Fully Dynamic $(\Delta + 1)$-Coloring in $O(1)$ Update Time

SAYAN BHATTACHARYA*, University of Warwick, UK
FABRIZIO GRANDONI*, IDSIA, Switzerland
JANARDHAN KULKARNI*, Microsoft Research, Redmond, USA
QUANQUAN C. LIU*, MIT CSAIL, USA
SHAY SOLOMON*, Tel Aviv University, Israel

The problem of $(\Delta+1)$-vertex coloring a graph of maximum degree Δ has been extremely well-studied over the years in various settings and models. Surprisingly, for the dynamic setting, almost nothing was known until recently. In SODA'18, Bhattacharya, Chakrabarty, Henzinger and Nanongkai devised a randomized algorithm for maintaining a $(\Delta + 1)$-coloring with $O(\log \Delta)$ expected amortized update time. In this paper, we present an improved randomized algorithm for $(\Delta + 1)$-coloring that achieves $O(1)$ amortized update time and show that this bound holds not only in expectation but also with high probability.

Our starting point is the state-of-the-art randomized algorithm for maintaining a maximal matching (Solomon, FOCS'16). We carefully build on the approach of Solomon, but, due to inherent differences between the maximal matching and $(\Delta + 1)$-coloring problems, we need to deviate significantly from it in several crucial and highly nontrivial points.[1]

ACM Reference Format:
Sayan Bhattacharya, Fabrizio Grandoni, Janardhan Kulkarni, Quanquan C. Liu, and Shay Solomon. 2020. Fully Dynamic $(\Delta + 1)$-Coloring in $O(1)$ Update Time. *ACM Trans. Algor.* 37, 4, Article 111 (August 2020), 25 pages. https://doi.org/10.1145/1122445.1122456

1 INTRODUCTION

Vertex coloring is one of the most fundamental and most well-studied graph problems. Consider any integral parameter $\lambda > 0$, an undirected graph $G = (V, E)$ with n nodes and m edges, and a *palette* $C = \{1, \ldots, \lambda\}$ of λ colors. A *λ-coloring* in G is simply a function $\chi : V \to C$ which assigns a color $\chi(v) \in C$ to each vertex $v \in V$. Such a coloring is called *proper* iff no two neighboring nodes in G get the same color. The main goal is to compute a proper λ-coloring in the input graph $G = (V, E)$ such that λ is as small as possible. Unfortunately, this problem is NP-hard and even extremely hard to approximate: for any constant $\epsilon > 0$, there is no polynomial-time approximation algorithm with approximation factor $n^{1-\epsilon}$ unless $P \neq NP$ [12, 22, 32]. Vertex coloring remains NP-hard even in graphs of small chromatic number. In particular, recognizing 3-colorable graphs is a classic NP-hard problem [15], and there is a deep line of work on coloring 3-colorable graphs in polynomial time with as few colors as possible [21].

[1] An earlier version of this paper started to circulate in early July 2019.

Partially supported by the SNSF Excellence Grant 200020B_182865/1.
Authors' addresses: Sayan Bhattacharya, S.Bhattacharya@warwick.ac.uk, University of Warwick, UK; Fabrizio Grandoni, fabrizio@idsia.ch, IDSIA, Switzerland ; Janardhan Kulkarni, jakul@microsoft.com, Microsoft Research, Redmond, USA; Quanquan C. Liu, quanquan@mit.edu, MIT CSAIL, USA; Shay Solomon, solo.shay@gmail.com, Tel Aviv University, Israel.

1549-6325/2020/8-ART111 $15.00
https://doi.org/10.1145/1122445.1122456

Since the problem is computationally hard in general, much of the work on vertex coloring has focused on restricted families of graphs in different settings. In the static case, results have included settings such as bounded arboricity, with the classic paper by Matula and Beck [28] in the sequential setting, and more recent results in a variety of models including the streaming [5], Congested Clique [5, 16], MPC [5], the general graph query [5], and Local models [2, 3, 5, 24]; other graph classes in which various vertex coloring problems were explored in the static case include bounded treewidth graphs [1, 13, 20], bounded clique-width graphs [14, 23], n-uniform hypergraphs [29], bounded diameter [10, 26, 27], and bounded degree graphs [11]. In the dynamic case, past results on restricted families of graphs have focused on the class of bounded arboricity graphs [18, 31].

Obviously it is always possible to $\Delta + 1$ color a graph $G = (V, E)$ with maximum degree Δ, and a simple linear time algorithm can achieve this goal in the classical centralized off-line setting. Achieving the same goal in different computational models is however not trivial. For example this problem was extensively studied in the distributed literature, both as a classical symmetry breaking problem, and due to its intimate connections with other fundamental distributed problems such as maximal matching and maximal independent set (MIS) [25].

In this work we study the $(\Delta + 1)$-coloring problem in the *fully dynamic* setting. Here, the input graph $G = (V, E)$ changes via a sequence of *updates*, where each update consists of the insertion or deletion of an edge in G. There is a *fixed* parameter $\Delta > 0$ such that the maximum degree in G remains upper bounded by Δ throughout this update sequence. We want to design an algorithm that is capable of maintaining a proper $(\Delta + 1)$-coloring in such a dynamic graph G. The time taken by the algorithm to handle an update is called its *update time*. We say that an algorithm has an *amortized* update time of $O(\gamma)$ iff starting from an empty graph,[2] it takes at most $O(t \cdot \gamma)$ time to handle any sequence of t updates. Our goal is to ensure that the amortized update time of our algorithm is as small as possible. Our focus is on amortized time bounds, and we henceforth use the phrase "update time" to refer to "amortized update time".

There is a naive dynamic algorithm for this problem that has $O(\Delta)$ update time, which works as follows. Suppose that we are maintaining a proper $\Delta + 1$-coloring $\chi : V \to C$ in G. At this point, if an edge gets deleted from the graph, then we do nothing, as the coloring χ continues to remain proper provided that Δ remains fixed throughout the update sequence. Otherwise, if an edge uv gets inserted into G, then we first check if $\chi(u) = \chi(v)$. If not, we do nothing. If yes, then we pick an arbitrary endpoint $x \in \{u, v\}$, and by scanning all its neighbors we identify a *blank* color $c' \in C$ for x (one that is not assigned to any of its neighbors). Such a blank color is guaranteed to exist, since x has at most Δ neighbors and the palette C consists of $\Delta + 1$ colors. We now *recolor* the node x by assigning it the color c'. This results in a proper $(\Delta + 1)$-coloring in the current graph. The time taken to implement this procedure is proportional to the degree of x, hence it is at most $O(\Delta)$. Bhattacharya et al. [7] significantly improved the $O(\Delta)$ time bound, obtaining the following result.

Theorem 1.1. *[7] There is a randomized dynamic algorithm that can maintain a $\Delta + 1$-coloring in a dynamic graph with $O(\log \Delta)$ update time in expectation.*

A fundamental open question left by [7] is whether one can improve the update time to constant. A constant update time is the holy grail for any graph problem that admits a linear time (static) algorithm, and thus far was obtained only for a handful of problems. Building on the algorithm of Baswana et al. [4], in FOCS'16 Solomon [30] presented a randomized algorithm for maintaining a maximal matching with constant update time. Remarkably, Solomon's algorithm was the first to

[2]In this paper, when we refer to an *empty graph*, we mean a graph that has n vertices and no edges. However, our algorithm can be modified to handle insertions and deletions of vertices with 0 degree. Such vertices are inserted into and deleted from the bottommost level of our structure.

achieve a constant update time *for any nontrivial problem in general dynamic graphs*. A maximal matching provides a 2-approximation for both the maximum matching and the minimum vertex cover. An alternative deterministic primal-dual approach, introduced by Bhattacharya et al. [8], maintains a fractional "almost-maximal" matching, and thus a $(2 + \epsilon)$-approximation for the maximum matching size, and also an integral $(2 + \epsilon)$-approximation for the minimum vertex cover. This line of work culminates in the SODA'19 paper of Bhattacharya and Kulkarni [9], which achieves an update time $O(1/\epsilon^2)$. There is an intimate connection between the two approaches, which is hard to formalize, but at a very high level, the randomness encapsulated within the maximal matching algorithms of [4, 30] naturally correspond to the fractional deterministic almost-maximal solutions of [6, 8, 9, 17].

Our main result is summarized in Theorem 1.2 below. We design a randomized algorithm for $(\Delta + 1)$-coloring with $O(1)$ update time in expectation and with high probability (for a sufficiently long update sequence). This constitutes a *dramatic* improvement over the update time of [7] as stated in Theorem 1.1. As with most existing randomized dynamic algorithms, both Theorems 1.1 and 1.2 hold only when the adversary deciding the next update is *oblivious* to the past random choices made by the algorithm. We emphasize that, unlike several related results in the literature–including the previous result for $(\Delta + 1)$-coloring [7], our bound holds also with high probability.

Theorem 1.2. *There is a randomized algorithm for maintaining a $(\Delta + 1)$-coloring in a dynamic graph that, given any sequence of t updates, takes total time $O(n \log n + n\Delta + t)$ in expectation and with high probability. The space usage is $O(n\Delta + m)$, where m is the maximum number of edges present at any time. For $t = \Omega(n \log n + n\Delta)$, we obtain $O(1)$ amortized update time in expectation and with high probability.*

To provide a very quick explanation of our bound: the factor of $O(n\Delta)$ comes from our data structure for maintaining free colors in Lemma 2.1, and the factor of $O(n \log n)$ comes from Lemma 3.14 and 3.15 in our analysis.

Previous work: We start with a high level overview of the dynamic algorithm in [7]. They maintain a *hierarchical partition* of the node-set V into $O(\log \Delta)$ levels. Let $\ell(v) \in \{1, \ldots, \log \Delta\}$ denote the *level* of a node $v \in V$. For every edge $(u, v) \in E$, say that u is a *same-level-neighbor*, *down-neighbor* and *up-neighbor* of v respectively iff $\ell(u) = \ell(v)$, $\ell(u) < \ell(v)$ and $\ell(u) \geq \ell(v)$. The following invariant is maintained.

Invariant 1.3. *Each node $v \in V$ has $\Omega(2^{\ell(v)})$ down-neighbors and $O(2^{\ell(v)})$ same-level neighbors.*

In order to ensure that Invariant 1.3 holds, the nodes need to keep changing their levels as the input graph keeps getting updated via a sequence of edge insertions/deletions. It is important to note that the subroutine in charge of maintaining this invariant is *deterministic* and has $O(\log \Delta)$ amortized update time.

The algorithm in [7] uses a separate (randomized) subroutine to maintain a proper $(\Delta + 1)$-coloring in the input graph, on top of the hierarchical partition. To appreciate the main intuition behind this *recoloring subroutine*, consider the insertion of an edge (u, v) at some time-step τ, and suppose that both u and v had the same color just before this insertion. Pick any arbitrary endpoint $x \in \{u, v\}$. The algorithm picks a new color for x as follows. Let $C_x \subseteq C$ denote the subset of colors that satisfy the following property at time-step τ: A color $c \in C$ belongs to C_x iff (a) no up-neighbor of x has color c, and (b) at most one down-neighbor of x has color c. Since the node x has at most Δ neighbors and the palette C consists of $\Delta + 1$ colors, a simple counting argument (see the proof of Lemma 3.2) along with Invariant 1.3 implies that the size of the set C_x is at least $\Omega(2^{\ell(x)})$. Furthermore, using appropriate data structures, the set C_x can be computed in time proportional to the number of down-neighbors and same-level neighbors of x, which is at most $O(2^{\ell(x)})$ by

Invariant 1.3. The algorithm picks a color c' uniformly at random from the set C_x, and then recolors x by assigning it the color c'. By definition of the set C_x, at most one neighbor (say, y) of x has the color c', and, furthermore, if such a neighbor y exists then $\ell(y) < \ell(x)$. If the down-neighbor y exists, then we recursively recolor y in the same manner. Note that this entire procedure leads to a *chain* of recolorings. However, the levels of the nodes involved in these successive recolorings form a strictly decreasing sequence. Thus, the total time taken by the subroutine to handle the edge insertion is at most $\sum_{\ell=1}^{\ell(x)} O(2^{\ell}) = O(2^{\ell(x)})$.

Now comes the most crucial observation. Note that each time the algorithm recolors a node x, it picks a new color uniformly at random from a set of size $\Omega(2^{\ell(x)})$. Thus, intuitively, if the adversary deciding the update sequence is oblivious to the random choices made by the algorithm, then in expectation at least $\Omega(2^{\ell(x)}/2) = \Omega(2^{\ell(x)})$ edge insertions incident on x should take place before we encounter a *bad event* (where the other endpoint of the edge being inserted has the same color as x). The discussion in the preceding paragraph implies that we need $O(2^{\ell(x)})$ time to handle the bad event. Thus, overall we get an amortized update time of $O(1)$ in expectation.

Our contribution: To summarize, the algorithm in [7] has two components – (1) a deterministic subroutine for maintaining the hierarchical partition which takes $O(\log \Delta)$ amortized update time, and (2) a randomized subroutine for maintaining a proper $(\Delta + 1)$-coloring which takes $O(1)$ amortized update time. The analysis of the amortized update time of the first subroutine is done via an intricate potential function, and it is not clear if it is possible to improve the update time of this subroutine to $O(1)$.

To get an overall update time of $O(1)$, our algorithm merges these two components together in a very careful manner. Our starting point is to build on the high-level strategy used for maximal matching in [30]. Suppose that we decide to recolor a node x during the course of our algorithm (either due to the insertion of an edge incident on it, or because one of its up-neighbors took up the same color as x while recoloring itself). Let $\ell(x)$ be the current level of x. We first check if the number of down-neighbors of x is $\Omega(3^{\ell(x)})$. If the answer is yes, then we move up the node x to the minimum level $\ell'(x) > \ell(x)$ where the number of its down-neighbors becomes $\Theta(3^{\ell'(x)})$, following which we recolor the node x in the same manner as in [7]. In contrast, if the answer is no, then we find a new color for x that does not conflict with any of its neighbors and move the node x down to the smallest possible level. Thus, in our algorithm, the hierarchical partition itself is determined by the random choices made by the nodes while they recolor themselves. This makes the analysis of our algorithm significantly more challenging than that of [7], as Invariant 1.3 is no longer satisfied all the time.

Furthermore, our analysis is more challenging than that of [30] in one central aspect, which we discuss next. As mentioned, due to the oblivious adversary assumption, at least $\Omega(3^{\ell(x)}/2) = \Omega(3^{\ell(x)})$ edge insertions incident on node x are expected to occur before a *bad event* is encountered, i.e., when the other endpoint x' of the edge being inserted has the same color as x. Importantly, the color of that other endpoint x' at the time of that edge insertion (x, x') might have been chosen *after* the color of x was chosen, which may create dependencies between the (random variables corresponding to the) colors of x and x'. A similar reasoning was applied to the maximal matching problem [4, 30]; if the matched edge incident on a node x, denoted by (x, x'), was sampled uniformly at random among $\Omega(3^{\ell(x)})$ edges incident on x, then $\Omega(3^{\ell(x)})$ edge deletions incident on x are expected to occur (among the sampled ones) before a bad event is encountered, where the bad event here is that the deleted edge on x is its matched edge (x, x'). There is an inherent difference, however, between these two bad events. In the maximal matching problem, the time step of the bad event is fully determined by the adversarial updates that occur after the creation of that matched edge (under the oblivious adversary assumption), and in particular it is independent of future

random choices made by the algorithm. On the other hand, in our coloring problem the time step of the bad event may depend on random choices made by the algorithm after the random color of x has been chosen, due to nodes that become neighbors of x in the future and whose colors are chosen after x's color has been chosen. Thus, we must cope with subtle conditional probability issues that did not effect the analysis in [4, 30]. Note that in our analysis, the value of Δ is the maximum Δ over the course of the edge updates. The main difficulty with getting our running time for Δ that is the maximum degree of the current graph is that a single update may decrease the maximum degree of the current graph by 1; and so every vertex which is colored with the $(\Delta+1)$-th color needs to be recolored in our algorithm and recoloring all such nodes may be expensive in total. Specifically, we think it may be possible to modify our algorithm to obtain a $(\deg(v)+1)$-coloring where $\deg(v)$ is the current degree of vertex v, in which case, the returned coloring will trivially be a $(\Delta_{current}+1)$-coloring. So far no work has obtained such a dynamic $(\deg(v)+1)$-coloring in $O(1)$ amortized running time; it is an interesting open question whether there exists a dynamic algorithm that maintains a $(\deg(v)+1)$-coloring for all vertices $v \in V$ in the input graph in $O(1)$ amortized time per update. Furthermore, it is an open question to obtain $(\Delta_{current}+1)$-coloring *with high probability* in $O(1)$ amortized running time.

Independent work: Independently of our work, Henzinger and Peng [19] have obtained an algorithm for $(\Delta+1)$-vertex coloring with $O(1)$ *expected amortized update time*. Note that our work achieves $(\Delta+1)$-vertex coloring with $O(1)$ amortized update time not only in expectation, but also with *with high probability*.

2 OUR ALGORITHM

Consider a graph $G=(V,E)$ with $|V|=n$ nodes that is changing via a sequence of *updates* (edge insertions and deletions). The graph initially starts off as empty (containing n vertices and no edges). Let $\Delta > 0$ be a fixed integer such that the maximum degree of any node in the dynamic graph G is always upper bounded by Δ. In other words, Δ represents the maximum degree that any vertex can take throughout the update sequence. Let $C=\{1,\ldots,\Delta+1\}$ denote a palette of $\Delta+1$ colors. Our algorithm will maintain a proper $\Delta+1$-coloring $\chi : V \to C$ in the dynamic graph G.

A hierarchical partition of the node-set V: Fix a parameter $L = \lceil \log_3(n-1) \rceil - 1$. Our dynamic algorithm will maintain a hierarchical partition of the node-set V into $L+2$ distinct *levels* $\{-1,0,\ldots,L\}$. We let $\ell(v) \in \{-1,0,\ldots,L\}$ denote the level of a given node $v \in V$. The levels of the nodes will vary over time. Consider any edge $(u,v) \in E$ in the dynamic graph G at any given point in time: We say that u is an *up-neighbor* of v iff $\ell(u) \geq \ell(v)$, and a *down-neighbor* of v iff $\ell(u) < \ell(v)$.

Notations: Fix any node $v \in V$. Let $\mathcal{N}_v = \{u \in V : uv \in E\}$ denote the set of neighbors of v. Furthermore, let $C_v^+ = \{c \in C : c = \chi(u) \text{ for some } u \in \mathcal{N}_v \text{ with } \ell(u) \geq \ell(v)\}$ denote the set of colors assigned to the up-neighbors of v. We say that $c \in C$ is a *blank* color for v iff no neighbor of v currently has the color c. Similarly, we say that $c \in C$ is a *unique* color for v iff $c \notin C_v^+$ *and* exactly one down-neighbor of v currently has the color c. C_v, as defined before,[3] then, consists of the blank and unique colors of v. Finally, for every $\ell \in \{-1,\ldots,L\}$, we let $\phi_v(\ell) = |\{u \in \mathcal{N}_v : \ell(u) < \ell\}|$ denote the number of neighbors of v that currently lie below level ℓ. We are now ready to describe our dynamic algorithm.

Preprocessing: In the beginning, the input graph $G=(V,E)$ has an empty edge-set, i.e., $E=\emptyset$, and the algorithm starts with any arbitrary coloring $\chi : V \to C$. All the relevant data structures

[3] C_v denotes the subset of colors that satisfy the following property at timestep τ: A color $c \in C$ belongs to C_v iff (a) no up-neighbor of v has color c and (b) at most one down-neighbor of v has color c.

are initialized. Subsequently, the algorithm handles the sequence of updates to the input graph in the following manner.

Handling the deletion of an edge: Suppose that an edge (u, v) gets deleted from G. Just before this deletion, the coloring $\chi : V \to C$ maintained by the algorithm was proper (no two adjacent nodes had the same color). So the coloring χ continues to remain proper even after the deletion of the edge. So the deletion of an edge does *not* lead to any change in the levels of the nodes and the coloring maintained by the algorithm.

Handling the insertion of an edge: This procedure is described in Figure 1. Suppose that an edge (u, v) gets inserted into G. If, just before this insertion, we had $\chi(u) \neq \chi(v)$, then we call this insertion *conflict-less*, and otherwise *conflicting*. In case of a conflict-less insertion, the coloring χ continues to remain proper even after insertion of the edge. In this case, the insertion does *not* lead to any change in the levels of the nodes or the colors assigned to them. Otherwise, we pick the endpoint $x \in \{u, v\}$ that was most recently recolored and call the subroutine `recolor`(x). Such a choice of which vertex to recolor is crucial for our proof of the running time. This call to `recolor`(x) changes the color assigned to x and it might also change the level of x. However, there is a possibility that the new color assigned to x might be the same as the color of (at most one) down-neighbor of x. If this is the case, then we go to that neighbor of x it conflicts with, and keep repeating the same process until we end up with a proper coloring in G.

Procedure `recolor`(x) (see Figure 2), depending on whether $\phi_x(\ell(x) + 1) < 3^{\ell(x)+2}$ or not, calls one of the procedures `det-color`(x) and `rand-color`(x).

`det-color`(x): This subroutine first picks a *blank* color (say) c for the node x. By definition no neighbor of x has the color c. It now recolors the node x by setting $\chi(x) \leftarrow c$. Finally, it moves the node x down to level -1, by setting $\ell(x) \leftarrow -1$. It then updates all the relevant data structures.

`rand-color`(x): This subroutine works as follows. Let $\ell = \ell(x)$ be the level of the node x when this subroutine is called. Step 04 in Figure 2 implies that at that time we have $\phi_x(\ell + 1) \geq 3^{\ell+2}$. It identifies the *minimum* level $\ell' > \ell$ where $\phi_x(\ell' + 1) < 3^{\ell'+2}$. Such a level ℓ' must exist because $\phi_x(L + 1) \leq (n - 1) < 3^{L+2}$. The subroutine then moves the node x up to level ℓ', by setting $\ell(x) \leftarrow \ell'$, and updates all the relevant data structures. After this step, the subroutine computes the set $C_x \subseteq C$ of colors that are either blank or unique for x, next called *palette*. It picks a color $c \in C_x$ uniformly at random, and recolors the node x with color c, by setting $\chi(x) \leftarrow c$. It then updates all the relevant data structures. If c happens to be a blank color for x, then no neighbor of x has the same color as c. In other words, this recoloring of x does *not* lead to any new conflict. Accordingly, in this case the subroutine returns NULL. Otherwise, if c happens to be an unique color for x, then by definition exactly one down-neighbor (and zero up-neighbors) of x also has color c. Let this down-neighbor be y. In other words, the recoloring of x creates a new *conflict* along the edge (x, y), and we need to recolor y to ensure a proper coloring. Thus, in this case the subroutine returns the node y.

```
01. If χ(u) = χ(v), Then
02.     Let x ∈ {u, v} be the endpoint was most recently recolored.
03.     While x ≠ NULL:
04.         x ← recolor(x).
```

Fig. 1. Handling the insertion of an edge uv.

It is not difficult to come up with suitable data structures for the algorithm described above such that the following result holds (more details in Appendix A). Due to the complexity of the

```
01. If φ_x(ℓ(x) + 1) < 3^{ℓ(x)+2}, Then
02.     det-color(x).
03.     Return NULL.
04. Else :
05.     y ← rand-color(x).
06.     Return y.
```

Fig. 2. recolor(x).

data structures from the need to maintain many low-level details, we defer the full details of such structures to Appendix A so as not to interrupt the core ideas and analysis in this paper. However, we describe the main functionalities of the data structures here as is necessary in our main analysis.

Lemma 2.1. *There is an implementation of the above dynamic algorithm such that:*

(1) *The preprocessing time is $O(\Delta n)$;*
(2) *The space usage is $O(\Delta n + m)$, where m is the maximum number of edges present at any time;*
(3) *Each deletion and conflict-less insertion takes $O(1)$ time deterministically;*
(4) *Procedure* det-color(x) *takes time $O(3^{\ell(x)})$;*
(5) *Procedure* rand-color(x) *takes time $O(3^{\ell'(x)})$ where $\ell'(x) > \ell(x)$ is the new level of node x at the end of the procedure.*

Proof. We now justify the five claims made in the statement of the lemma. A full, detailed implementation section of the data structures can be found in Appendix A.1.

(1) We initialize a *dynamic* array $\mathcal{U}_v$ for each vertex v that contains $O(\log n)$ entries (specifically, let L be the set of non-empty levels for v, the dynamic array contains $O(L)$ entries) that stores the up-neighbors of v. Each index of the array $\mathcal{U}_v$ contains a pointer to a linked list containing the up-neighbors of v at that level. For example, suppose that w is an up-neighbor of v at level i. Then, the i-th entry of $\mathcal{U}_v$ contains a linked list which contains w. We initialize another linked list $\mathcal{D}_v$ which contains the down-neighbors of v. Furthermore, we initialize two linked lists, C_v^+ and C_v. C_v^+ contains exactly one copy of each color held by up-neighbors stored in $\mathcal{U}_v$. $C \backslash C_v^+$ then represents the colors of the down-neighbors stored in $\mathcal{D}_v$ that are not in C_v^+ and the blank colors. The palette C_v containing the unique and blank colors of v can thus be computed from $C \backslash C_v^+$. Each $\mathcal{U}_v$ has size $O(1)$ initially when there are no edges (see Appendix A for details); C_v^+ and C_v each has size $O(\Delta)$; and $\mathcal{D}_v$ is initially empty. Thus, the preprocessing time necessary to initialize these structures is $O(\Delta n)$. [4] More details on these structures can be found in Section A.1.
(2) The total space used by $\mathcal{D}_v$ for all v is $O(m)$ since $\mathcal{D}_v$ for vertex v stores at most the number of neighbors of v. All other data structures are initialized during preprocessing. Therefore, the space cost of the other data structures is $O(\Delta n)$. For each edge $e = (u, v)$, we maintain pointers that represent e between copies of u and v in the various data structures. Refer to Section A.1 for a detailed description of the pointer management.
(3) Deleting an edge uv requires deleting u from $\mathcal{U}_v$ and v from $\mathcal{D}_u$ (or vice versa). Inserting an edge uv requires inserting u into $\mathcal{U}_v$ and v into $\mathcal{D}_u$ (or vice versa). The colors for u and v can be moved in between C_v^+ and C_v and between C_u^+ and C_u via a set of pointers connecting the colors to the vertices. Refer to Fig. 3, 4 and Section A.3 for a detailed description of these elementary operations. The total cost of these operations is then $O(1)$.

[4] We require such a list of blank and unique colors for each vertex v in order to ensure our running time. It is an interesting open question whether we can remove the need for such lists of blank and unique colors.

(4) In this procedure `det-color`(v), the level of v is set deterministically to -1 and the color for v is chosen deterministically from its set of blank colors. In this case, all the data structures of vertices in levels $[-1, \ell(v)]$ (where $\ell(v)$ is the old level of v) must be updated with the new level of v. Due to the existence of pointers in between vertices and its neighbors in $\mathcal{D}_v$ and $\mathcal{U}_v$ in all the data structures, the cost of updating each individual neighbor is $O(1)$. To update the colors of the data structures requires following $O(1)$ pointers for each $w \in \mathcal{D}_v$. By the definition of `recolor`(v) (which calls `det-color`(v)), $\phi_v(\ell(v)+1) < 3^{\ell(v)+2}$. Hence, there are $O(3^{\ell(v)})$ neighbors in levels $[\ell'(v), \ell(v)]$ to update and the cost of the procedure is $O(3^{\ell(v)})$. Refer to Section A.1 and Fig. 7 for a complete description of this procedure.

(5) Since $\ell'(v) > \ell(v)$, all the data structures of vertices in levels $[\ell(v), \ell'(v)]$ must be updated with the new level of v. The data structures can be updated in the same way as given above. Since $\ell'(v) > -1$ (it must be, by definition of the procedure), then, $\phi_v(\ell'(v)+1) < 3^{\ell'(v)+2}$. Hence, this procedure takes $O(3^{\ell'(v)})$ time. Refer to Section A.1 and Fig. 8 for a complete description of this procedure.

□

Again, a complete, detailed description of our data structures (with pseudocode) can be found in Appendix A.

3 ANALYSIS

We assume that our graph is empty at the end, meaning no edges exist on the graph after we perform all the updates in our update sequence. To ensure we end with an empty graph, we append additional edge deletions at the end of the original update sequence. Since we begin with an empty graph, this at most doubles the number of updates in our update sequence, but simplifies our analysis. Because edge deletions will never cause a recoloring of any vertex and the number of updates increases by at most a factor of 2, an amortized runtime bound of our algorithm with respect to the new update sequence will imply the same (up to a factor of 2) amortized bound with respect to the original sequence. We now show that our dynamic algorithm maintains the following invariant.

Invariant 3.1. *Consider a vertex v at level $\ell(v) \geq 0$ at a given point of time τ. When v was most recently recolored prior to τ, it chose a color uniformly at random from a palette of size at least $3^{\ell(v)+1}/2 + 1$. Furthermore, at that time v has at least $3^{\ell(v)+1}$ down-neighbors. For $\ell(v) = -1$, the color of v is set deterministically.*

Lemma 3.2. *Invariant 3.1 holds for all vertices at the beginning of each update.*

Proof. During the preprocessing step the color of each node v is set deterministically to some arbitrary color and $\ell(v) = -1$. Hence the claim holds initially. The color of v changes only due to a call to `recolor`(v). Let $\ell(v)$ and $\ell'(v)$ denote the level of v at the beginning and end of this call. If `recolor`(v) calls `det-color`(v), the color of v is set determinstically and $\ell'(v) = -1$. Hence the invariant holds. Otherwise, `recolor`(v) invokes `rand-color`(v). The latter procedure sets $\ell'(v)$ to the smallest value (larger than $\ell(v)$) such that $\phi_v(\ell'(v)+1) < 3^{\ell'(v)+2}$. Recall that $\phi_v(\ell)$ is the number of neighbors of v of level smaller than ℓ. This implies that the number of down-neighbors of v (at level $\ell'(v)$) are $\phi := \phi_v(\ell'(v)) \geq 3^{\ell'(v)+1}$.

It is then sufficient to argue that the palette used by `rand-color`(v) has size at least $\phi/2 + 1$. We use exactly the same argument as in [7]. One has $|C \setminus C_v^+| = (\Delta+1) - |C_v^+|$ since C_v^+ contains exactly one copy of each color occupied by an up-neighbor of v. Since the degree of any vertex is at most Δ and the number of down-neighbors of v is ϕ, $|C_v^+| \leq (\Delta - \phi)$ since the number of colors occupied

by the up-neighbors is at most the number of up-neighbors. Then, $|C \setminus C_v^+| = (\Delta+1) - |C_v^+| \geq (\Delta+1) - (\Delta - \phi) = \phi + 1$, where equality holds when v has degree Δ and up-neighbors of v all have distinct colors. For any color $c \in C_v$ that is occupied by at most one down-neighbor of v, c is a blank or unique color. Let x be the number of down-neighbors of v that occupy a unique color. Then, the size of v's palette is at least $|C_v| \geq |C \setminus C_v^+| - (\phi - x)/2$; this is due to the fact that there can be at most $(\phi - x)/2$ colors that are occupied by at least *two* down-neighbors of v. Then, $|C_v| \geq |C \setminus C_v^+| - (\phi - x)/2 \geq 1 + |\phi| - (\phi - x)/2 \geq \phi/2 + 1$. □

Let t be the total number of updates. Excluding the preprocessing time, the running time of our algorithm is given by the cost of handling insertions and deletions. By Lemma 2.1-3, the total cost of deletions and insertions that do not cause conflicts is $O(t)$. We thus focus on insertions that cause conflicts. Modulo $O(1)$ factors, the total cost of the latter insertions is bounded by the total cost of the calls to `recolor`$(\cdot)$.

Epochs: It remains to bound the total cost of the calls to `recolor`$(\cdot)$. To that aim, and inspired by [4], we introduce the following notion of epochs. An epoch $\mathcal{E}$ is associated with a node $v = v(\mathcal{E})$, and consists of any maximal time interval in which v does not get recolored. So $\mathcal{E}$ starts with a call to `recolor`(v), and ends immediately before the next call to `recolor`(v) is executed. Note that even if v gets recolored with the same color that it occupied before, the epoch still ends and a new epoch begins. Observe that there are potentially multiple epochs associated with the same node v. Notice that by construction, during an epoch $\mathcal{E}$ the level and color of $v(\mathcal{E})$ does not change: we refer to that level and color as $\ell(\mathcal{E})$ and $\chi(\mathcal{E})$, resp. By $\mathcal{E}_\ell$ we denote the set of epochs at level ℓ. We define the *cost* $c(\mathcal{E})$ of an epoch $\mathcal{E}$ as the time spent by the call to `recolor`$(v(\mathcal{E}))$ that starts it, and then we charge the cost of every epoch $\mathcal{E}$ at level $\ell(\mathcal{E}) = -1$ to the previous epoch involving the same node $v(\mathcal{E})$.

Lemma 3.3. *Excluding the preprocessing time, the total running time of the dynamic algorithm is given by:* $O(\sum_\ell \sum_{\mathcal{E}\in\mathcal{E}_\ell} c(\mathcal{E})) = O(\sum_\ell |\mathcal{E}_\ell| \cdot 3^{\ell(\mathcal{E})})$.

Proof. By the above discussion and Lemma 2.1 (points 4-5), the cost of any epoch $\mathcal{E}$ is given by $c(\mathcal{E}) = O\left(3^{\ell(\mathcal{E})}\right)$. The claim follows. □

A classification of epochs: It will be convenient to classify epochs as follows. An epoch $\mathcal{E}$ is *final* if it is not concluded by a call to `recolor`$(v(\mathcal{E}))$. Thus, for a final epoch $\mathcal{E}$, $v(\mathcal{E})$ keeps the same color from the beginning of $\mathcal{E}$ till the end of all the updates. Otherwise $\mathcal{E}$ is *terminated*. A terminated epoch $\mathcal{E}$, $v = v(\mathcal{E})$, terminates for one of the following possible events: (1) some edge (u, v) is inserted, with $\chi(u) = \chi(v)$, hence leading to a call to `recolor`(v); (2) a call to `recolor`(w) for some up-neighbor w of v forces a call to `recolor`(v) (without the insertion of any edge incident to v). We call the epochs of the first and second type *original* and *induced*, resp. In the second case, we say that the epoch $\mathcal{E}'$ that starts with the recoloring of w *induces* $\mathcal{E}$.

Lemma 3.4. *The total cost of induced epochs is (deterministically) at most* $O(1)$ *times the total cost of original and final epochs.*

Proof. Let us construct a directed *epoch graph*, with node set the set of epochs, and a directed edge $(\mathcal{E}, \mathcal{E}')$ iff $\mathcal{E}'$ induced $\mathcal{E}$. Notice that, for any edge $(\mathcal{E}, \mathcal{E}')$ in the epoch graph, $\ell(\mathcal{E}') > \ell(\mathcal{E})$. Observe also that this graph consists of a collection of disjoint, directed paths starting at original, induced, or final epochs and ending at original and final epochs. Let us charge the cost of each induced epoch $\mathcal{E}$ to the root $r(\mathcal{E})$ of the corresponding path in the epoch graph. All the cost is charged to original and final epochs, and the cost charged to one epoch $\mathcal{E}$ of the latter type is at most $\sum_{\ell<\ell(\mathcal{E})} O(3^\ell) = O(3^{\ell(\mathcal{E})})$. The claim follows. □

Lemma 3.5. *Given any sequence of t updates, the total cost of final epochs is (deterministically) $O(t)$.*

Proof. By Invariant 3.1, for any final epoch $\mathcal{E}$, $v = v(\mathcal{E})$ and $\ell = \ell(\mathcal{E})$, v must have at least $3^{\ell+1}$ down-neighbors at the beginning of $\mathcal{E}$. Since by assumption at the end of the process the graph is empty, there must be at least $3^{\ell+1}$ deletions with one endpoint being v during $\mathcal{E}$. By charging the $O(3^{\ell})$ cost of $\mathcal{E}$ to the later deletions, and considering that each deletion is charged at most twice, we achieve a average cost per deletion in $O(1)$, hence a total cost in $O(t)$. □

A classification of levels: Recall that $\mathcal{E}_\ell$ denotes the set of epochs at level ℓ. We now classify the levels into 3 types, as defined below.

- A level ℓ is *induced-heavy* iff at least 1/2-fraction of the epochs in $\mathcal{E}_\ell$ are induced.
- A level ℓ is *final-heavy* iff (a) it is not induced-heavy *and* (b) at least 1/8-fraction of the epochs in $\mathcal{E}_\ell$ are final.
- A level ℓ is *original-heavy* iff it is neither induced-heavy nor final-heavy. Note that if a level ℓ is original-heavy, then $\geq 3/8$-fraction of the epochs in $\mathcal{E}_\ell$ are original.

Henceforth, we say that an epoch is induced-heavy, final-heavy and original-heavy if it respectively belongs to an induced-heavy, final-heavy and original-heavy level. We use the term "*cost of a level ℓ*" to refer to the total cost of all the epochs at level ℓ.

Lemma 3.6. *The total cost of all the induced-heavy levels is (deterministically) at most $O(1)$ times the total cost of all the original-heavy and final-heavy levels.*

Proof. We perform charging level by level, starting from the lowest level -1. Given a level ℓ, if it is either original-heavy or final-heavy then we do nothing. Otherwise, we match each epoch $\mathcal{E} \in \mathcal{E}_\ell$ that is either original or final with some distinct induced epoch $\mathcal{E}' \in \mathcal{E}_\ell$. We next charge the cost of $\mathcal{E}$ (as obtained from the proof of Lemma 3.4) to $\mathcal{E}'$. Finally, we charge the cost of $\mathcal{E}'$ to some original or final epoch $\mathcal{E}''$ at a higher level following the same scheme as in the proof of Lemma 3.4. At the end of this process, only original and final epochs at the original-heavy and final-heavy levels are charged. By an easy induction, when we start processing level ℓ the total charge on an original or final epoch at level ℓ coming from the lower levels is at most $\sum_{\ell' < \ell} O(3^{\ell'}) = O(3^{\ell-1})$. The lemma follows. □

Lemma 3.7. *Given any sequence of t updates, the total cost of all the final-heavy levels is (deterministically) at most $O(t)$.*

Proof. Note that at each final-heavy level at least 1/8-fraction of the epochs are final. Thus, the cost of the other epochs given by Lemma 3.3 can be charged to the final epochs in the layer. There is already $O(3^{\ell})$ cost charged to the final epoch; thus, the additional cost of other epochs in the same level only increases this cost by an 8-factor. The proof now follows from Lemma 3.5. □

Corollary 3.8. *The total cost of the dynamic algorithm, excluding the preprocessing time and a term $O(t)$, is $O(1)$ times the total cost of the original-heavy levels.*

Proof. It follows from the above discussion and Lemmas 3.6, 3.7. □

Bounding the Cost of the Original-Heavy Levels: It now remains to bound the total cost of the original-heavy levels. Recall that at each original-heavy level, at least 3/8-fraction of the epochs are original. Thus, using a simple charging scheme, the task of bounding the total cost of all the original-heavy levels reduces to bounding the total cost of all the original epochs in these levels. At this point, it is tempting to use the following argument. By Invariant 3.1, for each epoch $\mathcal{E}$, $\ell = \ell(\mathcal{E})$, the corresponding color $\chi(\mathcal{E})$ is chosen uniformly at random in a palette of size at least $3^{\ell(\mathcal{E})}/2 + 1$. Therefore, if $\mathcal{E}$ is original, we expect to see at least $\Omega(3^{\ell})$ edge insertions having $v(\mathcal{E})$

as one endpoint before one such insertion causes a conflict with $v(\mathcal{E})$. This would imply an $O(1)$ amortized cost per edge insertion. The problem with this argument is that, conditioning on an epoch $\mathcal{E}$ being original, modifies a posteriori the distribution of colors taken at the beginning of $\mathcal{E}$. For example, the choice of certain colors might make more likely that the considered epoch is induced rather than original. To circumvent this issue, we need a more sophisticated argument that exploits the fact that we are considering original epochs in original-heavy levels only.

We define the *duration* $dur(\mathcal{E})$ of an epoch $\mathcal{E}$, $v = v(\mathcal{E})$, as the number of edge insertions of type (u, v) that happen during $\mathcal{E}$, plus possibly the final insertion that causes the termination of $\mathcal{E}$ (if $\mathcal{E}$ is original). We also define a critical notion of *pseudo-duration* $psdur(\mathcal{E})$ of $\mathcal{E}$ as follows. Let $(v, u_1), \ldots, (v, u_q)$ be the subsequence of insertions of edges incident to v in the input sequence after the creation of $\mathcal{E}$. For each u_i in the sequence of updates, let $\chi(u_i)$ represent the color of u_i *right before the creation* of $\mathcal{E}$. Consider the sequence of colors $\chi(u_1), \ldots, \chi(u_q)$. Remove from this sequence all colors not in the palette C used by $\mathcal{E}$ to sample $\chi(\mathcal{E})$, and then leave only the first occurrence of each duplicated color. Let $\chi(1), \ldots, \chi(k)$ be the obtained subsequence of (distinct) colors. We assume that $\chi(1), \ldots, \chi(k)$ is a permutation of C (so that $k = |C|$), and otherwise extend it arbitrarily to enforce this property. We define $psdur(\mathcal{E})$ to be the index i such that $\chi(i) = \chi(\mathcal{E})$. In other words, $psdur(\mathcal{E})$ is equal to the number of distinct colors in $\chi(u_1), \ldots, \chi(u_j)$ that are also in C where $\chi(u_j)$ is the first occurrence of the color $\chi(\mathcal{E})$. Such a j exists since $j = i$ if the first occurrence of $\chi(\mathcal{E})$ is at index i.

Lemma 3.9. *For an original epoch $\mathcal{E}$, $psdur(\mathcal{E}) \leq dur(\mathcal{E})$ deterministically.*

Proof. Let (v, u_i) be the edge insertion that causes the termination of $\mathcal{E}$, so that $dur(\mathcal{E}) = i$. Let $j \leq i$ be the smallest index with $\chi(u_j) = \chi(u_i)$. Let C be the palette used by v to sample $\chi(u_i)$. The value of $psdur(\mathcal{E})$ equals the number of distinct colors in the set $\chi(u_1), \ldots, \chi(u_j)$ that are also in C. The latter number is clearly at most $j \leq i$. (In this proof we crucially used the following property: If the insertion of an edge (x, y) creates a conflict, in the sense that both x and y have the same color, then our algorithm changes the color of the node $z \in \{x, y\}$ that was *most recently recolored*.) □

We say that an epoch $\mathcal{E}$ is *short* if $psdur(\mathcal{E}) \leq \frac{1}{32e} 3^{\ell(\mathcal{E})}$, and *long* otherwise. The following critical technical lemma upper bounds the probability that an epoch is short.

Lemma 3.10. *An epoch $\mathcal{E}$ is short with probability at most $\frac{1}{16e}$, independently from the random bits used by the algorithm other than the ones used to sample $\chi(\mathcal{E})$.*

Proof. Let C be the palette from which $v = v(\mathcal{E})$ took its color $c = \chi(\mathcal{E})$ uniformly at random. Let us condition on all the random bits used by the algorithm prior to the ones used to sample $\chi(\mathcal{E})$. Notice that this fixes C and the permutation $\chi(1), \ldots, \chi(|C|)$ of C used for the definition of $psdur(\mathcal{E})$ (see the paragraph before Lemma 3.9). The random bits used after the sampling of $\chi(\mathcal{E})$ clearly do not affect $psdur(\mathcal{E})$. The probability that $psdur(\mathcal{E}) = i$, i.e. $\chi(i) = \chi(\mathcal{E})$, is precisely $1/|C|$. The latter probability is deterministically at most $\frac{2}{3^{\ell(\mathcal{E})}}$ by Invariant 3.1. In particular, this upper bound holds independently from the random bits on which we conditioned earlier. The claim then follows since

$$\mathbf{P}[\mathcal{E} \text{ is short}] = \mathbf{P}\left[psdur(\mathcal{E}) \leq \frac{3^{\ell(\mathcal{E})}}{32e}\right] = \frac{3^{\ell(\mathcal{E})}}{32e} \cdot \frac{1}{|C|} \leq \frac{1}{16e}.$$

□

We next define some *bad* events, that happen with very small probability. Recall that $\mathcal{E}_\ell$ is the set of epochs at level ℓ. We define $\mathcal{E}_\ell^{short}$ (resp., $\mathcal{E}_\ell^{long}$) as the collection of all epochs in $\mathcal{E}_\ell$ that are short (resp., long).

Lemma 3.11. *Consider any $x \geq 0$, and let A_ℓ^x be the event that $|\mathcal{E}_\ell| > x$ and $|\mathcal{E}_\ell^{short}| \geq \frac{|\mathcal{E}_\ell|}{4}$. Then $\mathbf{P}(A_\ell^x) \leq \frac{4}{2^{x/2}}$.*

Proof. Fix two parameters q and j, with $j \geq q/4$, and consider any q level-ℓ epochs $\mathcal{E}^1, \ldots, \mathcal{E}^q$, ordered by their creation time. We argue that the probability that precisely j particular epochs $\mathcal{E}^{(1)}, \ldots, \mathcal{E}^{(j)}$ among these q are short is at most $\left(\frac{1}{16e}\right)^j$. Let $B^{(i)}$ be the event that $\mathcal{E}^{(i)}$ is short, $1 \leq i \leq j$. By a simple induction and Lemma 3.10, we have that $\mathbf{P}(B^{(i)} \mid B^{(1)} \cap B^{(2)} \cap \ldots B^{(i-1)}) \leq \frac{1}{16e}$. Consequently, we get: $\mathbf{P}(B^{(1)} \cap B^{(2)} \cap \ldots \cap B^{(j)}) = \mathbf{P}(B^{(1)}) \cdot \mathbf{P}(B^{(2)} \mid B^{(1)}) \cdot \ldots \cdot \mathbf{P}(B^{(j)} \mid B^{(1)} \cap B^{(2)} \cap \ldots \cap B^{(j-1)}) \leq \left(\frac{1}{16e}\right)^j$.

There are $\binom{q}{j}$ choices for the subsequence $\mathcal{E}^{(1)} \ldots \mathcal{E}^{(j)}$. Thus, we get: $\mathbf{P}[|\mathcal{E}_\ell| = q \cap |\mathcal{E}_\ell^{short}| = j] \leq \binom{q}{j}\left(\frac{1}{16e}\right)^j$. Since $\binom{q}{j} \leq (\frac{eq}{j})^j \leq (4e)^j$, we have $\binom{q}{j}\left(\frac{1}{16e}\right)^j \leq \frac{1}{4^j}$. Hence, $\mathbf{P}(A_\ell^x) = \sum_{q>x}\sum_{j=q/4}^{q} \mathbf{P}[|\mathcal{E}_\ell| = q \cap |\mathcal{E}_\ell^{short}| = j] \leq \sum_{q>x}\sum_{j=q/4}^{q} \frac{1}{4^j} \leq \sum_{q>x} \frac{4}{3} \cdot \frac{1}{2^{q/2}} \leq \frac{4}{2^{x/2}}$. □

Corollary 3.12. *For a large enough constant $a > 0$ and $x = 2a\log_2 n$, let A denote the event that A_ℓ^x happens for* some *level ℓ. Then $\mathbf{P}(A) = O(\frac{\log n}{n^a})$.*

Proof. It follows from Lemma 3.11 and the union bound over all levels ℓ. □

Lemma 3.13. *Let g be the number of level-ℓ epochs with duration $\geq \delta$, and IN_ℓ be the set of input insertions of edges incident to vertices at level ℓ. Then $g \leq 2|IN_\ell|/\delta$.*

Proof. Observe that for the duration of the concerned epochs, we consider only insertions in IN_ℓ. Furthermore, each such insertion can influence the duration of at most 2 such epochs. The claim follows by the pigeon-hole principle. □

Let $c(\mathcal{E}_\ell) = O(3^\ell \cdot |\mathcal{E}_\ell|)$ be the total cost of the epochs in level ℓ. We next relate the occurrence of event $\neg A_\ell^x$ to the value of the random variable $c(\mathcal{E}_\ell)$ for original epochs.

Lemma 3.14. *If $\neg A_\ell^x$ occurs and level ℓ is original-heavy, then $c(\mathcal{E}_\ell) = O(|IN_\ell| + 3^\ell x)$.*

Proof. If $|\mathcal{E}_\ell| \leq x$, then we clearly have $c(\mathcal{E}_\ell) = O(3^\ell x)$. For the rest of the proof, we assume that $|\mathcal{E}_\ell^{short}| < \frac{|\mathcal{E}_\ell|}{4}$, or equivalently: $|\mathcal{E}_\ell^{long}| \geq \frac{3}{4} \cdot |\mathcal{E}_\ell|$. Let $\mathcal{E}_\ell^* \subseteq \mathcal{E}_\ell$ be the set of original epochs at level ℓ. As the level ℓ is original-heavy, we have: $|\mathcal{E}_\ell^*| \geq \frac{3}{8} \cdot |\mathcal{E}_\ell|$. Since $|\mathcal{E}_\ell^{long}| \geq \frac{3}{4} \cdot |\mathcal{E}_\ell|$, applying the pigeon-hole principle we infer that at least $q \geq \frac{|\mathcal{E}_\ell|}{8}$ level-ℓ epochs are original and long at the same time. Specifically, we get: $|\mathcal{E}_\ell^* \cap \mathcal{E}_\ell^{long}| \geq \frac{1}{8} \cdot |\mathcal{E}_\ell|$, or equivalently: $|\mathcal{E}_\ell| \leq 8 \cdot |\mathcal{E}_\ell^* \cap \mathcal{E}_\ell^{long}|$.

Any epoch $\mathcal{E} \in \mathcal{E}_\ell^* \cap \mathcal{E}_\ell^{long}$ has duration $dur(\mathcal{E}) \geq psdur(\mathcal{E}) \geq \frac{3^\ell}{32e}$ by Lemma 3.9 and the definition of long epochs. Hence by applying Lemma 3.13 with $\delta = \frac{3^\ell}{32e}$, we can conclude that the number of such epochs is at most $\frac{2|IN_\ell|}{3^\ell/(32e)} = \frac{64e|IN_\ell|}{3^\ell}$. Hence, we get: $|\mathcal{E}_\ell| \leq 8 \cdot |\mathcal{E}_\ell^* \cap \mathcal{E}_\ell^{long}| \leq 8 \cdot \frac{64e}{3^\ell} \cdot |IN_\ell|$. The lemma follows if we multiply both sides of this inequality by the $O(3^\ell)$ cost charged to each epoch in $\mathcal{E}_\ell$. □

We are now ready to bound the amortized update time of our dynamic algorithm. Recall that t denotes the total number of updates.

Lemma 3.15. *For any fixed sequence of t updates, with probability $1 - O(\log n/n^a)$, the total running time of our algorithm is $O(t + an\log n + \Delta n)$.*

Proof. By Lemma 2.1-1, the preprocessing time is $O(\Delta n)$. The total cost of deletion and conflict-less insertions is $O(t)$, due to Lemma 2.1-3. Let us condition on the event $\neg A$, which happens with probability $1 - O(\log n/n^a)$ by Corollary 3.12. Then the total cost of the original-heavy levels is $O(\sum_\ell(|IN_\ell| + 3^\ell a \log n)) = O(t + n \log n)$ by Lemma 3.14. The lemma now follows from Corollary 3.8. □

In order to prove that the amortized update time of our algorithm is $O(1)$ in expectation, we also need the following upper bound on its worst-case total running time.

Lemma 3.16. *Our algorithm's total runtime is deterministically $O(tn^2 + n \log n + \Delta n)$.*

Proof. The preprocessing time is $O(n \log n + \Delta n)$ and the total cost of deletions and conflict-less insertions is $O(t)$ deterministically. Each conflicting insertion starts a sequence of calls to `recolor`$(\cdot)$ involving some nodes $w_1, \ldots, w_q$. A given node w can appear multiple times in the latter sequence. However, the sequence ends when some node w^* is moved to level -1, and in all other cases the level of w is increased by at least one. This means that the total cost associated with node w is $O(\sum_\ell 3^\ell) = O(n)$. The lemma follows by summing over the n nodes and the $O(t)$ insertions. □

Hence we can conclude:

Lemma 3.17. *The total expected running time of our algorithm is $O(t + n \log n + \Delta n)$.*

Proof of Lemma 3.17. When the event $\neg A$ happens, the total cost of the algorithm is $O(t + \Delta n)$ by Lemma 3.15. If instead the event A happens, then the cost is $O(tn^2 + \Delta n)$ by Lemma 3.16. However the latter event happens with probability at most $O(\frac{\log n}{n^a})$ by Corollary 3.12. So this second case adds $o(t)$ to the total expected cost for $a > 2$. □

We now have all the ingredients to prove the main theorem of this paper.

Proof of Theorem 1.2. Consider the dynamic algorithm described above. The space usage follows from Lemma 2.1-2 and the update time from Lemmas 3.15 and 3.17. □

A $(\Delta+1)$-COLORING UPDATE DATA STRUCTURES

In this section, we give a full detailed description of the data structures used by our dynamic algorithm.

The update algorithm is applied following edge insertions and deletions to and from the graph. In this section, we provide a complete description of the update data structures and algorithm. The pseudocode of this algorithm can be found in Appendix B. We begin with a description of the data structures and invariants that will be maintained by our algorithm. Throughout, we use the phrase *mutual pointers* between two elements a and b (i.e. specifically, we use the phrase "mutual pointers between a and b") to mean pointers from a to b and from b to a (hence the pointers are *mutual*).

A.1 Hierarchical Partitioning and Coloring Data Structures

Our algorithm maintains the following set of data structures which we divide into two groups: the data structures responsible for maintaining our hierarchical partitioning and the data structures used to maintain the set of colors associated with each vertex. Let C be the set of all $\Delta+1$ colors. The first group of data structures is a hierarchical partitioning of the vertices of the graph into different *levels* according to some procedures that maintain a set of invariants. A vertex at a level has some number of neighbors in other levels of the hierarchical partitioning structure. We refer to neighbors at the same or higher levels of the hierarchical partitioning structure as the *up-neighbors*. We refer to neighbors at lower levels of the hierarchical partitioning as the *down-neighbors*. Different

data structures will be used to maintain the colors of the down-neighbors and the colors of the up-neighbors of a vertex. We can obtain the palette, C_v, defined to consist of the blank and unique colors, by scanning through the list $C \setminus C_v^+$.

The second group of data structures deals with maintaining the colors of the vertices, inspired by the structures given in [7]. For the following data structures, we use logarithms in base 3 unless stated otherwise.

Let C be the set of all $\Delta + 1$ colors:

- **Hierarchical Partitioning:** We maintain the following data structures necessary for our hierarchical partitioning.

(1) For each vertex v:
  (a) $\mathcal{N}_v$: a linked list containing all neighbors of v.
  (b) $\mathcal{D}_v$: a linked list containing all down-neighbors of v.
  (c) $\mathcal{U}_v$: a dynamic array where each index corresponds to a distinct level $\ell \in \{0, \ldots, \log_3(n-1) - 1\}$. $\mathcal{U}_v[\ell]$ holds a level number, a pointer to the head of a non-empty doubly linked list containing all up-neighbors of v at level ℓ, and the size of the non-empty doubly linked list of neighbors. If this list is empty, then the corresponding pointer is not stored.

(2) For any vertex v and any neighbor u in $\mathcal{D}_v$, let $u_{\mathcal{D}_v}$ represent the copy of $u \in \mathcal{D}_v$, $v_{\mathcal{U}_u[\ell(v)]}$ be the copy of $v \in \mathcal{U}_u[\ell(v)]$, $v_{\mathcal{N}_u}$ be the copy of $v \in \mathcal{N}_u$, and, finally $u_{\mathcal{N}_v}$ be the copy of $u \in \mathcal{N}_v$. We maintain the following pairs of pointers, where for each pair, there exists a pointer from the first element in the pair to the second and vice versa: $(u_{\mathcal{D}_v}, v_{\mathcal{U}_u[\ell(v)]})$, $(u_{\mathcal{D}_v}, v_{\mathcal{N}_u})$, $(u_{\mathcal{D}_v}, u_{\mathcal{N}_v}\})$, $(v_{\mathcal{U}_u[\ell(v)]}, v_{\mathcal{N}_u})$, $(v_{\mathcal{U}_u[\ell(v)]}, u_{\mathcal{N}_v}\})$, and $(v_{\mathcal{N}_u}, u_{\mathcal{N}_v}\})$. In other words, there exists two pointers (one forwards and one backwards) between every pair of elements in $\{u_{\mathcal{D}_v}, v_{\mathcal{U}_u[\ell(v)]}, v_{\mathcal{N}_u}, u_{\mathcal{N}_v}\}$. The set of pointers means that given an edge insertion or deletion, we are able to quickly access the endpoints of the edge in each data structure once we locate one copy of an endpoint in memory.

(3) We define $\phi_v(\ell)$ to be the number of neighbors of v with level strictly lower than ℓ. We calculate the appropriate values for $\phi_v(\ell)$ as follows. For any level, $\ell' < \ell(v)$, we look through all neighbors stored in $\mathcal{D}_v$ (defined above) to calculate $\phi_v(\ell')$. For levels $\ell' \geq \ell(v)$, we use the sizes of the linked lists in $\mathcal{U}_v$ (defined above) to calculate $\phi_v(\ell')$.

- **Coloring:** We maintain the following data structures for our coloring procedures. These structures are similar to the structures used in [7].

(1) A static array χ of size $O(n)$ where $\chi[i]$ stores the *current* color of the i-th vertex.

(2) For each vertex v:
  (a) C_v^+: a doubly linked list of exactly one copy of each color occupied by vertices in $\mathcal{U}_v$. Each color contains a counter $\mu_v^+(c)$ counting the number of vertices in $\mathcal{U}_v$ that is colored color c.
  (b) The counters $\mu_v^+(c)$ are stored in a static array of size $\Delta + 1$ where index i contains the number of vertices in $\mathcal{U}_v$ that is colored with color i.
  (c) C_v: a doubly linked list of colors in $C \setminus C_v^+$ that are blank or unique.
  (d) A static array $\mathcal{P}_v$ of size $\Delta + 1$ containing mutual pointers (i.e. the pair of pointers from element a to element b and from element b to element a) to each color c in C_v or C_v^+ and to each of two additional nodes representing each color in C. Let i_c be the index of color c in $\mathcal{P}_v$. Suppose that $c \in C_v$. Let p_c and p_c^+ be the two additional nodes representing c. Then $\mathcal{P}_v[i_c]$ contains pointers to $c \in C_v$, p_c, and p_c^+. In addition, if $c \in C_v$, then it has mutual pointers to p_c. If, instead, $c \in C_v^+$, then it has mutual pointers to p_c^+ instead. In other words, p_c receives pointers from nodes in C_v (and has outgoing pointers to nodes

in C_v) and p_c^+ receives pointers from nodes in C_v^+ (and has outgoing pointers to nodes in C_v^+).

We define the set of *blank colors* for v to be colors in C_v which are not occupied by any vertex in $\mathcal{D}_v$. We define the set of *unique colors* of v to be colors in C_v which are occupied by at most one vertex in $\mathcal{D}_v$.

We now describe the pointers from the hierarchical partitioning structures to the coloring structures and vice versa.

- Each color c in C_v^+ has a pointer to p_c^+ and vice versa.
- Each color c' in C_v has a pointer to $p_{c'}$ and vice versa.
- Each vertex $u \in \mathcal{U}_v$ contains mutual pointers to the node p_c^+ representing its color in $\mathcal{P}_v$ that it is currently colored with. The color c is also in C_v^+ and has mutual pointers to p_c^+.
- Each vertex $u \in \mathcal{D}_v$ contains mutual pointers to p_c representing its corresponding color in $\mathcal{P}_v$. If its color is in C_v, then mutual pointers also exist between p_c and c in C_v.
- Each edge (u, v) contains two pointers, one to $u \in \mathcal{N}_v$ and one to $v \in \mathcal{N}_u$. u and v also contain pointers to edge (u, v).

Initial Data Structure Configuration, Time Cost, and Space Usage. There exist no edges in the graph initially; thus all vertices can be colored the same color. Such an arbitrary starting color is chosen. Before any edge updates are made, we assume that all vertices are on level -1, colored with the arbitrary starting color. Thus, all colors are also initially in C_v.

Before any edge insertions, the only structures that we initialize are an empty dynamic array $\mathcal{U}_v$ for each vertex v, the list of all colors C, and χ. When the first edge that contains vertex v as an endpoint is inserted, we initialize $\mathcal{N}_v$, $\mathcal{D}_v$, $\mathcal{U}_v$, C_v^+, C_v, μ_v^+ for v (as well as the associated pointers). The time for initializing these structures is $O(n\Delta)$ which means that the preprocessing time will result in $O(1)$ amortized time per update assuming $\Omega(n\Delta)$ updates.

The dynamic arrays in our data structure are implemented as follows. When the array contains no elements, we set the default size of the array to 8. Whenever the array is more than half full, we double the size of the array. Similarly, whenever the array is less than 1/4 full, we shrink the size of the array by half. Then, suppose we have an array which just shrank in size or doubled in size and the size of the array is L. We require at least either $L/4$ insertions or deletions to double or halve the size of the array again. Thus, the $O(L)$ cost of resizing the array can be amortized over the $L/4$ updates to $O(1)$ cost per update. Given that our algorithm is run on a graph which is initially empty, all dynamic arrays in our structure are initially empty and initialized to size 8.

We note a particular choice in constructing our data structures. In the case of $\mathcal{U}_v$, given our assumption of the number of updates, we can also implement $\mathcal{U}_v$ as a static array instead of a dynamic array. The maximum number of levels is bounded by $\log_3(n-1)+1$. Thus, if we instead implemented $\mathcal{U}_v$ as static arrays instead of dynamic arrays, the total space usage (and initialization cost) would be $O(n \log n)$, amortizing to $O(1)$ per update given $\Omega(n \log n)$ updates. There may be reasons to implement $\mathcal{U}_v$ as static arrays instead of dynamic dynamic arrays such as easier implementation of basic functions. However, we choose to use a dynamic array implementation for potential future work for the cases when the number of updates is $o(n \log n + n\Delta)$. The key property we can potentially take advantage of in using the dynamic array implementation is that the total space used (and the total time spent in initializing the data structure) is within a constant factor of the number of edges in the graph at any particular time.

Usefulness of the Pointers. Pointers between the various data structures used for the hierarchical partitioning and for maintaining the coloring allows for us to quickly update the state following an edge insertion or deletion. For example, when an edge uv is inserted or deleted, we get pointers to

$v \in \mathcal{N}_u$ and $u \in \mathcal{N}_v$, and through these pointers we delete all elements $u \in \mathcal{D}_v, v \in \mathcal{U}_u[\ell(v)], v \in \mathcal{N}_u, u \in \mathcal{N}_v$ and potentially move a color from C_u^+ to C_u. The exact procedure for handling edge deletions is described later.

A.2 Invariants

Our update algorithm and data structures maintain the following invariant (reproduced again here for readability).

Invariant A.1. *The following hold for all vertices:*

(1) *A vertex in level ℓ was last colored using a palette of size at least $(1/2)\,3^{\ell+1} + 1$. As a special case, a vertex in level -1 was last colored using a palette of size 1 (in other words, it was colored deterministically).*
(2) *The level of a vertex remains unchanged until the vertex is recolored.*

A.3 Edge Update Algorithm

We now describe the update algorithm in detail. The data structures are initialized as described in Section A.1. Then, edge updates are applied to the graph. Following an edge insertion or deletion, the procedure `handle-insertion`(u, v) or `handle-deletion`(u, v), respectively, is called. The descriptions of the insertion and deletion procedures are given below.

Procedures `handle-insertion`(u, v) *and* `handle-deletion`(u, v). `handle-deletion`(u, v) is called on an edge deletion uv. This case would not result in any need to recolor any vertices since a conflict will never be created. Thus, we update the relevant data structures in the obvious way (by deleting all relevant entries in all relevant structures); details of this set of deletions are given in the pseudocode in Fig. 4.

Procedure `handle-insertion`(u, v) is called on an edge insertion `handle-insertion`(u, v). The pseudocode for this procedure is given in Figure 3. If edge uv does not connect two vertices that are colored the same color (i.e. if the insertion is *conflict-less*), then we only need to update the relevant data structures with the inserted edge. Namely the vertices are added to the structures maintaining the neighbors of u and v. If u is on a higher level than v, then u is added to $\mathcal{U}_v$ and v is added to $\mathcal{D}_u$ (and vice versa). If u and v are on the same level, then u is added to $\mathcal{U}_v$ and v is added to $\mathcal{U}_u$. Furthermore, the colors that are associated with the vertices are moved in between the lists C_v^+ and C_v as necessary. See the pseudocode in Fig. 3 for exact details of these straightforward data structure updates.

In the case that edge uv connects two vertices of the same color (i.e. if the insertion is *conflicting*), we need to recolor at least one of these two vertices. We arbitrarily recolor *one* of the vertices u or v using procedure `recolor` (i.e. `recolor`(u) as given in the pseudocode in Fig. 6). Procedure `recolor` is the crux of the update algorithm and is described next.

Whenever a conflict is created following an edge insertion uv, procedure `recolor`(u) is called on one of the two endpoints. This procedure is described below.

Procedure `recolor`(v). The pseudocode for this procedure can be found in Figure 6. The procedure `recolor`(v) makes use of the level of v as well as the number of its down-neighbors to either choose a blank color deterministically to recolor v or to determine the palette from which to select a random color to recolor v. Recall that all vertices start in level -1 before any edges are inserted into the graph.

The procedure `recolor`(v) considers two cases:

- *Case 1:* $\phi_v(\ell(v)+1) < 3^{\ell(v)+2}$. In other words, the first case is when the number of down-neighbors and vertices on the same level as v is not much greater than $3^{\ell(v)+1}$. We show in the analysis that in this case, we can find the colors of all the neighbors in $\mathcal{D}_v$ and pick a color in C_v that does not conflict with any such neighbors (or the color that it currently has). Thus, we deterministically choose a blank color to recolor v, creating no further conflicts. The procedure to choose a blank color for v, `det-color`(v), is described in the following.
- *Case 2:* $\phi(\ell(v)+1) \geq 3^{\ell(v)+2}$. In this case, the number of down-neighbors and vertices on the same level as v is at least $3^{\ell(v)+2}$ and it will be too expensive to look for a blank color since we need to look at all neighbors in $\mathcal{D}_v$ to determine such a color and the size of $\mathcal{D}_v$ could be very large. Thus, we need to pick a random color from C_v to recolor v by running Procedure `rand-color`(v) as described below.

Procedures `det-color`(v) *and* `rand-color`(v). When called, the procedure `det-color`(v) starts by scanning the list C_v to find at least one blank color that we can use to color v. By the definition of $(\Delta+1)$-coloring, there must exist at least one blank color with which we can use to color v. We can deterministically find a blank color in the following way. The elements in C_v are stored in a doubly linked list. We start with the first element at the front of the list and scan through the list until we reach an element that does not have a pointer to a vertex in $\mathcal{D}_v$. We can determine whether a color $c \in C_v$ has a pointer to a vertex in $\mathcal{D}_v$ by following the pointer from c to p_c. From p_c, we can then determine whether any vertices in $\mathcal{D}_v$ are colored with c.

Let this first blank color be c_b. We assign color c_b to v, update $\chi(i_v)$ to indicate that the color of v is c_b, and update the lists C_w^+ and/or C_w of all $w \in \mathcal{D}_v$. To update all C_w^+ and C_w, we follow the following set of pointers:

(1) From $w \in \mathcal{D}_v$, follow pointers to reach $w \in \mathcal{N}_v$.
(2) From $w \in \mathcal{N}_v$, follow pointers to reach $v \in \mathcal{N}_w$.
(3) From $v \in \mathcal{N}_w$, follow pointers to reach $v \in \mathcal{U}_w[\ell(v)]$.
(4) Let c be the previous color of v as recorded in C_w^+. From $v \in \mathcal{U}_w[\ell(v)]$, follow pointers to reach $c \in C_w^+$.
(5) Decrement $\mu_w^+(c)$ by 1. Delete mutual pointers between v and p_c^+. If now $\mu_w^+(c) = 0$, remove c from C_w^+, append c to the end of C_w, delete mutual pointers between c and p_c^+, and add mutual pointers between c and p_c.
(6) Use $\mathcal{P}_w$ to find c_b in either C_w^+ or C_w. If $c_b \in C_w^+$, increment $\mu_w^+(c_b)$ by 1. Otherwise, if $c_b \in C_w$, remove c_b from C_w, append c_b to the end of C_w^+, increment $\mu_w^+(c_b)$ by 1, delete mutual pointers between c_b and p_{c_b}, and create mutual pointers between c_b and $p_{c_b}^+$. Create mutual pointers between $v \in \mathcal{U}_w[\ell(v)]$ and $p_{c_b}^+$.

After the above is done in terms of recoloring the vertex v, `set-level`$(v,-1)$ is called to bring the level of v down to -1. The description of `set-level`$(v,-1)$ is given in the following. See the pseudocode for `det-color`(v) in Fig. 7 for concrete details of this procedure.

The procedure `rand-color`(v) employs a *level-rising mechanism*. We mentioned before the concept of partitioning vertices into levels. Each level bounds the down-neighbbors of the vertices at that level, providing both an upper and lower bound on the number of down-neighbors of the vertex. Because there are at most $\log_3(n-1)$ levels, the number of vertices in each level is thus exponentially increasing. The procedure `rand-color`(v) takes advantage of this bound on the number of down-neighbors of the vertex v to find a level to recolor v with a color randomly chosen from its C_v. Specifically, `rand-color`(v) recolors v at some level ℓ^* higher than $\ell(v)$, with a random blank or unique color occupied by vertices of levels *strictly lower* than ℓ^*. At level ℓ^*, it attempts to select a color c within time $O(3^{\ell^*})$; this can only occur if $|\mathcal{D}_v| = O(3^{\ell^*})$. Upon failure, it calls itself

recursively to color v at yet a higher level. Again, `set-level`(v, ℓ^*) is called every time v moves to a high level.

Procedure `set-level`(v, ℓ). Procedures `det-color`(v) and `rand-color`(v) may set the level of v to a different level, in which case the procedure `set-level`(v, ℓ) is called with the new level ℓ as input. Let $\ell(v)$ be the previous level of v. The procedure does nothing if $\ell = \ell(v)$. Otherwise:

If v is set to a lower level $\ell < \ell(v)$: we need to update the data structures of vertices in levels $[\ell + 1, \ell(v)]$. For each vertex $w \in \mathcal{D}_v$ where $\ell + 1 \leq \ell(w) \leq \ell(v)$, we make the following data structure updates:

(1) Delete w from $\mathcal{D}_v$. Delete the mutual pointers between w and p_c. Let w's color be c. Move w's color, c, in C_v to C_v^+ if c is currently in C_v. Delete the mutual pointers between c and p_c. Create mutual pointers between c and p_c^+. Increment w's color count $\mu_v^+(c)$ by 1.
(2) Add w to $\mathcal{U}_v[\ell(w)]$. Add mutual pointers between w and p_c^+ where c is w's color.
(3) Delete v from $\mathcal{U}_w[\ell(v)]$. Let v's color be c'. Delete the mutual pointers between v and $p_{c'}^+$. Decrement v's color count $\mu_w^+(c')$ by 1. If $\mu_w^+(c')$ is now 0, move c' from C_w^+ to C_w, delete the mutual pointers between c' and $p_{c'}^+$, and create mutual pointers between c' and $p_{c'}$.
(4) Add v to $\mathcal{D}_w$. Add mutual pointers between v and $p_{c'}$ where c' is v's color if c' was moved to C_w.
(5) Add mutual pointers between all elements $v \in \mathcal{D}_w, w \in \mathcal{U}_v[\ell(w)], v \in \mathcal{N}_w, w \in \mathcal{N}_v$.
(6) Add mutual pointers between all copies of the same element: i.e. $w \in \mathcal{D}_v, w \in \mathcal{N}_v$, and/or $w \in \mathcal{U}_v[\ell(w)]$.
(7) Maintain mutual pointers between $\mathcal{P}_v[i_c]$, p_c, and p_c^+. Maintain mutual pointers between $\mathcal{P}_w[i_{c'}]$, $p_{c'}^+$, and p_c.

If v is set to a higher level $\ell > \ell(v)$: we need to update the data structures of vertices in levels $[\ell(v), \ell - 1]$. Specifically, for each non-empty list $\mathcal{U}_v[i]$, with $\ell(v) \leq i \leq \ell - 1$, and for each vertex $w \in \mathcal{U}_v[i]$, we perform the following operations:

(1) Delete w from $\mathcal{U}_v[i]$. Let c be the color of w. Delete the mutual pointers between w and p_c^+. Decrement $\mu_v^+(c)$ by 1. If $\mu_v^+(c) = 0$, then move c from C_v^+ to C_v, delete the mutual pointers between p_c^+ and c, and add mutual pointers between p_c and c.
(2) Add w to $\mathcal{D}_v$, create mutual pointers between w and p_c (where c is w's color), delete v from $\mathcal{D}_w$, and add v to $\mathcal{U}_w[\ell]$. Let v's color be c'. Delete the mutual pointers between v and $p_{c'}$. Add mutual pointers between v and $p_{c'}^+$. If c' is currently in C_w, move v's color, c', in C_w to C_w^+, delete mutual pointers between c' and $p_{c'}$, and add mutual pointers between c' and $p_{c'}^+$. Increment $\mu_w^+(c')$ by 1.
(3) Add mutual pointers between all elements $w \in \mathcal{D}_v, v \in \mathcal{U}_w[\ell], v \in N_w, w \in N_v$.
(4) Maintain mutual pointers between $\mathcal{P}_v[i_c]$, p_c, and p_c^+. Maintain mutual pointers between $\mathcal{P}_w[i_{c'}]$, $p_{c'}$, and $p_{c'}^+$.

The full pseudocode of this procedure can be found in Fig. 5.

B PSEUDOCODE

In the below pseudocode, we do not describe (most of) the straightforward but tedious pointer creation procedures. We assume that the corresponding pointers are created according to the procedure described in Section A.1. In the cases where the pointer change is significant, we describe it in the pseudocode.

`handle-insertion`(u, v):

(1) $\mathcal{N}_v \leftarrow \mathcal{N}_v \cup \{u\}$;
(2) $\mathcal{N}_u \leftarrow \mathcal{N}_u \cup \{v\}$;
(3) If $\ell(u) > \ell(v)$:
  (a) $\mathcal{D}_u \leftarrow \mathcal{D}_u \cup \{v\}$;
  (b) $\mathcal{U}_v[\ell(u)] \leftarrow \mathcal{U}_v[\ell(u)] \cup \{u\}$;
(4) Else if $\ell(u) = \ell(v)$:
  (a) $\mathcal{U}_v[\ell(u)] \leftarrow \mathcal{U}_v[\ell(u)] \cup \{u\}$;
  (b) $\mathcal{U}_u[\ell(v)] \leftarrow \mathcal{U}_u[\ell(v)] \cup \{v\}$;
(5) Else:
  (a) $\mathcal{D}_v \leftarrow \mathcal{D}_v \cup \{u\}$;
  (b) $\mathcal{U}_u[\ell(v)] \leftarrow \mathcal{U}_u[\ell(v)] \cup \{v\}$;
(6) `update-color-edge-insertion`(u, v, c_u, c_v);
(7) If color(u) = color(v): /* if u and v have the same color */
  (a) `recolor`(u); /* assuming u is the most recently colored */

Fig. 3. Handling edge insertion (u, v).

`handle-deletion`(u, v):

(1) $\mathcal{N}_v \leftarrow \mathcal{N}_v \setminus \{u\}$;
(2) $\mathcal{N}_u \leftarrow \mathcal{N}_u \setminus \{v\}$;
(3) If $v \in \mathcal{D}_u$:
  (a) $\mathcal{D}_u \leftarrow \mathcal{D}_u \setminus \{v\}$;
  (b) $\mathcal{U}_v[\ell(u)] \leftarrow \mathcal{U}_v[\ell(u)] \setminus \{u\}$;
(4) Else if $u \in \mathcal{D}_v$:
  (a) $\mathcal{D}_v \leftarrow \mathcal{D}_v \setminus \{u\}$;
  (b) $\mathcal{U}_u[\ell(v)] \leftarrow \mathcal{U}_u[\ell(v)] \setminus \{v\}$;
(5) Else:
  (a) $\mathcal{U}_u[\ell(v)] \leftarrow \mathcal{U}_u[\ell(v)] \setminus \{v\}$;
  (b) $\mathcal{U}_v[\ell(u)] \leftarrow \mathcal{U}_v[\ell(u)] \setminus \{u\}$;
(6) Remove all associate color pointers and shift colors between C_u, C_u^+ and C_w, C_w^+ as necessary;

Fig. 4. Handling edge deletion (u, v).

`set-level`(v, ℓ):

(1) For all $w \in \mathcal{D}_v$: /* update $\mathcal{U}_w$ regarding v's new level */
  (a) $\mathcal{U}_w[\ell(v)] \leftarrow \mathcal{U}_w[\ell(v)] \setminus \{v\}$;
  (b) $\mathcal{U}_w[\ell] \leftarrow \mathcal{U}_w[\ell] \cup \{v\}$;
(2) If $\ell < \ell(v)$: /* in this case the level of v is decreased by at least one */
  (a) For all $w \in \mathcal{D}_v$ such that $\ell \leq \ell(w) < \ell(v)$: /* reassign color pointers*/
    (i) $\mathcal{D}_v \leftarrow \mathcal{D}_v \setminus \{w\}$;
    (ii) Delete mutual pointers between w and $p_{\text{color}(w)}$;
    (iii) If $\text{color}(w) \in C_v$:
      (A) Move $\text{color}(w) \in C_v$ to C_v^+;
      (B) Delete mutual pointers between $\text{color}(w)$ and $p_{\text{color}(w)}$;
      (C) Create mutual pointers between $\text{color}(w)$ and $p^+_{\text{color}(w)}$;
    (iv) Increment $\mu_v^+(\text{color}(w))$ by 1;
    (v) $\mathcal{U}_v[\ell(w)] \leftarrow \mathcal{U}_v[\ell(w)] \cup \{w\}$;
    (vi) Create mutual pointers between w and $p^+_{\text{color}(w)}$ if such pointers do not already exist;
    (vii) $\mathcal{U}_w[\ell] \leftarrow \mathcal{U}_w[\ell] \setminus \{v\}$;
    (viii) Delete mutual pointers between $\text{color}(v)$ and $p^+_{\text{color}(v)}$;
    (ix) Decrement $\mu_w^+(\text{color}(v))$ by 1;
    (x) If $\mu_w^+(\text{color}(v)) = 0$:
      (A) Move $\text{color}(v) \in C_w^+$ to C_w;
      (B) Delete mutual pointers between $\text{color}(v)$ and $p^+_{\text{color}(v)}$;
      (C) Create mutual pointers between $\text{color}(v)$ and $p_{\text{color}(v)}$;
    (xi) $\mathcal{D}_w \leftarrow \mathcal{D}_w \cup \{v\}$;
    (xii) Create mutual pointers between v and $p_{\text{color}(v)}$ if such pointers do not already exist;
(3) If $\ell > \ell(v)$: /* in this case the level of v is increased by at least one */ [a]
  (a) For all $i = \ell(v), \ldots, \ell - 1$ and all $w \in \mathcal{U}_v[i]$:
    (i) $\mathcal{U}_v[i] \leftarrow \mathcal{U}_v[i] \setminus \{w\}$;
    (ii) Decrement $\mu_v^+(\text{color}(w))$ by 1;
    (iii) If $\mu_v^+(\text{color}(w)) = 0$: Move $\text{color}(w)$ from C_v^+ to C_v;
    (iv) $\mathcal{D}_v \leftarrow \mathcal{D}_v \cup \{w\}$;
    (v) $\mathcal{D}_w \leftarrow \mathcal{D}_w \setminus \{v\}$;
    (vi) $\mathcal{U}_w[\ell] \leftarrow \mathcal{U}_w[\ell] \cup \{v\}$;
    (vii) If $\text{color}(v) \in C_w$: Move $\text{color}(v)$ from C_w to C_w^+;
    (viii) Increment $\mu_w^+(\text{color}(v))$ by 1;
(4) $\ell(v) \leftarrow \ell$;

[a] For the sake of clarity and brevity, we do not describe the pointer deletions, creations, and changes in the case where $\ell > \ell(v)$ because these changes are almost identical to the changes given above for the case $\ell < \ell(v)$.

Fig. 5. Setting the old level $\ell(v)$ of v to ℓ.

`recolor`(v):
(1) If $\phi_v(\ell(v) + 1) < 3^{\ell(v)+2}$: `det-color`$(v)$;
(2) Else `rand-color`(v);

Fig. 6. Recoloring a vertex that collides with the color of an adjacent vertex after an edge insertion.

`det-color`(v):
(1) For all $c \in C_v$:
  (a) If c is not occupied by any vertex $w \in \mathcal{D}_v$ and $c \in C_v$: /* if c is a blank color, color v with c */
    (i) Set $\chi(i_v) = c$;
    (ii) For all $w \in \mathcal{D}_v$:
      (A) `update-color`(v, w, c).
    (iii) `set-level`$(v, -1)$;
    (iv) terminate the procedure; /* Note that the procedure will always terminate within this if statement because a blank color always exists by definition of $(\Delta+1)$-coloring. */

Fig. 7. Coloring v deterministically with a blank color. It is assumed that $\phi_v(\ell(v) + 1) < 3^{\ell(v)+2}$.

`rand-color`(v):
(1) $\ell^* \leftarrow \ell(v)$;
(2) while $\phi_v(\ell^* + 1) \geq 3^{\ell^*+2}$: $\ell^* \leftarrow \ell^* + 1$;
  /* ℓ^* is the minimum level after $\ell(v)$ with $\phi_v(\ell^* + 1) < 3^{\ell^*+2}$ */
(3) `set-level`(v, ℓ^*); /* after this call $\ell(v) = \ell^*$ and $3^{\ell^*+1} \leq d_{\mathsf{out}}(v) = \phi_v(\ell^*) < 3^{\ell^*+2}$ */
(4) Pick a blank or unique color c from C_v uniformly at random;
  /* c is chosen with probability at most $2/3^{\ell^*+1}$ and $\ell(w) \leq \ell^* - 1$ */
(5) If $c \neq$ color(v): /* If c is not the previous color of v. */
  (a) Set $\chi(i_v) = c$;
  (b) For all $z \in \mathcal{D}_v$:
    (i) `update-color`(v, z, c).
(6) If c is a unique color (let $w \in \phi_v(\ell^*)$ be the vertex that is colored with c):
  (a) `recolor`(w);

Fig. 8. Coloring v at level ℓ^* higher than $\ell(v)$, with a random blank or unique color of level lower than ℓ^*. If the procedure chose a unique color, it calls `recolor` (which may call itself recursively) to color w. It is assumed that $\phi_v(\ell(v) + 1) \geq 3^{\ell(v)+2}$.

`update-color-edge-insertion`(v, w, c_v, c_w):

(1) If $\ell(v) > \ell(w)$:
  (a) Locate $v \in \mathcal{U}_w[\ell(v)]$;
  (b) Delete the mutual pointers (if they exist) between v and $p^+_{c'}$ where c' is v's previous color; /* Note that v's previous color could be located by following pointers from v. */
  (c) Decrement $\mu^+_w(c')$ by 1 if pointers were deleted in the previous step; /* If no pointers were deleted, then w had no knowledge of v's previous color and we do not need to decrement */
  (d) If $\mu^+_w(c') = 0$:
    (i) Move c' from C^+_w to C_w by appending c' to the end of the linked list representing C_w;
  (e) Locate $p^+_{c_v}$ by following pointers from $\mathcal{P}_w$;
  (f) Create mutual pointers between v and $p^+_{c_v}$;
  (g) Increment $\mu^+_w(c_v)$ by 1;
  (h) If c_v is in C_w:
    (i) Move c_v from C_w to C^+_w by appending c_v to the end of the linked list representing C^+_w;
  (i) Locate $w \in \mathcal{D}_v$.
  (j) Delete the mutual pointers (if they exist) between w and $p_{c''}$ where c'' is w's previous color;
  (k) Locate p_{c_v} by following pointers from $\mathcal{P}_v$;
  (l) Create mutual pointers between $w \in \mathcal{D}_v$ and p_{c_w};
(2) Else if $\ell(v) < \ell(w)$:
  (a) /* Do the above except switch the roles of v and w as well as c_v and c_w. */
(3) Else:
  (a) Locate $v \in \mathcal{U}_w[\ell(v)]$ and $w \in \mathcal{U}_v[\ell(w)]$;
  (b) /* Do the above procedure given in the case when $\ell(v) > \ell(w)$ for $v \in \mathcal{U}_w[\ell(v)]$ for both $v \in \mathcal{U}_w[\ell(v)]$ and $w \in \mathcal{U}_v[\ell(w)]$.*/

Fig. 9. Updates the data structures with v and w's colors when an edge is inserted between v and w.

`update-color`(v, c_v):

(1) For $w \in \mathcal{D}_v$:
  (a) Locate $v \in \mathcal{U}_w[\ell(v)]$;
  (b) Delete the mutual pointers (if they exist) between v and $p^+_{c'}$ where c' is v's previous color; /* Note that v's previous color could be located by following pointers from v. */
  (c) Decrement $\mu^+_w(c')$ by 1 if pointers were deleted in the previous step; /* If no pointers were deleted, then w had no knowledge of v's previous color and we do not need to decrement */
  (d) If $\mu^+_w(c') = 0$:
    (i) Move c' from C^+_w to C_w by appending c' to the end of the linked list representing C_w;
  (e) Locate $p^+_{c_v}$ by following pointers from $\mathcal{P}_w$;
  (f) Create mutual pointers between v and $p^+_{c_v}$;
  (g) Increment $\mu^+_w(c_v)$ by 1;
  (h) If c_v is in C_w:
    (i) Move c_v from C_w to C^+_w by appending c_v to the end of the linked list representing C^+_w;

Fig. 10. Updates the color pointers of v of all of v's down-neighbors when v changes color.

REFERENCES

[1] S. Arnborg and A. Proskurowski. Linear time algorithms for np-hard problems restricted to partial k-trees. *Discrete Applied Mathematics*, 23(1):11–24, 1989.

[2] L. Barenboim and M. Elkin. Sublogarithmic distributed mis algorithm for sparse graphs using nash-williams decomposition. *Distributed Comput.*, 22(5-6):363–379, 2010.

[3] L. Barenboim and M. Elkin. Deterministic distributed vertex coloring in polylogarithmic time. *J. ACM*, 58(5):23:1–23:25, 2011. Announced at PODC 2010.

[4] S. Baswana, M. Gupta, and S. Sen. Fully dynamic maximal matching in $O(\log n)$ update time. *SIAM J. Comput.*, 44(1):88–113, 2015.

[5] S. K. Bera, A. Chakrabarti, and P. Ghosh. Graph coloring via degeneracy in streaming and other space-conscious models. In *ICALP*, volume 168 of *LIPIcs*, pages 11:1–11:21. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020.

[6] S. Bhattacharya, D. Chakrabarty, and M. Henzinger. Deterministic fully dynamic approximate vertex cover and fractional matching in O(1) amortized update time. In *IPCO*, volume 10328 of *Lecture Notes in Computer Science*, pages 86–98. Springer, 2017.

[7] S. Bhattacharya, D. Chakrabarty, M. Henzinger, and D. Nanongkai. Dynamic algorithms for graph coloring. In *SODA*, pages 1–20, 2018.

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

[9] S. Bhattacharya and J. Kulkarni. Deterministically maintaining a $(2+\epsilon)$-approximate minimum vertex cover in $o(1/\epsilon^2)$ amortized update time. In *SODA*, 2019.

[10] V. A. Campos, G. de C. M. Gomes, A. Ibiapina, R. Lopes, I. Sau, and A. Silva. Coloring problems on bipartite graphs of small diameter. *Electron. J. Comb.*, 28(2):P2.14, 2021.

[11] K. K. Dabrowski, F. Dross, M. Johnson, and D. Paulusma. Filling the complexity gaps for colouring planar and bounded degree graphs. *J. Graph Theory*, 92(4):377–393, 2019.

[12] U. Feige and J. Kilian. Zero knowledge and the chromatic number. *J. Comput. Syst. Sci.*, 57(2):187–199, 1998. Announced at CCC'96.

[13] J. Fiala, P. A. Golovach, and J. Kratochvíl. Parameterized complexity of coloring problems: Treewidth versus vertex cover. *Theoretical Computer Science*, 412(23):2513–2523, 2011.

[14] F. V. Fomin, P. A. Golovach, D. Lokshtanov, and S. Saurabh. Clique-width: on the price of generality. In *Proceedings of the Twentieth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 825–834. SIAM, 2009.

[15] M. R. Garey, D. S. Johnson, and L. Stockmeyer. Some simplified NP-complete problems. In *Proceedings of the Sixth Annual ACM Symposium on Theory of Computing*, STOC '74, pages 47–63, New York, NY, USA, 1974. ACM.

[16] M. Ghaffari and A. Sayyadi. Distributed Arboricity-Dependent Graph Coloring via All-to-All Communication. In C. Baier, I. Chatzigiannakis, P. Flocchini, and S. Leonardi, editors, *46th International Colloquium on Automata, Languages, and Programming (ICALP 2019)*, volume 132 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 142:1–142:14, Dagstuhl, Germany, 2019. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[17] A. Gupta, R. Krishnaswamy, A. Kumar, and D. Panigrahi. Online and dynamic algorithms for set cover. In *Proceedings of the 49th Annual ACM SIGACT Symposium on Theory of Computing, STOC 2017, Montreal, QC, Canada, June 19-23, 2017*, pages 537–550, 2017.

[18] M. Henzinger, S. Neumann, and A. Wiese. Explicit and implicit dynamic coloring of graphs with bounded arboricity. *CoRR*, abs/2002.10142, 2020.

[19] M. Henzinger and P. Peng. Constant-time dynamic (Δ+1)-coloring and weight approximation for minimum spanning forest: Dynamic algorithms meet property testing. *CoRR*, abs/1907.04745, 2019.

[20] K. Jansen and P. Scheffler. Generalized coloring for tree-like graphs. *Discrete Applied Mathematics*, 75(2):135–155, 1997.

[21] K.-I. Kawarabayashi and M. Thorup. Coloring 3-colorable graphs with less than $n^{1/5}$ colors. *J. ACM*, 64(1):4:1–4:23, Mar. 2017.

[22] S. Khot and A. K. Ponnuswami. Better inapproximability results for maxclique, chromatic number and min-3lin-deletion. In *ICALP*, pages 226–237, 2006.

[23] D. Kobler and U. Rotics. Edge dominating set and colorings on graphs with fixed clique-width. *Discrete Applied Mathematics*, 126(2):197–221, 2003.

[24] K. Kothapalli and S. Pemmaraju. Distributed graph coloring in a few rounds. In *Proceedings of the 30th Annual ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, PODC '11, page 31âĂŞ40, New York, NY, USA, 2011. Association for Computing Machinery.

[25] M. Luby. A simple parallel algorithm for the maximal independent set problem. *SIAM journal on computing*, 15(4):1036–1053, 1986.

[26] B. Martin, D. Paulusma, and S. Smith. Colouring H-Free Graphs of Bounded Diameter. In P. Rossmanith, P. Heggernes, and J.-P. Katoen, editors, *44th International Symposium on Mathematical Foundations of Computer Science (MFCS 2019)*,

volume 138 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 14:1–14:14, Dagstuhl, Germany, 2019. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[27] B. Martin, D. Paulusma, and S. Smith. Colouring graphs of bounded diameter in the absence of small cycles. In *CIAC 2021*, Lecture Notes in Computer Science. Springer, 2021.

[28] D. W. Matula and L. L. Beck. Smallest-last ordering and clustering and graph coloring algorithms. *J. ACM*, 30(3):417âĂŞ427, July 1983.

[29] J. Radhakrishnan, S. Shannigrahi, and R. Venkat. Hypergraph two-coloring in the streaming model. *CoRR*, abs/1512.04188, 2015.

[30] S. Solomon. Fully dynamic maximal matching in constant update time. In *FOCS*, pages 325–334, 2016.

[31] S. Solomon and N. Wein. Improved dynamic graph coloring. *ACM Trans. Algorithms*, 16(3):41:1–41:24, 2020.

[32] D. Zuckerman. Linear degree extractors and the inapproximability of max clique and chromatic number. *Theory of Computing*, 3(1):103–128, 2007. Announced at STOC'06.