Constant-Time Dynamic $(\Delta + 1)$-Coloring

Monika Henzinger
University of Vienna, Faculty of Computer Science, Vienna, Austria
monika.henzinger@univie.ac.at

Pan Peng
Department of Computer Science, University of Sheffield, Sheffield, UK
p.peng@sheffield.ac.uk

Abstract

We give a fully dynamic (Las-Vegas style) algorithm with *constant expected amortized* time per update that maintains a proper $(\Delta + 1)$-vertex coloring of a graph with maximum degree at most Δ. This improves upon the previous $O(\log \Delta)$-time algorithm by Bhattacharya et al. (SODA 2018). Our algorithm uses an approach based on assigning random ranks to vertices and does not need to maintain a hierarchical graph decomposition. We show that our result does not only have optimal running time, but is also optimal in the sense that already deciding whether a Δ-coloring exists in a dynamically changing graph with maximum degree at most Δ takes $\Omega(\log n)$ time per operation.

2012 ACM Subject Classification Theory of computation → Dynamic graph algorithms

Keywords and phrases Dynamic graph algorithms, Graph coloring, Random sampling

Digital Object Identifier 10.4230/LIPIcs.STACS.2020.53

Related Version A full version of the paper is available at `https://arxiv.org/abs/1907.04745`.

Funding The research leading to these results has received funding from the European Research Council under the European Union's Seventh Framework Programme (FP/2007-2013) / ERC Grant Agreement no. 340506.

1 Introduction

A (fully) dynamic graph algorithm is a data structure that provides information about a graph property while the graph is being modified by *edge updates* such as edge insertions or deletions. When designing a dynamic graph algorithm the goal is to minimize the time per update or query operation. The lower bounds of Patrascu and Demaine [24] showed that in the cell-probe model many fundamental graph properties, such as asking whether the graph is connected, require $\Omega(\log n)$ time per operation, where n is the number of nodes in the graph. Their lower bound technique also gives logarithmic time lower bounds for further dynamic problems such as higher types of connectivity, planarity and bipartiteness testing, and minimum spanning forest, and it is an open research question for which other dynamic graph problems non-constant time lower bounds exist.

Furthermore, there are only very few graph problems for which it is known that no such lower bounds can exist. These are the following problems, which all have constant-time, and thus optimal, algorithms: maintaining (a) a maximal matching (randomized) [25], (b) a $(2 + \varepsilon)$-approximate vertex cover (deterministic) [7], and (c) a $(2k - 1)$-stretch spanner of size $O(n^{1+\frac{1}{k}} \log^2 n)$ for *constant* k (randomized) [3]. All these are *amortized* time bounds and each of these algorithms maintains a dynamically-changing sophisticated hierarchical graph decomposition.

In this paper we present a dynamic algorithm with constant update time for a new graph problem, expanding the above list. Additionally, our algorithm does not rely on a dynamically changing hierarchical graph decomposition, making it (but not its analysis) simpler. Our new result is a dynamic algorithm for the following problem: We call a dynamic

graph Δ-*bounded* if throughout the updates, the graph has maximum degree at most Δ. A *proper coloring* assigns to each vertex an integer value, called *color*, such that the endpoints of every edge have a different color. A $(\Delta + 1)$-vertex coloring is a proper coloring that uses only colors from the range $[1, \ldots, \Delta + 1]$. Note that a proper $(\Delta + 1)$-vertex coloring in a (static) graph with maximum degree at most Δ always exists and can be found in linear time by a simple greedy algorithm [27]. A *fully dynamic* graph algorithm is a data structure that maintains a graph $G = (V, E)$ while it is undergoing an arbitrary sequence of the following operations: 1) **Insert**(u, v): insert the edge (u, v) in G; 2) **Delete**(u, v): delete the edge (u, v) from G. In the dynamic $(\Delta + 1)$-vertex coloring problem, the fully dynamic graph algorithm maintains after each update operation a proper $(\Delta + 1)$-vertex coloring of the current graph in a Δ-bounded dynamic graph. When asked to perform a **Query**(u) operation, the algorithm returns the color of the given vertex u.

Maintaining a proper $(\Delta + 1)$-vertex coloring in a Δ-bounded dynamic graph can be done trivially in $O(\Delta)$ worst-case update time: the algorithm does nothing after an edge deletion or an edge insertion between two nodes of different colors; once an edge is inserted between two nodes of the same color it scans the whole neighborhood of one of the nodes and chooses an unused color. Recently Bhattacharya et al. [5] presented a randomized $(\Delta+1)$-vertex coloring algorithm with $O(\log \Delta)$ expected amortized update time and a deterministic algorithm that maintains a $(\Delta + o(\Delta))$-vertex coloring with $O(\text{poly} \log \Delta)$ amortized time. Their randomized algorithm works against the *oblivious adversary*: It is assumed that the sequence of update operations is generated by an adversary whose goal is to maximize the running time, but has to fix the sequence *before* the algorithm starts to run. This guarantees that the adversary is *oblivious* to the random choices of the algorithm. Note that if Δ is polynomial in n, their algorithm takes time $O(\log n)$. In this paper, we improve upon this result as follows.

▶ **Theorem 1.** *There exists a fully dynamic algorithm for maintaining a proper* $(\Delta+1)$*-vertex coloring for a* Δ*-bounded graph against an oblivious adversary with* $O(1)$ *expected amortized update time.*

Unlike the algorithm in [5] our algorithm does not need to maintain a hierarchical graph decomposition. Furthermore, apart from having optimal running time, our result is also optimal in the sense that deciding whether a proper coloring with only Δ colors exists in a dynamically changing graph (with maximum degree at most Δ) takes at least $\Omega(\log n)$ time per operation, as we show in Theorem 2. More precisely, we define the *dynamic* Δ*-colorability testing problem* as follows: Besides operations **Insert**(u, v) and **Delete**(u, v), there is a **Query**() operation that returns *yes* if the graph is Δ-colorable and *no* otherwise, where Δ is the maximum degree in the current graph. We show the following theorem.

▶ **Theorem 2.** *Any data structure for dynamic* Δ*-colorability testing, where* Δ *is the maximum degree in the graph, must perform* $\Omega(\log n)$ *cell probes, where each cell has size* $O(\log n)$*.*

Our Techniques. We first give a brief overview of the algorithm in [5] that maintains a proper $(\Delta + 1)$-vertex coloring for a dynamic graph with maximum degree at most Δ. Let χ be the current proper $\Delta + 1$-coloring. First note that after an edge deletion and after an edge insertion (u, v) that does not cause a conflict, i.e., if $\chi(u) \neq \chi(v)$, then the coloring remains unchanged. If a conflict occurs (i.e., $\chi(u) = \chi(v)$), then one needs to fix the coloring by recoloring one vertex from $\{u, v\}$, say u. Instead of scanning the whole neighborhood of u to find the color (called a *blank* color) that has not been used by any of its neighbors, the algorithm in [5] tries to *sample* a color from a set S that contains only blank colors and colors (called *unique* colors) that have been used by exactly one neighbor of u. Note that S

has size $\Omega(\Delta)$, which guarantees that a future conflict edge incident to u occurs with low probability (i.e., with probability $O(1/\Delta)$). On the other hand, if a unique color is chosen, one needs to recolor the corresponding vertex w (which is a neighbor of u), again, using a new color sampled from the set of blank and unique colors for w. This procedure might cause a cascade and even not terminate at all. The dynamic $(\Delta+1)$-vertex coloring algorithm of [5] resolves this problem by maintaining a hierarchical graph decomposition, and when recoloring a node it picks a color randomly out of all colors that are either (i) used by none of the neighbors or (ii) used by at most one of the neighbors *on a lower level* in the graph hierarchy. The resulting algorithm is then shown to have $O(\log \Delta)$ amortized update time for maintaining a proper coloring. However, maintaining such a hierarchical partition is not only complicated, but also inefficient, as it alone already takes $O(\log \Delta)$ amortized update time.

Now we describe our main ideas which lead to a constant-time dynamic coloring algorithm. We show that an approach based on *assigning random ranks* to vertices outperforms the graph-hierarchy based algorithm: During preprocessing each node v is assigned a random rank $r(v)$ from $[0,1]$ and a random color (assuming as usual that the initial graph is empty). Let L_v denote the set of neighbors of a node v with rank lower than $r(v)$ and for any set S of neighbors of a node let $S^<$ denote the subset of S whose rank is at most the median rank of the nodes in S. When recoloring v, we pick a color randomly out of all colors that are either (i) used by none of its neighbors (called *blank* colors) or (ii) by at most one neighbor in L_v *and* this node belongs to $L_v^<$. (We show that there are always $\Omega(|L_v|)$ many such colors.) In case (ii) this neighbor w must be recolored. Due to the definition of $L_v^<$ it is guaranteed that $r(w)$ is at most the median rank of the lower-ranked neighbors of v. Recoloring w is done with a more refined recoloring procedure that additionally to the above information takes into account which nodes of L_w also belong to $N(v)$, the neighborhood of v. This is necessary since on the one side (a) we need to guarantee that the new color is chosen randomly from a set of $\Omega(|L_w|)$ colors and the other side (b) we have to apply a different analysis depending on whether the new color belongs to $N(v)$ or not.

More formally let $L_{w,\text{new}} := L_w \setminus N(v)$, let $L_{w,\text{old}} := L_w \cap N(v)$, and let L^* equal $L^<_{w,\text{new}}$ if $|L_{w,\text{new}}| > |L_w|/10$ and $L^<_{w,\text{old}}$ otherwise. The algorithm randomly samples a color out of the set which consists of (i) all blank colors and (ii) all colors which are used by exactly one node in L_w and are used by a node in L^*. If the color of a node y in L^* was chosen, y will be recolored recursively taking $N(x)$ for *all* previously visited nodes x into account. If y was chosen from $L^<_{w,\text{new}}$, y is called a *good* vertex, otherwise a *bad* vertex. This results in a recoloring of nodes along a random *recoloring path* P in the graph until a blank color is chosen. The latter is guaranteed to happen when a node y with $L_y = \emptyset$ is reached. We give a data structure that implements each *coloring step*, i.e., the selection of a new color of a vertex y on P, in time $O(|L_y|)$. Thus, the total time for recoloring P is $O(\sum_{y\in P} |L_y|)$.

This sampling routine guarantees that the rank of the next node is at most the median rank of the lower-ranked neighbors of the previous node. If there were no dependencies between the rank of the current node and the previous nodes on P, the *expected rank* would halve in this coloring step. These dependencies are exactly why we introduced $L_{y,\text{new}}$, $L_{y,\text{old}}$, and L^*, and labeled the vertices on P as good and bad. More specifically, we show that at every good vertex y the *expected rank* and the *expected size of* $L_{y,new}$ halves. This by itself would not be sufficient, since we need the expected size of L_y, and not only the expected size of $L_{y,\text{new}}$, to halve. Here we use the definition of L^* to show that the expected size of L_y decreases by *a constant factor* whenever $L_{y,\text{new}}$ halves. This then implies that the total expected time at the good vertices on P, i.e. $O(\sum_{y\in P, y:\text{good}} |L_y|)$, forms a geometric series adding up to $O(r(v)\Delta)$, where v is the initial vertex of P.

The main difficulty that the analysis still has to overcome is the fact that there might be bad vertices. To deal with this we introduce a novel potential function Φ based on the nodes on P, which allows us to bound the work, i.e., the number of ("standard" word) operations that the algorithm performs, done at *bad* vertices by the work done at *good* vertices. More specifically, we show that, when traversing P from an initial vertex v, at every bad vertex Φ drops. As (i) Φ is always non-negative, (ii) Φ only increases at good vertices, and (iii) the drop of Φ gives an upper bound of the time spent at bad vertices, we can *bound the total time for coloring all the vertices on P by the total time spent at the good vertices on P times a constant.* This allows us to prove that the total work done for recoloring all vertices on P is $O(r(v)\Delta)$, where v is the initial vertex of P (Lemma 4).

Finally, we combine this bound with the fact that (a) for many operations (such as all deletions and many insertions) no recoloring is necessary and (b) the color of each node y was picked uniformly at random from a set of $\Omega(|L_y|)$ many colors, to show that the expected amortized time per update operation is constant.

Note that the refined sampling routine as well as the analysis that combines a potential function analysis with a careful analysis of the expected size of the sets L_y along a random path P is novel. The technique has the advantage that, unlike in a hierarchical graph decomposition where the ordering of nodes by levels might change and needs to be updated, the ordering of nodes by ranks is static and does not create update costs. However, it has the disadvantage that, unlike in the hierarchical graph decomposition of [5], (1) we do not have a worst-case upper bound on the number of nodes that are "lower" in the ordering and (2) the length of P, which is limited by the longest strictly decreasing path in the ordering, might be $\Theta(n)$ and not $\Theta(\log \Delta)$ in the worst case, as in [5].

As we recently learnt, Bhattacharya et al. [6] achieved the same result as Theorem 1 independently.

Our proof of Theorem 2 follows from a simple reduction from dynamic connectivity, whose cell probe lower bound was known to be $\Omega(\log n)$ [24].

Other Related Work. Partially due to the $\Omega(\log n)$ lower bound for the fundamental problem of testing connectivity [24], a large amount of previous research on dynamic graph algorithms has focused on algorithms with polylogarithmic or super-polylogarithmic update time. Examples include testing k-edge (or vertex) connectivity (see e.g., [14, 18, 17]), maintaining minimum spanning tree (see e.g., [15, 14, 17, 16, 18, 19, 20, 28, 22, 23]), and graph coloring [2, 1, 5, 26, 13]. There are also studies on *incremental algorithms* that only allow edge insertions, and *decremental algorithms* that only allow edge deletions throughout all the updates. In contrast to such studies, our work is focusing on *fully dynamic* algorithms, in which both edge insertions and deletions are allowed.

The technique of maintaining random ranks for vertices was previously used for dynamic maximal independent sets in the distributed setting [10] and very recently in the centralized setting [11, 4]. However, our analysis is quite different from theirs.

2 Maintaining a Proper $(\Delta + 1)$-Vertex Coloring

In this section, we give our constant-time dynamic algorithm and its analysis for maintaining a proper $(\Delta+1)$-coloring in a dynamic Δ-bounded graph and present the proof of Theorem 1. In Section 4, we discuss how to extend our algorithm to handle the case that the maximum degree Δ also changes. Recall that a dynamic graph is said to be Δ-bounded if throughout the updates, it is Δ-bounded. Given Δ, let $\mathcal{C} := \{1, \cdots, \Delta + 1\}$ denote the set of *colors*. A coloring $\chi : V \to \mathcal{C}$ is *proper* if $\chi(u) \neq \chi(v)$ for any $(u, v) \in E$.

2.1 Data Structures and the Algorithm

Data structures. We use the following data structures.

(1) We maintain a vertex coloring χ as an array such that $\chi(v)$ denotes the color of the current graph and guarantee that χ is a proper $(\Delta+1)$-vertex coloring after each update.

(2) For each vertex $v \in V$ we maintain: (a) its rank $r(v)$ that is chosen uniformly at random from $[0,1]$ during preprocessing; (b) its degree $\deg(v)$; (c) the last time stamp, denoted by τ_v, at which v was recolored; (d) two sets $L_v := \{u : (u,v) \in E, r(u) < r(v)\}, H_v := \{u : (u,v) \in E, r(u) \geq r(v)\}$, which contain all neighbors of v with ranks less than v, and all neighbors of v with ranks at least v (including v itself), respectively; (e) the sizes of the previous two sets, i.e., $|L_v|$ and $|H_v|$. Note that $\deg(v) = |L_v \cup H_v| = |L_v| + |H_v|$. For each vertex $v \in V$ note that every color of $\mathcal{C}$ is either (i) used by no neighbor of v (and we call such color a *blank* color for v), (ii) used by a neighbor in H_v, or (iii) used by a neighbor in L_v and by *no* neighbor in H_v. We call the corresponding sets of colors (i) $\mathcal{B}_v$, (ii) $\mathcal{C}_v(H)$, and (iii) $\mathcal{C}_v(L)$. We further partition $\mathcal{C}_v(L)$ into (iii.1) $\mathcal{U}_v(L)$, which denotes the set of *unique* colors for v that have been used by exactly one vertex in L_v and (iii.2) $\mathcal{M}_v(L)$, which denotes the set of colors that have been used by at least two vertices in L_v. Thus, $\mathcal{C} = \mathcal{C}_v(H) \dot\cup \mathcal{B}_v \dot\cup \mathcal{U}_v(L) \dot\cup \mathcal{M}_v(L)$. As it will be useful in the description of the algorithm, we finally define $\mathcal{C}_v(\overline{H}) := \mathcal{B}_v \cup \mathcal{U}_v(L) \cup \mathcal{M}_v(L)$. Note that for any fixed v, a color c can appear in exactly one of the two sets $\mathcal{C}_v(H)$ and $\mathcal{C}_v(\overline{H})$.

(3) (i) For every vertex v, we maintain $\mathcal{C}_v(H)$ and $\mathcal{C}_v(\overline{H})$ in doubly linked lists. (ii) For each color $c \in \mathcal{C}$ and vertex $v \in V$, we keep the following information: (a) a pointer $p_{c,v}$ from c to its position in either $\mathcal{C}_v(H)$ or $\mathcal{C}_v(\overline{H})$, depending on which list it belongs to; (b) a counter $\mu_v^H(c)$ such that $\mu_v^H(c)$ equals the number of neighbors in H_v with color c if $c \in \mathcal{C}_v(H)$; or equals 0 if $c \in \mathcal{C}_v(\overline{H})$. (iii) For any vertex v and color $c \in \mathcal{C}$ we keep the pointer $p_{c,v}$ in a hash table $\mathcal{A}_v$ which is indexed by c. (iv) For any vertex v and color $c \in \mathcal{C}_v(H)$, we maintain the pairs $(c, \mu_v^H(c))$ in a hash table $\mathcal{A}_v^H$ which is indexed by the pair (v,c).

More precisely, we use the dynamic perfect hashing algorithm by Dietzfelbinger et al. [12], which takes amortized expected constant time per update and worst-case constant time for lookups. (Alternatively we can get constant worst-case time for updates and lookups by spending time $O(n\Delta)$ during preprocessing to initialize suitable arrays).

To simplify the presentation and since the randomness in the hash tables is independent of the randomness used by the algorithm otherwise, we will not mention the randomness introduced through the usage of hash tables in the following.

Initialization. As the initial graph G_0 is empty, we initialize as follows: (1) For each vertex $u \in V$, sample a random number (called *rank*) $r(u) \in [0,1]$. (2) Color each vertex u by a random color $\chi(u) \in \mathcal{C} := \{1, \cdots, \Delta+1\}$ and initialize all the data structures suitably. In particular, for each $u \in V$, we initialize $\mathcal{C}_u(H)$ to be the empty list and $\mathcal{C}_u(\overline{H})$ to be the doubly linked list containing all colors in $\mathcal{C}$. Note that the latter takes $O(n\Delta)$ time. We discuss how to reduce the initialization time to $O(n)$ while keeping constant expected amortized update time in Section 4.

Time stamp reduction. Our algorithm does not use the actual values of the time stamps, only their relative order. Thus, every $\text{poly}(n)$ (say, n^4) number of updates we determine the order of the vertices according to the time stamps and set the time stamps of every vertex to equal its position in the order and set the current time stamp to $n+1$. This guarantees that we only need to use $O(\log n)$ bits to store the time stamp τ_v for each vertex v and it does

not affect the ordering of the time stamps. The cost of the recomputation of time stamps is $O(n \log n)$ and can be amortized over all the operations that are performed between two updates, increasing their running time only by an additive constant.

Handling an edge deletion. As any edge deletion (u, v) does not lead to a violation of the current proper coloring, we do not need to recolor any vertex, except to update the data structures corresponding to u, v, the details of which are deferred to Section 2.1.1.

Handling an edge insertion. For an edge insertion (u, v), we note that if $\chi(u) \neq \chi(v)$ before the insertion, then we only need to update the basic data structures corresponding to the two endpoints. If $\chi(u) = \chi(v)$, i.e, the current coloring χ is not proper any more, then we need to recolor one vertex $w \in \{u, v\}$ as well as to update the relevant data structures. We always recolor the vertex that was colored last, i.e., the one with larger τ_w. W.l.o.g., we assume this vertex is v. Then we invoke a subroutine RECOLOR(v) to recolor v and potentially some other lower level vertices, and update the corresponding data structures. That is, we will first update H_u, L_u, H_v, L_v and their sizes trivially in constant time. Then if $\chi(u) \neq \chi(v)$, we update the data structures corresponding to u, v as described in Section 2.1.1.

If $\chi(u) = \chi(v)$, and w.l.o.g., suppose that $\tau_v > \tau_u$, then we recolor v by invoking the procedure RECOLOR(v) below, where $\mathcal{U}_v(L)$ denotes the set of colors that have been used by *exactly one* vertex in L_v.

RECOLOR(v)

1. Run SETCOLOR(v) and obtain a new color c (from $\mathcal{B}_v \cup \mathcal{U}_v(L)$).
2. Set $\chi(v) = c$. Update the data structures by the process $(*)$ described in Section 2.1.1.
3. If $c \in \mathcal{U}_v(L)$,
   a. Find the unique neighbor $w \in L_v$ with $\chi(w) = c$.
   b. RECOLOR(w).
4. If $c \in \mathcal{B}_v$, then remove all the **visited** marks generated from the calls to SETCOLOR.

Note that the recursive calls will eventually terminate as for every call RECOLOR(w) in Step 3 it holds that $r(w) < r(v)$. Furthermore, no recursive call will be performed when $L_v = \emptyset$ as it implies that $\mathcal{U}_v(L) = \emptyset$. The subroutine RECOLOR$(v)$ calls the following subroutine SETCOLOR(v).

2.1.1 Updating the Data Structures

Case I: an edge deletion (u, v). Whenever an edge (u, v) gets deleted, we update the data structures corresponding to u and v as follows. More precisely, we first update the sets H_u, L_u, H_v, L_v and their sizes trivially in constant time. The lists $\mathcal{C}_u(H), \mathcal{C}_u(\overline{H}), \mathcal{C}_v(H), \mathcal{C}_v(\overline{H})$ can be updated in constant worst-case time. The hash tables $\mathcal{A}_u^H, \mathcal{A}_v^H$ can also be maintained in constant amortized expected update time. More precisely, suppose w.l.o.g., $u \in L_v$, then we do the following:

1. Delete $(\chi(v), \mu_u^H(\chi(v)))$ from $\mathcal{A}_u^H$; $\mu_u^H(\chi(v)) \leftarrow \mu_u^H(\chi(v)) - 1$.
2. If $\mu_u^H(\chi(v)) = 0$, then $\mathcal{C}_u(H) \leftarrow \mathcal{C}_u(H) \setminus \{\chi(v)\}$, $\mathcal{C}_u(\overline{H}) \leftarrow \mathcal{C}_u(\overline{H}) \cup \{\chi(v)\}$.
3. Otherwise, insert $(\chi(v), \mu_u^H(\chi(v)))$ to $\mathcal{A}_u^H$.

SetColor(v)

1. Mark v as **visited**. Initialize sets $L_{v,\text{old}} := \{v\}$ and $L_{v,\text{new}} := \emptyset$.
   Scan the list L_v: for any $u \in L_v$, if it is marked as **visited**, then add u to $L_{v,\text{old}}$; otherwise (i.e., it is not marked), then add u to $L_{v,\text{new}}$ and mark u as **visited**.
2. If $|L_v| + |H_v| < \frac{\Delta}{2}$ (i.e., $\deg(v) < \frac{\Delta}{2}$), repeatedly sample a color uniformly at random from $[\Delta + 1]$ until we get a color c that is contained in $\mathcal{B}_v$, the set of *blank* colors for v that have not been used by any neighbor of v.
3. Otherwise, we let $L^{<}_{v,\text{new}}$ denote the subset of vertices in $L_{v,\text{new}}$ with ranks at most the median of all ranks of vertices in $L_{v,\text{new}}$. We let $\mathcal{U}_v(L^{<}_{\text{new}})$ denote the set of colors that each has been used by exactly one vertex in $L_{v,\text{new}}$ and additionally this vertex belongs to $L^{<}_{v,\text{new}}$. Define $L^{<}_{v,\text{old}}$ and $\mathcal{U}_v(L^{<}_{\text{old}})$ similarly.
   a. If $|L_{v,\text{new}}| \geq \frac{1}{10}|L_v|$ or $L_v = \emptyset$, then we sample a random color c from the set of the first $\min\{|\mathcal{B}_v \cup \mathcal{U}_v(L^{g}_{\text{new}})|, |L^{<}_{v,\text{new}}| + 1\}$ elements of $\mathcal{B}_v \cup \mathcal{U}_v(L^{<}_{\text{new}})$.
   b. Else (i.e., $|L_{v,\text{old}}| > \frac{9}{10}|L_v|$) we sample a random color c from the set of the first $\min\{|\mathcal{B}_v \cup \mathcal{U}_v(L^{<}_{\text{old}})|, |L^{<}_{v,\text{old}}| + 1\}$ elements of $\mathcal{B}_v \cup \mathcal{U}_v(L^{<}_{\text{old}})$.
4. Update the relevant data structures (i.e. of v and its neighbors in L_v) and **Return** c.

Case II: an edge insertion (u, v) such that $\chi(u) \neq \chi(v)$. In this case, w.l.o.g., suppose that $r(u) < r(v)$, we update the data structures as follows:

1. $\mathcal{C}_u(H) \leftarrow \mathcal{C}_u(H) \cup \{\chi(v)\}$, $\mathcal{C}_u(\overline{H}) \leftarrow \mathcal{C}_u(\overline{H}) \setminus \{\chi(v)\}$, $\mu^H_u(\chi(v)) \leftarrow \mu^H_u(\chi(v)) + 1$
2. Delete $(\chi(v), \mu^H_u(\chi(v)) - 1)$ from $\mathcal{A}^H_u$ if $\mu^H_u(\chi(v)) > 1$, insert $(\chi(v), \mu^H_u(\chi(v)))$ to $\mathcal{A}^H_u$.

Case III: procedure (⁎) in the subroutine Recolor(v). In the subroutine RecolorMis(v), if the color of v is changed from c' to c, then we update the relevant data structure as follows:

(⁎) For every $w \in L_v$:

1. $\mu^H_w(c') \leftarrow \mu^H_w(c') - 1$
2. If $\mu_w(c') = 0$, then $\mathcal{C}_w(H) \leftarrow \mathcal{C}_w(H) \setminus \{c'\}$, $\mathcal{C}_w(\overline{H}) \leftarrow \mathcal{C}_w(\overline{H}) \cup \{c'\}$,
3. $\mathcal{C}_w(H) \leftarrow \mathcal{C}_w(H) \cup \{c\}$, $\mathcal{C}_w(\overline{H}) \leftarrow \mathcal{C}_w(\overline{H}) \setminus \{c\}$, $\mu^H_w(c) \leftarrow \mu^H_w(c) + 1$.
4. Delete $(c, \mu^H_w(c))$ from $\mathcal{A}^H_w$ if $\mu^H_w(c) > 1$, and insert $(c, \mu^H_w(c))$ to $\mathcal{A}^H_w$.

2.2 The Analysis

Next we prove Theorem 1. Let $v_0 := v$ be the vertex that needs to be recolored after an insertion and let $v_1, v_2, \cdots, v_\ell$ denote the vertices on which the recursive calls of Recolor() were executed. We call $v_0, v_1, \cdots, v_\ell$ the *recoloring path* originated from v. In the following lemma, we show that the expected total time for all calls Recolor(v_i) is $O(1 + \sum_{i=0}^{\ell} |L_{v_i}|)$, where the expectation is *not* over the random choices of ranks or colors at Step 3, but comes from the use of hash tables and sampling colors at Step 2.

▶ **Lemma 3.** *Subroutine* SetColor(v) *can be implemented to run in $O(1 + |L_v|)$ expected time. For any recoloring path $v_0, v_1, \cdots, v_\ell$, the expected time for subroutine* Recolor(u) *for any $u \in \{v_1, \ldots, v_l\}$ excluding the recursive calls to* Recolor() *is $O(|L_u|)$ if $u \neq v_\ell$, and is $O(1 + \sum_{i=0}^{\ell} |L_{v_i}|)$ if $u = v_\ell$.*

Proof. Recall that we store L_v, $\mathcal{C}_v(H)$, and $\mathcal{C}_v(\overline{H})$ for every vertex v. We use them to build all the sets needed in SetColor(v). First we use an array $R_{v,L_{\text{new}}}$ (resp. $R_{v,L_{\text{old}}}$) to store ranks of vertices in $L_{v,\text{new}}$ (resp. $L_{v,\text{old}}$), and then find the median $m_{v,\text{new}}$ (resp. $m_{v,L_{\text{old}}}$) of the set of ranks of vertices in $L_{v,\text{new}}$ (resp. $L_{v,\text{old}}$) deterministically in $O(|R_{v,L_{\text{new}}}|) = O(|L_v|)$

time [8]. Traversing L_v again (and using an empty array of length Δ that we clean again after this step) we compute (1) the sets $\mathcal{U}_v(L^<_{\text{new}})$ and $\mathcal{U}_v(L^<_{\text{old}})$ of colors that contain all colors that have been used by *exactly one* vertex in $L^<_{v,\text{new}}$, and by *exactly one* vertex in $L^<_{v,\text{old}}$, respectively, and (2) the sets $\mathcal{M}_v(L)$ of colors that contain all colors that have been used by *at least two* vertices in L_v. Note that $\mathcal{U}_v(L) = \mathcal{U}_v(L^<_{\text{new}}) \cup \mathcal{U}_v(L^<_{\text{old}})$, and, thus, it can be computed by copying these lists. All these lists have size $O(|L_v|)$ and, thus, all these steps take time $O(|L_v|)$.

We will keep the sets $\mathcal{M}_v(L)$, $\mathcal{U}_v(L)$, $\mathcal{U}_v(L^<_{\text{new}})$, $\mathcal{U}_v(L^<_{\text{old}})$ in four separate lists and build hash tables for these sets with pointers to their positions in the lists. Next we delete all colors in $\mathcal{M}_v(L) \cup \mathcal{U}_v(L)$ from the list $\mathcal{C}_v(\overline{H})$ and the resulting list will be $\mathcal{B}_v$. Note that the hash tables can be implemented in time linear in the size of corresponding sets, and each lookup (i.e., check if an element is in the set) takes constant worst-case time [12]. This completes the building of the data structure before Step 1.

Recall that $|L_v| + |H_v| = \deg(v)$. Then for Step 2, if $\deg(v) < \frac{\Delta}{2}$, we know that $|\mathcal{B}_v| > \Delta - \frac{\Delta}{2} = \frac{\Delta}{2}$. Thus, a randomly sampled color from $[\Delta+1]$ belongs to $\mathcal{B}_v$ with probability at least $1/2$, which implies that in $O(1)$ expected time, we will sample a color c from $\mathcal{B}_v$. Note that a color c belongs to $\mathcal{B}_v$ if and only if c is not contained in $\mathcal{M}_v(L) \cup \mathcal{U}_v(L) \cup \mathcal{C}_v(H)$, which can be checked by using the hash tables for $\mathcal{M}_v(L)$, for $\mathcal{U}_v(L)$ and the hash table $\mathcal{A}_v^H$.

All the other steps only write, read and/or delete lists or hash tables of size proportional to $|L_v|$ or $|\mathcal{M}_v(L) \cup \mathcal{U}_v(L)|$, which is at most $|L_v|$. Though the list $\mathcal{B}_v \cup \mathcal{U}_v(L^<_{\text{new}})$ might have size much larger than $|L^<_{v,\text{new}}|$, it suffices to read at most $|L^<_{v,\text{new}}|$ elements from it in Step 3 (similar for $\mathcal{B}_v \cup \mathcal{U}_v(L^<_{\text{old}})$ versus $|L^<_{v,\text{old}}|$). In Step 4, to update the relevant data structures, we add all colors in $\mathcal{M}_v(L) \cup \mathcal{U}_v(L)$ back to the list $\mathcal{B}_v$ to construct $\mathcal{C}_v(\overline{H})$. Thus, SetColor($v$) takes $O(1 + |L_v|)$ expected time.

To analyze the running time of Recolor(u) (apart from the recursive calls), for any $u \in v_0, v_1, \ldots, v_\ell$, note that apart from calling SetColor(u), Recolor updates the data structures, determines the neighbor w that needs to be recolored next (if any) and if no such neighbor w exists, i.e. c is a blank color and u is the last vertex of the recoloring path, then it unmarks all vertices that were marked by all the calls to SetColor on the recoloring path. For this SetColor has stored all the marked vertices on a list, which it returns to Recolor. This list is then used by recolor to unmark these vertices. The time to update the data structures is constant expected time (the expectation arises due to the use of hash tables) to update its own data structure and $O(|L_u|)$ to update the data structures of its lower neighbors. Determining w requires $O(|L_u|)$ time, as all lower neighbors of u have to be checked. Finally, Recolor(u) for the last vertex $u = v_\ell$ on the recoloring path takes expected time $O(1 + \sum_i |L_{v_i}|)$ as it unmarks all vertices on the recoloring path and their neighbors. ◀

Throughout the process we have two different types of randomness: one for sampling the ranks for the vertices and the other for sampling the colors. These two types of randomness are independent. Furthermore, only the very last vertex v_ℓ on the recoloring path $P = v_0, v_1, \cdots, v_\ell$ can satisfy the condition of Step 2 in SetColor, as once the condition is satisfied, we will sample a blank color which will not cause any further recursive calls. Thus, for all vertices on P, with the possible exception of v_ℓ, Step 3 will be executed. We call a vertex w with $\deg(w) < \frac{\Delta}{2}$ a *low degree* vertex. Note that for a low degree vertex w, SetColor(w) executes Step 2 and takes $O(1)$ expected time, as with probability at least $1/2$ a randomly sampled color will be blank. In the following, we consider the expected time T_v of recoloring P that excludes the time of recoloring any low degree vertex (which, if

exists, must be the last vertex on P). We first present a key property regarding the expected running time for recoloring a vertex v. Let $N(v)$ denote the set of all neighbors of v in the current graph.

▶ **Lemma 4.** *Let G denote the current graph. For any vertex v with rank $r(v) \leq \alpha$, the expected running time T_v (over the randomness of choosing ranks of other vertices) is*

$$E[T_v | r(v) \leq \alpha] = O(\alpha\Delta) \tag{1}$$

Furthermore, conditioned on ranks of vertices in $N(v)$ and $r(v) \leq \alpha$, it holds that the expected running time T_v (over the randomness of sampling ranks of $V \setminus (N(v) \cup \{v\})$) is

$$E[T_v | r(v) \leq \alpha, r(w) \forall w \in N(v)] = O(|L_v|) + O(\alpha\Delta) \tag{2}$$

The proof of the above lemma is deferred to Section 2.2.1. We remark that Lemma 4 assumes that for each operation, it is executed in any possible current graph G with any proper $(\Delta + 1)$-coloring (i.e. worst-case analysis for graph and coloring) and that each rank is sampled uniformly at random from $[0, 1]$ in G. This is true as the adversary is assumed to be oblivious, i.e., the sequence of all updates has been written down before the algorithm starts to process the updates. That is, for any current graph G, the random ranks of vertices still follows from the same distribution as the one in the beginning. The above further implies that we can bound the work for recoloring a conflicting vertex v in G by a function that depends only on the randomness for sampling ranks (and *not* on the randomness for selecting colors in previous updates).

We will also need the following lemma regarding the size of the sampled color set. The proof of the lemma follows from a more refined analysis of the proof of Claim 3.1 in [5] and can be found in the full version of the paper.

▶ **Lemma 5.** *Let v be any vertex that needs to be recolored. Let s denote the size of the set of colors that the algorithm samples from in order to choose a new color for v. Then it holds that 1) if $|L_v| + |H_v| < \frac{\Delta}{2}$, then $s \geq \frac{\Delta}{2} + 1$; 2) otherwise, $s \geq \frac{1}{100}|L_v| + 1$.*

With the lemmas above, we are ready to prove Theorem 1.

Proof of Theorem 1. Note that an edge deletion does not lead to the recoloring of any vertex. Let us consider an insertion (u, v). If $\chi(u) \neq \chi(v)$, we do not need to recolor any vertex. Otherwise, we need to recolor one vertex from $\{u, v\}$. Suppose w.l.o.g. that $\tau_v > \tau_u$, where τ_u denotes the last time that u has been recolored. This implies that v is recolored at the current time step, which we denote by τ. We will invoke RECOLOR(v) to recolor v. Note that by definition, after calling subroutine RECOLOR, there will be no conflict in the resulting coloring. This proves the correctness of the algorithm. In the following, we analyze its running time.

Recall that we let T_v denote the running time of calling RECOLOR(v), including all the recursive calls to RECOLOR, while *excluding* the time of recoloring any low degree vertex (i.e. a vertex where SETCOLOR(w) executed Step 2) on the recoloring path originated from v (which, if exists, must be the last vertex on the path). If the last vertex is indeed a low degree vertex, then the expected total running time (over all sources of randomness) of RECOLOR(v) will be $\mathrm{E}[T_v] + O(1)$, where the expectation $\mathrm{E}[T_v]$ in turn is over the randomness of sampling ranks of all vertices; otherwise, the expected total running time (over all sources of randomness) of RECOLOR(v) will be $\mathrm{E}[T_v]$. Let $\alpha_0 = \frac{4C \log \Delta}{\Delta}$ for some constant $C \geq 1$. Now we consider two cases:

Case I: $r(v) \leq \alpha_0$. First we note that this case happens with probability at most α_0 as $r(v)$ is chosen uniformly at random from $[0,1]$. Furthermore, by Lemma 4, conditioned on the event that $r(v) \leq \alpha_0$, the expected time of the subroutine $\textsc{Recolor}(v)$ is $\mathrm{E}[T_v | r(v) \leq \alpha_0] = O(\alpha_0 \Delta)$, where the expectation is taken over the randomness of choosing ranks of all other vertices except v. Therefore, the expected time of $\textsc{Recolor}(v)$ (over the randomness of choosing ranks of all vertices) is at most $\alpha_0 \cdot O(\alpha_0\Delta) = O(\alpha_0^2\Delta) = O(\frac{\log^2 \Delta}{\Delta}) = O(1)$.

Case II: $r(v) > \alpha_0$. Let $r(v) = \alpha$. Conditioned on the event that $r(v) = \alpha$, by Lemma 4, the expected running time (over the randomness of choosing ranks of other vertices) of $\textsc{Recolor}(v)$ at time τ is $O(\alpha\Delta)$.

We let L_v and L'_v denote the set of neighbors of v with ranks lower than v in the graph at (current) time τ and at time τ_v, (the latest time that v was recolored), respectively. Note that $\tau_u < \tau_v$ implies that *neither $\chi(u)$ nor $\chi(v)$ changed between τ_v and τ*. We define H_v, H'_v similarly. We let $\deg(v) = |L_v \cup H_v|$ and $\deg'(v) = |L'_v \cup H'_v|$ denote the degree of v at time τ and τ_v, respectively.

Case (a): $\deg'(v) < \Delta/2$. In this case, we know that at time τ_v, we will sample a color from the set of blank colors $\mathcal{B}(v)$, which has size at least $\Delta/2$. Thus, the probability that we sampled *any fixed* color at time τ_v is at most $2/\Delta$. This also applies to the color $\chi(u)$. Thus, the probability that $\chi(v) = \chi(u)$ at time τ_v is at most $2/\Delta$. As neither $\chi(v)$ nor $\chi(u)$ have changed between τ_v and τ (which implies that the random choices of the algorithm between τ_v and τ have no influence on $\chi(v)$ or $\chi(u)$), the probability that $\chi(v) = \chi(u)$ at time τ is at most $2/\Delta$. On the other hand, at time τ, we will spend at most $O(\alpha\Delta) = O(\Delta)$ expected time (over the randomness of sampling ranks of vertices in $V \setminus \{v\}$). Thus, the expected time (over the randomness of sampling ranks and of sampling colors at time τ_v) we spent on recoloring v at time τ is $O(\frac{1}{\Delta} \cdot \Delta) = O(1)$.

Case (b): $\deg'(v) \geq \Delta/2$. We now consider two sub-cases.

Case (b1): If $\deg(v) < \Delta/4$, then there must have been at least $\deg'(v)/2 = \Omega(\Delta)$ deletions of edges incident to v between τ_v and τ. We can recolor v at time τ in expected $O(\alpha\Delta) = O(\Delta)$ time. We charge this time to the updates incident to v between τ_v and τ. Note that each update is only charged twice in this way, once from each endpoint, adding a constant amount of work to each deletion.

Case (b2): If $\deg(v) \geq \Delta/4$, then $\mathrm{E}[|L_v|] = \alpha \deg(v) \geq \alpha\Delta/4 \geq \alpha_0\Delta/4 \geq C \log \Delta$ for some constant $C \geq 1$ and $\mathrm{E}[|L_v|] = \alpha \deg(v) \leq \alpha\Delta$. Then over the randomness of sampling ranks for vertices in $N(v)$, it follows from a Chernoff bound that with probability at least $1 - \frac{1}{\Delta}$, $\frac{\mathrm{E}[|L_v|]}{2} \leq |L_v| \leq \frac{3\mathrm{E}[|L_v|]}{2}$, which implies that with probability at least $1 - \frac{1}{\Delta}$,

$$(\alpha\Delta)/8 \leq \mathrm{E}[|L_v|]/2 \leq |L_v| \leq (3\mathrm{E}[|L_v|])/2 \leq (3\alpha\Delta)/2 \tag{3}$$

By Ineq. (2) in Lemma 4, over the randomness of sampling ranks for $V \setminus (N(v) \cup \{v\})$, the expected work for recoloring v at time τ is $O(|L_v|) + O(\alpha\Delta) = O(\alpha\Delta)$. We first analyze the case that Ineq. (3) does not hold, which happens with probability at most $1/\Delta$. Then the work for recoloring is $O(\Delta)$ as $|L_v| \leq \Delta$. Thus the expected work of this case is $\frac{1}{\Delta} \cdot O(\Delta) = O(1)$.

Next we analyze the case that Ineq. (3) holds and further distinguish two sub-cases.

Case (b2-1): If $|L_v \triangle L'_v| > \frac{1}{10}|L_v|$, then there must have been at least $\frac{1}{10}|L_v| = \Theta(\alpha\Delta)$ edge updates incident to v between τ_v and τ. By the same argument as above we can amortize the expected work of $O(\alpha\Delta)$ over these edge updates, charging each edge update at most twice. This adds an expected amortized cost of $O(1)$ to each update.

Case (b2-2): If $|L_v \triangle L'_v| \leq \frac{1}{10}|L_v|$, then it holds that $|L'_v| \geq |L_v| - |L_v \triangle L'_v| \geq \frac{9}{10}|L_v|$. By Lemma 5, $\chi(v)$ was picked at time τ_v from a set of $\Omega(|L'_v|)$ many colors. By similar argument for the Case (a), the probability that we picked the color $\chi(u)$ at time τ_v is at most $O(\frac{1}{|L'_v|}) = O(\frac{1}{|L_v|})$. As the expected work at time τ is at most $O(\alpha\Delta) = O(|L_v|)$ (with the expectation over randomness of sampling ranks), the expected amortized update time is $O(\frac{1}{|L_v|}) \cdot O(|L_v|) = O(1)$.

This completes the proof of the theorem. ◀

2.2.1 Bounding the Expected Work per Recoloring: Proof of Lemma 4

Let $v_0, v_1, \cdots$ be the vertices on the recoloring path after an insertion. By Lemma 3 the total expected time for all calls RECOLOR(v_i) is $O(1 + \sum_{i \geq 0} |L_{v_i}|)$. Recall that the running time T_v excludes the time spent on recoloring a low degree vertex (and a low degree vertex can only be the last vertex of a recoloring path). Thus, for all vertices v_i that contribute to T_v only Step 3a or Step 3b of SETCOLOR can occur. Let $v_{i_0} = v_0, v_{i_1}, v_{v_2}, \cdots$ be the vertices for which Step 3a occurred during SETCOLOR(v), which we call *good* vertices. We bound the expected value of ranks of good vertices and the expected size of the lower-ranked neighborhood of these vertices in the following lemma. Note that the expectations are taken over the randomness for sampling ranks of vertices, whose ranks are *not* in the conditioned events.

▶ **Lemma 6.** *For any $j \geq 0$, it holds that*

$$E[r(v_{i_j+1})|r(v_0) \leq \alpha] \leq \alpha/2^j, \quad E[|L_{v_{i_j}}||r(v_0) \leq \alpha] \leq (10 \cdot \alpha \cdot \Delta)/2^{j-1}.$$

Furthermore, for any $j \geq 1$, it holds that

$$\begin{aligned} E[r(v_{i_j+1})|r(v_0) \leq \alpha, r(w) \forall w \in N(v_0)] &\leq \alpha/2^{j-1}, \\ E[|L_{v_{i_j}}||r(v_0) \leq \alpha, r(w) \forall w \in N(v_0)] &\leq (10 \cdot \alpha \cdot \Delta)/2^{j-2}. \end{aligned}$$

Proof. To prove the lemma, we use the principle of deferred decisions: Instead of sampling the ranks for all vertices (independently and uniformly at random from $[0, 1]$) at the very beginning, we sample the ranks of vertices sequentially by the following random process:

Starting from v_0 with rank $r(v_0)$, we sample all the ranks of vertices in $N(v_0)$. We will then choose v_1 as described in the algorithm RECOLOR (if a non blank color has been sampled). Now for each $i \geq 1$, we note that the ranks of all the vertices in $N_{\text{old}}(v_i) := N(v_i) \cap (\cup_{j<i} N(v_j) \cup \{v_0\})$ have already been sampled, and then we only need to sample (independently and uniformly at random from $[0, 1]$) the ranks for all vertices in $N_{\text{new}}(v_i) := N(v_i) \setminus N_{\text{old}}(v_i)$. In this case, we say that the ranks of vertices in $N_{\text{new}}(v_i)$ are sampled *when we are exploring* v_i. Then we will choose v_{i+1} in the algorithm RECOLOR (if a non blank color has been sampled). We iterate the above process until RECOLOR has sampled a blank color.

For any i, we call $N_{\text{new}}(v_i)$ the *free neighbors* of v_i with respect to $v_0, v_1, \cdots, v_{i-1}$. In particular, $N_{\text{new}}(v_0) = N(v_0)$ and $N(v_i) = N_{\text{new}}(v_i) \dot{\cup} N_{\text{old}}(v_i)$. Now a key observation is that

($\star$) for any vertex v_i, it holds that $L_{v_i,\text{new}}$ (as defined in the algorithm SETCOLOR(v_i)) is entirely determined by the ranks of the vertices $N_{\text{new}}(v_i)$ and is independent of the randomness for sampling ranks of $N_{\text{old}}(v_i)$.

This is true since $L_{v_i,\text{new}}$ contains all the neighbors of v_i with ranks less than $r(v_i)$ and have not been visited so far: for any vertex in $N_{\text{old}}(v_i)$, either its rank is higher than v_i, or its rank is less than v_i and it has been marked as **visited** before we invoke SetColor(v_i).

We first prove the first part of the lemma. We *assume for now that* $r(v_0)$ *is fixed* and we denote by $\mathcal{R}(i_j)$ the randomness of sampling ranks for vertices in $N_{\text{new}}(v_{i_j})$. We will prove by induction on the index j that

$$\mathrm{E}_{\mathcal{R}(i_j)}[r(v_{i_j+1})] \le r(v_0)/2^j \text{ and } \mathrm{E}_{\mathcal{R}(i_j)}[|L_{v_{i_j},\text{new}}|] \le (r(v_0)\cdot\Delta)/2^{j-1}. \tag{4}$$

Note that this holds for $j=0$ since $i_0=0$, $r(v_1)\le r(v_0)$, $L_{v_{i_0},\text{new}} = L_{v_0}$, and $\mathrm{E}_{\mathcal{R}(0)}[|L_{v_0}|] = r(v_0)\cdot|N(v_0)| \le r(v_0)\cdot\Delta$. Next we assume it holds for $j-1$, and prove it also holds for j. By the definition of the good vertex v_{i_j}, we know that $v_{i_j+1}\in L_{v_{i_j}}$, and that the rank of v_{i_j+1} is at most the median, denoted by $m_{v_{i_j},\text{new}}$, of all the ranks of vertices in $L_{v_{i_j},\text{new}}$, which in turn consists of all vertices in $N_{\text{new}}(v_{i_j})$ with rank not larger than $r(v_{i_j})$. Furthermore, by the observation $(\star)$, the rank of $r(v_{i_j+1})$ depends only on $r(v_{i_j})$ and the ranks in $N_{\text{new}}(v_{i_j})$. This implies that

$$\mathrm{E}_{\mathcal{R}(i_j)}[r(v_{i_j+1})|r(v_{i_j})] \le \mathrm{E}_{\mathcal{R}(i_j)}[m_{v_{i_j},\text{new}}|r(v_{i_j})] \le r(v_{i_j})/2,$$

where the last inequality follows from the fact that $m_{v_{i_j},\text{new}}$ is the median of a set of numbers chosen independently and uniformly at random from $[0,1]$, conditioned on that they are at most $r(v_{i_j})$ (see e.g., Lemma 8.2 and 8.3 in [21]). Since $r(v_{i_j}) \le r(v_{(i_{j-1})+1})$ in all cases and, by the induction assumption, $\mathrm{E}_{\mathcal{R}(i_{j-1})}[r(v_{(i_{j-1})+1})] \le \frac{r(v_0)}{2^{j-1}}$, it holds that

$$\begin{aligned}
&\mathrm{E}_{\mathcal{R}(i_j)}[r(v_{i_j+1})] \le \mathrm{E}_{r(v_{i_j})}[\mathrm{E}_{\mathcal{R}(i_j)}[r(v_{i_j+1})|r(v_{i_j})]] \le \frac{1}{2}\mathrm{E}_{r(v_{i_j})}[r(v_{i_j})] \\
\le\ & \frac{1}{2}\mathrm{E}_{\mathcal{R}(i_{j-1})}[\mathrm{E}_{r(v_{i_j})}[r(v_{i_j})|r(v_{(i_{j-1})+1})]] \le \frac{1}{2}\mathrm{E}_{\mathcal{R}(i_{j-1})}[r(v_{(i_{j-1})+1})] \le \frac{r(v_0)}{2^j}.
\end{aligned}$$

Furthermore, for any $j\ge 0$, by the observation $(\star)$, $L_{v_{i_j},\text{new}}$ depends only on $r(v_{i_j})$ and ranks in $N_{\text{new}}(v_{i_j})$. Thus

$$\mathrm{E}_{\mathcal{R}(i_j)}[|L_{v_{i_j},\text{new}}|\ |r(v_{i_j})] \le r(v_{i_j})\cdot|N_{\text{new}}(v_{i_j})| \le r(v_{i_j})\cdot\Delta.$$

This further implies that

$$\mathrm{E}_{\mathcal{R}(i_j)}[|L_{v_{i_j},\text{new}}|] = \mathrm{E}_{r(v_{i_j})}[\mathrm{E}_{\mathcal{R}(i_j)}[|L_{v_{i_j},\text{new}}|\ |r(v_{i_j})]] \le \mathrm{E}_{r(v_{i_j})}[r(v_{i_j})]\cdot\Delta \le \frac{r(v_0)\cdot\Delta}{2^{j-1}}.$$

Now let us no longer assume that $r(v_0)$ is fixed, but instead condition on the event that $r(v_0)\le\alpha$. Then it follows that $\mathrm{E}_{\mathcal{R}(i_j)}[r(v_{i_j+1})|r(v_0)\le\alpha] \le \frac{\alpha}{2^j}$ and $\mathrm{E}_{\mathcal{R}(i_j)}[|L_{v_{i_j},\text{new}}|\ |r(v_0)\le\alpha] \le \frac{\alpha\cdot\Delta}{2^{j-1}}$.

Now by the definition of good vertices, we have $|L_{v_{i_j},\text{new}}| \ge \frac{1}{10}|L_{v_{i_j}}|$. This implies that

$$\mathrm{E}_{\mathcal{R}(i_j)}[|L_{v_{i_j}}|\ |r(v_0)\le\alpha] \le 10\cdot\mathrm{E}_{\mathcal{R}(i_j)}[|L_{v_{i_j},\text{new}}|\ |r(v_0)\le\alpha] \le 10\cdot(\alpha\cdot\Delta)/(2^{j-1}).$$

This completes the proof of the first part of the lemma.

For the "Furthermore" part of the lemma, the analysis is similar as above. Now we start with the assumption that $r(v_0), r(w)\forall w\in N(v_0)$ are fixed. Note that $v_{i_1}\in N(v_0)$, which implies that $r(v_{i_1})$ is also fixed. We will then prove by induction on the index j that

$$\mathrm{E}_{\mathcal{R}(i_j)}[r(v_{i_j+1})] \le (r(v_{i_1}))/(2^{j-1}) \text{ and } \mathrm{E}_{\mathcal{R}(i_j)}[|L_{v_{i_j},\text{new}}|] \le (r(v_{i_1})\cdot\Delta)/(2^{j-2}).$$

In the case $j=1$, the above two inequalities hold as $r(v_{i_1+1}) \le r(v_{i_1})$ and $\mathrm{E}_{\mathcal{R}(i_1)}[|L_{v_{i_1},\text{new}}|] = r(v_{i_1})\cdot|N_{\text{new}}(v_{i_1})| \le r(v_{i_1})\cdot\Delta$. The inductive step from case $j-1$ to j can be then proven in the same way as we proved Inequalities (4). Then instead of assuming that

$r(v_0), r(w)\forall w \in N(v_0)$, we condition on the event that $r(v_0) \leq \alpha, r(w)\forall w \in N(v_0)$, which directly implies that $r(v_{i_1}) \leq \alpha$. Then it follows that $\mathrm{E}_{\mathcal{R}(i_j)}[r(v_{i_j+1})|r(v_0) \leq \alpha, r(w)\forall w \in N(v_0)] \leq \frac{\alpha}{2^{j-1}}$ and $\mathrm{E}_{\mathcal{R}(i_j)}[|L_{v_{i_j},\mathrm{new}}| \ |r(v_0) \leq \alpha, r(w)\forall w \in N(v_0)] \leq \frac{\alpha\cdot\Delta}{2^{j-2}}$. Finally, by the definition of good vertices, $|L_{v_{i_j},\mathrm{new}}| \geq \frac{1}{10}|L_{v_{i_j}}|$, which implies that $\mathrm{E}_{\mathcal{R}(i_j)}[|L_{v_{i_j}}| \ |r(v_0) \leq \alpha, r(w)\forall w \in N(v_0)] \leq 10 \cdot \mathrm{E}_{\mathcal{R}(i_j)}[|L_{v_{i_j},\mathrm{new}}| \ |r(v_0) \leq \alpha, r(w)\forall w \in N(v_0)] \leq \frac{10\alpha\cdot\Delta}{2^{j-2}}$. This completes the "Furthermore" part of the lemma. ◀

Now we relate the total work to the work incurred by Step 3a. Note that the total work T_v is proportional to the sum of sizes of all lower-ranked neighborhoods of $v_0, v_1, \ldots$. We will prove the following lemma, which implies that the total work of recoloring v is at most a constant factor of the total work for recoloring all the *good vertices* on the recoloring path.

▶ **Lemma 7.** *It holds that* $\sum_i |L_{v_i}| \leq 3\sum_{i\,:\,v_i \text{ is good}} |L_{v_i}| = 3\sum_j |L_{v_{i_j}}|$.

Proof. We first introduce the following definition. For any i and $k < i$, we let $\mathcal{F}(v_k, v_i)$ denote the set of vertices whose ranks are less than $r(v_i)$, and are sampled when we are exploring v_k, i.e., $\mathcal{F}(v_k, v_i) = \{w : w \in N_{\mathrm{new}}(v_k), r(w) < r(v_i)\}$. Note that as $r(v_{i+1}) < r(v_i)$, it always holds that for any $0 \leq k < i$, $\mathcal{F}(v_k, v_{i+1}) \subseteq \mathcal{F}(v_k, v_i)$. Now we define the following potential function Φ:

$$\Phi(-1) := 0 \text{ and } \ \Phi(i) := \sum_{k:k\leq i} |\mathcal{F}(v_k, v_{i+1})| \ \ \forall i \geq 0, \tag{5}$$

We have the following claim regarding the potential functions.

▷ Claim 8. For any $i \leq 0$, $\Phi(i) \geq 0$. Furthermore, if v_i is a good vertex, then $\Phi(i)-\Phi(i-1) \leq |L_{v_i}|/2$, otherwise $\Phi(i) - \Phi(i-1) \leq -7|L_{v_i}|/20$.

Proof. Note that if Step 3a in subroutine SetColor is executed at vertex v_i, i.e., v_i is good, then the potential $\Phi(i)$ might be larger or smaller than $\Phi(i-1)$. If v_i is good then $|\mathcal{F}(v_i, v_{1+i})| \leq \frac{|L^{<}_{v_i,\mathrm{new}}|}{2}$ by the fact that $r(v_{1+i})$ is at most the median rank in $L^{<}_{v_i,\mathrm{new}}$. Furthermore, it holds that

$$\begin{aligned}\Phi(i) = \sum_{k:k\leq i} |\mathcal{F}(v_k, v_{1+i})| \ &\leq \sum_{k:k\leq i-1} |\mathcal{F}(v_k, v_i)| + |\mathcal{F}(v_i, v_{i+1})| \\ &\leq \Phi(i-1) + |L^{<}_{v_i,\mathrm{new}}|/2 \leq \Phi(i-1) + |L_{v_i}|/2\end{aligned}$$

Now suppose that Step 3b is executed at vertex v_i, i.e., v_i is not good. Since v_{1+i} is a vertex from the lower half of the old lower neighbors of v_i (i.e., $v_{1+i} \in L^{<}_{v_i,\mathrm{old}} \subseteq \cup_{k<i}\mathcal{F}(v_k, v_i) \cap L_{v_i,\mathrm{old}}$), we have that to obtain the set $\cup_{k<i}\mathcal{F}(v_k, v_{1+i})$ from the set $\cup_{k<i}\mathcal{F}(v_k, v_i)$, we need to remove at least $\frac{1}{2}|L_{v_i,\mathrm{old}}| \geq \frac{1}{2}(1-\frac{1}{10})|L_{v_i}|$ vertices. Furthermore, $\mathcal{F}(v_i, v_{1+i})$ can contain at most $|L_{v_i,\mathrm{new}}| \leq \frac{1}{10}|L_{v_i}|$ vertices. This implies that

$$\begin{aligned}\Phi(i) &= \sum_{k:k\leq i} |\mathcal{F}(v_k, v_{1+i})| = \sum_{k:k\leq i-1} |\mathcal{F}(v_k, v_{1+i})| + |\mathcal{F}(v_i, v_{1+i})| \\ &\leq \sum_{k:k\leq i-1} |\mathcal{F}(v_k, v_i)| - \frac{1}{2}(1-\frac{1}{10})|L_{v_i}| + \frac{1}{10}|L_{v_i}| = \Phi(i-1) - \frac{7}{20}\cdot|L_{v_i}|\end{aligned}$$ ◁

Now we distinguish three types of indices. We call an index i, a *type I* index, if Step 3a occurred during SetColor(v) *and* the $\Phi(i) - \Phi(i-1) \geq 0$. By Claim 8 it holds that for such an index i, $|L_{v_i}| \geq 2(\Phi(i) - \Phi(i-1))$. We call i a *type II* index, if Step 3a occurred during SetColor(v) *and* the $\Phi(i) - \Phi(i-1) \leq 0$. It holds that for such an index i (as for any index), $|L_{v_i}| \geq 0$. We call i a *type III* index, if Step 3boccurred during SetColor(v), i.e. v_i is not a good vertex. By Claim 8 it holds that for such an index i, Φ decreases and

$$|L_{v_i}| \leq (\Phi(i-1) - \Phi(i)) \cdot \frac{20}{7} < 3 \cdot (\Phi(i-1) - \Phi(i)).$$

Now we bound the sum of sizes of lower-ranked neighborhoods of vertices corresponding to Step 3b. It holds that

$$\begin{aligned}\sum_{i:\text{ Step 3b}} |L_{v_i}| \leq \sum_{i:\text{ type III}} 3(\Phi(i-1)-\Phi(i)) &\leq \sum_{i:\text{ type II or III}} 3(\Phi(i-1)-\Phi(i)) \\ \leq \sum_{i:\text{ type I}} 3(\Phi(i)-\Phi(i-1)) &\leq \sum_{i:\text{ type I}} 3\cdot\frac{1}{2}|L_{v_i}| < \sum_{i:\text{ type I}} 2|L_{v_i}|\end{aligned}$$

where the third inequality follows from the fact that Φ starts at 0 and is non-negative at the end, and, thus, the total decrease of Φ is at most its total increase. Thus, it follows that

$$\sum_i |L_{v_i}| = \sum_{i:\text{ type I or II}} |L_{v_i}| + \sum_{i:\text{ type III}} |L_{v_i}| \leq 3 \sum_{i:\text{ type I or II}} |L_{v_i}| = 3\sum_j |L_{v_{i_j}}| \qquad \blacktriangleleft$$

Now we finish the proof of Lemma 4. By Lemma 7 and Lemma 6, it holds that

$$\mathrm{E}[\sum_i |L_{v_i}| \mid r(v)\leq\alpha] \leq 3\cdot\mathrm{E}[\sum_j |L_{v_{i_j}}| \mid r(v)\leq\alpha] = O(\alpha\cdot\Delta\cdot\sum_j \frac{1}{2^j}) = O(\alpha\Delta).$$

Since the expected work T_v satisfies that $T_v = O(\sum_i |L_{v_i}|)$, the first part of the lemma follows. By the "Furthermore" part of Lemma 6, it holds that

$$\begin{aligned}&\mathrm{E}[\sum_i |L_{v_i}| \,|\, r(v)\leq\alpha, r(w)\forall w\in N(v)] \\ \leq& 3\cdot|L_v| + 3\cdot\mathrm{E}[\sum_{j\geq 1} |L_{v_{i_j}}| \,|\, r(v)\leq\alpha, r(w)\forall w\in N(v)] \\ \leq& 3\cdot|L_v| + 3\cdot 10\cdot\alpha\cdot\Delta\cdot\sum_j \frac{1}{2^{j-2}} = 3\cdot|L_v| + O(\alpha\cdot\Delta\cdot\sum_j\frac{1}{2^j}) = O(|L_v|)+O(\alpha\Delta).\end{aligned}$$

Then the "Furthermore" part of Lemma 4 follows from the fact that $T_v = O(\sum_i |L_{v_i}|)$.

3 Lower Bound for Dynamic Δ-Colorability Testing: Proof of Theorem 2

In [24] Patrascu and Demaine construct an n-node graph and show that there exists a sequence $\mathcal{S}$ of T edge insertion, edge deletion, and query operations such that any data structure for dynamic connectivity must perform $\Omega(T\log n)$ cell probes to process the sequence, where each cell has size $O(\log n)$. This shows that the amortized number of cell probes per operation is $\Omega(\log n)$.

We now show how to use this result to get a lower bound for the dynamic Δ-colorability testing problem with $\Delta = 2$.

The graph G in the proof of [24] consists of a $\sqrt{n}\times\sqrt{n}$ grid, where each node in column 1 has exactly 1 edge to a node of column 2 and no other edges, each node in column i, with $1 < i < \sqrt{n}$ has exactly 1 edge to a node of column $i-1$ and 1 edge to a node of column $i+1$ and no other edges, and each node in column $\sqrt{n}$ has exactly 1 edge to a node of column $\sqrt{n}-1$ and no other edges. Thus, the graph consists of $\sqrt{n}$ paths of length $\sqrt{n}-1$ and the edges between column i and $i+1$ for any $1\leq i<\sqrt{n}$ represent a permutation of the $\sqrt{n}$ rows. The sequence $\mathcal{S}$ consists of "batches" of $O(\sqrt{n})$ edge updates, replacing the permutation of some column i by a new permutation for column i. Between the batches of updates are "batches" of connectivity queries, each consisting of $\sqrt{n}$ connectivity queries and a parameter $1\leq k\leq\sqrt{n}$, where the j-th query for $1\leq j\leq\sqrt{n}$ of each batch tests whether the j-th vertex of column 1 is connected with a specific vertex of column k.

Note that the maximum degree Δ is 2. We now show how to modify each connectivity query (u, v) such that it consists of a constant number of edge updates and one query whether the resulting graph is Δ-colorable. The answer will be *no* iff u and v are connected. Thus, in the resulting sequence $\mathcal{S}'$ the number of query operations equals the number of query operations in $\mathcal{S}$ and the number of update operations is linear in the number of update and query operations in $\mathcal{S}$. Thus the total number of operations in $\mathcal{S}'$ is only a constant factor larger than the number of operations in $\mathcal{S}$, which, together with the result of [24], implies that the amortized number of cell probes per operation is $\Omega(\log n)$.

We now show how to simulate a connectivity query(u, v), where u is in column 1 and v is in column k for some $1 \leq k\sqrt{n}$. We assume that k is even and explain below how to deal with the case that k is odd. The instance for the dynamic Δ-colorability testing consists of G with an additional node s added. To simulate a connectivity query(u, v) we (1) remove the edge from v to its neighbor in column $k+1$ if $k < \sqrt{n}$, (2) add the edges (u, s) and (v, s) and then (3) ask a Δ-colorability query. Note that the resulting graph still has maximum degree 2. Furthermore, if u and v are connected in G then there exists a unique path of odd length $k-1$ between them. Together with the edges (u, s) and (v, s) and the assumption that k is even, this results in an odd length cycle, so that the answer to the 2-colorability query is *no*. If, however, u and v are not connected in G, then adding the edges (u, s) and (v, s) creates a path of length $2 + \sqrt{n} - 1 + k - 1 = \sqrt{n} + k$, but no cycle. Thus, the 2-colorability query returns *yes*. Thus u and v are connected in G iff the 2-colorability query in the modified graph returns *no*. Afterwards we remove the edges (u, s) and (v, s). Finally if k is odd, we do not add a vertex s to G and to simulate the connectivity query(u, v) we simply insert the edge (u, v). As before there exists an odd length cycle in the graph iff u and v are connected. The rest of the proof remains unchanged.

This finishes the proof of Theorem 2.

▶ Remark 9. Let us recall Brooks' theorem [9]: every *connected* graph admits a Δ-coloring, except that it is an odd cycle or a complete graph. This implies that if the dynamic graph is guaranteed to be connected, then we can answer Δ-colorability in constant time for $\Delta \geq 3$ by checking if the graph is complete. However, since the graph is not necessarily connected, it is unclear if the query can be answered in constant time for $\Delta \geq 3$. In particular, testing whether a dynamic graph is connected or not requires $\Omega(\log n)$ time per operation [24].

4 Further Discussions

Initialization in $O(n)$ Time. Now we describe how we can reduce the initialization time from $O(n\Delta)$ to $O(n)$. Note that the only part that takes $O(n\Delta)$ time is to initialize $\mathcal{C}_u(\overline{H})$ for each vertex u, and the rest part of initialization already only takes $O(n)$ time. The main observation is that $\mathcal{C}_u(\overline{H})$ is only needed in the sampling subroutine of SetColor(u) and even there only once the degree of a vertex is at least $\Delta/2$. Since we make the standard assumption that we start with an empty graph, this means that $\Omega(\Delta)$ insertions incident to u must have happened. Thus, we build $\mathcal{C}_u(\overline{H})$ only once this is the case and amortize the cost of building it over these previous $\Omega(\Delta)$ insertions.

To be more precise, we change the initialization phase as follows: We do not build $\mathcal{C}_u(\overline{H})$ for any vertex u. Note that all other data structure are built as before, but they only have size $O(n)$ and only take time $O(n)$ to build.

When an edge (u, v) is inserted, we check whether one of the endpoints, say u, of the newly inserted edge reaches the degree $\Delta/2$ and does not yet have the data structure $\mathcal{C}_u(\overline{H})$. If so, we build $\mathcal{C}_u(\overline{H})$ and its hash table at this point in time $O(\Delta)$. We amortize this cost

over the $\Delta/2$ updates that increased the degree of u to $\Delta/2$, adding a constant amortized cost to each of them. (If the other endpoint v also reaches the degree $\Delta/2$, we handle it analogously.)

Note that this does not affect the SetColor algorithm: as long as the degree of a vertex u is less than $\Delta/2$, SetColor(u) selects a new color by sampling in Step 2 from $\mathcal{B}_u$. To do so $\mathcal{C}_u(\overline{H})$ is not needed: In time $O(|L_u|)$ time we build the lists and corresponding hash tables for $\mathcal{M}_u(L) \cup \mathcal{U}_u(L)$, which together with the maintained list and hash table for $\mathcal{C}_u(H)$ suffice for us to sample a color from $\mathcal{B}_u$ in $O(1)$ time: We pick a random color from $\mathcal{C}$ and test whether it belongs to $\mathcal{B}_u$ by making sure that it does not belong to $\mathcal{M}_u(L) \cup \mathcal{U}_u(L)$ or $\mathcal{C}_u(H)$. The fact that the degree of u is at most $\Delta/2$ implies that in expectation the second randomly chosen color will belong to $\mathcal{B}_u$.

Once $\mathcal{C}_u(\overline{H})$ and its hash table has been built, it is used in the way as we described before and updated as in Section 2.1.

Extension to Work for Changing Δ. As we mentioned, we can extend our algorithm to work with changing Δ. (A similar extension was also done in [5]). For any time stamp $t \geq 0$, we will maintain a global value $\Delta_t := \max_{j=1}^{t} \max_{v \in V} \deg_j(v)$, where $\deg_j(v)$ denotes the degree of v in the graph after j edge updates, that is, Δ is the maximum degree seen so far (till time t). Then we have a randomized algorithm for maintaining a $(\Delta_t + 1)$-coloring. More precisely, for any time stamp j, for each vertex v, we only need to guarantee that the color $\chi(v)$ is chosen from $\{1, \ldots, \deg_j(v) + 1\}$. Then for each vertex $v \in V$, we let $\mathcal{C}_v(\overline{H}) \subseteq \mathcal{C}$ consist of all the colors in $\{1, \ldots, \deg_j(v) + 1\}$ that have not been assigned to any neighbor u of v for $u \in H_v$. It is easy to see that Lemma 3, 4 and 5 still hold, and our randomized dynamic coloring algorithm maintains a proper $(\Delta_t + 1)$-coloring of the graph G_t at time t with constant amortized update time, for any $t \geq 0$.

Additionally we can keep a variable Δ such that we rebuild the data structure every Δn operations as follows: We determine the list of current edges and set Δ to be the maximum degree of the current graph. Then we build the data structure for an empty graph and insert all edges using the insert operation. This increases the running time by an amortized constant factor and guarantees that Δ is the maximum degree in the graph *within the last* Δn *updates.*

References

1. Luis Barba, Jean Cardinal, Matias Korman, Stefan Langerman, André van Renssen, Marcel Roeloffzen, and Sander Verdonschot. Dynamic graph coloring. In *Workshop on Algorithms and Data Structures*, pages 97–108. Springer, 2017.
2. Leonid Barenboim and Tzalik Maimon. Fully-dynamic graph algorithms with sublinear time inspired by distributed computing. *Procedia Computer Science*, 108:89–98, 2017.
3. Surender Baswana, Sumeet Khurana, and Soumojit Sarkar. Fully dynamic randomized algorithms for graph spanners. *ACM Transactions on Algorithms (TALG)*, 8(4):35, 2012.
4. Soheil Behnezhad, Mahsa Derakhshan, MohammadTaghi Hajiaghayi, Cliff Stein, and Madhu Sudan. Fully dynamic maximal independent set with polylogarithmic update time. In *IEEE 56th Annual Symposium on Foundations of Computer Science (FOCS), 2019*. IEEE, 2019.
5. Sayan Bhattacharya, Deeparnab Chakrabarty, Monika Henzinger, and Danupon Nanongkai. Dynamic algorithms for graph coloring. In *Proceedings of the Twenty-Ninth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 1–20. SIAM, 2018.
6. Sayan Bhattacharya, Fabrizio Grandoni, Janardhan Kulkarni, Quanquan C. Liu, and Shay Solomon. Fully dynamic $(\delta + 1)$ coloring in constant update time. Private communication.

7 Sayan Bhattacharya and Janardhan Kulkarni. Deterministically maintaining a $(2+\varepsilon)$-approximate minimum vertex cover in $O(1/\varepsilon^2)$ amortized update time. In *Proceedings of the Thirtieth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 1872–1885. SIAM, 2019.

8 Manuel Blum, Robert W Floyd, Vaughan Pratt, Ronald L Rivest, and Robert E Tarjan. Time bounds for selection. *Journal of Computer and System Sciences*, 7(4):448–461, 1973.

9 R. L. Brooks. On colouring the nodes of a network. *Mathematical Proceedings of the Cambridge Philosophical Society*, 37(2):194–197, 1941.

10 Keren Censor-Hillel, Elad Haramaty, and Zohar Karnin. Optimal dynamic distributed mis. In *Proceedings of the 2016 ACM Symposium on Principles of Distributed Computing*, pages 217–226. ACM, 2016.

11 Shiri Chechik and Tianyi Zhang. Fully dynamic maximal independent set in expected poly-log update time. In *IEEE 56th Annual Symposium on Foundations of Computer Science (FOCS), 2019*. IEEE, 2019.

12 Martin Dietzfelbinger, Anna R. Karlin, Kurt Mehlhorn, Friedhelm Meyer auf der Heide, Hans Rohnert, and Robert Endre Tarjan. Dynamic perfect hashing: Upper and lower bounds. *SIAM J. Comput.*, 23(4):738–761, 1994.

13 Ran Duan, Haoqing He, and Tianyi Zhang. Dynamic edge coloring with improved approximation. In *Proceedings of the Thirtieth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 1937–1945. SIAM, 2019.

14 David Eppstein, Zvi Galil, Giuseppe F. Italiano, and Amnon Nissenzweig. Sparsification - a technique for speeding up dynamic graph algorithms. *J. ACM*, 44(5):669–696, 1997.

15 Greg N. Frederickson. Data structures for on-line updating of minimum spanning trees (preliminary version). In *STOC*, pages 252–257, 1983.

16 Monika R Henzinger and Valerie King. Maintaining minimum spanning trees in dynamic graphs. In *International Colloquium on Automata, Languages, and Programming*, pages 594–604. Springer, 1997.

17 Monika Rauch Henzinger and Valerie King. Randomized fully dynamic graph algorithms with polylogarithmic time per operation. *J. ACM*, 46(4):502–516, 1999. doi:10.1145/320211.320215.

18 Jacob Holm, Kristian De Lichtenberg, and Mikkel Thorup. Poly-logarithmic deterministic fully-dynamic algorithms for connectivity, minimum spanning tree, 2-edge, and biconnectivity. *Journal of the ACM (JACM)*, 48(4):723–760, 2001.

19 Jacob Holm, Eva Rotenberg, and Christian Wulff-Nilsen. Faster fully-dynamic minimum spanning forest. In *Algorithms-ESA 2015*, pages 742–753. Springer, 2015.

20 Bruce M. Kapron, Valerie King, and Ben Mountjoy. Dynamic graph connectivity in polylogarithmic worst case time. In *SODA*, pages 1131–1141, 2013.

21 Michael Mitzenmacher and Eli Upfal. *Probability and computing: Randomized algorithms and probabilistic analysis*. Cambridge university press, 2005.

22 Danupon Nanongkai and Thatchaphol Saranurak. Dynamic spanning forest with worst-case update time: adaptive, Las Vegas, and $O(n^{1/2-\varepsilon})$-time. In *Proceedings of the 49th Annual ACM SIGACT Symposium on Theory of Computing*, pages 1122–1129. ACM, 2017.

23 Danupon Nanongkai, Thatchaphol Saranurak, and Christian Wulff-Nilsen. Dynamic minimum spanning forest with subpolynomial worst-case update time. In *Foundations of Computer Science (FOCS), 2017 IEEE 58th Annual Symposium on*, pages 950–961. IEEE, 2017.

24 Mihai Patrascu and Erik D. Demaine. Logarithmic lower bounds in the cell-probe model. *SIAM J. Comput.*, 35(4):932–963, 2006. doi:10.1137/S0097539705447256.

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

26 Shay Solomon and Nicole Wein. Improved dynamic graph coloring. In *26th Annual European Symposium on Algorithms*, 2018.

27 Dominic JA Welsh and Martin B Powell. An upper bound for the chromatic number of a graph and its application to timetabling problems. *The Computer Journal*, 10(1):85–86, 1967.

28 Christian Wulff-Nilsen. Fully-dynamic minimum spanning forest with improved worst-case update time. In *Proceedings of the 49th Annual ACM SIGACT Symposium on Theory of Computing*, pages 1130–1143. ACM, 2017.