Dynamic $(\Delta + 1)$ Vertex Coloring

Noam Benson-Tilsen

12 January 2026

Abstract

Several recent results from dynamic and sublinear graph coloring are surveyed. This problem is widely studied and has motivating applications like network topology control, constraint satisfaction, and real-time resource scheduling. Graph coloring algorithms are called *colorers*. In §1 are defined graph coloring, the dynamic model, and the notion of performance of graph algorithms in the dynamic model. In particular $(\Delta + 1)$-coloring, sublinear performance, and oblivious and adaptive adversaries are noted and motivated. In §2 the pair of approximately optimal dynamic vertex colorers given in [BCKLVRV17] are summarized as a warmup for the $(\Delta+1)$-colorers. In §3 the state of the art in dynamic $(\Delta+1)$-coloring is presented. This section comprises a pair of papers ([BCHN18] and [BGKLS22]) that improve dynamic $(\Delta+1)$-coloring from the naive algorithm with $O(\Delta)$ expected amortized update time to $O(\log \Delta)$, then to $O(1)$ with high probability. In §4 the results in [BRW25], which gives a sublinear algorithm for $(\Delta+1)$-coloring that generalizes oblivious adversaries to adaptive adversaries, are presented.

Contents

1 Preliminaries

1.1 Graph coloring

A vertex coloring $\chi : V \longrightarrow C$ of a graph $G \coloneqq (V, E)$ is an assignment of the vertices V to colors $\mathcal{C} \coloneqq \{1, \ldots, k\} \subseteq \mathbb{N}_+$. χ is proper if $\chi(u) \neq \chi(v)$ for all edges $\{u, v\} \in E$. G is k-colorable if χ is a proper coloring of G with largest color $\max_{c \in C} c = k$. The smallest k such that there exists a proper coloring of G is the chromatic number $\chi(G)$.

In general, however, determining $\chi(G)$ is NP-hard, even for $\chi(G) = 3$. This motivates studying approximation algorithms (§2), as well as algorithms for finding $(\Delta + 1)$-colorings (§3 and on), which always exist. See §1.3 for more basic facts about $(\Delta + 1)$-coloring performance.

1.2 Dynamic model

In the dynamic model, G undergoes a sequence of updates $\mathcal{U}$ consisting of insertions and deletions of edges and vertices. Note that neither vertex deletions nor edge deletions corrupt a proper coloring. In the context of vertex coloring, one can choose between accounting solely for (1) vertex (plus incident edge) insertions or (2) edge insertions.

1. Edge updates can be factored out by
   (a) simulating any edge insertion by removing and adding back one of its endpoints, and
   (b) delaying accounting for any edge deletion until one of its endpoints is deleted; or
2. vertex updates can be factored out by assuming (WLOG) that G begins with the maximal order n it would have at any point during $\mathcal{U}$, in which case vertex updates are the same as edge updates with respect to maintaining a proper vertex coloring.

In §2 the first approach is taken; in §3, the second.

Maintaining a proper coloring across $\mathcal{U}$ requires recoloring vertices; i.e., upon an edge insertion a dynamic graph colorer must update the current coloring χ by modifying the colors $\chi(V')$ of some vertices $V' \subseteq V$. The maximum degree $\max^{\mathcal{U}}_{v \in V(G^{\mathcal{U}})} d(V)$ across $\mathcal{U}$ of any vertex is denoted by Δ.

1.3 Performance

The performance of a dynamic graph colorer is defined by its (expected amortized) number of recolorings per update, also known as update time, as follows. For some underlying set $\mathscr{U}$ of inputs, let update time $T_{\mathcal{A}}(\mathcal{U})$ be the performance of dynamic colorer $\mathcal{A}$ against an input sequence $\mathcal{U} \sim \mathfrak{S}_{\mathscr{U}}$ drawn from the symmetric group $\mathfrak{S}_{\mathscr{U}}$ of $\mathscr{U}$.[1] Then the task is to find

$$\arg\min_{\mathcal{A}} \mathop{\mathbb{E}}_{\mathbf{U} \sim \mathfrak{S}_{\mathscr{U}}} [T_{\mathcal{A}}(\mathcal{U})].$$

Linear-time dynamic $(\Delta+1)$-colorers are trivial, but sublinear dynamic $(\Delta+1)$-colorers are nontrivial. This is the case even when inputs are oblivious — that is, when input sequences are generated independently of the colorer, such as being drawn uniformly at random from $\mathfrak{S}_{\mathscr{U}}$. The end of this report will discuss a paper that gives sublinear performance against adaptively adversarial

[1]When considering only edge insertions, as in most of the report, T will specialize to $T(\mathcal{U}, n)$, where n is the order of the initially empty graph.

input sequences, which can force pathological, worst-case behavior based on the colorer. In [BCK-LVRV17], presented in §2, the colorers use $k(N) \cdot \chi(G)$ colors for a parameter $k > 0$ depending on the maximum number N of vertices of graph G across all updates. The expected amortized number of recolorings per update is traded against approximation tightness in the two colorers. In particular, for a k-colorable graph with parameter $d > 0$, two algorithms are given in which a factor in $O(N^{1/d})$ is traded between performance and approximation tightness.

2 Warmup: Approximately optimal dynamic coloring

[BCKLVRV17] provides two variations of the same algorithm. The first algorithm, $\mathcal{A}_1$, uses a one-level vertex partition; $\mathcal{A}_2$ uses a two-level partition.[2] In both, vertices within a first-level partition are colored properly using $\chi(G)$ colors. Also in both, V is split into buckets $\bigoplus_{i=1}^{d} V_i$ such that V_i has capacity $N_R^{(i-1)/d}$, where N_R is the value of n at the last update. Note that

$$\sum_{i=1}^{d} N_R^{(i-1)/d} = \frac{N_R - 1}{N_R^{1/d} - 1} \in O(N_R^{1-1/d}) \subseteq O(n).$$

Buckets, however, do not reach their capacity. Instead, V_i has an invariant high point of at most $h_i = N_R^{i/d} - N_R^{(i-1)/d}$ vertices. In $\mathcal{A}_2$, each first-level bucket V_i has its up to h_i vertices partitioned into $N_R^{1/d} - 1$ buckets $V_{i,1}, \ldots, V_{i,\lceil N_R^{1/d} - 1 \rceil}$ each of capacity $N_R^{(i-1)/d}$, with an extra empty *reset bucket*. That is, in $\mathcal{A}_2$, sub-buckets are left-packed to leave room for the reset bucket. The reset bucket is used during insertions.

2.1 Vertex insertion

When vertex v is inserted, it is placed in V_1. In $\mathcal{A}_2$, v goes into the first empty sub-bucket $V_{1,j}$. If V_1 has fewer than h_i vertices, v is assigned one of the leftover colors local to V_1. In $\mathcal{A}_2$, this color is local to $V_{1,j}$. But if inserting v to V_1 violates the high-point invariant, all vertices in V_1 are moved to V_2 (or in $\mathcal{A}_2$, moved to the empty sub-bucket, guaranteed to exist by the invariant). The vertices that were transferred are colored again, if possible; if they violate the invariant in V_2 as well, then V_2 is transferred up to V_3. This can propagate up to the last partition, V_d. In this case, the entire graph is reset and recolored.

2.2 Performance

In the worst case, $\mathcal{A}_1$ requires shifting and recoloring $O(N^{1/d})$ vertices per bucket across all d buckets. Each recoloring is trivial, given that there are more than enough colors to assign each vertex a distinct color. In total gives recolorings in $O(dN^{1/d})$. Similarly, in $\mathcal{A}_2$, vertices are moved and trivially recolored at most once per first-level partition, giving total recolorings in $O(d)$.

For $\mathcal{A}_1$, each bucket uses $O(\chi(G))$ colors, and there are $O(d)$ buckets, giving an $O(d)$-approximate coloring. For $\mathcal{A}_2$, each bucket within a partition uses $O(\chi(G))$ colors, and with $O(N^{1/d})$ buckets in each of the d partitions, this gives an $O(dN^{1/d})$-approximate coloring.

[2]The paper presents $\mathcal{A}_1$ and $\mathcal{A}_2$ in the other order, which is less clear. I also combine their presentation because they have one difference.

Table 1: Performance and tightness

	Recolorings	**Approximation tightness**
$\mathcal{A}_1$	$O(dN^{1/d})$	$O(d)$
$\mathcal{A}_2$	$O(d)$	$O(dN^{1/d})$

3 Dynamic $(\Delta + 1)$-coloring with oblivious adversaries

In this section, I'll give an overview of the techniques used in [BCHN18] and [BGKLS22]. For the remainder all updates will be edge insertions, and $k \coloneqq \Delta + 1$, so that $\mathcal{C} = \{1, \ldots, \Delta + 1\}$. χ^* will denote the coloring during $\mathcal{U}$; $\chi^* = \chi$ exactly when $\mathcal{U}$ finishes, and between atomic steps of $\mathcal{A}$ χ^* is not guaranteed to be proper.

3.1 Logarithmic-time dynamic $(\Delta + 1)$-coloring

There is a trivial algorithm with $O(\Delta)$ update time. When edge uv is inserted, if $\chi^*(u) = \chi^*(v)$, choose one x of u and v and assign x a color not used by any of its neighbors. Such a color is a blank color of x, and the set of such colors is $\mathcal{B}_{N(x)}$. Note that for any $x \in V$, $|N(x)| \leq \Delta < \Delta + 1 = |C|$, so by the pigeonhole principle a blank color always exists.

3.1.1 Level data structure

To improve update times to $O(\log \Delta)$, [BCHN18] presents $\mathcal{A}_{\log}$, a dynamic colorer with logarithmic update time that uses a level data structure $\ell : V \longrightarrow \{4, \ldots, L\}$, where $L \coloneqq \log_\beta \Delta$ for sufficiently large constant $\beta > 0 \in \mathbb{N}$. In other words, $\ell(v)$ is the level of v, where there are $O(\log \Delta)$ possible levels. The motive for ℓ is to expose a heuristic for which colors to use during recoloring that minimizes the expected length of cascading recoloring chains. For each vertex v, let the down-neighbors $\mathcal{L}(v)$ of v be the neighbors of v with levels lower than $\ell(v)$, the up-neighbors $\mathcal{H}(v)$ those with level at least $\ell(v)$, and same-level neighbors $\mathcal{S}(v)$ those with level equal to $\ell(v)$. See Figure 1. $\mathcal{A}_{\log}$ maintains two invariants:

Invariant 3.1. *The number $\phi \coloneqq |\mathcal{L}(v)|$ of down-neighbors is at least $\beta^{\ell(v)-5} \in \Omega(\Delta)$ for each $v \in V$.*

Invariant 3.2. *The number $|\mathcal{L}(v) \cup \mathcal{S}(v)|$ of down- and same-level neighbors is at most $\beta^{\ell(v)} \in O(\Delta)$ for each $v \in V$.*

Let $\mathcal{C}_{\mathcal{H}(v)}$ be the colors used by an up-neighbor of v, $\mathcal{U}_{\mathcal{L}(v)}$ those used by exactly one down-neighbor of v, and $\mathcal{M}_{\mathcal{L}(v)}$ those used by at least two down-neighbors of v. Let $\mathcal{D}_{N(v)} \coloneqq \mathcal{C} \setminus \mathcal{C}_{\mathcal{H}(v)} \setminus \mathcal{M}_{\mathcal{L}(v)}$ be the colors used by no up-neighbor of v and at most one down-neighbor of v. Let τ_v be the last timestamp at which v was recolored. See Table 2 for these definitions. Sets are denoted by calligraphic characters. Sets of vertices are named by functions $V \longrightarrow \mathbb{P}(V)$ and sets of colors have names subscripted by the set of vertices over which they are defined.

$\mathcal{D}_{N(v)}$ is the most important object. In the worst-case edge insertion, where there might be a recursive cascade of recolorings, $\mathcal{D}_{N(v)}$ will help limit the recursion fan-out to one by exposing

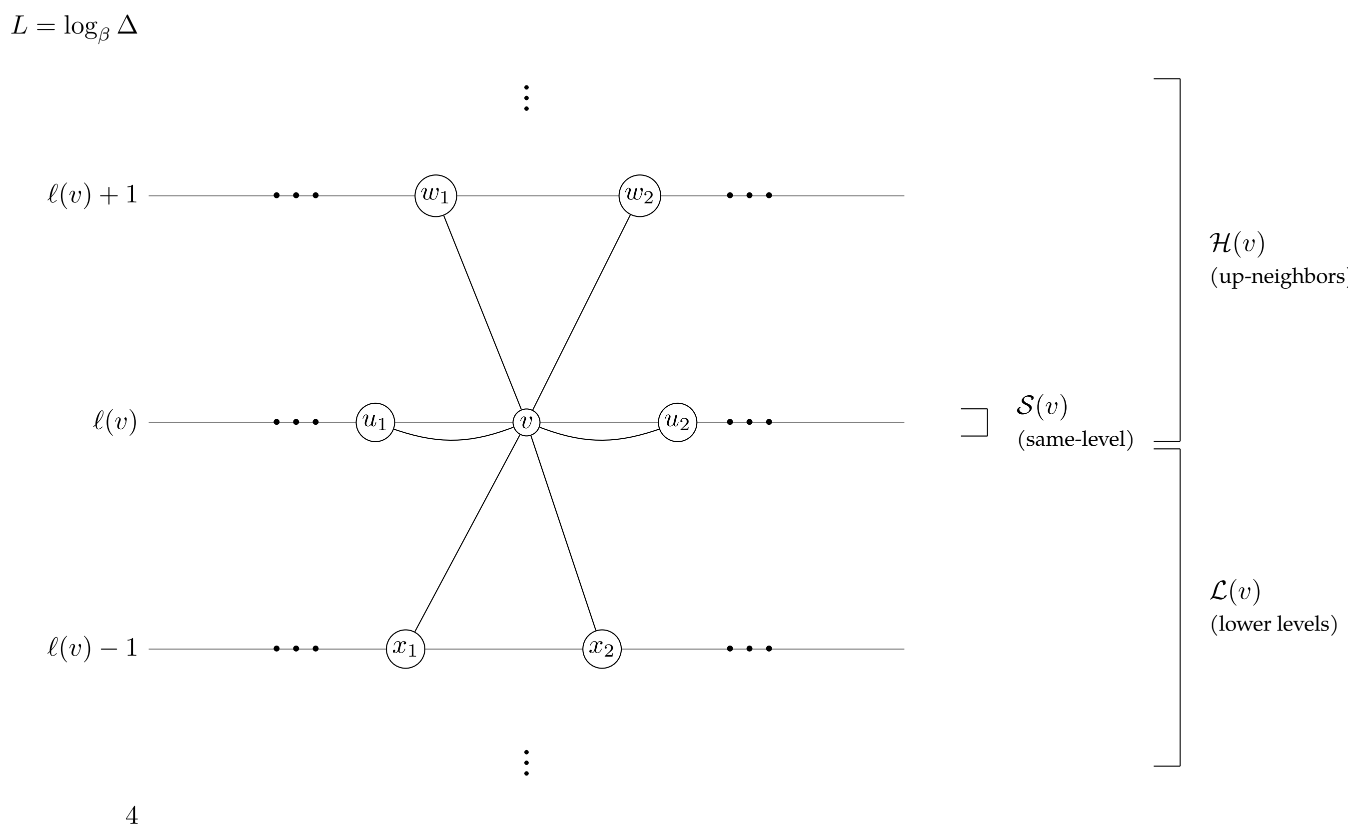

Figure 1: $\ell(v)$ for each $v \in V$, with up-neighbors $\mathcal{H}(v)$, same-level neighbors $\mathcal{S}(v) \subseteq \mathcal{H}(v)$, and down-neighbors $\mathcal{L}(v)$.

colors used by few of the recolored vertex's down-neighbors. By Lemma 3.3, $\mathcal{D}_{N(v)}$ is large enough that sampling uniformly at random from it is unlikely to conflict with the vertex being recolored.

Lemma 3.3. *If Invariant 3.1 and Invariant 3.2 hold,* $\left|\mathcal{D}_{N(v)}\right| \in \Omega(\Delta)$ *for each* $v \in V$.

Proof. Let $x := \left|\{v \mid \chi^*(v) \in \mathcal{U}_{\mathcal{L}(v)}\}\right|$ be the number of down-neighbors with a color used exactly once in $N(v)$. Note that $\left|\mathcal{M}_{\mathcal{L}(v)}\right| \leq \frac{\phi - x}{2}$: Each of the ϕ down-neighbors of v, except for the x down-neighbors with a color used exactly once, has at least one partner with the same color. Therefore

$$\left|\mathcal{D}_{N(v)}\right| \geq \left|\mathcal{C} \setminus \mathcal{C}_{\mathcal{H}(v)}\right| - \frac{\phi - x}{2} = (\Delta + 1) - (\Delta - \phi) - \frac{\phi - x}{2} \leq \frac{\phi}{2} + 1,$$

so $\left|\mathcal{D}_{N(v)}\right| \in \Omega(\phi) = \Omega(\Delta)$. □

Table 2: Data for level data structure for each $v \in V$

Name	Definition	Description
L	$\log_\beta \Delta$	Highest level
$\ell(v)$	$\in \{4, \ldots, L\}$	Level of v
$\deg(v)$	$\lvert N(v) \rvert$	Degree of v
τ_v	$\in \mathbb{N}$	Last time at which v was recolored
$\mathcal{L}(v)$	$\{u \mid \{u, v\} \in E \land \ell(u) < \ell(v)\}$	Neighbors of v of lower level
$\mathcal{H}(v)$	$\{u \mid \{u, v\} \in E \land \ell(u) \geq \ell(v)\}$	Neighbors of v with at least v's level
$\mathcal{S}(v)$	$\{u \mid \{u, v\} \in E \land \ell(u) = \ell(v)\}$[1]	Neighbors of v with the same level as v
$\mathcal{B}_{N(v)}$	$\{c \mid \forall u \in N(v) : \chi^*(u) \neq c\}$	Colors of no neighbor of v
$\mathcal{C}_{\mathcal{H}(v)}$	$\{c \mid \exists u \in \mathcal{H}(v) \mid \chi^*(u) = c\}$	Colors of an up-neighbor of v
$\mathcal{U}_{\mathcal{L}(v)}$	$\{c \mid \exists!\, u \in \mathcal{L}(v) \mid \chi^*(u) = c\}$	Colors of exactly one down-neighbor of v
$\mathcal{M}_{\mathcal{L}(v)}$	$\{c \mid \exists_{\geq 2} u \in \mathcal{L}(v) \mid \chi^*(u) = c\}$[2]	Colors of at least two down-neighbors of v
$\mathcal{D}_{N(v)}$	$\mathcal{C} \setminus \mathcal{C}_{\mathcal{H}(v)} \setminus \mathcal{M}_{\mathcal{L}(v)}$	Colors of no up-neighbor and at most one down-neighbor of v

[1] Note that this is a subset of $\mathcal{H}(v)$.
[2] Note that this is $\mathcal{C} \setminus \mathcal{B}_{N(v)} \setminus \mathcal{C}_{\mathcal{H}(v)} \setminus \mathcal{U}_{\mathcal{L}(v)}$.

3.1.2 Algorithm

Algorithm 1: EDGE INSERTION$_{\text{log}}$

Input: G; $\{u, v\} \leftrightsquigarrow E(G)$ for deletion or insertion; ℓ and associated data structures
Output: Updated $(\Delta + 1)$-coloring $\chi*$

```
UPDATE LEVELS;
if {u, v} is being inserted and χ*(u) = χ*(v) then
    x ← arg max_{x∈{u,v}} τ_w;          // More-recently recolored
    RECOLOR_log(x);
```

Algorithm 2: UPDATE LEVELS

Input: G; χ^*; ℓ and associated data structures
Output: ℓ and associated data structures with Invariant 3.1 and Invariant 3.2 satisfied

while *Invariant 3.1 or Invariant 3.2 is violated* **do**
  if *there exists $x \in V$ that violates Invariant 3.2 (having more than $\beta^{\ell(x)}$ down-neighbors)* **then**
    find the minimum level $k > \ell(x)$ for x that would have $|\mathcal{L}(x)| + |\mathcal{S}(x)| \leq \beta^k$;
    $\ell(x) \leftarrow k$;
    update any auxiliary data structures;
  else
    find a vertex $x \in V$ that violates Invariant 3.1 (having fewer than $\beta^{\ell(x)-5}$ same-level neighbors);
    if *there exists a level $k < \ell(x)$ for x that would have $|\mathcal{L}(x)| \geq \beta^{k-1}$* **then**
      let k' be the maximum such level;
      $\ell(x) \leftarrow k'$;
    else
      $\ell(x) \leftarrow 4$;
  update any auxiliary data structures;

Algorithm 3: RECOLOR$_{\log}$

Input: G; $v \in V(G)$ for recoloring; χ^*; ℓ and associated data structures
Output: χ^* with new color for v

$c \sim \mathcal{D}_{N(v)}$ uniformly at random;
$\chi^*(v) \leftarrow c$;
Update relevant auxiliary data structures;
if $\exists\, w \in \mathcal{L}(v) : \chi^*(w) = c$ **then**
  RECOLOR$_{\log}(w)$;

3.1.3 Analysis

Insertion (Algorithm 1) relies on two subroutines: UPDATE LEVELS (Algorithm 2) and RECOLOR$_{\log}$ (Algorithm 3). To insert an edge $\{u, v\}$ to $E(G)$, first the level data structure ℓ is repaired so that Invariants 3.1 and 3.2 are satisfied, then $\{u, v\}$ is added to $E(G)$. If $\chi^*(u) = \chi^*(v)$, the one more recently recolored is recolored. An auxiliary list consisting of invariant-violating *dirty* vertices is also maintained.

The idea in UPDATE LEVELS is to move vertices in ℓ to ensure the invariants hold before repairing χ^* if needed. First, as long as there is a vertex x with too many (more than $\beta^{\ell(x)}$) down-neighbors, x is moved up to the lowest level such that Invariant 3.2 holds of it at that level. Note that x moves up levels in ℓ, the number of down-neighbors of x increases more slowly than $\beta^{\ell(x)}$; in particular it is always possible to fix x this way. On the other hand, promoting x might make some other vertices dirty, but by the end of the while loop all vertices are clean with respect to Invariant 3.2. (For a proof of this, see [BCHN18] Lemma 3.1.) After cleaning each vertex with respect to Invariant 3.1, UPDATE LEVELS cleans the vertices with respect to Invariant 3.1. Namely, while there is a dirty vertex x, x is moved down to the highest level such that it is clean. The properties of the cleaning loop for Invariant 3.2 also hold in this loop.

UPDATE LEVELS is the source of EDGE INSERTION$_{\log}$'s $O(\log \Delta)$ runtime. Details of the proof of UPDATE LEVEL's $O(\beta) = O(\log \Delta)$ running time are in [BGKLS22] Theorem 3.2. More importantly, the other component of EDGE INSERTION$_{\log}$ — RECOLOR$_{\log}$ — is implementable in constant expected amortized time.

In RECOLOR$_{\log}$, if $\chi^*(u) \neq \chi^*(v)$ then χ^* remains proper. Otherwise, the more-recently recolored vertex x of u and v, tracked by τ_u, τ_v, is recolored. To recolor x, $\chi^*(x)$ is set to a random sample $c \sim \mathcal{D}_{N(x)}$. Note that each color in $\mathcal{D}_{N(x)}$ is assigned by χ^* to at most one neighbor w of x. If $c = \chi^*(w)$, then $\{x, w\}$ is a monochromatic edge. To fix this corruption, RECOLOR$_{\log}$ recurses on w. Observe that this can cascade down $O(\beta^{\ell(x)})$ levels, each requiring a sample from $\mathcal{D}_{N(x)}$ and a scan of $\mathcal{L}(x)$ for w. Sampling is difficult technically because $\left|\mathcal{D}_{N(x)}\right| \gg \beta^{\ell(x)}$ in general, but truncating the sample at the first $\beta^{\ell(x)}$ elements of $\mathcal{D}_{N(x)}$ works.

In any case, each initial call to RECOLOR$_{\log}$ takes time in $O(\beta^{\ell(x)}) = O(\Delta)$. To show that RECOLOR$_{\log}$ has expected amortized time in $O(1)$, it is necessary to show that it is called with sufficiently low probability given χ^* and x's new color. By Lemma 3.3 and the definition of $\mathcal{D}_{N(x)}$, at most one of $\mathcal{D}_{N(x)}$'s $\Omega(\Delta)$ elements conflicts with a down-neighbor $w \in \mathcal{L}(x)$ of x; therefore $c \sim \mathcal{D}_{N(x)}$ is a conflict with probability $\Pr[\kappa_{\chi^*,c,w}] \in O(\frac{1}{\Delta})$, so the expected amortized runtime $T_{\text{RECOLOR}_{\log}}$ of RECOLOR$_{\log}$ is

$$\mathbb{E}[T_{\text{RECOLOR}_{\log}}] = \frac{\Pr[\kappa_{\chi^*,c,w}]}{\Delta} \in O\left(\frac{\Delta}{\Delta}\right) = O(1).$$

Then the total expected amortized update time for $\mathcal{A}_{\log}$ is in $O(\log \Delta)$.

3.2 Constant-time dynamic $(\Delta + 1)$-coloring

To improve the performance of dynamic $(\Delta + 1)$-coloring from $O(\log \Delta)$ to $O(1)$ update time, [BGKLS22] builds on [BCHN18] by disjoining RECOLOR$_{\text{constant}}$ into two cases. One case corresponds to RECOLOR$_{\text{constant}}$ from [BCHN18]; in the other case, when the recoloring palette is sufficiently small, a deterministic recoloring subroutine is used with time in $O(n)$.

For this algorithm, $\mathcal{A}_{\text{constant}}$, the level data structure ℓ is spiritually the same as in $\mathcal{A}_{\log}$. It has levels $\{-1, \ldots, \lceil \log_3(n) \rceil\}$ with associated data similar to those in $\mathcal{A}_{\log}$. Note that in both $\mathcal{A}_{\text{constant}}$ and $\mathcal{A}_{\log}$ there are numerous implementation details that are laid out in the papers. In this report I focus instead on the crucial algorithmic and formal details that distinguish these methods from others and from each other. The analysis and its apparatus in $\mathcal{A}_{\text{constant}}$ is significantly more involved than that of $\mathcal{A}_{\log}$, but the key invariant is similar.

Invariant 3.4. *If $v \in V$ was moved to level $\ell(v) \neq -1$ during a recoloring, the recoloring used a palette with size most $\frac{3^{\ell(v)}+1}{2}$. If $\ell(v) = 1$ the palette had size 1.*

$\mathcal{A}_{\text{constant}}$ has update time in $O(1 + n\frac{\log n + \Delta}{t})$ with high probability. For $t \in \Omega(n(\log n + \Delta))$ this is $O(1)$. The factor of Δn comes from initializing the associated data structures that constitute the level data structure and in particular for each $v \in V$ there are $O(\Delta)$ elements of $\mathcal{D}_{N(v)}$. The factor of $n \log n$ comes from the total update time of vertices at levels containing at least a constant fraction of vertices that are recolored because they are at some point the more recently recolored endpoint of a conflicting inserted edge. The level data structure is reproduced in Figure 2 with several new definitions, including some artifacts of the analysis whose explanation I omit; the colored terms are defined in 3.2.2. In addition to numbering levels differently from $\mathcal{A}_{\log}$, proving constant update

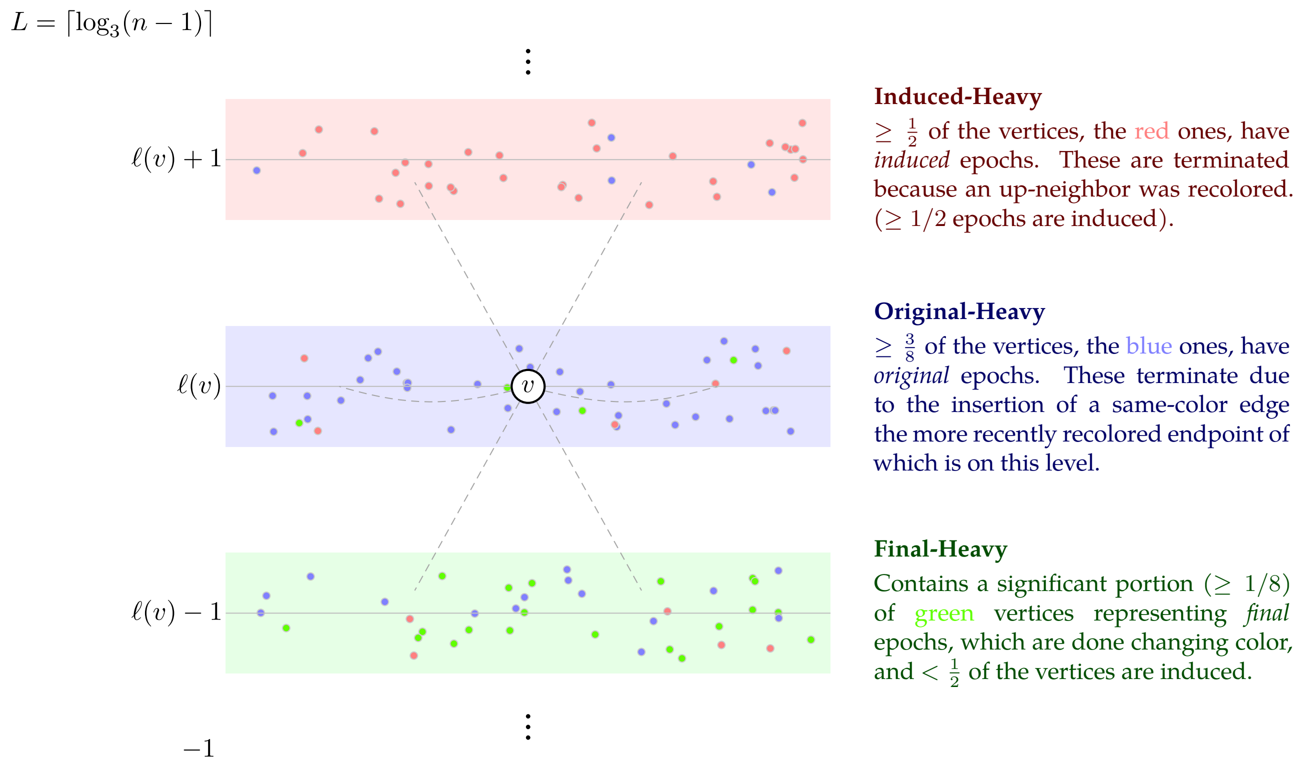

Figure 2: Classification of levels based on epoch types. The small colored nodes represent individual epochs, colored by their termination cause. Vertex v is shown at level $\ell(v)$, with weak dashed lines indicating its neighborhood in different levels.

time with high probability for $\mathcal{A}_{\text{constant}}$ involves tracking which vertices are recolored at an *original* root call to RECOLOR$_{\text{constant}}$ — endpoints of an inserted edge that conflicts with χ^* — or from an *induced*, descendant call to RECOLOR$_{\text{constant}}$.

3.2.1 Algorithm

Let $\phi(v, \ell^*) = |\{u \in N(v) \mid \ell(u) < \ell^*\}|$ be the number of neighbors of v below level ℓ^* for each $v \in V$. EDGE INSERTION$_{\text{constant}}$ in the nontrivial case of a conflicting edge is almost the same as EDGE INSERTION$_{\text{log}}$, without the step to move vertices that break the invariants associated with $\mathcal{A}_{\text{log}}$ between levels, and with retries when ϕ is too large.

Algorithm 4: EDGE INSERTION$_{\text{constant}}$

Input: G; $\{u, v\} \leftrightsquigarrow E(G)$ for deletion or insertion; ℓ and associated data structures
Output: Updated $(\Delta + 1)$-coloring $\chi*$

if $\{u, v\}$ *is being inserted and* $\chi^*(u) = \chi^*(v)$ **then**
  $x \leftarrow \arg\max_{x \in \{u,v\}} \tau_w$; `// More-recently recolored`
  RECOLOR$_{\text{constant}}(x)$;

Algorithm 5: RECOLOR$_{\text{constant}}$

Input: G; $v \in V(G)$ for recoloring; χ^*; ℓ and associated data structures
Output: χ^* with new color for v

if $\phi(x, \ell(x)) < 3^{\ell(x)+2}$ **then**
  DETERMINISTIC RECOLOR(x);
  return null;
else
  return RANDOM RECOLOR(x);

Algorithm 6: DETERMINISTIC RECOLOR

Input: G; $v \in V(G)$ for recoloring; χ^*; ℓ and associated data structures
Output: χ^* with new color for v

for $c \in \mathcal{D}_{N(v)}$ **do**
  if *there is no vertex* $w \in \mathcal{L}(v)$ *with color* $\chi^*(w) = c$ **then**
    Set $\chi^*(v) \leftarrow c$;
    Update down-neighbor data structure of v;
    Set $\ell(v) = -1$;
    return;

Algorithm 7: RANDOM RECOLOR

Input: G; $v \in V(G)$ for recoloring; χ^*; ℓ and associated data structures
Output: χ^* with new color for v

Set $\ell' \leftarrow \ell(v)$;
while $\phi(v, \ell' + 1) \geq 3^{\ell'+2}$ **do**
  Increment ℓ';
Set $\ell(v) \leftarrow \ell'$;
Draw $c \sim \mathcal{D}_{N(v)}$ uniformly at random;
if $c \neq \chi^*(v)$ **then**
  Set $\chi^*(v) \leftarrow c$;
  Update down-neighbor data structure of v;
if $c \in \mathcal{U}_{\mathcal{L}(v)} \setminus \mathcal{C}_{\mathcal{H}(v)}$ **then**
  Let $w \in \mathcal{L}(v)$ be such that $\chi^*(w) = c$;
  RECOLOR$_{\text{constant}}(w)$;

3.2.2 Analysis

The essence of the analysis for $\mathcal{A}_{\text{constant}}$ is the notion of an epoch $\mathcal{E}$, which characterizes the infrequency with which some vertices are recolored. Namely, epoch $\mathcal{E}$ corresponds to a vertex $x = v(\mathcal{E})$ during which x maintains its color $\chi^*(\mathcal{E})$ and level $\ell(v)$. The cost $c(\mathcal{E})$ of an epoch is the time required for the call to RECOLOR$_{\text{constant}}$, applied to $v(\mathcal{E})$, that began $\mathcal{E}$ by changing $\chi^*(v(\mathcal{E}))$ and $\ell(v(\mathcal{E}))$. When $\ell(x) = -1$ for $x = v(\mathcal{E})$, and therefore x is done being recolored, the analysis registers $c(\mathcal{E})$. In other words, each vertex's contribution to $\mathcal{A}_{\text{constant}}$'s total update time comes from the sum of the cost of its epochs. An epoch is demarcated by either an original call to RECOLOR$_{\text{constant}}$, when it is called original, or an induced call to RECOLOR$_{\text{constant}}$, when it is called induced. See Figure 3.

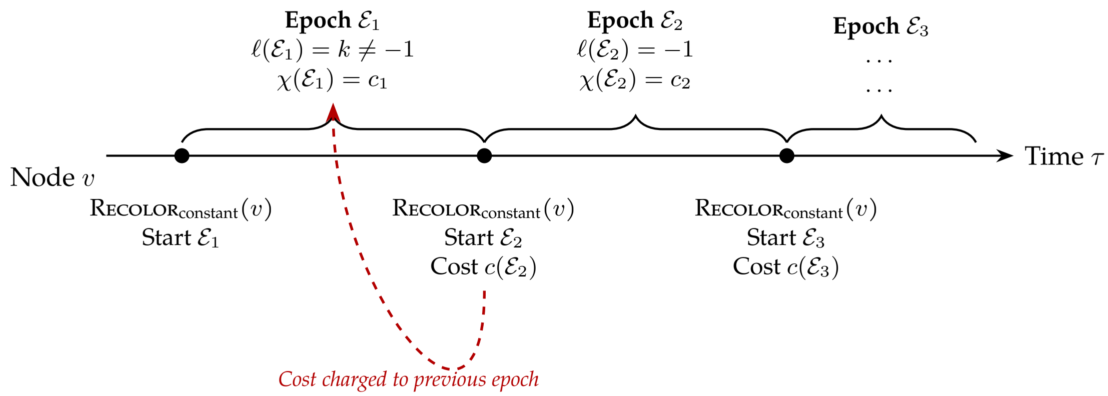

Figure 3: Epochs $\mathcal{E}_1, \mathcal{E}_2, \mathcal{E}_3$ for vertex $x = v(\mathcal{E}_1) = v(\mathcal{E}_2) = v(\mathcal{E}_3)$. $\mathcal{E}$ corresponds to the maximal time interval between consecutive recolorings of x. The cost of an epoch with level $\ell(\mathcal{E}) = -1$ ($\mathcal{E}_2$) is charged to the preceding epoch ($\mathcal{E}_1$). Here $\mathcal{E}_3$ could be induced.

Because no vertex $v(\mathcal{E})$ changes level during $\mathcal{E}$, the set $\mathscr{E}$ of epochs can be partitioned across ℓ. The bottleneck set of levels in ℓ is the set of *original-heavy* levels, in which less than half of the vertices have induced epochs and less than an eighth have final epochs; that is, at least $\frac{3}{8}$ of the vertices in an original-heavy level have an original call to RECOLOR$_{\text{constant}}$. The notation c for cost is extended to the sum of costs of epochs in a level.

Then the main task of the analysis is to bound the cost of original-heavy levels, which is non-trivial because the cost of an original-heavy level depends on the fact that it is original-heavy. This means that Invariant 3.4 does not help to lower bound the expected number of insertions before a conflicting edge is inserted analogously to how Invariants 3.1 and 3.2 ensured a sufficiently small probability of conflict for each edge insertion in $\mathcal{A}_{\log}$.

To overcome this obstacle in the analysis, two time functions are defined. First, the duration $dur(\mathcal{E})$ of epoch $\mathcal{E}$ is the number of edge insertions during $\mathcal{E}$. Second, the pseudo-duration $psdur(\mathcal{E})$ of $\mathcal{E}$ is the number of distinct colors that neighbors of $v(\mathcal{E})$ take during $\mathcal{E}$ until one of them takes the color $\ell(\mathcal{E})$ that $v(\mathcal{E})$ chose. For example, an adaptive adversary would ensure that the other endpoint, u, of each insertion $(v(\mathcal{E}), u)$ incident with $v(\mathcal{E})$ has the same color $\chi^*(\mathcal{E})$ as what $v(\mathcal{E})$ took at the beginning of $\mathcal{E}$. See Figure 4.

By definition, $psdur(\mathcal{E}) \leq dur(\mathcal{E})$, and it turns out that there is a low upper bound on the probability that $\mathcal{E}$ is short ([BGKLS22] Lemma 3.11). Further, there is a low upper bound on the probability that a level has too many epochs while having more than a small number of short epochs. In fact, roughly speaking, the probability that any level has this undesirable trait is in $O(\frac{\log n}{n^a})$ for large constant a. This is responsible for the factor of $\log n$ in the total runtime, which is amortized out by sufficiently many edge insertions. Without this bound on the chance of epochs being too short in pseudo-duration, the deterministic runtime of $\mathcal{A}_{\log}$ would be in $O(tn^2 + \Delta n)$; in this case, even with $t \in \Omega(n(\log n + \Delta))$, $\mathcal{A}_{\log}$'s update time would have at least two extra factors of n.

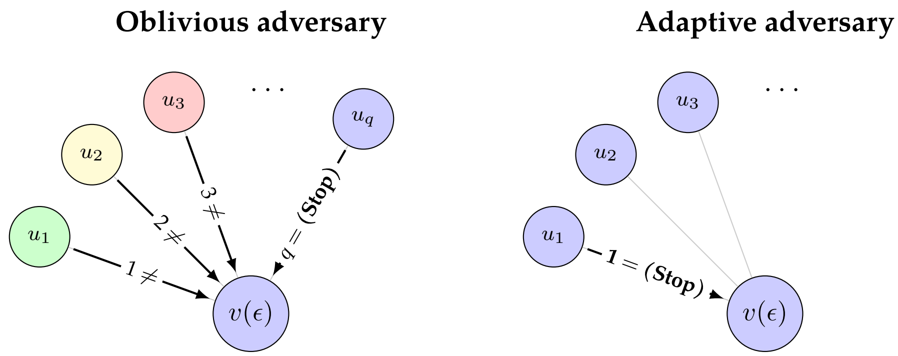

Figure 4: Example pseudo-duration, first in the oblivious case as in $\mathcal{A}_{\log}$, then how an adaptive adversary would exploit pseudo-duration to try to decrease epoch length.

4 Dynamic $(\Delta + 1)$-coloring with adaptive adversaries

$\mathcal{A}_1$, $\mathcal{A}_2$, $\mathcal{A}_{\log}$, and $\mathcal{A}_{\text{constant}}$ assume oblivious inputs. That is, they assume inserted edges are drawn uniformly at random from the set $\binom{V(G)}{2} \setminus E(G)$ of non-edges of G. Obliviousness is congruous with various application paradigms in which inputs come from complicated real-world systems with no tractable distribution. [BRW25] examines $(\Delta + 1)$-coloring that performs robustly against adaptively adversarial inputs, which break uniformity assumptions of the input and pessimally induce worst-case performance. An adaptive adversary $\forall$ can stealthily inspect or simulate $(\Delta+1)$-colorer $\mathcal{A}$, determine $\mathcal{A}$'s performance on each permutation $\mathcal{U} \in \mathfrak{S}_{\mathscr{U}}$ of the desired inputs $\mathscr{U}$, and input the $\mathcal{U}$ that maximizes $\mathcal{A}$'s amortized update time to $\mathcal{A}$. To wit, [BRW25] presents a dynamic $(\Delta + 1)$-colorer, $\mathcal{A}_\forall$, with update time in $\tilde{O}(n^{8/9})$ against such an adaptive adversary. To obtain sublinear time against $\forall$, $\mathcal{A}_\forall$ takes a phase-based approach that handles so-called sparse and dense vertices separately. Sparse vertices are recolored randomly at deterministic intervals so that their palettes typically have sufficiently many surplus colors; dense vertices are recolored repeatedly a bounded number of times in expectation until they are colored properly.

In the previous $(\Delta + 1)$-colorers, foreknowledge of Δ allows the algorithm to determine the number of buckets or height of the hierarchical vertex partition. In this case, using all $O(\Delta)$ available colors is an explicit invariant.

Invariant 4.1. *Each color $c \in \{1, \ldots, \Delta + 1\}$ is assigned to a number of vertices in $\tilde{O}(\frac{n}{\Delta})$.*

With Invariant 4.1, verifying that a color is proper to assign to some vertex takes time in $\tilde{O}(\frac{n}{\Delta})$ instead of in $\Theta(\Delta)$. $\mathcal{A}_\forall$ uses a sparse–dense vertex decomposition to make it easier to track which colors are available for each vertex. The decomposition is roughly the Harris–Schneider–Su [HSS16] (HSS) decomposition, which partitions $V(G)$ into sparse vertices V_S and dense vertices V_D. Let $G_S \coloneqq G[V_S]$ and $G_D \coloneqq G[V_D]$. V_S is the set of vertices each v of which has $\left|E(G) \cap N(v)^2\right| \leq (1 - \epsilon^2)\binom{\Delta}{2}$; i.e., the neighborhood of each sparse vertex is ϵ^2-far from being a Δ-clique. It further partitions $G[V_D]$ into $C_1, \ldots, C_k$:

$$V(G) = V_S \oplus \bigoplus_{i=1}^{k} V(C_i).$$

Each $C_i \in \{C_1, \ldots, C_k\}$ is an *almost-clique*, so for some $\epsilon > 0$

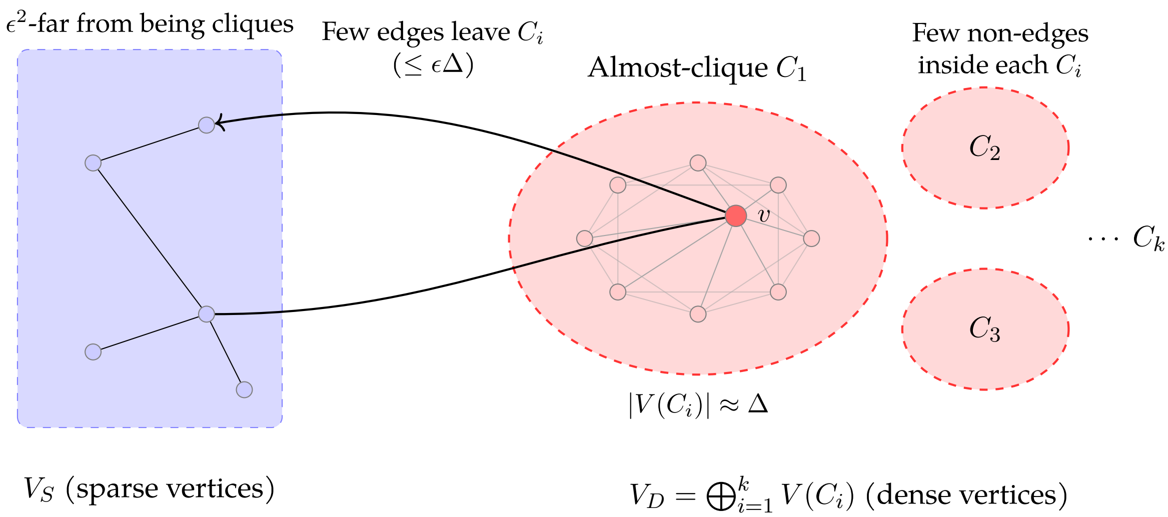

Figure 5: HSS decomposition scheme. $V(G)$ is partitioned into sparse vertices V_S and dense vertices V_D. V_D is a disjoint union of almost-cliques $C_1, C_2, \ldots, C_k$. Vertex $v \in C_1$ illustrates the property of being connected to almost all of its own clique with few edges leaving it.

- $|V(C_i)| \in [(1-\epsilon)\Delta, (1+\epsilon)\Delta]$ (C_i has order close to Δ)

and for any $v \in C_i$

- $|V(C_i)| - |N(v)| \leq \epsilon\Delta$ (v is adjacent to almost all of C_i) and
- $|N(v) \cap V(G) \setminus V(C_i)| \leq \epsilon\Delta$ (few edges leave C_i).

See Figure 5. (There are a few more properties that hold of the almost-cliques involving adjustment complexity (see [BRW25] Theorem 2).)

Decomposition is useful because $\mathcal{A}_\forall$ maintains colorings of G_S and the C_i independently. For G_S, $\mathcal{A}_\forall$ runs a one-shot refresh colorer that ensures each sparse vertex has $\Omega(\epsilon^2\Delta)$ surplus colors in expectation. Namely, One-shot sparse coloring samples a random color for each vertex uniformly at random, then assigns the nonconflicting colors. See Algorithm 8. Typically, this successfully recolors a constant fraction of G_S and assigns identical colors to many neighbors of the typical vertex, thereby leaving $\Omega(\epsilon^2\Delta)$ surplus colors for each vertex's palette. To ensure these colors are available, One-shot sparse recoloring is run after every $\Theta(\epsilon^2\Delta)$ updates; each update recolors at most one vertex, so the surplus colors at vertex are reduced by at most one, giving remaining surplus colors in $\Omega(\epsilon^2\Delta) - \Theta(\epsilon^2\Delta) = \Omega(\epsilon^2\Delta)$. In other words, for sparse vertices there are $\Theta(\epsilon^2\Delta)$-update batches of updates, demarcated by One-shot sparse recoloring. Then to recolor a sparse vertex, the probability that one of these surplus vertices is selected uniformly at random is in $\frac{\Omega(\epsilon^2\Delta)}{O(\Delta+1)} = \Omega(\epsilon^2)$. From this, $O(\epsilon^{-2})$ such samples, each with a $\tilde{O}(\frac{n}{\Delta})$ feasibility check thanks to Invariant 4.1, will produce a good color, giving $\tilde{O}(\frac{n}{\epsilon^2\Delta}) = n^{1-\Omega(1)}$ — sublinear — recoloring time for sparse vertices.[3]

Algorithm 8: ONE-SHOT SPARSE COLORING

Input: Graph G; sparse vertices $V_S \subset V(G)$
Output: Recolored vertices $V' \subseteq V_S$

Initialize $V' \leftarrow \emptyset$;
for $v \in V_S$ **do**
  Sample $\chi'(v) \sim [\Delta + 1]$ uniformly at random;
for $v \in V_S$ **do**
  if $c(v) \neq \chi^*(w)$ *for each* $w \in N(v)$ **then**
    Assign $\chi^*(v) \leftarrow c(v)$;
    Add v to V';
return V'_S;

For V_D, dynamic coloring is more involved. The key technique is a dynamized static algorithm from [ACK19] in which each almost-clique C_i is colored as follows. First, an approximate maximum matching of non-edges $\binom{C_i}{2} \setminus E(C_i)$ is found, and matching vertices are colored identically. In other words, as many nonadjacent vertices as possible are colored, using two vertices per color. Second, a perfect matching from the remaining vertices to the remaining colors is found. The matching is on a bipartite graph $\mathcal{H}$ with parts $\mathcal{V}$ (remaining vertices) and $\mathcal{C}$ (remaining colors), with $\{v, c\} \in E(\mathcal{H})$ if $c \notin N(v) \setminus \mathcal{V}$. A perfect matching on $\mathcal{H}$ takes each vertex to an unused color, providing a proper coloring. The existence of the matching is nonobvious, and making this step dynamic without spending time linear in Δ (and therefore possibly linear in n, against $\forall$) is highly nontrivial.

To dynamize this static algorithm, there are two cases. In the first case, for almost-cliques with order at least $\Delta + 1$, colors are partitioned into heavy and light colors. A color c is heavy in almost-clique C_i if there is a number in $\Omega(\Delta)$ of edges (a_i, b_i) such that $a_i \in V(C_i), b_i \notin V(C_i), \chi^*(a_i) = c$. Light colors are not heavy. Heavy colors are less useful than light colors. Fortunately, $\mathcal{H}$ is still colorable if the heavy colors are excluded from $\mathcal{C}$, using an augmenting path that exists with high constant probability. In the second case, for almost-cliques with size less than $\Delta + 1$, there are two subcases. If the non-edge matching is sufficiently large (at least $\frac{\Delta}{10}$), $\mathcal{A}_\forall$ behaves similarly to the large almost-clique case, because enough of the vertices have already been handled. Otherwise, to recolor $v \in V(C_i)$, a random color c yet unused in C_i is picked uniformly at random. If v has sparse neighbors, c might cause a conflict, with probability in $\Omega(\frac{1}{k})$, where $k = \Delta + 1 - |V(C_i)|$. Then in $\tilde{O}(\epsilon\Delta)$ such samples, $\mathcal{A}_\forall$ will typically find a usable color. If this does not work, then an augmenting path through $\mathcal{H}$ of length five, using three vertices $v_1, v_2, v_3 \in \mathcal{V}$ and three colors in $\mathcal{C}$, gives sufficiently large constant probability of properly coloring v_1, v_2, v_3. Finding the augmenting path takes the most runtime: $\tilde{O}(k)$ samples, each taking $\frac{\tilde{n}}{\Delta}$-time feasibility checks, contribute time in $\tilde{O}(\epsilon n)$ — which is sublinear.

5 Acknowledgments

Thanks to the authors for their contributions to the field and to Quanquan Liu for comments. Thanks also to OpenAI and Google's generative pre-trained transformer LLMs (Large LaTeX Mud-

[3]After each batch begins with ONE-SHOT SPARSE RECOLORING, an extra greedy algorithm from [AY25] colors the constant fraction of V_S that was not successfully recolored.

slingers), which helped make the figures.

References

[ACK19] Sepehr Assadi, Yu Chen, and Sanjeev Khanna. "Sublinear algorithms for $(\Delta + 1)$ vertex coloring". In: *Proceedings of the Thirtieth Annual ACM-SIAM Symposium on Discrete Algorithms*. SIAM. 2019, pp. 767–786 (cit. on p. 14).

[AY25] Sepehr Assadi and Helia Yazdanyar. "Simple sublinear algorithms for $(\Delta + 1)$ vertex coloring via asymmetric palette sparsification". In: *2025 Symposium on Simplicity in Algorithms (SOSA)*. SIAM. 2025, pp. 1–8 (cit. on p. 14).

[BCKLVRV17] Luis Barba, Jean Cardinal, Matias Korman, Stefan Langerman, André Van Renssen, Marcel Roeloffzen, and Sander Verdonschot. "Dynamic graph coloring". In: *Workshop on algorithms and data structures*. Springer. 2017, pp. 97–108 (cit. on pp. 1, 3).

[BRW25] Soheil Behnezhad, Rajmohan Rajaraman, and Omer Wasim. "Fully dynamic $(\Delta + 1)$-coloring against adaptive adversaries". In: *Proceedings of the 2025 Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*. SIAM. 2025, pp. 4983–5026 (cit. on pp. 1, 12, 13).

[BCHN18] Sayan Bhattacharya, Deeparnab Chakrabarty, Monika Henzinger, and Danupon Nanongkai. "Dynamic algorithms for graph coloring". In: *Proceedings of the Twenty-Ninth Annual ACM-SIAM Symposium on Discrete Algorithms*. SIAM. 2018, pp. 1–20 (cit. on pp. 1, 4, 7, 8).

[BGKLS22] Sayan Bhattacharya, Fabrizio Grandoni, Janardhan Kulkarni, Quanquan C Liu, and Shay Solomon. "Fully dynamic $(\Delta + 1)$-coloring in $O(1)$ update time". In: *ACM Transactions On Algorithms (TALG)* 18.2 (2022), pp. 1–25 (cit. on pp. 1, 4, 8, 11).

[HSS16] David G Harris, Johannes Schneider, and Hsin-Hao Su. "Distributed $(\Delta + 1)$-coloring in sublogarithmic rounds". In: *Proceedings of the forty-eighth annual ACM symposium on Theory of Computing*. 2016, pp. 465–478 (cit. on p. 12).