-approximation in polylog update time. In *Proceedings of the 2024 Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 3040–3061, 2024.

[ACC+18] Moab Arar, Shiri Chechik, Sarel Cohen, Cliff Stein, and David Wajc. Dynamic matching: Reducing integral algorithms to approximately-maximal fractional algorithms. In *45th International Colloquium on Automata, Languages, and Programming, ICALP 2018, Prague, Czech Republic, July 9-13, 2018*, volume 107 of *LIPIcs*, pages 7:1–7:16, 2018.

[AKK25] Sepehr Assadi, Sanjeev Khanna, and Peter Kiss. Improved bounds for fully dynamic matching via ordered ruzsa-szemerédi graphs. In *Proceedings of the 2025 Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 2971–2990, 2025.

[AKL19] Sepehr Assadi, Sanjeev Khanna, and Yang Li. The stochastic matching problem with (very) few queries. *ACM Transactions on Economics and Computation (TEAC)*, 7(3):1–19, 2019.

[BBF+26] Aaron Bernstein, Sayan Bhattacharya, Nick Fischer, Peter Kiss, and Thatchaphol Saranurak. Separations between oblivious and adaptive adversaries for natural dynamic graph problems. In *to appear in Proceedings of the 2026 Annual ACM-SIAM Symposium on Discrete Algorithms*, 2026.

[BBKS25] Aaron Bernstein, Sayan Bhattacharya, Peter Kiss, and Thatchaphol Saranurak. Deterministic dynamic maximal matching in sublinear update time. In *Proceedings of the 57th Annual ACM Symposium on Theory of Computing*, pages 132–143, 2025.

[BCH17] Sayan Bhattacharya, Deeparnab Chakrabarty, and Monika Henzinger. Deterministic fully dynamic approximate vertex cover and fractional matching in o (1) amortized update time. In *International Conference on Integer Programming and Combinatorial Optimization*, pages 86–98. Springer, 2017.

[BDH+19] Soheil Behnezhad, Mahsa Derakhshan, MohammadTaghi Hajiaghayi, Cliff Stein, and Madhu Sudan. Fully dynamic maximal independent set with polylogarithmic update time. In *2019 IEEE 60th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 382–405. IEEE, 2019.

[BDL21] Aaron Bernstein, Aditi Dudeja, and Zachary Langley. A framework for dynamic matching in weighted graphs. In *STOC '21: 53rd Annual ACM SIGACT Symposium on Theory of Computing, Virtual Event, Italy, June 21-25, 2021*, pages 668–681. ACM, 2021.

[Beh23] Soheil Behnezhad. Dynamic algorithms for maximum matching size. In *Proceedings of the 2023 Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 129–162, 2023.

[BFH21] Aaron Bernstein, Sebastian Forster, and Monika Henzinger. A deamortization approach for dynamic spanner and dynamic maximal matching. *ACM Transactions on Algorithms (TALG)*, 17(4):1–51, 2021.

[BG24] Soheil Behnezhad and Alma Ghafari. Fully dynamic matching and ordered ruzsa-szemerédi graphs. In *2024 IEEE 65th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 314–327. IEEE, 2024.

[BGK+22] Sayan Bhattacharya, Fabrizio Grandoni, Janardhan Kulkarni, Quanquan C Liu, and Shay Solomon. Fully dynamic (δ+ 1)-coloring in o (1) update time. *ACM Transactions On Algorithms (TALG)*, 18(2):1–25, 2022.

[BGS11] Surender Baswana, Manoj Gupta, and Sandeep Sen. Fully dynamic maximal matching in O (log n) update time. In *IEEE 52nd Annual Symposium on Foundations of Computer Science, FOCS 2011, Palm Springs, CA, USA, October 22-25, 2011*, pages 383–392. IEEE Computer Society, 2011.

[BGS18] Surender Baswana, Manoj Gupta, and Sandeep Sen. Fully dynamic maximal matching in o(log n) update time (corrected version). *SIAM Journal on Computing*, 47(3):617–650, 2018.

[BHI18] Sayan Bhattacharya, Monika Henzinger, and Giuseppe F. Italiano. Deterministic fully dynamic data structures for vertex cover and matching. *SIAM J. Comput.*, 47(3):859–887, 2018.

[BHN16] Sayan Bhattacharya, Monika Henzinger, and Danupon Nanongkai. New deterministic approximation algorithms for fully dynamic matching. In *Proceedings of the 48th Annual ACM SIGACT Symposium on Theory of Computing, STOC 2016, Cambridge, MA, USA, June 18-21, 2016*, pages 398–411. ACM, 2016.

[BK22] Soheil Behnezhad and Sanjeev Khanna. New trade-offs for fully dynamic matching via hierarchical edcs. In *Proceedings of the 2022 Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 3529–3566, 2022.

[BKS23] Sayan Bhattacharya, Peter Kiss, and Thatchaphol Saranurak. Dynamic $(1+\epsilon)$-approximate matching size in truly sublinear update time. In *2023 IEEE 64th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 1563–1588. IEEE, 2023.

[BKSW24a] Sayan Bhattacharya, Peter Kiss, Thatchaphol Saranurak, and David Wajc. Dynamic matching with better-than-2 approximation in polylogarithmic update time. *Journal of the ACM*, 71(5):1–32, 2024.

[BKSW24b] Sayan Bhattacharya, Peter Kiss, Aaron Sidford, and David Wajc. Near-optimal dynamic rounding of fractional matchings in bipartite graphs. In *Proceedings of the 56th Annual ACM Symposium on Theory of Computing, STOC 2024, Vancouver, BC, Canada, June 24-28, 2024*, pages 59–70. ACM, 2024.

[BRW25] Soheil Behnezhad, Rajmohan Rajaraman, and Omer Wasim. Fully dynamic (δ+ 1)-coloring against adaptive adversaries. In *Proceedings of the 2025 Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 4983–5026, 2025.

[BS15] Aaron Bernstein and Cliff Stein. Fully dynamic matching in bipartite graphs. In *International Colloquium on Automata, Languages, and Programming*, pages 167–179. Springer, 2015.

[BS16] Aaron Bernstein and Cliff Stein. Faster fully dynamic matchings with small approximation ratios. In *Proceedings of the twenty-seventh annual ACM-SIAM symposium on Discrete algorithms*, pages 692–711, 2016.

[CZ19] Shiri Chechik and Tianyi Zhang. Fully dynamic maximal independent set in expected poly-log update time. In *2019 IEEE 60th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 370–381. IEEE, 2019.

[DZ18] Yuhao Du and Hengjie Zhang. Improved algorithms for fully dynamic maximal independent set, 2018.

[FH25] Maxime Flin and Magnús M. Halldórsson. Faster dynamic (Δ+1)-coloring against adaptive adversaries. In *52nd International Colloquium on Automata, Languages, and Programming, ICALP 2025, Aarhus, Denmark, July 8-11, 2025*, volume 334 of *LIPIcs*, pages 79:1–79:21. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2025.

[GK18] Manoj Gupta and Shahbaz Khan. Simple dynamic algorithms for maximal independent set and other problems, 2018.

[GP13] Manoj Gupta and Richard Peng. Fully dynamic (1+ e)-approximate matchings. In *2013 IEEE 54th Annual Symposium on Foundations of Computer Science*, pages 548–557. IEEE, 2013.

[GSSU22] Fabrizio Grandoni, Chris Schwiegelshohn, Shay Solomon, and Amitai Uzrad. Maintaining an edcs in general graphs: Simpler, density-sensitive and with worst-case time bounds. In *Symposium on Simplicity in Algorithms (SOSA)*, pages 12–23, 2022.

[HP22] Monika Henzinger and Pan Peng. Constant-time dynamic (δ+ 1)-coloring. *ACM Transactions on Algorithms (TALG)*, 18(2):1–21, 2022.

[IL93] Zoran Ivković and Errol L Lloyd. Fully dynamic maintenance of vertex cover. In *International Workshop on Graph-Theoretic Concepts in Computer Science*, pages 99–111. Springer, 1993.

[Kis22] Peter Kiss. Deterministic dynamic matching in worst-case update time. In *13th Innovations in Theoretical Computer Science Conference, ITCS 2022*, volume 215 of *LIPIcs*, pages 94:1–94:21, 2022.

[Liu24] Yang P Liu. On approximate fully-dynamic matching and online matrix-vector multiplication. In *2024 IEEE 65th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 228–243. IEEE, 2024.

[NS15] Ofer Neiman and Shay Solomon. Simple deterministic algorithms for fully dynamic maximal matching. *ACM Transactions on Algorithms (TALG)*, 12(1):1–15, 2015.

[OR10] Krzysztof Onak and Ronitt Rubinfeld. Maintaining a large matching and a small vertex cover. In *Proceedings of the forty-second ACM symposium on Theory of computing*, pages 457–464, 2010.

[RSW22] Mohammad Roghani, Amin Saberi, and David Wajc. Beating the folklore algorithm for dynamic matching. In *13th Innovations in Theoretical Computer Science Conference, ITCS 2022, Berkeley, CA, USA, January 31 - February 3, 2022*, volume 215 of *LIPIcs*, pages 111:1–111:23, 2022.

[Sol16] Shay Solomon. Fully dynamic maximal matching in constant update time. In *2016 IEEE 57th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 325–334. IEEE, 2016.

[Viz64] Vadim G Vizing. On an estimate of the chromatic class of a p-graph. *Diskret analiz*, 3:25–30, 1964.

[Waj20] David Wajc. Rounding dynamic matchings against an adaptive adversary. In *Proceedings of the 52nd Annual ACM SIGACT Symposium on Theory of Computing, STOC 2020, Chicago, IL, USA, June 22-26, 2020*, pages 194–207, 2020.

Received 2025-11-03; accepted 2026-02-01