Dynamic Algorithms for Graph Coloring

Sayan Bhattacharya* Deeparnab Chakrabarty† Monika Henzinger‡
Danupon Nanongkai§

Abstract

We design fast dynamic algorithms for proper vertex and edge colorings in a graph undergoing edge insertions and deletions. In the static setting, there are simple linear time algorithms for $(\Delta+1)$- vertex coloring and $(2\Delta-1)$-edge coloring in a graph with maximum degree Δ. It is natural to ask if we can efficiently maintain such colorings in the dynamic setting as well. We get the following three results. (1) We present a *randomized* algorithm which maintains a $(\Delta+1)$-vertex coloring with $O(\log\Delta)$ expected amortized update time. (2) We present a *deterministic* algorithm which maintains a $(1+o(1))\Delta$-vertex coloring with $O(\text{polylog}\,\Delta)$ amortized update time. (3) We present a simple, deterministic algorithm which maintains a $(2\Delta-1)$-edge coloring with $O(\log\Delta)$ *worst-case* update time. This improves the recent $O(\Delta)$-edge coloring algorithm with $\tilde{O}(\sqrt{\Delta})$ worst-case update time [BM17].

*Corresponding author. University of Warwick, UK. Email: S.Bhattacharya@warwick.ac.uk
†Dartmouth College, USA. Email: deeparnab.chakrabarty@dartmouth.edu
‡University of Vienna, Austria. Email: monika.henzinger@univie.ac.at
§KTH, Sweden. Email: danupon@gmail.com

Contents

1 Introduction

Graph coloring is a fundamental problem with many applications in computer science. A proper c-vertex coloring of a graph assigns a color in $\{1, \ldots, c\}$ to every node, in such a way that the endpoints of every edge get different colors. The chromatic number of the graph is the smallest c for which a proper c-vertex coloring exists. Unfortunately, from a computational perspective, approximating the chromatic number is rather futile: for any constant $\epsilon > 0$, there is no polynomial time algorithm that approximates the chromatic number within a factor of $n^{1-\epsilon}$ in an n-vertex graph, assuming $P \neq NP$ [FK98; Zuc07] (see [KP06] for a stronger bound). On the positive side, we know that the chromatic number is at most $\Delta + 1$ where Δ is the maximum degree of the graph. There is a simple linear time algorithm to find a $(\Delta + 1)$-coloring: pick any uncolored vertex v, scan the colors used by its neighbors, and assign to v a color not assigned to any of its neighbors. Since the number of neighbors is at most Δ, by pigeon hole principle such a color must exist.

In this paper, we consider the graph coloring problem in the dynamic setting, where the edges of a graph are being inserted or deleted over time and we want to maintain a proper coloring after every update. The objective is to use as few colors as possible while keeping the update time[1] small. Specifically, our main goal is to investigate whether a $(\Delta + 1)$-vertex coloring can be maintained with small update time. Note that the greedy algorithm described in the previous paragraph can easily be modified to give a worst-case update time of $O(\Delta)$: if an edge (u, v) is inserted between two nodes u and v of same color, then scan the at most Δ neighbors of v to find a free color. A natural question is whether we can get an algorithm with significantly lower update time. We answer this question in the affirmative.

- We design and analyse a *randomized* algorithm which maintains a $(\Delta + 1)$-vertex coloring with $O(\log \Delta)$ expected amortized update time.[2]

It is not difficult to see that if we had $(1 + \epsilon)\Delta$ colors, then there would be a simple randomized algorithm with $O(1/\epsilon)$-expected amortized update time (see Section 2.1 for details). What is challenging in our result above is to maintain a $\Delta + 1$ coloring with small update time. In contrast, if randomization is not allowed, then even maintaining a $O(\Delta)$-coloring with $o(\Delta)$-update time seems non-trivial. Our second result is on deterministic vertex coloring algorithms: although we do not achieve a $\Delta + 1$ coloring, we come close.

- We design and analyse a *deterministic* algorithm which maintains a $(\Delta + o(\Delta))$-vertex coloring with $O(\text{polylog}\, \Delta)$ amortized update time.

Note that in a dynamic graph the maximum degree Δ can change over time. Our results hold with the changing Δ as well. However, for ease of explaining our main ideas we restrict most of the paper to the setting where Δ is a static upper bound known to the algorithm. In Section 6 we point out the changes needed to make our algorithms work for the changing-Δ case.

Our final result is on maintaining an *edge coloring* in a dynamic graph with maximum degree Δ. A proper edge coloring is a coloring of edges such that no two adjacent edges have the same color.

- We design and analyze a simple, deterministic $(2\Delta - 1)$-edge coloring algorithm with $O(\log \Delta)$ *worst-case* update time.

This significantly improves upon the recent $O(\Delta)$-edge coloring algorithm of Barenboim and Maimon [BM17] which needs $\tilde{O}(\sqrt{\Delta})$-worst-case update time.

[1] There are two notions of update time: *amortized update time* – an algorithm has amortized update time of α if for any t, after t insertions or deletions the total update time is $\leq \alpha t$, and *worst case update time* – an algorithm has worst case update time of α if every update time $\leq \alpha$. As typical for amortized update time guarantees, we assume that the input graph is empty initially.

[2] As typically done for randomized dynamic algorithms, we assume that the adversary who fixes the sequence of edge insertions and deletions is *oblivious* to the randomness in our algorithm.

Perspective: An important aspect of $(\Delta + 1)$-vertex coloring is the following *local-fixability property*: Consider a graph problem $\mathcal{P}$ where we need to assign a *state* (e.g. color) to each node. We say that a constraint is *local to a node* v if it is defined on the states of v and its neighbors. We say that a problem $\mathcal{P}$ is locally-fixable iff it has the following three properties. (i) There is a local constraint on every node. (ii) A solution S to $\mathcal{P}$ is *feasible* iff S satisfies the local constraint at every node. (iii) If the local constraint C_v at a node v is unsatisfied, then we can change only the state of v to satisfy C_v without creating any new unsatisfied constraints at other nodes. For example, $(\Delta + 1)$-vertex coloring is locally-fixable as we can define a constraint local to v to be satisfied if and only if v's color is different from all its neighbors, and if not, then we can always find a recoloring of v to satisfy its local constraint and introduce no other constraint violations. On the other hand, the following problems do not seem to be locally-fixable: globally optimum coloring, the best approximation algorithm for coloring [Hal93], Δ-coloring (which always exists by Brook's theorem, unless the graph is a clique or an odd cycle), and $(\Delta+1)$-edge coloring (which always exists by Vizing's theorem).

Observe that if we start with a feasible solution for a locally-fixable problem $\mathcal{P}$, then after inserting or deleting an edge (u, v) we need to change only the states of u and v to obtain a new feasible solution. For instance in the case of $(\Delta + 1)$-vertex coloring, we need to recolor only the nodes incident to the inserted edge. Thus, the number of changes is guaranteed to be small, and the main challenge is to *search* for these changes in an efficient manner without having to scan the whole neighborhood. In contrast, for non-locally-fixable problems, the main challenge seems to be analyzing how many nodes or edges we need to recolor (even with an inefficient algorithm) to keep the coloring proper. A question in this spirit has been recently studied in [BCKLRRV17].

It can be shown that the $(2\Delta-1)$-edge coloring problem is also locally-fixable (see Appendix A). Given our current results on $(\Delta+1)$-vertex coloring and $(2\Delta-1)$-edge coloring, it is inviting to ask whether there is some deeper connections that exist in designing dynamic algorithms for these problems. In particular, are there reductions possible among these problems? Or can we find a *complete* locally-fixable problem? It is also very interesting to understand the power of randomization for these problems.

Indeed, in the *distributed computing* literature, there is deep and extensive work on and beyond the locally-fixable problems above. (In fact, it can be shown that any locally-fixable problem is in the SLOCAL complexity class studied in distributed computing [GKM17]; see Appendix A.) Coincidentally, just like our findings in this paper, there is still a big gap between deterministic and randomized distributed algorithms for $(\Delta + 1)$-vertex coloring. For further details we refer the to the excellent monograph by Barenboim and Elkin [BE13] (see [GKM17; FGK17], and references therein, for more recent results).

Finally, we also note that the dynamic problems we have focused on are *search problems*; i.e., their solutions always exist, and the hard part is to find and maintain them. This posts a new challenge when it comes to proving conditional lower bounds for dynamic algorithms for these locally-fixable problems: while a large body of work has been devoted to decision problems [Pat10; AW14; HKNS15; KPP16; Dah16], it seems non-trivial to adapt existing techniques to search problems.

Other Related Work. Dynamic graph coloring is a natural problem and there have been many works on this [DGOP07; OB11; SIPGRP16; HLT17]. Most of these papers, however, have proposed heuristics and described experimental results. The only two theoretical papers that we are aware of are [BM17] and [BCKLRRV17], and they are already mentioned above.

Organisation of the rest of the paper. In Section 2 we give the high level ideas behind our vertex coloring result. In particular, Section 2.1 contains the main ideas of the randomized algorithm, whereas Section 3 contains the full details. Similarly, Section 2.2 contains the main ideas of the deterministic algorithm, whereas Section 4 contains the full details. Section 5 contains the edge-coloring result. *We emphasize that Sections 3, 4, 5 are completely self contained, and they can be read independently of each other.* As mentioned earlier, in Sections 3, 4, 5 we assume that the parameter Δ is known and that the maximum

degree never exceeds Δ. We do so solely for the better exposition of the main ideas. Our algorithms easily modify to give results where Δ is the current maximum degree. See Section 6 for the details.

2 Our Techniques for Dynamic Vertex Coloring

2.1 An overview of our randomized algorithm.

We present a high level overview of our randomized dynamic algorithm for $(\Delta + 1)$-vertex coloring that has $O(\log \Delta)$ expected amortized update time. The full details can be found in Section 3. We start with a couple of warm-ups before sketching the main idea.

Warmup I: Maintaining a 2Δ-coloring in $O(1)$ expected amortized update time. We first observe that maintaining a 2Δ coloring is easy using randomization against an oblivious adversary – we need only $O(1)$ expected amortized time. The algorithm is this. Let $\mathcal{C}$ be the palette of 2Δ colors. Each vertex v stores the last time τ_v at which it was recolored. If an edge gets deleted or if an edge gets inserted between two vertices of different colors, then we do nothing. Next, consider the scenario where an edge gets inserted at time τ between two vertices u and v of same color. Without any loss of generality, suppose that $\tau_v > \tau_u$, i.e., the vertex v was recolored last. In this event, we scan all the neighbors of v and store the colors used by them in a set S, and select a random color from $\mathcal{C} \setminus S$. Since $|\mathcal{C}| = 2\Delta$, we have $|\mathcal{C} \setminus S| \geq \Delta$ as well. Clearly this leads to a proper coloring since the new color of v, by design, is not the current colors of any of v's neighbors.

The time taken to compute the set S can be as high as $O(\Delta)$ since v can have Δ neighbors. Now, let us analyze the probability that the insertion of the edge (u, v) at time τ leads to a conflict. Suppose that at time τ_v, just before v recolored itself, the color of u was c. The insertion at time τ creates a conflict only if v chose the color c at time τ_v. However the probability of this event is at most $1/\Delta$, since v had at least Δ choices to choose its color from at time τ_v. Therefore the expected time spent on the addition of edge (u, v) is $O(\Delta) \cdot (1/\Delta) = O(1)$.

In the analysis described above, we have crucially used the fact that the insertion of the edge (u, v) at time τ is oblivious to the random choice made while recoloring the vertex v at time τ_v. It should also be clear that the constant 2 is not sacrosanct and a $(1 + \epsilon)\Delta$ coloring can be obtained in $O(1/\epsilon)$-expected amortized time. However this fails to give a $\Delta + 1$ or even $\Delta + c$ coloring in $o(\Delta)$ time for any constant c.

Warmup II: A simple algorithm for $(\Delta + 1)$ coloring that is difficult to analyze. In the previous algorithm, while recoloring a vertex we made sure that it never assumed the color of any of its neighbors. We say that a color c is *blank* for a vertex v iff no neighbor of v gets the color c. Since we have $\Delta + 1$ colors, every vertex has at least one blank color. However, if there is only one blank color to choose from, then an adversarial sequence of updates may force the algorithm to spend $\Omega(\Delta)$ time after every edge insertion. A polar-opposite idea would be to randomly recolor a vertex without considering the colors of its neighbors. This has the problem that a recoloring may lead to one or more neighbors of the vertex being unhappy (i.e., having the same color as v), and then there is a cascading effect which is hard to control.

We take the middle ground: Define a color c to be *unique* for a vertex v if it is assigned to *exactly one* neighbor of v. Thus, if v is recolored using a unique color then the cascading effect of unhappy vertices doesn't explode. Specifically, after recoloring v we only need to consider recoloring v's unique neighbor, and so on and so forth. Why is this idea useful? This is because although the number of *blank* colors available to a vertex (i.e., the colors which none of its neighbors are using) can be as small as 1, the number of blank+unique colors is always at least $\Delta/2$. This holds since any color which is neither blank nor unique accounts for at least two neighbors of v, whereas v has at most Δ neighbors.

The above observation suggests the following natural algorithm. When we need to recolor a vertex v, we first scan all its neighbors to identify the set S of all unique *and* blank colors for v, and then we pick a new

color c for v uniformly at random from this set S. By definition of the set S, at most one neighbor y of x will have the same color c. If such a neighbor y exists, then we recolor y recursively using the same algorithm. We now state three important properties of this scheme. (1) While recoloring a vertex x we have to make *at most one* recursive call. (2) It takes $O(\Delta)$ time to recolor a vertex x, ignoring the potential recursive call to its neighbor. (3) When we recolor a vertex x, we pick its new color uniformly at random from a set of size $\Omega(\Delta)$. Note that the properties (2) and (3) served as the main tools in establishing the $O(1)$ bound on the expected amortized update time as discussed in the previous algorithm. For property (1), if we manage to upper bound the length of the *chain of recursive calls* that might result after the insertion of an edge in the input graph between two vertices of same color, then we will get an upper bound on the overall update time of our algorithm. This, however, is not trivial. In fact, the reader will observe that it is not necessary to have $\Delta + 1$ colors in order to ensure the above three properties. They hold even with Δ colors. Indeed, in that case the algorithm described above might never terminate. We conclude that another idea is required to achieve $O(\log \Delta)$ update time. This turns out to be the concept of a hierarchical partition of the set of vertices of a graph. We describe this and present an overview of our final algorithm below.

An overview of the final algorithm. Fix a large constant $\beta > 1$, and suppose that we can partition the vertex-set of the input graph $G = (V, E)$ into $L = \log_\beta \Delta$ *levels* $1, \ldots, L$ with the following property.

Property 2.1. *Consider any vertex v at a level $1 \leq \ell(v) \leq L$. Then the vertex v has at most $O(\beta^{\ell(v)})$ neighbors in levels $\{1, \ldots, \ell(v)\}$, and at least $\Omega(\beta^{\ell(v)-5})$ neighbors in levels $\{1, \ldots, \ell(v) - 1\}$.*

It is not clear at first glance that there even exists such a partition of the vertex-set: Given a *static* graph $G = (V, E)$, there seems to be no obvious way to assign a level $\ell(v) \in \{1, \ldots, L\}$ to each vertex $v \in V$ satisfying Property 2.1. One of our main technical contributions is to present an algorithm that maintains a hierarchical partition satisfying Property 2.1 in a dynamic graph. Initially, when the input graph $G = (V, E)$ has an empty edge-set, we place every vertex at level 1. This trivially satisfies Property 2.1. Subsequently, after every insertion or deletion of an edge in G, our algorithm updates the hierarchical partition in a way which ensures that Property 2.1 continues to remain satisfied. This algorithm is deterministic, and using an intricate charging argument we show that it has an amortized update time of $O(\log \Delta)$. This also gives a constructive proof of the existence of a hierarchical partition that satisfies Property 2.1 in any given graph.

We now explain how this hierarchical partition, in conjunction with the ideas from Warmup II, leads to an efficient randomized $\Delta + 1$ vertex coloring algorithm. In this algorithm, we require that a vertex u keeps all its neighbors v at levels $\ell(v) \leq \ell(u)$ informed about its own color $\chi(u)$. This requirement allows a vertex x to maintain: (1) the set $\mathcal{C}_x^+$ of colors assigned to its neighbors y with $\ell(y) \geq \ell(x)$, and (2) the set $\mathcal{C}_x = \mathcal{C} \setminus \mathcal{C}_x^+$ of remaining colors. We say that a color $c \in \mathcal{C}_x$ is *blank* for x iff no neighbor y of x with $\ell(y) < \ell(x)$ has the same color c. On the other hand, we say that a color $c \in \mathcal{C}_x$ is *unique* for x iff exactly one neighbor y of x with $\ell(y) < \ell(x)$ has the same color c. Note the crucial change in the definition of a unique color from Warmup II. Now, for a color c to be unique for x it is not enough that x has exactly one neighbor with the same color; in addition, this neighbor has to lie at a level *strictly* below the level of x. Using the property of the hierarchical partition that x has $\Omega(\beta^{\ell(x)-5})$ neighbors in levels $\{1, \ldots, \ell(x) - 1\}$ and an argument similar to one used in Warmup II, we can show that there are a large number of colors that are either blank or unique for x.

Claim 2.1. *For every vertex x, there are at least $1 + \Omega(\beta^{\ell(x)-5})$ colors that are either blank or unique.*

We now implement the same template as in Warmup II. When a vertex x needs to be recolored, it picks its new color uniformly at random from the set of its blank + unique colors. This can cause some other vertex y to be unhappy, but such a vertex y lies at a level strictly lower than $\ell(x)$. As there are $O(\log \Delta)$ levels, this bounds the depth of any recursive call: At level 1, we just use a blank color. Further, whenever we recolor x, the time it needs to inform all its neighbors y with $\ell(y) \leq \ell(x)$ is bounded by $O(\beta^{\ell(x)})$ (by

the property of the hierarchical partition). Since each recursive call is done on a vertex at a strictly lower level, the total time spent on all the recursive calls can also be bounded by $O(\beta^{\ell(x)})$ due to a geometric sum. Finally, by Claim 2.1, each time x picks a random color it does so from a palette of size $\Omega(\beta^{\ell(x)-5})$. If the order of edge insertions and deletions is oblivious to this randomness, then the probability that an edge insertion is going to be problematic is $O(1/\beta^{\ell(x)-5})$, which gives an expected amortized time bound of $O(1/\beta^{\ell(x)-5}) \times O(\beta^{\ell(x)}) = O(\beta^5) = O(1)$.

2.2 An overview of our deterministic algorithm.

We present a high level overview of our deterministic dynamic algorithm for $(1+o(1))\Delta$-vertex coloring that has an amortized update time of $O(\text{polylog}\,\Delta)$. The full details are in Section 4. As in Section 2.1, we start with a warmup before sketching the main idea.

Warmup: Maintaining a 4Δ coloring in $O(\sqrt{\Delta})$ amortized update time. Let $\mathcal{C}$ be the palette of 4Δ colors. We partition the set $\mathcal{C}$ into $2\sqrt{\Delta}$ equally sized subsets: $\mathcal{C}_1, \ldots, \mathcal{C}_{2\sqrt{\Delta}}$ each having $2\sqrt{\Delta}$ colors. Colors in $\mathcal{C}_t$ are said to be of *type* t and we let $t(v)$ denote the type of the color assigned to a node v. Furthermore, we let $d_t(v)$ denote the number of neighbors of v that are assigned a type t color. We refer to the neighbors u of v with $t(u) = t$ as *type t neighbors of v*. For every node v, we let $\Gamma(v)$ denote the set of neighbors u of v with $t(u) = t(v)$. Every node v maintains the set $\Gamma(v)$ in a doubly linked list. Note that if the node v gets a color from $\mathcal{C}_t$, then we have $d_t(v) = |\Gamma(v)|$. We maintain a proper coloring with the following extra property: If a node v is of type t, then it has at most $2\sqrt{\Delta} - 1$ type t neighbors.

Property 2.2. *If any node v is assigned a color from $\mathcal{C}_t$, then we have $d_t(v) < 2\sqrt{\Delta}$.*

Initially, the input graph $G = (V, E)$ is empty, every vertex is colored arbitrarily, and the above property holds. Note that the deletion of an edge from G does not lead to a violation of the above property, nor does it make the existing coloring invalid. We now discuss what we do when an edge (u, v) gets inserted into G, by considering three possible cases.

Case 1: $t(u) \neq t(v)$. There is nothing to be done since u and v have different types of colors.

Case 2: $t(u) = t(v) = t$, *but both* $d_t(u)$ *and* $d_t(v) < 2\sqrt{\Delta} - 1$ *after the insertion of the edge* (u, v). The colors assigned to the vertices u and v are of the same type. In this event, we first set $\Gamma(u) = \Gamma(u) \cup \{v\}$ and $\Gamma(v) = \Gamma(v) \cup \{u\}$. There is nothing further to do if u and v don't have the same color since the property continues to hold. If they have the same color c, then we pick an arbitrary endpoint u and find a type t color $c' \neq c$ that is not assigned to any of the neighbors of u in the set $\Gamma(u)$. This is possible since $|\Gamma(u)| = d_t(u) < 2\sqrt{\Delta} - 1$ and there are $2\sqrt{\Delta}$ colors of each type. We then change the color of u to c'. These operations take $O(|\mathcal{C}_t| + |\Gamma(u)|) = O(\sqrt{\Delta})$ time.

Case 3: $t(u) = t(v) = t$ *and* $d_t(u) = 2\sqrt{\Delta} - 1$ *after the insertion of the edge* (u, v). Here, after the addition of the edge (u, v), the vertex u violates Property 2.2. We run the following subroutine RECOLOR(u):

- Since u has at most Δ neighbors and there are $2\sqrt{\Delta}$ types, there must exist a type t' with $d_{t'}(u) \leq \sqrt{\Delta}/2$. Such a type t' can be found by doing a linear scan of all the neighbors of u, and this takes $O(\Delta)$ time since u has at most Δ neighbors.

  From the set $\mathcal{C}_{t'}$ we choose a color c that is not assigned to any of the neighbors of u: Such a color must exist since $|\mathcal{C}_{t'}| = 2\sqrt{\Delta} > d_{t'}(u)$. Next, we update the set $\Gamma(u)$ as follows: We delete from $\Gamma(u)$ every neighbor x of u with $t(x) = t$, and insert into $\Gamma(u)$ every neighbor x of u with $t(x) = t'$. We similarly update the set Γ_x for every neighbor x of u with $t(x) \in \{t, t'\}$. It takes $O(d_t(u) + d_{t'}(u)) = O(\sqrt{\Delta})$ time to implement this step.

  Accordingly, the total time spent on this call to the RECOLOR(.) subroutine is $O(\Delta) + O(\sqrt{\Delta}) = O(\Delta)$. However, Property 2.2 may now be violated for one or more neighbors u' of u. If this is the

case, then we recursively call RECOLOR(u') and keep doing so until all the vertices satisfy Property 2.2. In the end, we have a proper coloring with all the vertices satisfying Property 2.2.

A priori it may not be clear that the above procedure even terminates. However, we now argue that the amortized time spent in all the calls to the RECOLOR subroutine is $O(\sqrt{\Delta})$ (and in particular the chain of recursive calls to the subroutine terminates). To do so we introduce a potential $\Phi := \sum_{v \in V} d_{t(v)}(v)$, which sums over all vertices the number of its neighbors which are of the same type as itself. Note that when an edge (u, v) is inserted or deleted the potential can increase by at most 2. However, during a call to RECOLOR(u) the potential Φ drops by at least $3\sqrt{\Delta}$. This is because u moves from a color of type t_{old} to a color of type t_{new} where $d_{t_{old}}(u) = 2\sqrt{\Delta}$ and $d_{t_{new}}(u) \leq \sqrt{\Delta}/2$; this leads to a drop of $1.5\sqrt{\Delta}$ and we get the same amount of drop when considering u's neighbors. Therefore, during T edge insertions or deletions starting from an empty graph, we can have at most $O(T/\sqrt{\Delta})$ calls to the RECOLOR subroutine. Since each such call takes $O(\Delta)$ time, we get the claimed $O(\sqrt{\Delta})$ amortized update time.

Getting $O(\text{polylog}\,\Delta)$ amortized update time. One way to interpret the previous algorithm is as follows. Think of each color $c \in \mathcal{C}$ as an ordered pair $c = (c_1, c_2)$, where $c_1, c_2 \in \{1, \ldots, 2\sqrt{\Delta}\}$. The first coordinate c_1 is analogous to the notion of a *type*, as defined in the previous algorithm. For any vertex $v \in V$ and $j \in \{1, 2\}$, let $\chi^*_j(v)$ denote the j-tuple consisting of the first j coordinates of the color assigned to v. For ease of exposition, we define $\chi^*_0(v) = \perp$. Furthermore, for every vertex $v \in V$ and every $j \in \{0, 1, 2\}$, let $N^*_j(v) = \{u \in N_v : \chi^*_j(u) = \chi^*_j(v)\}$ denote the set of neighbors u of v with $\chi^*_j(u) = \chi^*_j(v)$. With these notations, Property 2.2 can be rewritten as: $|N^*_1(v)| < 2\sqrt{\Delta}$ for all $v \in V$.

To improve the amortized update time to $O(\text{polylog}\,\Delta)$, we think of every color as an L tuple $c = (c_1, \ldots, c_L)$, whose each coordinate can take λ possible values. The total number of colors is given by $\lambda^L = |\mathcal{C}|$. The values of L and λ are chosen in such a way which ensures that $\lambda = O(\lg^{1+o(1)} \Delta)$ and $L = O(\lg \Delta / \lg\lg \Delta)$. We maintain the invariant that $|N^*_j(v)| \leq (\Delta/\lambda^j) \cdot f(j)$ for all $v \in V$ and $j \in [0, L]$, for some carefully chosen function $f(j)$. We then implement a generalization of the previous algorithm on these colors represented as L tuples. Using some carefully chosen parameters we show how to deterministically maintain a $(\Delta + o(\Delta))$ vertex coloring in a dynamic graph in $O(\text{polylog}\,\Delta)$ amortized update time. See Section 4 for the details.

3 A Randomized Dynamic Algorithm for $\Delta + 1$ Vertex Coloring

As discussed in Section 2.1, our randomized dynamic algorithm for $\Delta + 1$ vertex coloring has two main components. The first one is a hierarchical partition of the vertices of the input graph into $O(\log \Delta)$-many levels. In Section 3.2, we show how to maintain such a hierarchical partition dynamically. The second component is the use of randomization while recoloring a conflicted vertex v so as to ensure that (a) at most one new conflict is caused due to this recoloring, and (b) if so, the new conflicted vertex lies at a level strictly lower than $\ell(v)$. We describe this second component in Section 3.3. The complete algorithm, which combines the two components, appears in Section 3.4. The theorem below captures our main result.

Theorem 3.1. *There is a randomized, fully dynamic algorithm to maintain a $\Delta + 1$ vertex coloring of a graph whose maximum degree is Δ with expected amortized update time $O(\log \Delta)$.*

3.1 Preliminaries.

We start with the definition of a hierarchical partition. Let $G = (V, E)$ denote the input graph that is changing dynamically, and let Δ be an upper bound on the maximum degree of any vertex in G. For now we assume that the value of Δ does not change with time. In Section 6, we explain how to relax this assumption. Fix a constant $\beta > 20$. For simplicity of exposition, assume that $\log_\beta \Delta = L$ (say) is an integer

and that $L > 3$. The vertex set V is partitioned into $L-3$ subsets $V_4, \ldots, V_L$. The *level* $\ell(v)$ of a vertex v is the index of the subset it belongs to. For any vertex $v \in V$ and any two indices $4 \leq i \leq j \leq L$, we let $\mathcal{N}_v(i,j) = \{u : (u,v) \in E,\ \ i \leq \ell(u) \leq j\}$ be the set of neighbors of v whose levels are between i and j. For notational convenience, we define $\mathcal{N}_v(i,j) = \emptyset$ whenever $i > j$. A hierarchical partition satisfies the following two properties/invariants. Note that since $\beta^L = \Delta$, Invariant 3.3 is trivially satisfied by every vertex at the highest level L. Invariant 3.2, on the other hand, is trivially satisfied by the vertices at level 4.

Invariant 3.2. *For every vertex $v \in V$ at level $\ell(v) > 4$, we have $|\mathcal{N}_v(4, \ell(v)-1)| \geq \beta^{\ell(v)-5}$.*

Invariant 3.3. *For every vertex $v \in V$, we have $|\mathcal{N}_v(4, \ell(v))| \leq \beta^{\ell(v)}$.*

Let $\mathcal{C} = \{1, \ldots, \Delta+1\}$ be the set of all possible colors. A coloring $\chi : V \to \mathcal{C}$ is *proper* for the graph $G = (V, E)$ iff for every edge $(u,v) \in E$, we have $\chi(u) \neq \chi(v)$. Given the hierarchical partition, a coloring $\chi : V \to \mathcal{C}$, and a vertex x at level $i = \ell(x)$, we define a few key subsets of $\mathcal{C}$. Let $\mathcal{C}_x^+ := \bigcup_{y \in \mathcal{N}_x(i,L)} \chi(y)$ be the colors used by neighbors of x lying in levels i and above. Let $\mathcal{C}_x := \mathcal{C} \setminus \mathcal{C}_x^+$ denote the remaining set of colors. We say a color $c \in \mathcal{C}_x$ is *blank for* x if no vertex in $\mathcal{N}_x(4, i-1)$ is assigned color c. We say a color $c \in \mathcal{C}_x$ is *unique for* x if *exactly one* vertex in $\mathcal{N}_x(4, i-1)$ is assigned color c. We let B_x (respectively U_x) denote the blank (respectively unique) colors for x. Let $T_x := \mathcal{C}_x \setminus (B_x \cup U_x)$ denote the remaining colors in C_x. Thus, for every color $c \in T_x$, there are at least two vertices $u \in \mathcal{N}_x(4, i-1)$ that are assigned color c. We end this section with a crucial observation.

Claim 3.1. *For any vertex x at level i, we have $|B_x \cup U_x| \geq 1 + \frac{|\mathcal{N}_x(4,i-1)|}{2}$.*

Proof. Since $|\mathcal{C}| = 1 + \Delta \geq 1 + |\mathcal{N}_x(4, L)|$ and $|\mathcal{C}_x^+| \leq |\mathcal{N}_x(i, L)|$, we get $|\mathcal{C}_x| \geq 1 + |\mathcal{N}_x(4, i-1)|$. The following two observations, which in turn follow from definitions, prove the claim; (a) $|\mathcal{C}_x| = |B_x \cup U_x| + |T_x|$ and (b) $2|T_x| \leq |\mathcal{N}_x(4, i-1)|$. □

Data Structures. We now describe the data structures used by our dynamic algorithm. The first set is used to maintain the hierarchical partition and the second set is used to maintain the sets of colors.

(1) For every vertex $v \in V$ and every level $\ell(v) \leq i \leq L$, we maintain the neighbors $\mathcal{N}_v(i,i)$ of v in level i in a doubly linked list. If $\ell(v) > 4$, then we also maintain the set of neighbors $\mathcal{N}_v(4, \ell(v)-1)$ in a doubly linked list. We use the phrase *neighborhood list* of v to refer to any one of these lists. For every neighborhood list we maintain a counter which stores the number of vertices in it. Every edge $(u,v) \in E$ keeps two pointers – one to the position of u in the neighborhood list of v, and the other vice versa. Therefore when an edge is inserted into or deleted from G the linked lists can be updated in $O(1)$ time. Finally, we keep two queues of *dirty* vertices which store the vertices not satisfying either of the two invariants.

(2) We maintain the coloring χ as an array where $\chi(v)$ contains the current color of v. Every vertex v maintains the colors $\mathcal{C}_v^+$ and $\mathcal{C}_v$ in doubly linked lists. For each color c and vertex v, we keep a pointer from the color to its position in either $\mathcal{C}_v^+$ or $\mathcal{C}_v$ depending on which list c belongs to. This allows us to add and delete colors from these lists in $O(1)$ time. We also maintain a counter $\mu_v^+(c)$ associated with each color c and each vertex v. If $c \in \mathcal{C}_v^+$, then the value of $\mu_v^+(c)$ equals the number of neighbors $y \in \mathcal{N}_v(i, L)$ with color $\chi(y) = c$. Otherwise, if $c \in \mathcal{C}_v$, then we set $\mu_v^+(c) \leftarrow 0$. For each vertex v, we keep a *time* counter τ_v which stores the last "time" (edge insertion/deletion) at which v was recolored[3], i.e., its $\chi(v)$ was changed.

3.2 Maintaining the hierarchical partition.

Initially when the graph is empty, all the vertices are at level 4. This satisfies both the invariants vacuously. Subsequently, we ensure that the hierarchical partition satisfies Invariants 3.2 and 3.3 by using a simple

[3]Note that as long as the number of edge insertions and deletions are polynomial, τ_v requires only $O(\log n)$ bits to store; if the number becomes superpolynomial then every n^3 rounds or so we recompute the full coloring in the current graph.

greedy heuristic procedure. To describe this procedure, we define a vertex to be *dirty* if it violates any one of the invariants, and *clean* otherwise. Our goal then is to ensure that every vertex in the hierarchical partition remains clean. By inductive hypothesis, we assume that every vertex is clean before the insertion/deletion of an edge. Due to the insertion/deletion of an edge (u, v), some vertices of the form $x \in \{u, v\}$ might become dirty. We fix the dirty vertices as per the procedure described in Figure 1. In this procedure, we always fix the dirty vertices violating Invariant 3.3 before fixing any dirty vertex that violates Invariant 3.2. This will be crucial in bounding the amortized update time. Furthermore, note that as we change the level of a vertex x during one iteration of the WHILE loop in Figure 1, this might lead to a change in the *below or side degrees*[4] of the neighbors of x. Hence, one iteration of the WHILE loop might create multiple new dirty vertices, which are dealt with in the subsequent iterations of the same WHILE loop. It is not hard to see that any iteration of the while loop acting on a vertex x ends with making it clean. We encapsulate this in the following lemma and the subsequent corollary.

```
01. WHILE Invariant 3.2 or Invariant 3.3 is violated:
02.     IF there is some vertex x ∈ V that violates Invariant 3.3, THEN
03.         Find the minimum level k > ℓ(x) where |N_x(4, k)| ≤ β^k.
04.         Move the vertex x up to level k, and update the relevant data structures as
            described in the proof of Lemma 3.5.
05.     ELSE
06.         Find a vertex x ∈ V that violates Invariant 3.2.
07.         IF there is a level 4 < k < ℓ(x) where |N_x(4, k − 1)| ≥ β^{k−1}, THEN
08.             Move the vertex x down to maximum such level k, and update the relevant
                data structures as described in the proof of Lemma 3.6.
09.         ELSE
10.             Move the vertex x down to level 4, and update the relevant data structures.
```

Figure 1: Subroutine: MAINTAIN-HP is called when an edge (u, v) is inserted into or deleted from G.

Lemma 3.4. *Consider any iteration of the* WHILE *loop in Figure 1 which changes the level of a vertex $x \in V$ from i to k. The vertex x becomes clean (i.e., satisfies both the invariants) at the end of the iteration. Furthermore, at the end of this iteration we have: (a) $|\mathcal{N}_x(4, k)| \leq \beta^k$, (b) $|\mathcal{N}_x(4, k-1)| \geq \beta^{k-1}$ if $k > 4$.*

Proof. There are three cases to consider, depending on how the vertex moves from level i to level k.

Case 1. The vertex x moves up from a level $i \in [4, L-1]$. In this case, the vertex moves up to the *minimum* level $k > i$ where $|\mathcal{N}_x(4, k)| \leq \beta^k$. This implies that $|\mathcal{N}_x(4, k-1)| > \beta^{k-1} > \beta^{k-5}$. Thus, the vertex x satisfies both the invariants after it moves to level k, and both the conditions (a) and (b) hold.

Case 2. The vertex x moves down from level i to a level $4 < k < i$. In this case, steps 07, 08 in Figure 1 imply that $k < i$ is the *maximum* level where $|\mathcal{N}_x(4, k-1)| \geq \beta^{k-1}$. Hence, we have $|\mathcal{N}_x(4, k)| < \beta^k$. So the vertex satisfies both the invariants after moving to level k, and both the conditions (a) and (b) hold.

Case 3. The vertex x moves down from level i to level $k = 4$. Here, steps 07, 09 in Figure 1 imply that $|\mathcal{N}_x(4, j-1)| < \beta^{j-1}$ for every level $4 < j < i$. In particular, setting $j = 5 = k + 1$, we get: $|\mathcal{N}_x(4, k)| < \beta^0 = 1 < \beta^4$. Thus, the vertex satisfies both the invariants after it moves down to level 4, and both the conditions (a) and (b) hold. □

Lemma 3.4 states that during any given iteration of the WHILE loop in Figure 1, we pick a dirty vertex x and make it clean. In the process, some neighbors of x become dirty, and they are handled in the subsequent

[4]The terms *below-degree* and *side-degree* of a vertex v refer to the values of $|\mathcal{N}_v(4, \ell(v)-1)|$ and $|\mathcal{N}_v(\ell(v), \ell(v))|$ respectively.

iterations of the same WHILE loop. When the WHILE loop terminates, every vertex is clean by definition. It now remains to analyze the time spent on implementing this WHILE loop after an edge insertion/deletion in the input graph. Lemma 3.4 will be crucial in this analysis. The intuition is as follows. The lemma guarantees that whenever a vertex x moves to a level $k > 4$, its below-degree is at least β^{k-1}. In contrast, Invariant 3.2 and Figure 1 ensure that whenever the vertex moves down from the same level k, its below-degree is less than β^{k-5}. Thus, the vertex loses at least $\beta^{k-1} - \beta^{k-5}$ in below-degree before it moves down from level k. This *slack* of $\beta^{k-1} - \beta^{k-5}$ help us bound the amortized update time. We next bound the time spent on a single iteration of the WHILE loop in Figure 1.

Lemma 3.5. *Consider any iteration of the* WHILE *loop in Figure 1 where a vertex x moves up to a level k from a level $i < k$ (steps 2 – 4). It takes $\Theta(\beta^k)$ time to implement such an iteration.*

Proof. First, we claim that the value of k (the level where the vertex x will move up to) can be identified in $\Theta(k - i)$ time. This is because we explicitly store the sizes of the lists $\mathcal{N}_x(4, i-1)$ and $\mathcal{N}_x(j, j)$ for all $j \geq i$. Next, we update the lists $\{\mathcal{C}_v^+, \mathcal{C}_v\}$ and the counters $\{\mu_v^+(c)\}$ for x and its neighbors as follows. FOR every level $j \in \{i, \ldots, k\}$ and every vertex $y \in \mathcal{N}_x(j, j)$:

- $\mathcal{C}_y^+ \leftarrow \mathcal{C}_y^+ \cup \{\chi(x)\}$, $\mathcal{C}_y \leftarrow \mathcal{C}_y \setminus \{\chi(x)\}$ and $\mu_y^+(\chi(x)) \leftarrow \mu_y^+(\chi(x)) + 1$.
- IF $j < k$, THEN
  - $\mu_x^+(\chi(y)) \leftarrow \mu_x^+(\chi(y)) - 1$.
  - If $\mu_x^+(\chi(y)) = 0$, then $\mathcal{C}_x^+ \leftarrow \mathcal{C}_x^+ \setminus \{\chi(y)\}$ and $\mathcal{C}_x \leftarrow \mathcal{C}_x \cup \{\chi(y)\}$.

The time spent on the above operations is bounded by the number of vertices in $\mathcal{N}_x(4, k)$.

Since the vertex x is moving up from level i to level $k > i$, we have to update the position of x in the neighborhood lists of the vertices $u \in \mathcal{N}_x(4, k)$. We also need to merge the lists $\mathcal{N}_x(4, i-1)$ and $\mathcal{N}_x(j, j)$ for $i \leq j < k$ into a single list $\mathcal{N}_x(4, k-1)$. In the process if some vertices $u \in \mathcal{N}_x(4, k)$ becomes dirty, then we need to put them in the correct dirty queue. This takes $\Theta(|\mathcal{N}_x(4, k)|)$ time.

By Lemma 3.4, we have $|\mathcal{N}_x(4, k)| \leq \beta^k$, and $|\mathcal{N}_x(4, k)| \geq |\mathcal{N}_x(4, k-1)| \geq \beta^{k-1}$. Since β is a constant, we conclude that it takes $\Theta(\beta^k)$ time to implement this iteration of the WHILE loop in Figure 1. □

Lemma 3.6. *Consider any iteration of the* WHILE *loop in Figure 1 where a vertex x moves down to a level k from a level $i > k$ (steps 5 – 10). It takes $O(\beta^i)$ time to implement such an iteration.*

Proof. We first bound the time spent on identifying the level $k < i$ the vertex x will move down to. Since the vertex x violates Invariant 3.2, we know that $|\mathcal{N}_x(4, i-1)| < \beta^{i-5} = O(\beta^i)$. Therefore, the algorithm can scan through the list $\mathcal{N}_x(4, i-1)$ and find the required level k in $\Theta(i + |\mathcal{N}_x(4, i-1)|)$ time. Next, we update the lists $\{\mathcal{C}_v^+, \mathcal{C}_v\}$ and the counters $\{\mu_v^+(c)\}$ for x and its neighbors as follows.

FOR every vertex $y \in \mathcal{N}_x(4, i-1) \cup \mathcal{N}_x(i, i)$:

- IF $i \geq \ell(y) > k$, THEN
  - $\mu_y^+(\chi(x)) \leftarrow \mu_y^+(\chi(x)) - 1$.
  - If $\mu_y^+(\chi(x)) = 0$, then $\mathcal{C}_y^+ \leftarrow \mathcal{C}_y^+ \setminus \{\chi(x)\}$ and $\mathcal{C}_y \leftarrow \mathcal{C}_y \cup \{\chi(x)\}$.
- IF $i > \ell(y) \geq k$, THEN
  - $\mathcal{C}_x^+ \leftarrow \mathcal{C}_x^+ \cup \{\chi(y)\}$, $\mathcal{C}_x \leftarrow \mathcal{C}_x \setminus \{\chi(y)\}$ and $\mu_x^+(\chi(y)) \leftarrow \mu_x^+(\chi(y)) + 1$.

The time spent on the above operations is bounded by the number of vertices in $\mathcal{N}_x(4,i)$.

Since the vertex x is moving down from level i to level $k < i$, we have to update the position of x in the neighborhood lists of the vertices $u \in \mathcal{N}_x(4,i)$. We also need to split the list $\mathcal{N}_x(4,i-1)$ into the lists $\mathcal{N}_x(4,k-1)$ and $\mathcal{N}_x(j,j)$ for $k \le j < i$. In the process if some vertices $u \in \mathcal{N}_x(4,i)$ become dirty, then we need to put them in the correct dirty queue. This takes $\Theta(|\mathcal{N}_x(4,i)|)$ time.

Figure 1 ensures that x satisfies Invariant 3.3 at level i before it moves down to a lower level. Thus, we have $|\mathcal{N}_x(4,i)| \le \beta^i$, and we spend $\Theta(1 + |\mathcal{N}_x(4,i)|) = O(\beta^i)$ time on this iteration of the WHILE loop. □

Corollary 3.7. *It takes $\Omega(\beta^k)$ time for a vertex x to move from a level i to a different level k.*

Proof. If $4 \le i < k$, then the corollary follows immediately from Lemma 3.5. For the rest of the proof, suppose that $i > k$. In this case, as per the proof of Lemma 3.6, the time spent is at least the size of the list $\mathcal{N}_x(4,k-1)$, and Lemma 3.4 implies that $|\mathcal{N}_x(4,k-1)| \ge \beta^{k-1}$. Hence, the total time spent is $\Omega(\beta^{k-1})$, which is also $\Omega(\beta^k)$ since β is a constant. Note that we ignored the scenarios where $k = 4$ since in that event β^k is a constant anyway. □

In Theorem 3.8, we bound the amortized update time for maintaining a hierarchical partition.

Theorem 3.8. *We can maintain a hierarchical partition of the vertex set V that satisfies Invariants 3.2 and 3.3 in $O(\log \Delta)$ amortized update time.*

We devote the rest of Section 3.2 to the proof of the above theorem using a token based scheme. The basic framework is as follows. For every edge insertion/deletion in the input graph we create at most $O(L)$ tokens, and we use one token to perform $O(\beta^2)$ units of computation. This implies an amortized update time of $O(\beta^2 \cdot L) = O(\beta^2 \cdot \log_\beta \Delta)$, which is $O(\log \Delta)$ since β is a constant.

Specifically, we associate $\theta(v)$ many tokens with every vertex $v \in V$ and $\theta(u,v)$ many tokens with every edge $(u,v) \in E$ in the input graph. The values of these tokens are determined by the following equalities.

$$\theta(u,v) \;=\; L - \max(\ell(u), \ell(v)). \tag{3.1}$$

$$\begin{aligned} \theta(v) \;&=\; \frac{\max\left(0, \beta^{\ell(v)-1} - |\mathcal{N}_v(4, \ell(v)-1)|\right)}{2\beta} && \text{if } \ell(v) > 4; \\ &=\; 0 && \text{otherwise.} \end{aligned} \tag{3.2}$$

Initially, the input graph G is empty, every vertex is at level 4, and $\theta(v) = 0$ for all $v \in V$. Due to the insertion of an edge (u,v) in G, the total number of tokens increases by at most $L - \max(\ell(u), \ell(v)) < L$, where $\ell(u)$ and $\ell(v)$ are the levels of the endpoints of the edge just before the insertion. On the other hand, due to the deletion of an edge (u,v) in the input graph, the value of $\theta(x)$ for $x \in \{u,v\}$ increases by at most $1/(2\beta)$, and the tokens associated with the edge (u,v) disappears. Overall, the total number of tokens increases by at most $1/(2\beta) + 1/(2\beta) \le O(L)$ due to the deletion of an edge. We now show that the work done during one iteration of the WHILE loop in Figure 1 is proportional to $O(\beta^2)$ times the net decrease in the total number of tokens during the same iteration. Accordingly, we focus on any single iteration of the WHILE loop in Figure 1 where a vertex x (say) moves from level i to level k. We consider two cases, depending on whether x moves to a higher or a lower level.

Case 1: The vertex x moves up from level i to level $k > i$. Immediately after the vertex x moves up to level k, we have $\mathcal{N}_x(4,k-1) \ge \beta^{k-1}$ and hence $\theta(x) = 0$. This follows from (3.2) and Lemma 3.4. Since $\theta(x)$ is always nonnegative, the value of $\theta(x)$ does not increase as x moves up to level k. We now focus on bounding the change in the total number of tokens associated with the neighbors of x. Note

that the event of x moving up from level i to level k affects only the tokens associated with the vertices $u \in \mathcal{N}_x(4,k)$. Specifically, from (3.2) we infer that for every vertex $u \in \mathcal{N}_x(4,k)$, the value of $\theta(u)$ increases by at most $1/(2\beta)$. On the other hand, for every vertex $u \in \mathcal{N}_x(k+1,L)$, the value of $\theta(u)$ remains unchanged. Thus, the total number of tokens associated with the neighbors of x increases by at most $(2\beta)^{-1} \cdot |\mathcal{N}_x(4,k)| \leq (2\beta)^{-1} \cdot \beta^k = \beta^{k-1}/2$. The inequality follows from Lemma 3.4. To summarize, the total number of tokens associated with all the vertices increases by at most $\beta^{k-1}/2$.

We now focus on bounding the change in the total number of tokens associated with the edges incident on x. From (3.1) we infer that for every edge (x,u) with $u \in \mathcal{N}_x(4,k-1)$, the value of $\theta(x,u)$ drops by at least one as the vertex x moves up from level $i < k$ to level k. For every other edge (x,u) with $u \in \mathcal{N}_x(k,L)$, the value of $\theta(x,u)$ remains unchanged. Overall, this means that the total number of tokens associated with the edges drops by at least $|\mathcal{N}_x(4,k-1)| \geq \beta^{k-1}$. The inequality follows from Lemma 3.4. To summarize, the total number of tokens associated with the edges decreases by at least β^{k-1}.

From the discussion in the preceding two paragraphs, we reach the following conclusion: As the vertex x moves up from level $i < k$ to level k, the total number of tokens associated with all the vertices and edges decreases by at least $\beta^{k-1} - \beta^{k-1}/2 \geq \beta^{k-1}/2$. In contrast, Lemma 3.5 states that it takes $O(\beta^k)$ time taken to implement this iteration of the WHILE loop in Figure 1. Hence, we derive that the time spent on updating the relevant data structures is at most $O(2\beta) = O(\beta^2)$ times the net decrease in the total number of tokens. This concludes the proof of Theorem 3.8 for Case 1.

Case 2: The vertex x moves down from level i to level $k < i$. As in Case 1, we begin by observing that immediately after the vertex x moves down to level k, we have $|\mathcal{N}_x(4,k-1)| \geq \beta^{k-1}$ if $k > 4$, and hence $\theta(x) = 0$. This follows from Lemma 3.4. The vertex x violates Invariant 3.2 just before moving from level i to level k (see step 6 in Figure 1). In particular, just before the vertex moves down from level i to level k, we have $|\mathcal{N}_x(4,i-1)| < \beta^{i-5}$ and $\theta(x) \geq (2\beta)^{-1} \cdot \left(\beta^{i-1} - \beta^{i-5}\right) = \beta^{i-2}/2 - \beta^{i-6}/2 \geq \beta^{i-2}/3$. The last inequality holds since β is a sufficiently large constant. So the number of tokens associated with x drops by at least $\beta^{i-2}/3$ as it moves down from level i to level k. Also, from (3.2) we infer that the value of $\theta(u)$ does not increase for any $u \in \mathcal{N}_x$ as x moves down to a lower level. Hence, we conclude that:

$$\text{The total number of tokens associated with all the vertices drops by at least } \beta^{i-2}/3. \tag{3.3}$$

We now focus on bounding the change in the number of tokens associated with the edges incident on x. From (3.1) we infer that the number of tokens associated with an edge $(u,x) \in \mathcal{N}_x(4,i-1)$ increases by $(i - \max(k, \ell(u)))$ as x moves down from level i to level k. In contrast, the number of tokens associated with any other edge $(u,x) \in \mathcal{N}_x(i,L)$ does not change as the vertex x moves down from level i to a lower level. Let Γ be the increase in the total number of tokens associated with all the edges. Thus, we have:

$$\Gamma = \sum_{(u,x) \in \mathcal{N}_x(4,i-1)} (i - \max(k, \ell(u))) = \sum_{j=k}^{i-1} |\mathcal{N}_x(4,j)|, \tag{3.4}$$

where the last equality follows by rearrangement. Next, recall that the vertex x moves down from level i to level k during the concerned iteration of the WHILE loop in Figure 1. Accordingly, steps 7 – 10 in Figure 1 implies that $|\mathcal{N}_x(4,j-1)| < \beta^{j-1}$ for all levels $i > j > k$. This is equivalent to the following statement:

$$|\mathcal{N}_x(4,j)| < \beta^j \text{ for all levels } i - 1 > j \geq k. \tag{3.5}$$

Next, step 6 in Figure 1 implies that the vertex x violates Invariant 3.2 at level i. Thus, we get: $|\mathcal{N}_x(4,i-1)| < \beta^{i-5}$. Note that for all levels $j < i$, we have $\mathcal{N}_x(4,j) \subseteq \mathcal{N}_x(4,i-1)$ and $|\mathcal{N}_x(4,j)| \leq |\mathcal{N}_x(4,i-1)|$. Hence, we get: $|\mathcal{N}_x(4,j)| < \beta^{i-5}$ for all levels $j < i$. Combining this observation with (3.5), we get:

$$|\mathcal{N}_x(4,j)| < \min(\beta^{i-5}, \beta^j) \text{ for all levels } i - 1 \geq j \geq k. \tag{3.6}$$

Plugging (3.6) into (3.4), we get:

$$\Gamma < 5\beta^{i-5} + \sum_{j=k}^{i-6} \beta^j < \beta^{i-3}. \tag{3.7}$$

In the above derivation, the last inequality holds since β is a sufficiently large constant.

From (3.3) and (3.7), we reach the following conclusion: As the vertex x moves down from level i to a level $k < i$, the total number of tokens associated with all the vertices and edges decreases by at least $\beta^{i-2}/3 - \Gamma > \beta^{i-2}/3 - \beta^{i-3} = \Omega(\beta^{i-2})$. In contrast, by Lemma 3.6 it takes $O(\beta^i)$ time to implement this iteration of the WHILE loop in Figure 1. Hence, we derive that the time spent on updating the relevant data structures is at most $O(\beta^2)$ times the net decrease in the total number of tokens. This concludes the proof of Theorem 3.8 for Case 2.

3.3 The recoloring subroutine.

Whenever we want to change the color of a vertex $v \in V$, we call the subroutine RECOLOR(v) as described in Figure 2. We ensure that the hierarchical partition does not change during a call to this subroutine. Specifically, throughout the duration of any call to the RECOLOR subroutine, the value of $\ell(x)$ remains the same for every vertex $x \in V$. We also ensure that the hierarchical partition satisfies Invariants 3.2 and 3.3 before any making any call to the RECOLOR subroutine.

During a call to the subroutine RECOLOR(v), we randomly choose a color c for the vertex v from the subset $B_v \cup U_v \subseteq \mathcal{C}_v$. In case the random color c lies in U_v, we find the unique neighbor $v' \in \mathcal{N}_v(4, \ell(v)-1)$ of v which is assigned this color, and then we recursively recolor v'. Since the level of v' is strictly less than that of v, the maximum depth of this recursion is L. We now bound the time spent on a call to RECOLOR(v).

1. Choose $c \in B_v \cup U_v$ uniformly at random. *// These notations are defined in Section 3.1.*
2. Set $\chi(v) \leftarrow c$.
3. Update the relevant data structures as described in the proof of Lemma 3.9.
4. IF $c \in U_v$:
5. Find the *unique* vertex $v' \in \mathcal{N}_v(4, \ell(v) - 1)$ with $\chi(v') = c$.
6. RECOLOR(v').

Figure 2: Subroutine RECOLOR(v)

Lemma 3.9. *It takes $O(\beta^{\ell(v)})$ time to implement one call to RECOLOR(v). This includes the total time spent on the chain of subsequent recursive calls that originate from the call to RECOLOR(v).*

Proof. Let us assume that $\ell(v) = i$ and $\chi(v) = c'$ just before the call to RECOLOR(v). To implement Step 01 in Figure 2, the vertex v scans the neighborhood list $\mathcal{N}_v(4, i-1)$ and computes the subset colors $T_v \subseteq \mathcal{C}_v$ which appear twice or more among the these vertices. The vertex v keeps these colors T_v in a separate list and deletes every color in T_v from the list $\mathcal{C}_v$. On completion, the list $\mathcal{C}_v$ consists of the colors in $B_v \cup U_v$ and the algorithm samples a random color c from this list.[5] Next, the algorithm adds all the colors in T_v back to the list $\mathcal{C}_v$, thereby restoring the list $\mathcal{C}_v$ to its actual state. The algorithm can also do another scan of $\mathcal{N}_v(4, i-1)$ to check whether $c \in B_v$ or $c \in U_v$. The total time taken to do all this is $\Theta(|\mathcal{N}_v(4, i-1)|)$ which by Invariants 3.2 and 3.3 is $\Theta(\beta^{\ell(v)})$. After changing the color of the vertex v from c' to c in step 02, the algorithm needs to update the data structures (see Section 3.1) as follows.

[5]Note that we might have $|B_v \cup U_v| \gg \beta^{\ell(v)}$ and so it is not clear how to sample in $O(\beta^{\ell(v)})$ time. The modification required here is that it is sufficient to sample from the first $\beta^{\ell(v)}$ elements of $B_v \cup U_v$. For clarity of exposition, we ignore this issue.

- FOR every vertex $w \in \mathcal{N}_v(4, i)$:
  - $\mu_w^+(c') \leftarrow \mu_w^+(c') - 1$.
  - IF $\mu_w^+(c') = 0$, THEN $\mathcal{C}_w^+ \leftarrow \mathcal{C}_w^+ \setminus \{c'\}$ and $\mathcal{C}_w \leftarrow \mathcal{C}_w \cup \{c'\}$.
  - $\mathcal{C}_w^+ \leftarrow \mathcal{C}_w^+ \cup \{c\}$, $\mathcal{C}_w \leftarrow \mathcal{C}_w \setminus \{c\}$ and $\mu_w^+(c) \leftarrow \mu_w^+(c) + 1$.

The above operations also take $\Theta(|\mathcal{N}_v(4, i)|) = \Theta(\beta^{\ell(v)})$ time, as per Invariants 3.2 and 3.3.

Finally, in the subsequent recursive calls suppose we recolor the vertices $y_1, y_2, \ldots$. Note that $\ell(x) > \ell(y_1) > \ell(y_2) > \cdots$. Therefore the total time taken can be bounded by $\Theta(\beta^{\ell(x)} + \beta^{\ell(y_1)} + \cdots) = \Theta(\beta^{\ell(x)})$ since it is a geometric series sum. This completes the proof. □

3.4 The complete algorithm and analysis.

Initially, when the graph $G = (V, E)$ is empty, every vertex $v \in V$ belongs to level 4 and picks a random color $\chi(v) \in \mathcal{C}$. At this point, the coloring χ is proper since there are no edges, and Invariants 3.2 and 3.3 are vacuously satisfied. Now, by inductive hypothesis, suppose that before the insertion or deletion of an edge in G, we have the guarantee that: (1) χ is a proper coloring and (2) Invariants 3.2, 3.3 are satisfied. We handle the insertion or deletion of this edge in G according to the procedure in Figure 3.

Specifically, after the insertion or deletion of an edge (u, v), we first update the hierarchical partition by calling the subroutine MAINTAIN-HP (see Figure 1). At the end of the call to this subroutine, we know for sure that Invariants 3.2 and 3.3 are satisfied. At this point, we check if the existing coloring χ is proper. The coloring χ can become invalid only if the edge (u, v) is getting inserted and $\chi(u) = \chi(v)$. In this event, we find the endpoint $x \in \{u, v\}$ that was recolored last, i.e., the one with the larger τ_x. Without any loss of generality, let this endpoint be v. We now change the color of v by calling the subroutine RECOLOR(v). At the end of the call to this subroutine, we know for sure that the coloring is proper. Thus, we can now apply the inductive hypothesis for the next insertion or deletion of an edge.

On INSERT/DELETE((u, v)):

MAINTAIN-HP. *// See Figure 1.*

In case (u, v) is inserted and $\chi(u) = \chi(v)$:

Suppose that $\tau_v > \tau_u$. *// This notation is defined in Section 3.1.*

RECOLOR(v).

Figure 3: Dynamic algorithm to maintain $\Delta + 1$ vertex coloring

We first bound the amortized time spent on all the calls to the RECOLOR subroutine in Lemma 3.10. From Theorem 3.8 and Lemma 3.10, we get the main result of this section, which is stated in Theorem 3.1.

Lemma 3.10. *Consider a sequence of T edge insertions/deletions starting from an empty graph $G = (V, E)$. Let T_R and T_{HP} respectively denote the total time spent on all the calls to the RECOLOR and* MAINTAIN-HP *subroutines during these T edge insertions/deletions. Then $\mathbf{E}[T_R] \leq O(T) + O(T_{HP})$.*

Proof. Since edge deletions don't lead to recoloring, we need to bother only with edge insertions. Consider the scenario where an edge (u, v) is being inserted into the graph at time τ. Without any loss of generality, assume that $\tau_v > \tau_u$. Recall that these are the last times before τ when v and u were recolored. Suppose that the vertex v is at level i immediately after we have updated the hierarchical partition following the insertion of the edge at time τ. Thus, if $\chi(u) = \chi(v)$ at this point in time, then the subroutine RECOLOR(v) will

be called to change the color of the endpoint v. On the other hand, if $\chi(u) \neq \chi(v)$ at this point in time, then no vertex will be recolored. Furthermore, suppose that the vertex v was at level j during the call to RECOLOR(v) at time τ_v. The analysis is done via three cases.

Case 1: $i > j$. In this case, at some point in time during the interval $[\tau_v, \tau]$, the subroutine MAINTAIN-HP raised the level of the vertex v to i. Corollary 3.7 implies that this takes $\Omega(\beta^i)$ time. On the other hand, even if the subroutine RECOLOR(v) is called at time τ, by Lemma 3.9 it takes $O(\beta^i)$ time to implement that call. So the total time spent on all such calls to the RECOLOR subroutine is at most $O(T_{HP})$.

Case 2: $4 < i \leq j$. In this case, we use the fact that the vertex v picks a random color at time τ_v. In particular, by Lemma 3.9 the expected time spent on recoloring the vertex v at time τ is at most $O(\beta^i) \cdot \Pr[\mathcal{E}_\tau]$, where $\mathcal{E}_\tau$ is the event that $\chi(u) = \chi(v)$ just before the insertion at time τ. We wish to bound this probability $\Pr[\mathcal{E}_\tau]$, which is evaluated over the past random choices of the algorithm *which the adversary fixing the order of edge insertions is oblivious to*.[6] We do this by using the principle of deferred decision.

Let c_u and c_v respectively denote the colors assigned to the vertices u and v during the calls to the subroutines RECOLOR(u) and RECOLOR(v) at times τ_u and τ_v. Note that the event $\mathcal{E}_\tau$ occurs iff $c_u = c_v$. Condition on all the random choices made by the algorithm till just before the time τ_v. Since $\tau_u < \tau_v$, this fixes the color c_u. At time τ_v, the vertex v picks the color c_v uniformly at random from the subset of colors $B_v \cup U_v$. Let λ denote the size of this subset $B_v \cup U_v$ at time τ_v. Clearly, the event $c_v = c_u$ occurs with probability $1/\lambda$, i.e., we have $\Pr[\mathcal{E}_\tau] = 1/\lambda$. It now remains to lower bound λ. Since $4 < i \leq j$, Claim 3.1 and Invariant 3.2 imply that when the vertex v gets recolored at time τ_v, we have: $\lambda = |B_v \cup U_v| \geq 1 + |\mathcal{N}_v(4, j-1)|/2 \geq 1 + \beta^{j-5}/2 = \Omega(\beta^{j-5}) = \Omega(\beta^{i-5})$. To summarize, the expected time spent on the possible call to RECOLOR(v) at time τ is at most $O(\beta^i) \cdot \Pr[\mathcal{E}_\tau] = O(\beta^i) \cdot (1/\lambda) = O(\beta^5) = O(1)$. Hence, the total time spent on all such calls to the RECOLOR subroutine is at most $O(T)$.

Case 3. $i = 4$. Even if RECOLOR(v) is called at time τ, by Lemma 3.9 at most $O(\beta^4) = O(1)$ time is spent on that call. So the total time spent on all such calls to the RECOLOR subroutine is $O(T)$. □

Proof of Theorem 3.1. The theorem holds since $\mathbf{E}[T_R] + T_{HP} \leq O(T) + O(T_{HP}) \leq O(T) + O(T \log \Delta) = O(T \log \Delta)$. The first and the second inequalities respectively follow from Lemma 3.10 and Theorem 3.8.

4 A Deterministic Dynamic Algorithm for $(1 + o(1))\Delta$ Vertex Coloring

Let $G = (V, E)$ denote the input graph that is changing dynamically, and let Δ be an upper bound on the maximum degree of any vertex in G. For now we assume that the value of Δ does not change with time. In Section 6, we explain how to relax this assumption. Our main result is stated in the theorem below.

Theorem 4.1. *We can maintain a $(1 + o(1))\Delta$ vertex coloring in a dynamic graph deterministically in $O(\lg^{5+o(1)} \Delta / \lg \lg^2 \Delta)$ amortized update time.*

4.1 Notations and preliminaries.

Throughout Section 4, we define three parameters η, L, λ as follows.

$$\eta = e^{16/\lg \lg \Delta}, L = \left\lfloor \frac{\lg(\eta\Delta)}{\lg \lg \Delta} \right\rfloor \text{ and } \lambda = \left\lceil 2^{\frac{\lg(\eta\Delta)}{L}} \right\rceil. \tag{4.1}$$

We will use λ^L colors. From (4.1) and Lemma 4.2, it follows that $\lambda^L \leq \eta\Delta = (1 + o(1))\Delta$ when $\Delta = \omega(1)$. In Lemma 4.2, we establish a couple of useful bounds on the parameters η, L and λ.

[6] In case 1, we used the trivial upper bound $\Pr[\mathcal{E}_\tau] \leq 1$.

Lemma 4.2. *We have:*

1. $\lg \Delta \leq \lambda \leq 2 \lg^{1+o(1)} \Delta$, *and*
2. $\lambda^L \leq \eta\Delta \leq (\lambda+1)^L$.

Proof. From (4.1) we infer that:

$$\lambda \geq 2^{\frac{\lg(\eta\Delta)}{L}} \geq 2^{\frac{\lg(\eta\Delta)}{\lg(\eta\Delta)/\lg\lg\Delta}} = \lg\Delta.$$

From (4.1) we also infer that:

$$\begin{aligned}
\lambda &\leq 2 \cdot 2^{\frac{\lg(\eta\Delta)}{L}} \\
&\leq 2 \cdot 2^{\frac{\lg(\eta\Delta)}{\lg(\eta\Delta)/\lg\lg\Delta - 1}} \\
&= 2 \cdot 2^{\frac{\lg(\eta\Delta)}{\lg(\eta\Delta) - \lg\lg\Delta} \cdot \lg\lg\Delta} \\
&= 2 \cdot 2^{(1+o(1)) \cdot \lg\lg\Delta} = 2\lg^{1+o(1)}\Delta.
\end{aligned}$$

The proves part (1) of the lemma. Next, note that:

$$\lambda^L \leq \left(2^{\frac{\lg(\eta\Delta)}{L}}\right)^L = \eta\Delta \leq \left\lceil 2^{\frac{\lg(\eta\Delta)}{L}} \right\rceil^L = (\lambda+1)^L.$$

This proves part (2) of the lemma. □

We let $\mathcal{C} = \{1, \ldots, \lambda^L\}$ denote the palette of all colors. Note that $|\mathcal{C}| = \lambda^L = (1+o(1))\Delta$. Indeed, we view the colors available to us as L-tupled vectors where each coordinate takes one of the values from $\{1, \ldots, \lambda\}$. In particular, the color assigned to any vertex v is denotes as $\chi(v) = (\chi_1(v), \ldots, \chi_L(v))$, where $\chi_i(v) \in [\lambda]$ for each $i \in [L]$. Given such a coloring $\chi : V \to \mathcal{C}$, for every index $i \in [L]$ we define $\chi_i^*(v) := (\chi_1(v), \ldots, \chi_i(v))$ to be the i-tuple denoting the first i coordinates of $\chi(v)$. For notational convenience, we define $\chi_0^*(v) := \perp$ for all v. For all $i \in [L]$ and $\alpha \in [\lambda]$, we let $\chi_{i\to\alpha}^*(v) = (\chi_1(v), \ldots, \chi_{i-1}(v), \alpha)$ denote the i-tuple whose first $(i-1)$ coordinates are the same as that of χ but whose i^{th} coordinate is α.

For all $i \in [L]$ and $\alpha \in [\lambda]$, we define the subsets $N_i^*(v) = \{u \in V : (u,v) \in E \text{ and } \chi_i^*(u) = \chi_i^*(v)\}$ and $N_{i\to\alpha}^*(v) = \{u \in V : (u,v) \in E \text{ and } \chi_i^*(u) = \chi_{i\to\alpha}^*(v)\}$. In other words, the set $N_i^*(v)$ consists of all the neighbors of a vertex $v \in V$ whose colors have the same first i coordinates as the color of v. On the other hand, the set $N_{i\to\alpha}^*(v)$ denotes the status of the set N_i^* *in the event* that the vertex v decides to change the ith coordinate of its color to α. In particular, if $\chi_i(v) \neq \alpha$, then $N_i^*(v) \cap N_{i\to\alpha}^*(v) = \emptyset$. Going over all possible choices of $\chi_i(v)$ we get the following

$$N_{i-1}^*(v) = \bigcup_{\alpha\in[\lambda]} N_{i\to\alpha}^*(v)$$

Also note that $N_0^*(v)$ is the full neighborhood of v. Define $D_i^*(v) = |N_i^*(v)|$ and $D_{i\to\alpha}^*(v) = |N_{i\to\alpha}^*(v)|$. The above observations is encapsulated in the following corollary.

Corollary 4.3. *For every vertex $v \in V$, and every index $j \in \{1, \ldots, L\}$, the set $N_{j-1}^*(v)$ is partitioned into the subsets $N_{j\to\alpha}^*(v)$ for $\alpha \in \{1, \ldots, \lambda\}$. In particular, we have: $D_{j-1}^*(v) = \sum_{\alpha\in[\lambda]} D_{j\to\alpha}^*(v)$.*

We maintain the following invariant.

Invariant 4.4. *For all $v \in V$, $i \in [0, L]$, we have $D_i^*(v) \leq (\Delta/\lambda^i) \cdot f(i)$, where $f(i) = ((\lambda+1)/(\lambda-1))^i$.*

For every j, we have $f(j) > 1$ and $f(j-1) \leq f(j)$. We now give a brief intuitive explanation for the above invariant. Associate a rooted λ-ary tree T_v of depth L with every vertex $v \in V$. We shall refer to the vertices of this tree T_v as meta-vertices, to distinguish them from the vertices of the input graph $G = (V, E)$. The total number of leaves in this tree is λ^L, which is the same as the total number available colors. Thus, we can ensure that each root to leaf path in this tree corresponds to a color in a natural way, and any internal meta-vertex at depth i corresponds to the ith coordinate of a color. The quantity $D_i^*(v)$ can now be interpreted as follows. Consider the meta-vertex (say) x_i at depth i on the unique root to leaf path corresponding to the color of v. Let μ_i denote the number of all neighbors of v in G such that this meta-vertex x_i also belongs to the root to leaf paths for their corresponding colors. Then we have $D_i^*(v) = \mu_i$. Note that if $i = 0$, then the meta-vertex x_i is the root of the tree, which is at depth zero. It follows that if $i = 0$, then μ_i equals the degree of the vertex v in the input graph G, which is at most Δ. Thus, we have $\mu_0 \leq \Delta$. Now, let $y_1, \ldots, y_\lambda$ denote the children of the root x_0 in this λ-ary tree T_v. A simple counting argument implies that there exists an index $j \in \{1, \ldots, \lambda\}$ with the following property: At most a $1/\lambda$ fraction of the neighbors of v in G have colors whose corresponding root to leaf paths contain the meta-vertex y_j. Thus, if it were the case that the root to leaf path corresponding to the color of v also passes through such a meta-vertex y_j, then we would have $D_i^*(v) = \mu_i \leq \Delta/\lambda$ for $i = 1$. Invariant 4.4, on the other hand, gives a *slack* of $f(1)$ and requires that $D_i^*(v) \leq (\Delta/\lambda) \cdot f(1)$ for $i = 1$. We can interpret the invariant in this fashion for every subsequent index $i \in \{2, \ldots, L\}$ by iteratively applying the same principle. The reader might find it helpful to keep this interpretation in mind while going through the formal description of the algorithm and its analysis.

Lemma 4.5. *If Invariant 4.4 holds then χ is a proper vertex coloring.*

Proof. Claim 4.1 implies that for $i = L$, the invariant reduces to: $D_L^*(v) < 1$. Since $D_L^*(v)$ is a nonnegative integer, we get $D_L^*(v) = 0$. Since $D_L^*(v)$ is the number of neighbors of v who are assigned the color $\chi(v)$, no two adjacent vertices can get the same color. The invariant thus ensures a proper vertex coloring.

Claim 4.1. *We have: $(\Delta/\lambda^L) \cdot f(L) < 1$.*

In order to prove Claim 4.1, we derive that:

$$
\begin{aligned}
& (\Delta/\lambda^L) f(L) \\
\leq\ & (((\lambda+1)^L/\eta)/\lambda^L) \cdot f(L) && (4.2)\\
=\ & (1/\eta) \cdot (1+1/\lambda)^L \cdot (1+2/(\lambda-1))^L \\
\leq\ & (1/\eta) \cdot (1+1/\lambda)^L \cdot (1+4/\lambda)^L && (4.3)\\
\leq\ & (1/\eta) \cdot (1+7/\lambda)^L && (4.4)\\
\leq\ & (1/\eta) \cdot e^{7L/\lambda} \\
\leq\ & (1/\eta) \cdot e^{7 \lg(\eta\Delta)/(\lambda \lg\lg \Delta)} && (4.5)\\
\leq\ & (1/\eta) \cdot e^{7 \lg(\Delta^2)/(\lg \Delta \lg\lg \Delta)} && (4.6)\\
\leq\ & (1/\eta) \cdot e^{14/\lg\lg \Delta} \\
<\ & 1. && (4.7)
\end{aligned}
$$

Step (4.2) follows from part (2) of Lemma 4.2. Steps (4.3), (4.4) hold as long as $\lambda \geq 2$.[7] Step (4.5) follows from (4.1). Step (4.6) follows from (4.1) and part (1) of Lemma (4.2). Step (4.7) follows from (4.1). □

Data structures. For every vertex $v \in V$, our dynamic algorithm maintains the following data structures.

[7] When $\lambda < 2$, we have $\Delta = O(1)$ and so we can trivially maintain a $(\Delta+1)$-vertex coloring in $O(\Delta) = O(1)$ update time.

- For all $i \in [0, L]$, the set $N_i^*(v)$ as a doubly linked list and the counter $D_i^*(v)$. It will be the responsibility of the neighbors of v to update the list $N_i^*(v)$ when they change their own colors. Using appropriate pointers, we will ensure that any given node x can be inserted into or deleted from any given list $N_i^*(y)$ in $O(1)$ time. Note that each vertex figures out if it satisfies the Invariant 4.4 or not.
- The color $\chi(v) = (\chi_1(v), \ldots, \chi_L(v))$ assigned to the vertex v.

4.2 The algorithm.

Initially, since the edge-set of the input graph is empty, we can assign each vertex an arbitrary color. For concreteness, we set $\chi_i(v) = 1$ for all $i \in [L]$ and $v \in V$. Since there are no edges, $D_i^*(v) = 0$ for all i, v and so Invariant 4.4 vacuously holds at this point. We show how to ensure that the invariant continues to remain satisfied even after any edge insertion or deletion, and bound the amortized update time.

Deletion of an edge. Suppose that an edge (u, v) gets deleted from G. No vertex changes its color due to this deletion. We only need to update the relevant data structures. Without any loss of generality, suppose that $i \in [0, L]$ is the largest index for which we have $\chi_i^*(v) = \chi_i^*(u)$. Then for every vertex $x \in \{u, v\}$ and every $j \in [0, i]$, we delete the vertex $y \in \{u, v\} \setminus \{x\}$ from the set $N_i^*(x)$ and decrement the value of the counter $D_j^*(x)$ by one. This takes $O(L)$ time. To summarize, deletion of an edge can be handled in $O(L)$ worst case update time. If Invariant 4.4 was satisfied just before the edge deletion, then the invariant continues to remain satisfied after the edge deletion since the LHS of the invariant can only decrease.

Insertion of an edge. Suppose that an edge (u, v) gets inserted into the graph. We first update the relevant data structures as follows. Let $i \in [0, L]$ be the largest index such that $\chi_i^*(u) = \chi_i^*(v)$. Note that if $i = L$ then the colors of u and v are the same. For every vertex $x \in \{u, v\}$ and every $j \in [0, i]$, we insert the vertex $y \in \{u, v\} \setminus \{x\}$ into the set $N_j^*(x)$ and increment the value of the counter $D_j^*(x)$ by one. This takes $O(L)$ time in the worst case. Next, we focus on ensuring that Invariant 4.4 continues to hold. Towards this end, we execute the subroutine described in Figure 4. Lemma 4.6 implies the correctness of the algorithm. In Lemma 4.7, we upper bound the time spent on a given iteration of the WHILE loop in Figure 4.

1. WHILE there is some vertex $x \in V$ that violates Invariant 4.4:
2. Let $k \in [L]$ be the smallest index such that $D_k^*(x) > (\Delta/\lambda^k) \cdot f(k)$.
3. FOR $j = k$ to L:
4. Find an $\alpha \in [\lambda]$ that minimizes $D_{j \to \alpha}^*(x)$.
5. Set $\chi_j(x) = \alpha$ and update the relevant data structures. // See Lemma 4.7.

Figure 4: Ensuring Invariant 4.4 after an edge insertion.

Lemma 4.6. *Consider an iteration of the* WHILE *loop in Figure 4 for a vertex $x \in V$. At the end of this iteration, the vertex x satisfies Invariant 4.4.*

Proof. Consider any iteration of the FOR loop in steps 3 – 5. By induction hypothesis, suppose that just before this iteration we have $D_i^*(x) \leq (\Delta/\lambda^i) \cdot f(i)$ for all $i \in [0, j-1]$. Due to step 2, the induction hypothesis holds just before the first iteration of the FOR loop, when we have $j = k$. Since $D_{j-1}^*(x) = \sum_{\alpha \in [\lambda]} D_{j \to \alpha}^*(x)$ by Corollary 4.3, the α that minimizes $D_{j \to \alpha}^*(x)$ satisfies: $D_{j \to \alpha}^*(x) \leq (1/\lambda) \cdot D_{j-1}^*(x) \leq (\Delta/\lambda^j) \cdot f(j-1) \leq (\Delta/\lambda^j) \cdot f(j)$. Accordingly, after executing step 6 we get: $D_j^*(x) \leq (\Delta/\lambda^j) \cdot f(j)$. Thus, the induction hypothesis remains valid for the next iteration of the FOR loop. At the end of the FOR loop, we get $D_j^*(x) \leq (\Delta/\lambda^j) \cdot f(j)$ for all $j \in [0, L]$, and hence the vertex x satisfies Invariant 4.4. □

Lemma 4.7. *It takes $O(L \cdot \lambda + L \cdot \frac{\Delta}{\lambda^{k-1}} \cdot f(k-1))$ time for one iteration of the* WHILE *loop in Figure 4, where k is defined as per Step 2 in Figure 4.*

Proof. Consider any iteration of the WHILE loop that changes the color of a vertex $x \in V$. Since we store the value of $D^*_j(x)$ for every $j \in [0, L]$, it takes $O(L)$ time to find the index $k \in [0, L]$ as defined in step 2 of Figure 4. Next, for every index $j \in [k, L]$ and every vertex $u \in N^*_j(x)$, we set $N^*_j(u) = N^*_j(u) \setminus \{x\}$ and $D^*_j(u) = D^*_j(u) - 1$. Since the vertex x is going to change the kth coordinate of its color, the previous step is necessary to ensure that no vertex u mistakenly continues to include x in the set $N^*_j(u)$ for $j \in [k, L]$. This takes $O(\sum_{j=k}^{L} |N^*_j(x)|)$ time. Since $N^*_j(x) \subseteq N^*_{j-1}(x)$ for all $j \in [k, L]$, the time taken is actually $O((L-k-1) \cdot |N^*_{k-1}(x)|) = O(L \cdot (\Delta/\lambda^{k-1}) \cdot f(k-1))$. The last equality follows from step 2 in Figure 4. At this point, we also set $N^*_j(x) = \emptyset$ and $D^*_j(x) = 0$ for all $j \in [k, L]$. We shall rebuild the sets $N^*_j(x)$ during the FOR loop in steps 3 - 5. Applying a similar argument as before, we conclude that this also takes $O(L \cdot (\Delta/\lambda^{k-1}) \cdot f(k-1))$ time. It now remains to bound the time spent on the FOR loop.

Consider any iteration of the FOR loop as described by steps 3 - 5 in Figure 4. By inductive hypothesis, suppose that for every index $i \in [0, j-1]$ and every vertex $v \in V$, the list $N^*_i(v)$ is now consistent with the changes we have made to the color of x in the earlier iterations of the FOR loop. Our first goal is to compute the index $\alpha \in [\lambda]$, which we do by scanning the vertices $u \in N^*_{j-1}(x)$. In the beginning of this scan, we initialize a counter $Z_\alpha = 0$ for every $\alpha \in [\lambda]$. Subsequently, while considering any vertex $u \in N^*_{j-1}(x)$ during the scan, we set $Z_{c_j(u)} = Z_{c_j(u)} + 1$. At the end of the scan, we return the index $\alpha \in [\lambda]$ that minimizes the value of Z_α. Thus, overall it takes $O(\lambda + |N^*_{j-1}(x)|)$ time to find the index α. Next, for every vertex $u \in N^*_{j-1}(x)$ with $c_j(u) = \alpha$, we set $N^*_j(u) = N^*_j(u) \cup \{x\}$, $N^*_j(x) = N^*_j(x) \cup \{u\}$, $D^*_j(u) = D^*_j(u) + 1$ and $D^*_j(x) = D^*_j(x) + 1$. It takes $O(|N^*_{j-1}(x)|)$ time to implement this step. Now, we set $c_j(x) = \alpha$ in $O(1)$ time. At this stage, we have updated all the relevant data structures for the index j and changed the jth coordinate of the color of x. This concludes the concerned iteration of the FOR loop. Since $N^*_{j-1}(x) \subseteq N^*_{k-1}(x)$, the total time spent on this iteration is $O(\lambda + |N^*_{j-1}(x)|) = O(\lambda + |N^*_{k-1}(x)|) = O(\lambda + (\Delta/\lambda^{k-1}) \cdot f(k-1))$, as per step 2 in Figure 4.

Since the FOR loop runs for at most L iterations, the total time spent on the FOR loop is at most $O(L \cdot \lambda + L \cdot \frac{\Delta}{\lambda^{k-1}} \cdot f(k-1))$. Combining this with the discussion in the first paragraph of the proof of this lemma, we infer that the total time spent on one iteration of the WHILE loop is also $O(L \cdot \lambda + L \cdot \frac{\Delta}{\lambda^{k-1}} \cdot f(k-1))$. □

4.3 Bounding the amortized update time.

In this section, we prove Theorem 4.1 by bounding the amortized update time of the algorithm described in Section 4.2. Recall that handling the deletion of an edge takes $O(L)$ time in the worst case. Furthermore, ignoring the time spent on the WHILE loop in Figure 4, handling the insertion of an edge also takes $O(L)$ time in the worst case. From Lemma 4.2 we have $L = O(\lg \Delta / \lg \lg \Delta)$. Thus, it remains to bound the time spent on the WHILE loop in Figure 4. We focus on this task for the remainder of this section.

The main idea is the following: by Lemma 4.7 the WHILE loop processing vertex x takes a *long* time when k is *small* which in turn implies $D^*_k(x)$ is *large*. However, in the FOR loop we choose the colors which minimize precisely the values of $D^*_{j \to \alpha}(x)$. Therefore these quantities cannot be large too often.

Consider any iteration of the WHILE loop in Figure 4, which changes the color of a vertex $x \in V$. Let S^+_x (resp. S^-_x) be the set of all ordered pairs (i, v) such that the value of $D^*_i(v)$ increases (resp. decreases) due to this iteration. The following lemma precisely bounds the increases and decreases of these D^* values.

Lemma 4.8. *During any single iteration of the* WHILE *loop in Figure 4, we have:*

$$|S^-_x| > (\Delta/\lambda^k) \cdot f(k) \text{ and } |S^+_x| < (\Delta/\lambda^k) \cdot f(k-1) \cdot \lambda/(\lambda - 1).$$

Proof. Throughout the proof, we let t^- and t^+ respectively denote the time-instant just before and just after the concerned iteration of the WHILE loop. Let $k \in [L]$ be the smallest index such that $D^*_k(x) > (\Delta/\lambda^k) \cdot f(k)$ at time t^-. For every index $j \in [0, L]$ and every vertex $v \in V$, let $N^*_j(v, t^-)$ and $N^*_j(v, t^+)$ respectively denote the set of vertices in $N^*_j(v)$ at time t^- and at time t^+.

Consider any vertex $u \in N^*_k(x, t^-)$. At time t^-, we had $\chi^*_k(u) = \chi^*_k(x)$. The concerned iteration of the WHILE loop changes the kth coordinate of the color of x, but the vertex u does not change its color during this iteration. Thus, we have $\chi^*_k(u) \neq \chi^*_k(x)$ at time t^+. We therefore infer that $x \in N^*_k(u, t^-)$ and $x \notin N^*_k(u, t^+)$. Hence, for every vertex $u \in N^*_k(x, t^-)$, the value of $D^*_k(u)$ drops by one due to the concerned iteration of the WHILE loop. It follows that $\{(k, u) | u \in N^*_k(x, t^-)\} \subseteq S^-_x$, and we get: $|S^-_x| \geq |N^*_k(x, t^-)| > (\Delta/\lambda^k) \cdot f(k)$. The last inequality follows from step 2 in Figure 4. This gives us the desired lower bound on the size of the set S^-_x. It now remains to upper bound the size of the set S^+_x.

Consider any ordered pair $(j, u) \in S^+_x$. Since the concerned iteration of the WHILE loop increases the value of $D^*_j(u)$ and does not change the color of any vertex other than x, we infer that:

$$x \notin N^*_j(u, t^-) \text{ and } x \in N^*_j(u, t^+). \tag{4.8}$$

The concerned iteration of the WHILE loop does not change the ith coordinate of the color of x for any $i < k$. Thus, the i-tuple $\chi^*_i(x)$ does not change from time t^- to time t^+. Furthermore, the vertex u does not change its color during the time-interval $[t^-, t^+]$. It follows that if $x \notin N^*_i(u, t^-)$ for some $i < k$, then we also have $x \notin N^*_i(u, t^+)$. From (4.8) we therefore get $j \in [k, L]$. Next, note that since $x \in N^*_j(u, t^+)$, we have $\chi^*_j(x) = \chi^*_j(u)$ at time t^+, and accordingly we also have $u \in N^*_j(x, t^+)$. To summarize, if an ordered pair (j, u) belongs to the set S^+_x, then we must have $j \in [k, L]$ and $u \in N^*_j(x, t^+)$. Thus, we get:

$$|S^+_x| \leq \sum_{j=k}^{L} |N^*_j(x, t^+)| \tag{4.9}$$

Note that step 4 in Figure 4 picks an $\alpha \in [\lambda]$ that minimizes $D^*_{j \to \alpha}(x)$. This gives us the following guarantee.

$$|N^*_k(x, t^+)| \leq |N^*_{k-1}(x, t^-)|/\lambda. \tag{4.10}$$

$$|N^*_j(x, t^+)| \leq |N^*_{j-1}(x, t^+)|/\lambda \text{ for all } j \in [k+1, L]. \tag{4.11}$$

Using (4.9), (4.10) and (4.11), we can upper bound $|S^+_x|$ by the sum of a geometric series, and get:

$$\begin{aligned}
|S^+_x| &\leq \sum_{j=k}^{L} |N^*_j(x, t^+)| \\
&\leq |N^*_{k-1}(x, t^-)| \cdot \left(\frac{1}{\lambda} + \cdots + \frac{1}{\lambda^{L-k+1}} \right) \\
&\leq |N^*_{k-1}(x, t^-)| \cdot \frac{1}{(\lambda - 1)} \\
&\leq \frac{\Delta}{\lambda^{k-1}} \cdot f(k-1) \cdot \frac{1}{(\lambda - 1)}
\end{aligned}$$

The last inequality holds due to step 2 in Figure 4. This gives us the desired upper bound on $|S^+_x|$. □

We now use a potential function based argument to prove Theorem 4.1, where the potential associated with the graph at any point in time is given by $\Phi = \sum_{v \in V, i \in [0,L]} D^*_i(v)$. Note that the potential Φ is always nonnegative. The bound on the amortized update now follows from the three observations stated below.

Observation 4.9. *Due to the insertion or deletion of an edge in G, the potential Φ changes by at most $O(L)$.*

Proof. Consider the insertion or deletion of an edge (u, v) in the input graph $G = (V, E)$. For all $x \in V \setminus \{u, v\}$ and $i \in [0, L]$, the value of $D_i^*(x)$ remains unchanged. Furthermore, for all $x \in \{u, v\}$ and $i \in [0, L]$, the value of $D_i^*(x)$ changes by at most one. Hence, the potential changes by at most $2(L+1)$. □

Observation 4.10. *Due to one iteration of the* WHILE *loop in Figure 4, the potential Φ decreases by at least $\Gamma = (\Delta/\lambda^k) \cdot f(k-1)/(\lambda-1)$, where k is defined as per Step 2 in Figure 4.*

Proof. Note that the concerned iteration of the WHILE loop does not change the color of any vertex $v \neq x$. Thus, for every vertex $v \in V$ and every index $i \in [0, L]$, the value of $D_i^*(v)$ changes by at most one. As in Lemma 4.8, let S_x^+ (resp. S_x^-) denote the set of all ordered pairs (i, v) such that the value of $D_i^*(v)$ increases (resp. decreases) due to this iteration. We therefore infer that the net decrease in Φ is equal to $|S_x^-| - |S_x^+|$, and we have:

$$\begin{aligned} |S_x^-| - |S_x^+| &\geq (\Delta/\lambda^k) \cdot f(k) - (\Delta/\lambda^k) \cdot f(k-1) \cdot \lambda/(\lambda-1) \\ &= (\Delta/\lambda^k) \cdot f(k-1) \cdot ((\lambda+1)/(\lambda-1) - \lambda/(\lambda-1)) \\ &= (\Delta/\lambda^k) \cdot f(k-1)/(\lambda-1). \end{aligned}$$

The first step in the above derivation follows from Lemma 4.8, and the second step follows from Invariant 4.4. □

Observation 4.11. *The time taken to implement one iteration of the* WHILE *loop in Figure 4 is $O((\lambda-1) \cdot \lambda^2 \cdot L \cdot \Gamma)$, where Γ is the net decrease in the potential due to the same iteration of the* WHILE *loop.*

Proof. This follows from Observation 4.10 and Lemma 4.7. □

Proof of Theorem 4.1. In the beginning when the edge set is empty we have $\Phi = 0$. After T edge insertions and deletions, the total time taken by the algorithm is $O(LT) + W$, where W is the total time taken by the WHILE loops. From Observation 4.11 we know that $W = O(\lambda^3 L \cdot \sum_t \Gamma_t)$, where Γ_t is the decrease in the potential due to the tth WHILE loop. On the other hand, from Observation 4.9 we get $\sum_t \Gamma_t = O(TL)$. Putting it all together, we get that the total time taken by the algorithm is $O(\lambda^3 L^2 T)$. This proves the theorem since $\lambda = O(\log^{1+o(1)} \Delta)$ and $L = O(\log \Delta / \log\log \Delta)$ as per (4.1) and part (1) of Lemma 4.2.

5 A Deterministic Dynamic Algorithm for $(2\Delta - 1)$ Edge Coloring

Let $G = (V, E)$ be the input graph that is changing dynamically, and let Δ be an upper bound on the maximum degree of any vertex in G. For now we assume that the value of Δ does not change with time. In Section 6, we explain how to relax this assumption. We present a simple, deterministic dynamic algorithm for maintaining a $2\Delta - 1$ edge coloring algorithm in G.

Data Structures. For every vertex $v \in V$, we maintain the following data structures.

1. An array C_v of length $2\Delta - 1$. Each entry in this array corresponds to a color. For each color c, the entry $C_v[c]$ is either null or points to the unique edge incident on v which is colored c.
2. A bit vector A_v of length $2\Delta - 1$, where $A_v[c] = 0$ iff $C_v[c]$ is null, and $A_v[c] = 1$ otherwise.
3. A balanced binary search tree T_v with $2\Delta - 1$ leaves. We refer to the vertices in the tree as *meta-nodes*, to distinguish them from the vertices in the input graph G. We maintain a counter $val(T_v, x)$ at every meta-node x in the tree T_v. The value of this counter at the cth leaf of T_v is given by $A_v[c]$. Furthermore, the value of this counter at any internal meta-node x of T_v is given by: $val(T_v, x) = \sum_{c: c \text{ is a leaf in the subtree rooted at } x} val(T_v, c)$.

We use the notation $A_v[a:b]$ to denote $\sum_{a \le c < b} A_v[c]$.

Initialization. Initially when the graph $G = (V, E)$ is empty, C_v is set to all nulls, A_v is set to all 0's, and the counters $val(T_v, x)$ are set to all 0's.

Coloring subroutine. When an edge $e = (u, v)$ is inserted, we need to assign it a color in $\{1, 2, \ldots, 2\Delta - 1\}$. We do so using the following binary-search like procedure described in Figure 5.

1. Set $\ell = 1$ and $r = 2\Delta$.
2. WHILE $\ell < r - 1$: // *Invariant:* $A_u[\ell : r] + A_v[\ell : r] < r - \ell$
3. $z := \lceil (\ell + r)/2 \rceil$
4. If $A_u[\ell : z] + A_v[\ell : z] < z - \ell$, THEN set $r = z$.
5. Else if $A_u[z : r] + A_v[z : r] < r - z$, THEN set $\ell = z$.
6. Assign color $\chi(e) = \ell$ to edge e.
7. Update the following data structures: $C_u[\ell], C_v[\ell], A_u[\ell], A_v[\ell], T_u, T_v$.

Figure 5: COLOR(e): Coloring an edge $e = (u, v)$ that has been inserted.

Claim 5.1. *The subroutine COLOR(e) returns a proper coloring of an edge e in $O(\log \Delta)$ time.*

Proof. At the end of the WHILE loop we have $\ell = r - 1$ since $z \le r - 1$. Assume that the invariant stated in the while loop holds at the end; in that case we have $A_u[\ell] + A_v[\ell] < 1$ implying both are 0. This, in turn, means that there is no edge incident on either u or v with the color ℓ. Thus, coloring (u, v) with ℓ is proper. We now show that the invariant always hold. It holds in the beginning since at that point both u and v have degree $\le \Delta - 1$ since (u, v) is being added. This implies the total sum of A_u and A_v are $\le 2\Delta - 2$, which implies the invariant. The while loop then makes sure that the invariant holds subsequently. For time analysis, note that there are $O(\log \Delta)$ iterations, and the values of $A_v[\ell : z]$'s are stored in the counters $val(T_v, x)$ at the internal meta-nodes x of the trees T_v, and these values can be obtained in $O(1)$ time. The data structures can also be maintained in $O(\log \Delta)$ time since in the tree T_v the values of the meta-nodes only in the path from ℓ to the root has to be increased by 1. □

The full algorithm is this: initially the graph is empty. When an edge $e = (u, v)$ is added, we run COLOR(e). When an edge $e = (u, v)$ is deleted and its color was c, we point $C_u[c]$ and $C_v[c]$ to null, set $A_u[c] = A_v[c] = 0$ and then update T_u and T_v in $O(\log \Delta)$ time. Thus we have the following theorem.

Theorem 5.1. *In a dynamic graph with maximum degree Δ, we can deterministically maintain a $(2\Delta - 1)$-edge coloring in $O(\log \Delta)$ worst case update time.*

6 Extensions to the Case where Δ Changes with Time

For ease of exposition, in the whole paper we have maintained that Δ is a parameter known to the algorithm up front with the promise that maximum degree of the graph remains at most Δ at all times. In fact all our algorithms, with some work, can work with the changing Δ as well. That is, if Δ_t is the maximum degree of the graph after t edge insertions and deletions, then in fact we have a randomized algorithm maintaining a $(\Delta_t + 1)$ vertex coloring, a deterministic algorithm maintaining a $(1 + o(1))\Delta_t$ vertex coloring, and a deterministic algorithm maintaining a $(2\Delta_t - 1)$ edge coloring. Our running times take a slight hit. For the first two algorithms the amortized running time is $O(\text{polylog}\, \Delta)$ where Δ is the maximum degree seen

so far (till time t); for the edge coloring the worst case update time becomes $O(\log \Delta_t)$. In the following three subsections we give a brief sketch of the differences in the algorithm and how the analysis needs to be modified. In all cases, we achieve this by making the requirement on the algorithm stronger.

6.1 Randomized $(\Delta_t + 1)$ vertex coloring.

Let $D_v = |\mathcal{N}_v(4, L)|$ be the degree of a vertex $v \in V$ in the current input graph. To extend our dynamic algorithm in Section 3 to the scenario where Δ changes with time, we simply ensure that the following property holds.

Property 6.1. *Every vertex v to have a color $\chi(v) \in \{1, \ldots, D_v + 1\}$.*

To see why it is easy to ensure Property 6.1, we only need to make the following two observations.

1. The algorithm that maintains the hierarchical partition in Section 3.2 is oblivious to the value of Δ.
2. For every vertex $v \in V$, change the definition of the subset of colors $\mathcal{C}_v \subseteq \mathcal{C}$ as follows. Now, the subset $\mathcal{C}_v \subseteq \mathcal{C}$ consists of all the colors in $\{1, \ldots, D_v + 1\}$ that are not assigned to any neighbor u of v with $\ell(u) \geq \ell(v)$. We can maintain these modified sets $\mathcal{C}_v$ by incurring only a $O(1)$ factor cost in the update time. Finally, note that even with this modified definition, Claim 3.1 continues to hold. Hence, the RECOLOR subroutine in Figure 2 in particular and our randomized dynamic coloring algorithm in general continue to remain valid.

6.2 Deterministic $(1 + o(1))\Delta_t$ vertex coloring.

Whenever $\deg_t(v)$ is less than a (very) large constant, whenever we need to change its color we do the greedy step taking $\deg_t(v) = O(1)$ time to find a free color. Henceforth assume $\deg_t(v) = \omega(1)$. Instead of having a fixed λ, L and η, for every vertex v we have separate parameters which depend on the degree of v at time t. Note these can be maintained at each insertion and deletion with $O(1)$ time per update.

$$\eta_t(v) := e^{\frac{16}{\lg \lg \deg_t(v)}}, \quad L_t(v) := \left\lfloor \frac{\lg(\eta_t(v) \cdot \deg_t(v))}{\lg \lg \deg_t(v)} \right\rfloor \text{ and } \lambda_t(v) := \left\lceil 2^{\frac{\lg(\eta_t(v) \cdot \deg_t(v))}{L_t(v)}} \right\rceil.$$

At time t, each vertex is assigned a color from $\{1, 2, \ldots, \lambda_t(v)^{L_t(v)}\}$. As before, this color $\chi(v)$ is assumed to be a $L_t(v)$-dimensional tuple where each entry takes positive integer values in $[\lambda_t(v)]$. Note that the dimension of tuple and the range of each dimension of the tuple can change with time and we need to be careful about that. The definitions of $N_i^*(v)$ and $D_i^*(v)$ remains the same, except that the range of i is now only till $L_t(v)$.

The invariant that we maintain for every vertex is similar to Invariant 4.4 changed appropriately *and* we add the condition that at time t each coordinate is in $[\lambda_t(v)]$.

Invariant 6.2. *For all t, for all $v \in V$, $i \in [0, L_t(v)]$, we have (1) $D_i^*(v) \leq (\deg_t(v)/\lambda_t(v)^i) \cdot f_{t,v}(i)$, where $f_{t,v}(i) = ((\lambda_t(v) + 1)/(\lambda_t(v) - 1))^i$, and (2) $\chi_i^{(t)}(v) \in \{1, 2, \ldots, \lambda_t(v)\}$.*

Let us take care of situations when part (2) of the above invariant is violated because the other part is similar to as done in Section 4. To do so, for a positive integer p, define d_p to be the largest value of $\deg_t(v)$ for which $\lambda_t(v)$ evaluates to p. By the definition of the parameters (since $\lambda_t(v) = \Theta(\lg \deg_t(v))$) we get $d_{p+1} - d_p = \Theta(d_p)$. To take care of part (2) of Invariant 6.2, the algorithm given in Figure 4 needs to be changed in Step 4 as follows: if $\deg_t(v) \in [d_p, d_{p+1}]$ (and therefore $\lambda_t(v) = p + 1$), we search for α in $[\lambda_t(v)]$ if $\deg_t(v) > d_p + (d_{p+1} - d_p)/2$, otherwise we search for α in $[\lambda_t(v) - 1]$.

Whenever part (2) of Invariant 6.2 is violated by vertex v at time t, we perform the following changes. We take $\Theta(\deg_t(v))$ time to find a color for v such that $\chi_i(v) \in [\lambda_t(v)]$ for all $i \in [L_t(v)]$ satisfying Invariant 6.2. We now show that this time can be charged to edge deletions *incident on* v that has happened in the past, and furthermore these edge deletions will not be charged to again.

Firstly note that v violated part (2) on Invariant 6.2 only because we delete an edge (u, v) and $\deg_t(v)$ and therefore $\lambda_t(v)$ has gone down. Note that it must be the case $\chi_i^{(t)}(v) = \lambda_t(v) + 1 = \lambda_{t^-}(v)$, where the third term is the value of $\lambda_t(v)$ just before the edge deletion. Suppose $\lambda_t(v) = p$. Note that $\deg_t(v)$ must be d_p since at time $\deg_{t^-}(v) = \deg_t(v) + 1$ and $\lambda_{t^-}(v) = p + 1$. Look at the last time t' before t at which $\chi_i^{(t')}(v)$ was set to $p + 1$. At that time, because of the modification made above to the algorithm, we must have had $\deg_{t'}(v) > d_p + (d_{p+1} - d_p)/2$. Therefore between t' and t we must have had at least $(d_{p+1} - d_p)/2 = \Theta(d_p) = \Theta(\deg_t(v))$ edge deletions incident on v. We charge the $\Theta(\deg_t(v))$ time taken to recolor v due to violation of part (2) of invariant to these deletions. Note that since we choose t' to be the last time before t, we won't charge to these edge deletions again.

To maintain part (1) of the invariant, the algorithm is similar as in the previous section with two changes: (1) firstly, the WHILE loop checks the new invariant, and (2) is the technical change described above. Observe that even when we delete an edge we run the risk of the invariant getting violated since the RHS of the invariant also goes down. For the analysis, the one line argument why everything generalizes is that our analysis is in fact vertex -by-vertex. More precisely, we have a version of Lemma 4.7 where the Δ, L, λ, f are replaced by $\deg_t(v), L_t(v), \lambda_t(v), f_{t,v}$. Similar changes are in all the other claims and lemmas and for brevity we don't mention the subscripts below. For instance, time analysis of Lemma 4.7 for the WHILE loop taking care of vertex x generalizes with $L_t(x), \lambda_t(x), f_{t,x}$ and $\deg_t(x)$ replacing L, λ, f and Δ. Similarly, in Lemma 4.8 we have exactly the same changes which reflects in Observation 4.10. That is, the decrease in potential in a single while loop is charged to the running time of that while loop. The rest of the analysis as in Section 4. There is an extra change in Lemma 4.8 where because of the technical change we made, we only get $|S_x^+| < (\Delta/\lambda^k) \cdot f(k-1) \cdot \lambda/(\lambda - 2)$ since we could be searching over a range of $[\lambda - 1]$. This only increases the update time by an extra factor which is $O(1)$ if $\lambda \geq 2$.

6.3 Deterministic $(2\Delta_t - 1)$ edge coloring.

We assert the invariant that every edge (u, v) gets a color from the palette $\{1, \ldots, 2\max(\deg_t(u), \deg_t(v)) - 1\}$. In the subroutine COLOR(e), the upper bound u is then set to $u = \deg_t(u) + \deg_t(v)$, and the rest of the algorithm remains the same. The trees T_u and T_v now need to be dynamically balanced; but this can be done in $O(\log \Delta_t)$ time using say red-black trees. The other place where the algorithm needs to change is that when an edge $e = (u, v)$ is deleted, the degree of both u and v go down. That may lead to at most four edges (two incident on u and two incident on v) violating the invariant. They need to be re-colored using COLOR() procedure again. But this takes $O(\log \Delta_t)$ time in all.

7 Open Problems

One obvious open question left from this work is whether we can maintain a $(\Delta + 1)$-vertex coloring in polylogarithmic time using a *deterministic* algorithm. We believe that this is an important question, since it may help in understanding how to develop deterministic dynamic algorithms in general. It is very challenging and interesting to design deterministic dynamic algorithms with performances similar to the randomized ones for many dynamic graph problems such as maximal matching [BGS11; BHI15b; BHI15a; BHN16; BCH17; BHN17], connectivity [KKM13; NSW17; Wul17; NS17], and shortest paths [Ber17; BC17; BC16; HKN14; HKN16].

Another obvious question is whether our deterministic update time for $(1+o(\Delta))\Delta$-vertex coloring can be improved. We did not try to optimize the polylog factors hidden inside Theorem 4.1 and believe that it can be improved; however, getting an $O(\log \Delta)$ deterministic update time seems challenging. It will also be interesting to get $O(\text{poly} \log \Delta)$ worst-case update time for dynamic vertex coloring.

Finally, one other direction is to study the classes of locally-fixable problems and SLOCAL [GKM17]. Does every locally-fixable problem admit polylogarithmic update time? How about problems in SLOCAL such as maximal independent set and minimal dominating set?

8 Acknowledgements

We thank an anonymous reviewer for the proofs of Lemma 4.2 and Claim 4.1.

This project has received funding from the European Research Council (ERC) under the European Unions Horizon 2020 research and innovation programme under grant agreement No 715672. Danupon Nanongkai was also partially supported by the Swedish Research Council (Reg. No. 2015-04659). 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.

References

[AW14] Amir Abboud and Virginia Vassilevska Williams. "Popular Conjectures Imply Strong Lower Bounds for Dynamic Problems". In: *FOCS*. IEEE Computer Society, 2014, pp. 434–443 (cit. on p. 2).

[BC16] Aaron Bernstein and Shiri Chechik. "Deterministic decremental single source shortest paths: beyond the o(mn) bound". In: *STOC*. ACM, 2016, pp. 389–397 (cit. on p. 23).

[BC17] Aaron Bernstein and Shiri Chechik. "Deterministic Partially Dynamic Single Source Shortest Paths for Sparse Graphs". In: *SODA*. SIAM, 2017, pp. 453–469 (cit. on p. 23).

[BCH17] Sayan Bhattacharya, Deeparnab Chakrabarty, and Monika Henzinger. "Deterministic Fully Dynamic Approximate Vertex Cover and Fractional Matching in O(1) Amortized Update Time". In: *IPCO*. Vol. 10328. Lecture Notes in Computer Science. Springer, 2017, pp. 86–98 (cit. on p. 23).

[BCKLRRV17] Luis Barba, Jean Cardinal, Matias Korman, Stefan Langerman, André van Renssen, Marcel Roeloffzen, and Sander Verdonschot. "Dynamic Graph Coloring". In: *WADS*. 2017 (cit. on p. 2).

[BE13] Leonid Barenboim and Michael Elkin. *Distributed Graph Coloring: Fundamentals and Recent Developments*. Synthesis Lectures on Distributed Computing Theory. Morgan & Claypool Publishers, 2013. ISBN: 9781627050180 (cit. on p. 2).

[BGS11] Surender Baswana, Manoj Gupta, and Sandeep Sen. "Fully Dynamic Maximal Matching in O (log n) Update Time". In: *IEEE 52nd Annual Symposium on Foundations of Computer Science, FOCS 2011, Palm Springs, CA, USA, October 22-25, 2011*. Ed. by Rafail Ostrovsky. IEEE Computer Society, 2011, pp. 383–392 (cit. on p. 23).

[BHI15a] S. Bhattacharya, M. Henzinger, and G. F. Italiano. "Design of Dynamic Algorithms via Primal-Dual Method". In: *ICALP*. 2015 (cit. on p. 23).

[BHI15b] S. Bhattacharya, M. Henzinger, and G. F. Italiano. "Deterministic Fully Dynamic Data Structures for Vertex Cover and Matching". In: *SODA*. 2015 (cit. on p. 23).

[BHN16] S. Bhattacharya, M. Henzinger, and D. Nanongkai. “New deterministic approximation algorithms for fully dynamic matching”. In: *STOC*. 2016 (cit. on p. 23).

[BHN17] Sayan Bhattacharya, Monika Henzinger, and Danupon Nanongkai. “Fully Dynamic Approximate Maximum Matching and Minimum Vertex Cover in $O(\log^3 n)$ Worst Case Update Time”. In: *Proceedings of the Twenty-Eighth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2017, Barcelona, Spain, Hotel Porta Fira, January 16-19*. Ed. by Philip N. Klein. SIAM, 2017, pp. 470–489 (cit. on p. 23).

[BM17] Leonid Barenboim and Tzalik Maimon. “Fully-Dynamic Graph Algorithms with Sublinear Time Inspired by Distributed Computing”. In: *International Conference on Computational Science, ICCS 2017, 12-14 June 2017, Zurich, Switzerland*. Ed. by Petros Koumoutsakos, Michael Lees, Valeria V. Krzhizhanovskaya, Jack J. Dongarra, and Peter M. A. Sloot. Vol. 108. Procedia Computer Science. Elsevier, 2017, pp. 89–98 (cit. on pp. i, 1, 2).

[Ber17] Aaron Bernstein. “Deterministic Partially Dynamic Single Source Shortest Paths in Weighted Graphs”. In: *ICALP*. Vol. 80. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2017, 44:1–44:14 (cit. on p. 23).

[DGOP07] Antoine Dutot, Frederic Guinand, Damien Olivier, and Yoann Pign. “On the Decentralized Dynamic Graph-Coloring Problem.” In: *COSSOM: Complex Systems and Self-Organization Modelling. Satellite workshop within European Simulation and Modelling Conference (ESM'2007). Pages 259-261*. 2007 (cit. on p. 2).

[Dah16] Søren Dahlgaard. “On the Hardness of Partially Dynamic Graph Problems and Connections to Diameter”. In: *ICALP*. Vol. 55. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2016, 48:1–48:14 (cit. on p. 2).

[FGK17] Manuela Fischer, Mohsen Ghaffari, and Fabian Kuhn. “Deterministic Distributed Edge-Coloring via Hypergraph Maximal Matchings”. In: *FOCS*. IEEE, 2017 (cit. on p. 2).

[FK98] Uriel Feige and Joe Kilian. “Zero Knowledge and the Chromatic Number”. In: *J. Comput. Syst. Sci.* 57.2 (1998). announced at CCC'96, pp. 187–199 (cit. on p. 1).

[GKM17] Mohsen Ghaffari, Fabian Kuhn, and Yannic Maus. “On the complexity of local distributed graph problems”. In: *STOC*. ACM, 2017, pp. 784–797 (cit. on pp. 2, 24, 26, 27).

[HKN14] Monika Henzinger, Sebastian Krinninger, and Danupon Nanongkai. “Decremental Single-Source Shortest Paths on Undirected Graphs in Near-Linear Total Update Time”. In: *FOCS*. IEEE Computer Society, 2014, pp. 146–155 (cit. on p. 23).

[HKN16] Monika Henzinger, Sebastian Krinninger, and Danupon Nanongkai. “Dynamic Approximate All-Pairs Shortest Paths: Breaking the O(mn) Barrier and Derandomization”. In: *SIAM J. Comput.* 45.3 (2016), pp. 947–1006 (cit. on p. 23).

[HKNS15] Monika Henzinger, Sebastian Krinninger, Danupon Nanongkai, and Thatchaphol Saranurak. “Unifying and Strengthening Hardness for Dynamic Problems via the Online Matrix-Vector Multiplication Conjecture”. In: *STOC*. ACM, 2015, pp. 21–30 (cit. on p. 2).

[HLT17] Bradley Hardy, Rhyd Lewis, and Jonathan Thompson. “Tackling the edge dynamic graph colouring problem with and without future adjacency information”. In: *Journal of Heuristics* (2017), pp. 1–23 (cit. on p. 2).

[Hal93] Magnús M. Halldórsson. “A Still Better Performance Guarantee for Approximate Graph Coloring”. In: *Inf. Process. Lett.* 45.1 (1993), pp. 19–23 (cit. on p. 2).

[KKM13] Bruce M. Kapron, Valerie King, and Ben Mountjoy. “Dynamic graph connectivity in polylogarithmic worst case time”. In: *SODA*. SIAM, 2013, pp. 1131–1142 (cit. on p. 23).

[KP06] Subhash Khot and Ashok Kumar Ponnuswami. "Better Inapproximability Results for MaxClique, Chromatic Number and Min-3Lin-Deletion". In: *Automata, Languages and Programming, 33rd International Colloquium, ICALP 2006, Venice, Italy, July 10-14, 2006, Proceedings, Part I*. Ed. by Michele Bugliesi, Bart Preneel, Vladimiro Sassone, and Ingo Wegener. Vol. 4051. Lecture Notes in Computer Science. Springer, 2006, pp. 226–237 (cit. on p. 1).

[KPP16] Tsvi Kopelowitz, Seth Pettie, and Ely Porat. "Higher Lower Bounds from the 3SUM Conjecture". In: *SODA*. SIAM, 2016, pp. 1272–1287 (cit. on p. 2).

[NS17] Danupon Nanongkai and Thatchaphol Saranurak. "Dynamic spanning forest with worst-case update time: adaptive, Las Vegas, and O($n^{1/2 - \epsilon}$)-time". In: *STOC*. ACM, 2017, pp. 1122–1129 (cit. on p. 23).

[NSW17] Danupon Nanongkai, Thatchaphol Saranurak, and Christian Wulff-Nilsen. "Dynamic Minimum Spanning Forest with Subpolynomial Worst-case Update Time". In: *FOCS*. IEEE Computer Society, 2017 (cit. on p. 23).

[OB11] Linda Ouerfelli and Hend Bouziri. "Greedy algorithms for dynamic graph coloring". In: *Communications, Computing and Control Applications (CCCA), 2011 International Conference on*. IEEE. 2011, pp. 1–5 (cit. on p. 2).

[Pat10] Mihai Patrascu. "Towards polynomial lower bounds for dynamic problems". In: *STOC*. ACM, 2010, pp. 603–610 (cit. on p. 2).

[SIPGRP16] Scott Sallinen, Keita Iwabuchi, Suraj Poudel, Maya Gokhale, Matei Ripeanu, and Roger A. Pearce. "Graph colouring as a challenge problem for dynamic graph processing on distributed systems". In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2016, Salt Lake City, UT, USA, November 13-18, 2016*. Ed. by John West and Cherri M. Pancake. IEEE Computer Society, 2016, pp. 347–358 (cit. on p. 2).

[Wul17] Christian Wulff-Nilsen. "Fully-dynamic minimum spanning forest with improved worst-case update time". In: *STOC*. ACM, 2017, pp. 1130–1143 (cit. on p. 23).

[Zuc07] David Zuckerman. "Linear Degree Extractors and the Inapproximability of Max Clique and Chromatic Number". In: *Theory of Computing* 3.1 (2007). Announced at STOC'06, pp. 103–128 (cit. on p. 1).

A Locally-Fixable Problems

We consider graph problems in a way similar to [GKM17], where there is a set of *states* S_v associated with each node v, and each node v has to pick a state $s(v) \in S_v$. For a *locally-fixable* problem, once every node picks its own state, there is a function f_v that determines whether any given node $v \in V$ is *valid* or *invalid*. Crucially, the function f_v satisfies the two properties stated below. The goal in a locally-fixable problem is to assign a state to each node in such a way that all the nodes become valid.

Property A.1. *The output of the function f_v depends* only on *the states of v and its neighbors. (In other words, f_v is a constraint that is* local to a node *v in that it is defined on the states of v and its neighbors.)*

Property A.2. *Consider any assignment of states to v and its neighbors where v is invalid. Then, without modifying the states of v's neighbors, there is a way to change the state of v that (a) makes v valid, and (b) does not make any erstwhile valid neighbor of v invalid.*

We present two examples of locally-fixable problems. Note again that all graph problems below are viewed as assigning states to nodes, and their definitions below are standard. Our main task is to define a function f_v for each node v such that Properties A.1 and A.2 are satisfied.

$(\Delta+1)$-vertex coloring. In this problem, the set of states S_v associated with a node v is the set of $(\Delta+1)$ colors, and a feasible coloring is when every node has different color from its neighbors. To show that this problem is locally-fixable, consider the function f_v which determines that v is valid iff it none of its neighbors have the same state as v. This satisfies Properties A.1 and A.2, since a node v has at most Δ neighbors and there are $(\Delta+1)$ possible states.

$(2\Delta-1)$-edge coloring. The problem is as follows. Let $\mathcal{C}$ denote the palette of $2\Delta-1$ colors. Let $n=|V|$ denote the number of nodes. We identify these nodes as $V=\{1,\ldots,n\}$. The state $s(v)$ of a node v is an n-tuple $s(v)=(s_1(v),\ldots,s_n(v))$, where $s_i(v)\in\mathcal{C}\cup\perp$ for each $i\in[n]$. The set S_v consists of all such possible n-tuples. The element $s_u(v)$ is supposed to be the color of edge (u,v), which should be $\perp$ if edge (u,v) does not exist. Thus, the feasible solution is the one where

(1) for every nodes $u\neq v$, $s_u(v)\neq\perp$ iff there is an edge (u,v) (i.e. each node only assign colors to its incident edges), (2) for every edge (u,v), $s_u(v)=s_v(u)$ (i.e. u and v agree on the color of (u,v)), and (3) for every edges $(u,v)\neq(u',v)$, $s_u(v)\neq s_{u'}(v)$ (i.e. adjacent edges should have different colors).

To show that this problem is locally-fixable, consider the function f_v which determines that v is valid iff the following three conditions hold.

(1) $s_i(v)=\perp$ for every $i\in[n]$ that is not a neighbor of v. (2) $s_i(v)=s_v(i)\neq\perp$ for every neighbor $i\in[n]$ of v. (3) $s_i(v)\neq s_j(v)$ for every two neighbors $i,j\in[n]$ of v with $i\neq j$.

This satisfies Properties A.1 and A.2.

Lemma A.3. *In the dynamic setting, we can update a valid solution to a locally-fixable problem by making at most two changes after an insertion or deletion of an edge. Furthermore, the changes occur only at the endpoints of the edge being inserted or deleted.*

Proof. Consider a locally fixable problem on a dynamic graph $G=(V,E)$. Suppose that every node is valid just before the insertion or deletion of an edge (u,v). By Property A.1, due to the insertion/deletion of the edge, only the endpoints u and v can potentially become invalid. By Property A.2, we can consider u and v one at a time in any order and make them valid again without making any new node invalid. □

Locally-fixable problems constitute a subclass of SLOCAL problems. Roughly speaking, the complexity class SLOCAL(1) [GKM17] is as follows (the description closely follows [GKM17]). In the SLOCAL(1) model, nodes are processed in an arbitrary order. When a node v is processed, it can see the current state of its neighbors and compute its output as an arbitrary function of this. In addition, v can locally store an arbitrary amount of information, which can be read by later nodes as part of v's state.

Lemma A.4. *Any locally-fixable problem is in SLOCAL(1).*

Proof. We initially assign an arbitrary state $s(v)\in S_v$ to every node $v\in V$. Then we scan the nodes in V in any arbitrary order. While considering a node v during the scan, using Property A.2 we change the state of v in such a way which ensures that (1) the node v becomes valid and (2) no previously scanned neighbor of v becomes invalid. Thus, at the end of the scan, we are guaranteed that every node is valid. □

The maximal independent set (MIS) problem is not locally fixable. Recall the standard definition of MIS where the state $s(v)$ of a node v is either 0 or 1, and a feasible solution is such that (i) no two adjacent nodes are both 1, and (ii) no node whose neighbors are all 0 is in state 0. We claim that MIS under this standard

definition is not locally-fixable; i.e. we cannot define function f_v for every node v such that Properties A.1 and A.2 are satisfied.

To see this, consider the following graph instance $G = (V, E)$, where the node-set is defined as $V = \{u, v, u_1, \ldots, u_{\Delta-1}, v_1, \ldots, v_{\Delta-1}\}$. The edge-set E is defined as follows. For every $i \in [\Delta - 1]$, there are two edges (u, u_i) and (v, v_i). Now consider the following MIS solution for the graph G. We have $s(u) = s(v) = 1$ and $s(x) = 0$ for all $x \in V \setminus \{u, v\}$. Now, if the edge (u, v) gets inserted, we have to change either $s(u)$ or $s(v)$ to 0. Without any loss of generality, suppose that we set $s(u) \leftarrow 0$. Then we also need to set $s(u_i) \leftarrow 1$ for every $i \in [\Delta - 1]$. In other words, insertion or deletion of an edge can lead to $\Omega(\Delta)$ many changes in a valid solution. By a contrapositive of Lemma A.3, MIS is not locally-fixable.