Improved Dynamic Graph Coloring

Shay Solomon† Nicole Wein‡

Abstract

This paper studies the fundamental problem of graph coloring in fully dynamic graphs. Since the problem of computing an optimal coloring, or even approximating it to within $n^{1-\epsilon}$ for any $\epsilon > 0$, is NP-hard in static graphs, there is no hope to achieve any meaningful computational results for general graphs in the dynamic setting. It is therefore only natural to consider the combinatorial aspects of dynamic coloring, or alternatively, study restricted families of graphs.

Towards understanding the combinatorial aspects of this problem, one may assume a black-box access to a static algorithm for $C$-coloring any subgraph of the dynamic graph, and investigate the trade-off between the number of colors and the number of recolorings per update step. Optimizing the number of recolorings, sometimes referred to as the recourse bound, is important for various practical applications. In WADS’17, Barba et al. devised two complementary algorithms: For any $\beta > 0$, the first (respectively, second) maintains an $O(C\beta n^{1/\beta})$ (resp., $O(C\beta)$)-coloring while recoloring $O(\beta)$ (resp., $O(\beta n^{1/\beta})$) vertices per update. Barba et al. also showed that the second trade-off appears to exhibit the right behavior, at least for $\beta = O(1)$: Any algorithm that maintains a $c$-coloring of an $n$-vertex dynamic forest must recolor $\Omega(n^{c^2-1})$ vertices per update, for any constant $c \geq 2$. Our contribution is two-fold:

- We devise a new algorithm for general graphs that improves significantly upon the first trade-off in a wide range of parameters: For any $\beta > 0$, we get a $\tilde{O}(C\beta \log n)$-coloring with $O(\beta)$ recolorings per update, where the $\tilde{O}$ notation suppresses polyloglog($n$) factors. In particular, for $\beta = O(1)$ we get constant recolorings with polylog($n$) colors; not only is this an exponential improvement over the previous bound, but it also unveils a rather surprising phenomenon: The trade-off between the number of colors and recolorings is highly non-symmetric.
- For uniformly sparse graphs, we use low out-degree orientations to strengthen the above result by bounding the update time of the algorithm rather than the number of recolorings. Then, we further improve this result by introducing a new data structure that refines bounded out-degree edge orientations and is of independent interest.

1 Introduction

1.1 Background

Graph coloring is one of the most fundamental and well studied problems in computer science, having found countless applications over the years, ranging from scheduling and computational vision to biology and chemistry. A proper $C$-coloring of a graph $G = (V, E)$, for a positive integer $C$, assigns a color in $\{1, \ldots, C\}$ to every vertex, so that no two adjacent vertices are assigned

---

* A preliminary version of this paper appeared in the proceedings of ESA’18.
† IBM Research, TJ Watson Research Center, Yorktown Heights, New York, USA. Supported by the IBM Herman Goldstine Postdoctoral Fellowship.
‡ Massachusetts Institute of Technology, Cambridge, Massachusetts, USA. Supported by an NSF Graduate Fellowship and NSF Grant CCF-1514339.
the same color. The chromatic number of the graph is the smallest integer \( C \) for which a proper \( C \)-coloring exists. (We shall write “coloring” as a shortcut for “proper coloring”, unless otherwise specified.)

This paper studies the problem of graph coloring in fully dynamic graphs subject to edge updates. A dynamic graph is a graph sequence \( G = (G_0, G_1, \ldots, G_M) \) on a fixed vertex set \( V \), where the initial graph is \( G_0 = (V, \emptyset) \) and each graph \( G_i = (V, E_i) \) is obtained from the previous graph \( G_{i-1} \) in the sequence by either adding or deleting a single edge. We investigate general graphs as well as uniformly sparse graphs. The “uniform density” of the graph is captured by its arboricity: a graph \( G = (V, E) \) has arboricity \( \alpha \) if \( \alpha = \max_{U \subseteq V} \left\lceil \frac{|E(U)|}{|U|-1} \right\rceil \), where \( E(U) = \{(u, v) \in E \mid u, v \in U\} \).

That is, the arboricity is close to the maximum density \( |E(U)|/|U| \) over all induced subgraphs of \( G \).

The class of constant arboricity graphs, which contains planar graphs, bounded tree-width graphs, and in general all minor-free graphs, as well as some classes of “real-world” graphs, has been subject to extensive research in the dynamic algorithms literature [15, 16, 63, 70, 59, 60, 52, 35, 22]. A dynamic graph of arboricity \( \alpha \) is a dynamic graph such that all graphs \( G_i \) have arboricity bounded by \( \alpha \).

It is NP-hard to approximate the chromatic number of an \( n \)-vertex graph to within a factor of \( n^{1-\epsilon} \) for any constant \( \epsilon > 0 \), let alone to compute the corresponding coloring [74, 46]. Consequently, there is no hope to achieve any meaningful computational results for general graphs in the dynamic setting. It is perhaps for that reason that the literature on dynamic graph coloring is sparse (see Section 1.1.1). Nevertheless, as discussed next, one may view the area of dynamic graph algorithms as lying within the wider area of local algorithms, in which there has been tremendous success in the context of graph coloring.

When dealing with networks of large scale, it is important to devise algorithms that are intrinsically local. Roughly speaking, a local algorithm restricts its execution to a small part of the network, yet is still able to solve a global task over the entire network. There is a long line of work on local algorithms for graph coloring and related problems from various perspectives. For example, seminal papers on distributed graph coloring [26, 36, 53, 6, 54, 55] laid the foundation for the area of symmetry breaking problems, which remains the subject of ongoing intensive research. Refer to the book of Barenboim and Elkin [10] for a detailed account on this topic. Additionally, graph coloring is well-studied in the areas of property testing [37, 27] and local computation algorithms [65, 34].

### 1.1.1 Dynamic graph coloring

In light of the computational intractability of graph coloring, previous work on dynamic graph coloring is devoted mostly to heuristics and experimental results [56, 64, 78, 42, 41, 61, 66]. From the theoretical standpoint, it is natural to consider the combinatorial aspects of dynamic coloring or to study restricted families of graphs; to the best of our knowledge, the only work on this pioneering front is that of Barba et al. from WADS’17 [8] and Bhattacharya et al. from SODA’18 [17]. Additionally, Parter, Peleg, and Solomon [62] studied this problem in the dynamic distributed setting, and Barenboim and Maimon [11] studied the related problem of dynamic edge coloring. (Our work focuses on amortized time bounds; we henceforth do not distinguish between amortized and worst-case time bounds, unless explicitly specified.)

Barba et al. [8] studied the combinatorial aspects of dynamic coloring in general graphs. They assumed that at all times the graph can be \( C \)-colored and further assumed black-box access to a static algorithm for \( C \)-coloring any subgraph of the current graph. They investigated the trade-off between the number of colors and the number of recolorings (i.e., the number of vertices that change their color) per update step. The number of recolorings is an example of a recourse bound, which
counts the number of changes to the maintained graph structure done following a single update step. This measure has been well studied in the areas of dynamic and online algorithms for various fundamental problems, such as maximal matching, MIS, approximate matching, approximate vertex and set cover, network flow and job scheduling; see [38, 25, 19, 21, 13, 10, 7, 5, 23, 39, 67, 14] and the references therein. In some applications such as job scheduling and web hosting, a change to the underlying structure may be costly. A low recourse bound is particularly important when the dynamic algorithm is used as a black-box subroutine inside a larger data structure or algorithm [16, 2].

Barba et al. devised two complementary algorithms: for any $\beta > 0$, the first (respectively, second) maintains an $O(C \beta n^{1/\beta})$ (resp., $O(C \beta)$)-coloring while recoloring $O(\beta)$ (resp., $O(\beta n^{1/\beta})$) vertices per update step. While these trade-offs coincide at $\beta = \log n$, each providing $O(C \log n)$ recolorings per update, any slight improvement on one of these parameters triggers a significant blowup to the other. In particular, the extreme point $\beta = O(1)$ on the first and second trade-off curves yields a polynomial number of colors and recolorings, respectively. Barba et al. [8] also showed that the second trade-off exhibits the right behavior, at least for $\beta = O(1)$: Any algorithm that maintains a $c$-coloring of an $n$-vertex dynamic forest must recolor $\Omega(n^{c-1})$ vertices per update, for any constant $c \geq 2$. The following question was left open.

**Question 1.1.** Does the first trade-off of [8] exhibit the right behavior, and in particular, does a constant number of recolorings require a polynomial number of colors?

Bhattacharya et al. [17] studied the problem of dynamically coloring bounded degree graphs. For graphs of maximum degree $\Delta$ they presented a randomized (respectively deterministic) algorithm for maintaining a $(\Delta + 1)$ (resp., $\Delta(1 + o(1))$)-coloring with amortized expected $O(\log \Delta)$ (resp., $\text{polylog}(\Delta)$) update time. These results provide meaningful bounds only when all vertices have bounded degree. The following question naturally arises.

**Question 1.2.** Can we get meaningful results for the more general class of bounded arboricity graphs?

Question 1.2 is especially intriguing because, as shown in [8], dynamic forests (which have arboricity 1) appear to provide a hard instance for dynamic graph coloring.

Parter, Peleg, and Solomon [62] studied Question 1.2 in dynamic distributed networks: They showed that for graphs of arboricity $\alpha$ an $O(\alpha \cdot \log^* n)$-coloring can be maintained with $O(\log^* n)$ update time. The update time in this context, however, bounds the number of communication rounds per update, while the number of recolorings done (and number of messages sent) per update is polynomial in $n$, even for forests.

1.2 Our results

We use $\tilde{O}$ notation throughout to suppress polyloglog factors.

1.2.1 General graphs

The following theorem summarizes our main result for general graphs.

**Theorem 1.3.** For any $n$-vertex dynamic graph that can be $C$-colored at all times, there is a fully dynamic deterministic algorithm for maintaining an $O(C \beta \log^3 n)$-coloring with $O(\beta)$ (amortized) recolorings per update step, for any $\beta > 0$. Using randomization (against an oblivious adversary), the number of colors can be reduced by a factor of $\tilde{O}(\log n)$ while achieving an expected bound of $O(\beta)$ recolorings.
Theorem 1.3 with $\beta = O(1)$ yields $O(1)$ recolorings with $\text{polylog}(n)$ colors, thus answering Question 1.1 in the negative. Not only is this result an exponential improvement over the previous bound of $[8]$, but it also unveils a rather surprising phenomenon: The trade-off between the number of colors and recolorings is highly non-symmetric.

We also note that the number of recolorings can be de-amortized.

**A running time bound.** Assuming black-box access to two efficient coloring algorithms we can bound the running time of the algorithm from Theorem 1.3.

**Black-box static algorithm.** Let $A_{G,C}$ be a static algorithm that takes as input a graph $G$ from a graph class $\mathcal{G}$ and a subset $S$ of vertices in $G$, and computes the induced graph $G[S]$ and a $C$-coloring of $G[S]$ in time $T(|S|)$.

**Black-box dynamic algorithm.** Let $A'$ be a fully dynamic algorithm that colors graphs of maximum degree $\Delta$ using $O(\Delta)$ colors. Such algorithms exist: there is a randomized algorithm with $O(1)$ expected amortized update time and a deterministic algorithm with $O(\text{polylog}(\Delta))$ amortized update time $[17]$. Let $T'(\Delta, n) \leq \text{polylog}(\Delta)$ be the running time of an optimal deterministic algorithm for this problem. We state our results in terms of $T'(\Delta, n)$ to emphasize that any improvement over the deterministic algorithm of $[17]$ would yield an improvement to the running time of our algorithm.

**Theorem 1.4.** The randomized algorithm from Theorem 1.3 has expected amortized update time $O\left(\frac{\beta}{n \log n} \sum_{i=0}^{\log n} 2^i T\left(\frac{n}{2^i}\right)\right)$ and the deterministic algorithm from Theorem 1.3 has the same amortized update time with an additional additive factor of $T'(\frac{\log^2 n}{\beta}, n)$.

**Remark.** The randomized black-box dynamic algorithm of $[17]$ that we apply in Theorem 1.4 is actually a simple observation (referred to as a “warm-up result” in $[17]$) which gives a $2\Delta$-coloring with $O(1)$ expected update time. The main result of $[17]$, however, is an algorithm to bound the number of colors by only $\Delta + 1$ (or slightly more).

### 1.2.2 Uniformly sparse graphs

We answer Question 1.2 in the positive by showing that by applying the algorithms from Theorem 1.4 to arboricity $\alpha$ graphs we can obtain a bound on the update time rather than only the number of recolorings.

**Theorem 1.5.** There is a fully dynamic deterministic algorithm for graphs of arboricity $\alpha$ that maintains an $O(\alpha \log^2 n)$-coloring in amortized $T'(\frac{\alpha \log^2 n}{\beta}, n) + O(\beta)$ time per update for any $\beta > 0$. Using randomization (against an oblivious adversary), the number of colors can be reduced by a factor of $\tilde{O}(\log n)$ and the expected amortized update time becomes $O(\beta)$.

Furthermore, we improve over this result when $\beta = o(\sqrt{\log n})$ by designing an algorithm that specifically exploits the structure of arboricity $\alpha$ graphs.

**Theorem 1.6.** There is a fully dynamic deterministic algorithm for graphs of arboricity $\alpha$ that maintains an $O(\alpha \log^2 n)$-coloring in amortized $\tilde{O}(1)$ time.

The proof of Theorem 1.6 relies on a new layered data structure (LDS) for bounded arboricity graphs that we expect will be more widely applicable.

**Definition 1.1.** Given a dynamic graph $G$ of arboricity $\alpha$, a layered data structure (LDS) with parameters $k$ and $\Delta$ is a partition of the vertices into $k$ layers $L_1, \ldots, L_k$ so that all vertices $v$ have at most $\Delta$ neighbors in layers equal to or higher than the layer containing $v$. 4
Theorem 1.7. Let $A''$ be an algorithm for arboricity $\alpha$ graphs that maintains an orientation of the edges with out-degree at most $D$ that performs amortized $F(n)$ flips per update. Then there is an algorithm to maintain an LDS for a fully dynamic graph of arboricity $\alpha$ with $k = O(\log n)$ and $\Delta = O(D + \alpha \log n)$ in amortized deterministic time $O(F(n))$.

1.3 Technical overview

1.3.1 Low out-degree dynamic edge orientations

All of our results are, in different ways, intimately related to the dynamic edge orientation problem for arboricity $\alpha$ graphs, where the goal is to dynamically maintain a low out-degree orientation of the edges in a graph (an orientation with out-degree $\alpha$ always exists [58]). Our algorithm for general graphs (outlined in Section 1.3.2) is inspired by an algorithm for the dynamic edge orientation problem. Our algorithm for bounded arboricity graphs from Theorem 1.5 uses a dynamic edge orientation algorithm as a black-box. Our algorithm for bounded arboricity graphs from Theorem 1.6 uses a dynamic edge orientation to define a potential function useful in the running time analysis (outlined in Section 3.1).

Brodal and Fagerberg [22] initiated the study of the dynamic edge orientation problem and gave an algorithm that maintains an $O(\alpha)$ out-degree orientation in amortized $O(\alpha + \log n)$ time. To analyze this algorithm, they reduced the “online” setting, where we have no knowledge of the future, to the “offline” settings, where we know the entire sequence of edge updates in advance. Thus, in the the subsequent results, it sufficed to consider only the offline setting. Kowalik [49] used an elegant argument to derive a result complementary to [22]: one can maintain an $O(\alpha \log n)$ out-degree orientation in amortized $O(1)$ time. He, Tang, and Zeh [43] completed the picture with a trade-off bound: for all $\beta \geq 1$, one can maintain an $O(\beta \alpha)$ out-degree orientation in amortized $O(\log n / \beta)$ time. The worst-case update time of this problem has also been studied by Kopelowitz et al. [47] and Berglin and Brodal [12].

Dynamic bounded out-degree orientations are a key ingredient in a number of dynamic algorithms for graphs of bounded arboricity [50, 48, 15, 16, 59, 60, 35, 32], as well as in dynamic algorithms for general graphs [69, 15, 16, 18].

1.3.2 Overview of algorithm for general graphs

We apply two black-box coloring algorithms defined in Section 1.2.1, one static and one dynamic. For each vertex $v$, if it is assigned color $c_1$ by the static algorithm and color $c_2$ by the dynamic algorithm, its true color is defined by the pair $(c_1, c_2)$.

Periodically, we run the static algorithm using a carefully chosen subset of vertices as input. To select these subsets, we keep track of the recent degree of each vertex $v$: the number of edges incident to $v$ that were inserted since the last time $v$ was included as input to an instance of the static algorithm. Then, we choose the vertices of highest recent degree as input to the static algorithm, thus setting the recent degree of these vertices to zero. By repeatedly setting the recent degree of the highest recent degree vertices to zero, we obtain a bound on the maximum recent degree in the graph. Then we apply the dynamic algorithm for bounded degree graphs on only the edges that contribute to recent degrees.

We can further reduce the maximum recent degree in the graph by employing randomization: In addition to the vertices already chosen to participate in the static algorithm, we randomly select some vertices incident to newly inserted edges.

To obtain an upper bound on the maximum recent degree at all times, we model the changes in recent degree by an online 2-player balls and bins game. The game was first introduced in
1988 [51, 29] and has found a number of applications in the dynamic algorithms literature for obtaining worst-case guarantees [30, 24, 11, 71, 8, 57, 71, 72, 12, 31, 45]. To the best of our knowledge, our techniques are the first to demonstrate improved amortized guarantees using the game. We anticipate that this game will find additional applications in amortized algorithms as well as in translating offline strategies to online strategies.

The main technical content that remains are the details of each instance of the static algorithm: we have not specified when to run each instance, the precise subset of vertices to input, and which palette of colors to draw from. Understanding these details illuminates the key insight that allows us to improve the number of colors from the polynomial bound in [8] to polylogarithmic. We hierarchically bipartition the update sequence into \( \log_2 n \) levels of nested time intervals and at the end of each interval, we apply the static algorithm. We use a separate palette of colors for each level of intervals but for all instances of the static algorithm on the same level we use the same palette. Consequently, we need to ensure that vertices colored at the end of different intervals on the same level do not have conflicting colors. To do this, we ensure the structure of the intervals is such that if we color a vertex \( v \) at the end of an interval on some level \( L \), then before the end of the next interval on level \( L \), \( v \) has been recolored due to the end of an interval on a different level.

This partition of the update sequence is inspired by the offline algorithm of [43] for the dynamic edge orientation problem. Adapting their ideas to our setting requires overcoming two main hurdles: a) transitioning from graphs of bounded arboricity to general graphs, and b) transitioning from the offline setting to the online setting.

1.3.3 Overview of algorithm for low arboricity graphs

The proof of Theorem 1.5 is based on the following observation: the black-box static algorithm used in Theorem 1.3 can be made efficient if \( G \) is the class of arboricity \( \alpha \) graphs and we have access to a low out-degree orientation of the graph.

The proof of Theorem 1.6 concerns the LDS (defined in Section 1.2.2). The definition of the LDS is inspired by the following property of arboricity \( \alpha \) graphs: there exists an ordering of the vertices \( v_1, \ldots, v_n \) such that every vertex has at most \( 2\alpha \) neighbors that appear after it in the ordering [4]. Given such an ordering, consider the procedure of iteratively removing the vertices from the graph in order (or adding the vertices to the graph in reverse order) so that when each vertex is removed (or added) its degree with respect to the current graph is only \( 2\alpha \). This procedure has been a key ingredient in algorithms in a variety of settings including distributed algorithms [9], parallel algorithms [4], property testing [33], and social network analysis [44, 65, 28]. We are the first to devise a data structure that dynamically maintains (an approximate version of) this ordering.

The LDS is useful for maintaining a proper coloring of a graph because the graph induced by each layer of vertices has low degree. Thus, we can apply a dynamic algorithm for graphs of bounded maximum degree on the graph induced by each individual layer. Then, because there are not too many layers in total, we can use a disjoint palette of colors for each layer.

On the other hand, simply using a low out-degree orientation of the edges does not seem to suffice for solving dynamic coloring. In general, one shortfall of a low out-degree orientation is that it is an inherently local data structure; each vertex only keeps track of information about its immediate neighborhood. In contrast, the LDS maintains a global partition of the vertices into layers. Furthermore, the LDS is designed to store strictly more information than a bounded out-degree edge orientation; by orienting all edges in the LDS from lower to higher layers, we get a bounded out-degree edge orientation. We anticipate that the LDS could be useful for solving more dynamic problems for which a bounded out-degree edge orientation does not appear to suffice.
2 Algorithm for general graphs

In this section we prove Theorems 1.3, 1.4, and 1.5.

Theorem 2.1 (Restatement of Theorem 1.3). There is a fully dynamic deterministic algorithm for maintaining an \(O(\frac{C}{\beta} \log^3 n)\)-coloring with \(O(\beta)\) (amortized) recolorings per update step, for any \(\beta > 0\). Using randomization (against an oblivious adversary), the number of colors can be reduced to \(O(\frac{C}{\beta} \log^2 n (\log \log n + \log \beta))\) while achieving an expected bound of \(O(\beta)\) recolorings.

The algorithm is as follows. Periodically, we run the black-box static algorithm on a subset of vertices, to be specified later. At all times, each vertex \(v\) is assigned a color \(c_1\) by the black-box static algorithm (from the last time \(v\) was input to an instance of the static algorithm) and a color \(c_2\) by the black-box dynamic algorithm. The true color of \(v\) is defined by the pair \((c_1, c_2)\), so the total number of colors is the product of the number of colors used in each black-box algorithm. To specify the subsets of vertices taken as input to the static algorithm, we define a hierarchical partition of the update sequence. First, we describe this partition, then we describe how to apply the static algorithm, and then we describe how to apply the dynamic algorithm.

2.1 Partition of update sequence

We partition the update sequence (without knowing its contents) into a set of intervals as follows. An interval is said to be of length \(\ell\) if it contains \(\ell\) update steps. We partition the entire update sequence into intervals of length \(n\ell\) for some parameter \(\ell\) (which we will later set to \(\frac{\log n}{\beta}\)). We say that this set of intervals is on level 0. Next, for each \(i = 1, \ldots, \log_2 n\), the level-\(i\) intervals are obtained from the \(i - 1\)-level intervals by splitting each \(i - 1\) interval in two subintervals of equal length. Note that the intervals on level \(\log n\) are of length \(\ell\) and in general the intervals on level \(i\) are of length \(n\ell/2^i\).

It will be easier to work with these intervals if no two have the same ending point. So, for every set of intervals with the same endpoint, we remove all intervals except for the one with the lowest numbered level. The resulting set of intervals, shown in Figure 1 is the set of intervals that we work with in the algorithm.

![Figure 1: The set of intervals.](image)

2.2 Applying the black-box static algorithm

At the end of each interval, we apply the black-box static algorithm. For each interval \(I\), let \(A_I\) be the instance of the black-box algorithm that is executed at the end of interval \(I\). If \(I\) is an interval on level \(i\), we say that \(A_I\) is on level \(i\). For each level, we use a separate palette of \(C\) colors, and
all instances of the algorithm on the same level use the same palette of colors. In particular, if \(A_I\) is on level \(i\), it uses the \(C\) colors in the range from \(i \cdot C + 1\) to \((i + 1)C\).

We determine the input \(S\) to each \(A_I\) as follows. If \(I\) is on level 0, the input \(S\) is simply the entire vertex set. Otherwise, we decide the input based on the update sequence. For each vertex \(v\), we keep track of its recent degree, defined as the number of edges incident to \(v\) that were inserted since the last time \(v\) was included as input to an instance of the static algorithm. For each interval \(I\), we let \(v_I\) be the vertex of highest recent degree at the end of interval \(I\) (breaking ties arbitrarily). For the deterministic algorithm, the input \(S\) to each \(A_I\) is the set of vertices \(\{v_I'|I'\text{ is a subinterval of } I\}\) (where an interval is considered a subinterval of itself).

For the randomized algorithm, in addition to \(v_I\) we select another vertex \(u_I\) at the end of each interval \(I\). Specifically, we pick uniformly at random an edge insertion \((y, z)\) from the last \(\ell\) updates (if one exists) and then we let \(u_I\) be either \(y\) or \(z\), chosen at random. Then the input \(S\) to each \(A_I\) is the set of vertices \(\{v_I', u_I'\mid I'\text{ is a subinterval of } I\}\).

We note that each interval on level \(\log_2 n\) contains only 1 subinterval (itself), and generally, each interval on level \(i\) contains \(n/2^i\) subintervals. Thus, each \(A_I\) on level \(i\) takes \(O(n/2^i)\) vertices as input.

### 2.3 Applying the black-box dynamic algorithm

We apply the black-box dynamic algorithm on the graph with the full vertex set but only the edges that count towards the recent degree of both of its endpoints. Specifically, if \(G\) denotes the input dynamic graph then the dynamic graph \(G'\) that we input to the black-box dynamic algorithm is defined as follows. \(G'\) is initially the empty graph on the same vertex set as \(G\) and whenever there is an edge update to \(G\), the same edge is updated in \(G'\). Additionally, when a vertex \(v\) is included as input to the static algorithm, every edge incident to \(v\) is deleted from \(G'\).

To apply the black-box dynamic algorithm, we need to show that \(G'\) has bounded maximum degree. To do this, we apply an online 2-player balls and bins game. The game begins with \(N\) empty bins. The goal of Player 1 is to maximize the size of the largest bin and the goal of Player 2 is the opposite. At each step, the players each make a move according the following rules.

- Player 1 distributes at most \(k\) new balls to its choice of bins.
- Player 2 removes all of the balls from the largest bin (breaking ties arbitrarily).

**Theorem 2.2** ([29]). In the balls and bins game, every bin always contains \(O(k \log N)\) balls.

A randomized variant of the game will be useful in analyzing our randomized algorithm. In this variant, in addition to emptying the largest bin, Player 2 also chooses a number \(i\) from \([k]\) uniformly at random and empties the bin to which Player 1 added its \(i^{th}\) ball during its last turn. Player 1 is oblivious to the behavior of Player 2.

**Theorem 2.3** ([30]). In the randomized variant of the balls and bins game, in a game with \(N\) moves every bin always contains \(O(k \log \log N + k \log k)\) balls with high probability.

Recall that \(\ell\) is a parameter introduced in Section 2.1.

**Lemma 2.1.** In the deterministic algorithm the maximum degree of \(G'\) is always \(O(\ell \log n)\). In the randomized algorithm the maximum degree of \(G'\) is always \(O(\ell \log \log n + \ell \log \ell)\).

\(^1\) “High probability” means that for all \(c > 0\), there is an \(N\) such that the probability is at least \(1 - N^{-c}\)
Proof. We will argue that in the balls and bins game with \( N = n \) and \( k = 2\ell \), the number of balls in the largest bin is an upper bound for the maximum degree of \( G' \). Then, applying Theorems 2.2 and 2.3 completes the proof.

We first note that by construction, the degree of each vertex \( v \) in \( G' \) is at most the recent degree of \( v \) so it suffices to bound recent degree. (In particular, the recent degree of \( v \) could be larger because it counts edges to vertices that have recently been included as input to the static algorithm.)

The only way for the recent degree of a vertex \( v \) to increase is due to the insertion of an edge incident to \( v \). On the other hand, the recent degree of a vertex \( v \) decreases when a) an edge incident to \( v \) is deleted causing its recent degree to decrement, and b) \( v \) is included as input to the static algorithm causing its recent degree to be set to 0.

We consider the special case of the balls and bins game where for each edge insertion \((u, v)\), Player 1 places one ball in the bin corresponding to \( u \) and one ball in the bin corresponding to \( v \). Then, when each interval ends (which happens once every \( \ell \) updates), Player 2 moves. Recall that at this point the recent degree of \( v \) is set to 0 (and in the randomized algorithm, so is that of \( u \)). It is clear from this description that the deterministic and randomized balls and bins games parallel all of the increases and some of the decreases in recent degree in our deterministic and randomized algorithms, respectively. From here, it is easy to verify that the number of balls in the largest bin is an upper bound for the maximum recent degree in both the deterministic and randomized settings. For the sake of completeness, we prove this fact formally in the appendix.

2.4 Correctness

We will show that our algorithm produces a proper coloring after every update. Recall that the color of each vertex \( v \) is defined by the pair of colors \((c_1, c_2)\) where \( c_1 \) is the color assigned to \( v \) by the black-box static algorithm and \( c_2 \) is the color assigned to \( v \) by the black-box dynamic algorithm.

Consider an edge \((u, v)\) in the graph at a fixed point in time. We will show that our algorithm assigns different colors to \( u \) and \( v \). If \((u, v)\) is included in the input to the black-box dynamic algorithm (i.e. if \((u, v)\) is in \( G' \)), then its two endpoints are assigned different colors by this algorithm, and are thus assigned different colors by the overall algorithm.

Otherwise, by the definition of the input to the black-box dynamic algorithm, after the edge \((u, v)\) was last inserted at least one of \( u \) or \( v \) was included as input to the static algorithm. We claim that \( u \) and \( v \) are assigned different colors by the static algorithm. If \( u \) and \( v \) were last colored by the same instance \( A_I \) of the static algorithm, then \( A_I \) was executed after the edge \((u, v)\) was inserted (by assumption). Thus, the edge \((u, v)\) was included as input to \( A_I \), causing \( u \) and \( v \) to be assigned different colors. If \( u \) and \( v \) were last colored by instances of the static algorithm on different levels, then they are assigned different colors since each level uses a separate palette of colors.

The only remaining case is that \( u \) and \( v \) were last included as input to the static algorithm by two different instances of the static algorithm on the same level \( i \). We will show that this is impossible. This case is the crux of the correctness argument and the reason that we define the intervals in precisely the way that we do. It cannot be the case that \( i = 0 \) since every vertex is recolored at the end of every interval on level 0. Suppose by way of contradiction that \( u \) was most recently colored by \( A_I \) (the instance of the static algorithm at the end of interval \( I \)) and \( v \) was most recently colored by \( A_{I'} \) where interval \( I \) comes before interval \( I' \) and both are on level \( i \). We will show that between the end of interval \( I \) and the end of interval \( I' \), \( u \) is recolored by an instance of the static algorithm on a level \( j < i \) (a contradiction). By the construction of the intervals (see Figure 1), between the ending points of \( I \) and \( I' \) is the end of an interval \( I'' \) on a level \( j < i \).
that contains interval $I$ as a subinterval. By the definition of the algorithm, every vertex that is included as input to $A_I$ is also included as input to $A_{I'}$. Thus, $u$ is recolored on level $j$ before $A_I$ was executed, a contradiction.

2.5 Analysis

Static algorithm.

*Number of colors.* The static algorithm uses $C$ colors per level and there are $O(\log n)$ levels, for a total of $O(C \log n)$ colors.

*Number of recolorings.* In the deterministic algorithm, each interval $I$ has an associated vertex $v_I$ and in the randomized algorithm, each interval has two associated vertices $v_I$ and $u_I$. Each such vertex is included as input to the static algorithm for all superintervals of $I$. Since there are $O(\log n)$ levels and each level consists of a set of disjoint intervals, each interval has at most $O(\log n)$ superintervals. Thus, for each interval $I$, $v_I$ and $u_I$ are included as input to $O(\log n)$ instances of the static algorithm. Every interval ends after a multiple of $\ell$ updates so the number of recolorings is amortized $O\left(\frac{\log n}{\ell}\right)$.

Dynamic algorithm.

*Number of colors.* Given a dynamic graph of maximum degree $\Delta$, the black-box dynamic algorithm maintains an $O(\Delta)$-coloring. By Lemma 2.1, $G'$ (the graph input to the black-box dynamic algorithm) has maximum degree $O(\ell \log n)$ in the deterministic setting and $O(\ell \log n + \ell \log \ell)$ in the randomized setting. The randomized bound is with high probability and in the low probability event that the maximum degree exceeds the bound, we will immediately end all intervals, thereby recoloring the entire graph. Thus, the runtime bound is probabilistic but the bound on the number of colors is not.

*Number of recolorings.* Using the following simple greedy algorithm as our black-box dynamic algorithm, we get a constant number of recolorings per update. When an edge is added between two vertices of the same color, simply scan the neighborhood of one of them and recolor it with a non-conflicting color. If the maximum degree of the graph is $\Delta$, this algorithm produces a $\Delta + 1$ coloring.

Combining the static and dynamic algorithms

*Number of colors.* Recall that if a vertex $v$ is assigned color $c_1$ by the black-box static algorithm and color $c_2$ by the black-box dynamic algorithm, then our algorithm assigns $v$ the color $(c_1, c_2)$. So the number of colors is the product of the number of colors used in each black-box algorithm, which is $O(C\ell \log^2 n)$ for the deterministic algorithm and $O(C\ell \log n \log \log n \log \ell)$ for the randomized algorithm.

*Number of recolorings.* The total number of recolorings is the sum of the number of recolorings performed in each of the black-box algorithms, which is $O\left(\frac{\log n}{\ell}\right)$.

Setting $\ell = \frac{\log n}{\beta}$ completes the proof.

2.6 Time bound

**Theorem 2.4** (Restatement of Theorem 1.4). The randomized algorithm from Theorem 1.3 has expected amortized update time $O\left(\frac{\beta}{n \log n} \sum_{i=0}^{\log n} 2^i T(n/2^i)\right)$ and the deterministic algorithm from Theorem 1.3 has the same amortized update time with an additional additive factor of $T'(\frac{\log^2 n}{\beta}, n)$.

**Proof.**
Static algorithm  We defined \( v_I \) as the vertex of highest recent degree at the end of each interval \( I \). Although there are data structures to find \( v_I \) in constant time, it suffices for the analysis of this algorithm to spend \( O(\log n) \) time to find each \( v_I \). An interval ends once every \( \ell \) updates so the amortized time is \( \frac{\log n}{\ell} \).

At the end of every interval in the randomized algorithm, \( u_I \) is chosen randomly from a distribution over only the most recent \( \ell \) updates, so this takes constant amortized time. Then, after finding \( v_I \) and \( u_I \), determining the input to each \( A_I \) takes time linear in the size of the input to \( A_I \).

We now analyze the time to run all of the instances of the static algorithm. Recall that given a graph \( G \) and a subset \( S \) of the vertices, the static algorithm computes \( G[S] \) and a \( C \)-coloring of \( G[S] \) in time \( T(|S|) \). Consider a level 0 interval \( I \). At the end of interval \( I \), we run the algorithm on the entire vertex set, which takes time \( T(n) \). By construction of the intervals, each interval on level \( i \) is of length \( n\ell/2^i \) and contains \( n/2^i \) subintervals. Thus, the static algorithm at the end of each interval on level \( i \) takes \( n/2^i \) vertices as input so each such static algorithm runs in time \( T(n/2^i) \). For all levels \( i \), there are \( 2^i \) subintervals of \( I \) on level \( i \). Thus, it takes total time \( 2^i T(n/2^i) \) to run all instances of the static algorithm on level \( i \) that are executed during interval \( I \). Therefore, the total time to run all instances of the static algorithm that are executed during interval \( I \) (including the one at the end of interval \( I \)) is \( \sum_{i=0}^{\log n} 2^i T(n/2^i) \). Interval \( I \) is of length \( n\ell \) so the amortized time is \( \frac{1}{n\ell} \sum_{i=0}^{\log n} 2^i T(n/2^i) \).

Dynamic algorithm  We note that the number of updates to \( G'[u] \) is at most twice the number of updates to \( G \) since for each edge \((u,v)\) inserted to \( G \), \((u,v)\) is inserted to and deleted from \( G'[u] \) at most once. Thus, maintaining \( G'[u] \) takes constant amortized time.

Recall that the black-box dynamic algorithm has amortized expected update time \( O(1) \) in the randomized setting and amortized update time \( T'(\Delta, n) \leq O(\text{polylog}(\Delta)) \) in the deterministic setting.

In the randomized algorithm, when the maximum degree of \( G'[u] \) exceeds the stated bound, we immediately end all intervals, thereby recoloring the entire graph. For large enough \( n \), this happens with probability less than \( 1/n^2 \). Each time this happens, we pay an extra \( T(n) \) time to recolor the entire graph. Thus, this takes amortized time \( T(n)/n^2 \) in expectation.

Combining the static and dynamic algorithms  The total amortized update time is the sum of the amortized running times of each of the black-box algorithms, which is \( \frac{1}{n\ell} \sum_{i=0}^{\log n} 2^i T(n/2^i) \) in expectation for the randomized algorithm, and with an additional additive factor of \( T'(\frac{\log n}{\ell}, n) \) for the deterministic algorithm. (This expression subsumes the additive factor of \( T(n)/n^2 \) from the randomized algorithm assuming \( \ell \leq n \).)

Setting \( \ell = \frac{\log n}{\beta} \) completes the proof. \( \square \)

2.7 De-amortizing the number of recolorings

We note that our algorithm can be easily modified to achieve the same trade-off between number of colors and number of recolorings in the worst-case setting as in the amortized setting. This extension does not give a worst-case bound on the running time, only the number of recolorings. Our analysis of the amortized algorithm already uses a trivial black-box dynamic algorithm that performs a constant number of recolorings per update in the worst case. We need to show that the static algorithms can be applied with a worst-case number of recolorings per update.
The worst-case algorithm works as follows. Since we are not concerned with running time, we run our amortized algorithm in the background (without performing any actual colorings). At the end of each interval $I$, our worst-case algorithm immediately recolors $v_I$ and $u_I$ to the color assigned by $A_I$. We delay the recoloring of the rest of the vertices in the input of $A_I$. It is important to recolor $v_I$ and $u_I$ immediately because otherwise the balls and bins game does not apply.

From the proof of correctness of the amortized algorithm (Section 2.4), if $(u, v)$ is an edge and $u$ and $v$ were last recolored according to two different instances of the static algorithm on different levels, or the same instance, then $u$ and $v$ are assigned different colors. The only remaining case is if $u$ and $v$ were last recolored by different instances of the static algorithm on the same level. The proof that this is impossible from Section 2.4 holds if the following property holds: for every pair of adjacent intervals $I$ and $I'$ on the same level $i$ where $I$ comes before $I'$, all vertices colored by $A_I$ are recolored by some $A_{I''}$ on a level $j 
eq i$ before any vertices are colored by $A_{I'}$.

We design the worst-case algorithm to ensure that this property holds. It suffices to take all of the at most $n/2^i$ vertices input to $A_I$ and recoloring them to the color assigned by $A_{I''}$ throughout the course of interval $I'$. For ease of notation, we say that these colorings are performed by interval $I'$. We note that by the construction of the intervals, interval $I''$ ends when interval $I'$ begins so when interval $I'$ begins, it already has full information about all of the recolorings it will perform. Furthermore, when interval $I'$ performs a recoloring according to $A_{I''}$, the interval following $I''$ on level $j$ has not ended (or even started) yet so these recolorings cannot conflict with other vertices colored using the level $j$ color palette.

Interval $I'$ is of length $nl/2^i$ and performs at most $n/2^i$ recolorings, so on average $I'$ performs at most one recoloring every $\ell$ updates. To achieve $\frac{\log n}{\ell}$ recolorings per update in the worst case, we need to only allow intervals on a $1/\ell$ fraction of the levels to perform recolorings following each update. One way to do this is only allow intervals on level $i$ to perform recolorings after the $k^{th}$ update if $k \equiv i \mod \ell$.

3 Algorithms for low arboricity graphs

In this section we begin by proving Theorem 1.5. The proof follows from a combination of dynamically maintaining a bounded out-degree edge orientation and applying the algorithm from Section 2.

Our main goal in this section is to improve upon Theorem 1.5 by proving Theorem 1.6. To this end we refine the tool of dynamic edge orientations by introducing a new layered data structure.

**Theorem 3.1 (Restatement of Theorem 1.5).** There is a fully dynamic deterministic algorithm for graphs of arboricity $\alpha$ that maintains an $O((\frac{\alpha}{\beta})^2 \log^4 n)$-coloring in amortized $T(n,\frac{\alpha \log^3 n}{\beta^2}, n) + O(\beta)$ time per update for any $\beta > 0$. Using randomization (against an oblivious adversary), the number of colors can be reduced by a factor of $\tilde{O}(\log n)$ and the expected amortized update time becomes $O(\beta)$.

**Proof.** We run the dynamic edge orientation algorithm of [43], which maintains an $O(\beta' \alpha)$ out-degree orientation in amortized $O(\frac{\log n}{\beta'})$ time, for all $\beta' > 1$. Given this orientation, for any subset $S$ of the vertices in the current graph $G$, we can compute $G[S]$ and a $2\alpha$-coloring of $G[S]$ in time $O(n\beta \alpha)$. We compute $G[S]$ by simply scanning the out-neighborhood of every vertex in $S$ and including the edges whose other endpoint is also in $S$. Every edge between a pair of vertices $u, v \in S$ is oriented away from either $u$ or $v$ so this algorithm scans every edge in $G[S]$.

Every subgraph of an arboricity $\alpha$ graph also has arboricity $\alpha$, in particular $G[S]$.

We color $G[S]$ by considering an ordering of the vertices $v_1, v_2, \ldots$ in $S$ such that every vertex has at most $2\alpha$ neighbors that appear after it in the ordering. Such an ordering exists and can be computed
in time $O(|S| \alpha)$ [4]. We imagine starting with an empty graph iteratively adding the vertices in
$S$ to the graph in reverse order. When each vertex $v$ is added, $v$ has at most $2\alpha$ neighbors in the
current graph. Using a palette of $2\alpha + 1$ colors, we can always color $v$ a color different from all of its
neighbors in the current graph.

Applying Theorem [2.4] with $T(n) = O(n\beta'\alpha)$ and parameter $\beta''$, we see that the algorithm
from Theorem [2.1] runs in time $O(\beta''\beta'\alpha)$ per update in expectation in the randomized setting and
$T'((\log^{2} n)/\beta''), n) + O(\beta''\beta'\alpha)$ per update in the deterministic setting. The additional time for maintaining
the edge orientation is $O((\log n)/\beta'')$. Setting $\beta' = \sqrt{\log n/(\alpha\beta'')}$, the running time is $O(\sqrt{\alpha/\beta''}\log n)$
(with an additional additive factor of $T'( (\log^{2} n)/\beta''), n)$ in the deterministic setting).

Applying Theorem [2.1] with $C = O(\alpha)$, the number of colors is $O(\frac{\alpha}{\beta''} \log^{3} n)$ for the deter-
mimistic algorithm and $O(\frac{\alpha}{\beta''} \log^{2} n(\log \log n + \log \beta''))$ for the randomized algorithm. Setting
$\beta'' = \beta^{2}/(\alpha \log n)$ completes the proof.

For the remainder of this section we prove Theorem [1.6].

**Theorem 3.2** (Restatement of Theorem [1.6]). There is a fully dynamic deterministic algorithm
for graphs of arboricity $\alpha$ that maintains an $O(\alpha \log^{2} n)$-coloring in amortized $O(1)$ time.

Given a partition of the vertices of a graph into layers $L_{1}, L_{2}, \ldots$, for all vertices $v$ let $d_{up}(v)$
(the up-degree of $v$) be the number of neighbors of $v$ in layers equal to or higher than that of $v$.

**Definition 3.1.** Given a dynamic graph $G$ of arboricity $\alpha$, a *layered data structure* (LDS) with
parameters $k$ and $\Delta$ is a partition of the vertices into $k$ layers $L_{1}, \ldots, L_{k}$ so that for all vertices $v,$
$d_{up}(v) \leq \Delta$.

The bulk of the proof of Theorem 3.2 is to prove Theorem [1.7].

**Theorem 3.3** (Restatement of Theorem [1.7]). Let $A''$ be an algorithm for arboricity $\alpha$ graphs that
maintains an orientation of the edges with out-degree at most $D$ that performs amortized $F(n)$ flips
per update. Then there is an algorithm to maintain an LDS for a fully dynamic graph of arboricity
$\alpha$ with $k = O(\log n)$ and $\Delta = O(D + \alpha \log n)$ in amortized deterministic time $O(F(n))$.

### 3.1 Proof overview

The idea of the algorithm is essentially to move vertices to new layers when the required properties
of the data structure are violated. Roughly, when there is a vertex $v$ with $d_{up}(v) \geq \Delta$ we move $v$
to a higher layer so that $d_{up}(v)$ decreases to $O(\alpha)$. Additionally, to control the number of layers,
whenever a vertex $v$ has up-degree less than $d = O(\alpha)$ and $v$ can be moved to a lower layer while
maintaining up-degree less than $d$, we move $v$ to a lower layer. The fact that $d$ and $\Delta$ differ by a
logarithmic factor ensures that vertices don’t move between layers too often which is essential for
bounding the running time.

To help with the running time analysis, we maintain two dynamic orientations of the edges:
one is defined by the algorithm $A''$ and the other is maintained by our algorithm. The orientation
maintained by our algorithm has the property that all edges with endpoints in different layers are
oriented toward the higher layer. We compare the number of edge flips in the orientation defined
by our algorithm to the number of edge flips in the orientation algorithm $A''$ using a potential function: $\phi(i)$ = the number of edges oriented in opposite directions in the two algorithms. This
potential function is also used in [22].

The main idea of the analysis is to observe how $\phi$ changes in response to vertices moving between
levels. We claim that when we move a vertex to a higher level, $\phi$ decreases substantially for the
following reason. Our algorithm is defined so that we only move a vertex to a higher layer if its up-degree decreases substantially as a result. Because our algorithm orients edges from lower to higher layers, when we move a vertex \( v \) to a higher layer many edges incident to \( v \) are flipped towards \( v \). Then because \( A'' \) maintains an orientation of low out-degree, many of these edges flipped towards \( v \) end up oriented in the same direction in the two orientations. Thus, \( \phi \) decreases substantially as a result of \( v \) moving to a higher layer. On the other hand, when a vertex moves to a lower layer, \( \phi \) might increase. The idea of the argument is to use the substantial decreases in \( \phi \) that result from moving vertices to higher layers to pay for the increases in \( \phi \) that result from moving vertices to lower layers.

3.2 Invariants

In this section we introduce four invariants that together imply that \( d_{up}(v) \leq \Delta \) and \( k = O(\log n) \).

We maintain two dynamic orientations of the edges in the graph, one defined by our algorithm and the other defined by the algorithm \( A'' \). Unless otherwise stated, when we refer to an orientation, we mean the orientation defined by our algorithm.

For ease of notation, let \( d = 4\alpha \) and let \( d' = \Delta / 2 \).

We define the following for each vertex \( v \):

- \( L(v) \) is the layer containing \( v \).
- \( L_{max}(v) \) is the lowest layer for which if \( v \) were in this layer, \( d_{up}(v) \) would be at most \( d \).
- \( d^+(v) \) is the out-degree of \( v \).
- \( d^-_L(v) \) is the in-degree of \( v \) from neighbors in \( L(v) \).

Orientation invariants

Invariant 1 defines how edges are oriented between layers and is useful for analyzing the update time of the algorithm, as outlined in Section 3.1.

**Invariant 1.** All edges with endpoints in different layers are oriented towards the vertex in the higher layer.

The next two invariants bound \( d^+(v) \) and \( d^-_L(v) \), which helps to bound \( d_{up}(v) \).

**Invariant 2.** For all vertices \( v \), \( d^+(v) \leq d' \).

**Invariant 3.** For all vertices \( v \), \( d^-_L(v) \leq d' \).

**Claim 1.** Invariants 1, 2, 3 together imply that \( d_{up}(v) \leq 2d' = \Delta \).

**Proof.** By Invariant 1 for all vertices \( v \), every neighbor of \( v \) in a layer equal to or higher than \( L(v) \) is either an out-neighbor of \( v \) or an in-neighbor of \( v \) in \( L(v) \), so \( d_{up}(v) = d^+(v) + d^-_L(v) \). Then by Invariants 2 and 3 \( d^+(v) + d^-_L(v) \leq 2d' \).

**Number of layers invariant**

Invariant 4 serves to bound the number of layers \( k \).

**Invariant 4.** For all vertices \( v \), \( L(v) \leq L_{max}(v) \).

**Claim 2.** Invariant 4 implies that \( k = O(\log n) \).
Proof. First we observe that under Invariant 1, all vertices of degree at most \( d \) are in \( L_1 \). Now, consider removing all vertices in \( L_1 \) from the graph. In the remaining graph, all vertices of degree at most \( d \) are in layer \( L_2 \). More generally, after removing all vertices in layers 1 through \( i \) for any \( i \), all vertices of degree at most \( d \) must be in layer \( L_{i+1} \).

The total number of edges in a graph of arboricity \( \alpha \) is less than \( \alpha n \). So at least a \( (1 - 2\alpha/d) \) fraction of the vertices have degree at most \( d \). Any subgraph of an arboricity \( \alpha \) graph also has arboricity \( \alpha \) so after the vertices in any given layer are removed, the graph still has arboricity \( \alpha \). Thus, after removing the vertices in layers 1 through \( i \) for any \( i \), at least a \( (1 - 2\alpha/d) \) fraction of the remaining vertices are in \( L_{i+1} \). Therefore, the number \( k \) of layers total is at most \( \log \frac{d}{2\alpha} n = O(\log n) \).

3.3 Algorithm

The idea of the algorithm is essentially to move vertices to new layers when the required properties of the data structure are violated. We define two recursive procedures \( \text{Rise} \) and \( \text{Drop} \) which move vertices to higher and lower layers respectively. In particular, when a vertex \( v \) violates Invariant 2 or 3 (i.e. either \( d^+(v) > d' \) or \( d^-(v) > d' \)), we call the procedure \( \text{Rise}(v) \) which moves \( v \) up to the layer \( L_{\text{max}}(v) \). The movement of \( v \) to a new higher layer may increase the up-degree of some neighbors \( u \) of \( v \) causing \( u \) to violate Invariant 2 or 3, in which case we recursively call \( \text{Rise}(u) \). On the other hand, when a vertex \( v \) violates Invariant 4 (i.e. \( L_{\text{max}}(v) < L(v) \)), we call the procedure \( \text{Drop}(v) \) which moves \( v \) down to the layer \( L_{\text{max}}(v) \). The movement of \( v \) to a new lower layer may decrease \( L_{\text{max}}(u) \) for some neighbors \( u \) of \( v \) causing \( u \) to violate Invariant 4, in which case we recursively call \( \text{Drop}(u) \). See Algorithm 1 for the pseudocode.

3.4 Correctness

We will show that the four invariants are satisfied. By Claims 1 and 2 this implies that \( d_{up}(v) \leq 2d' = \Delta \) and \( k = O(\log n) \).

Invariant 1 is satisfied at all times because whenever a vertex changes layer all of its incident edges that are oriented towards the lower layer are immediately flipped.

We observe the following useful property of the algorithm:

Lemma 3.1.

1. Right after any vertex \( v \) is moved to a new layer \( L_i \), \( d_{up}(v) \leq d \).

2. While \( v \) remains in \( L_i \), the only way for \( d^-(v) \) to increase is by the insertion of an edge incident to \( v \).

Proof.

1. Whenever any vertex \( v \) is moved to a new layer (either by \( \text{Rise} \) or \( \text{Drop} \)), it is moved to the layer \( L_{\text{max}}(v) \). By the definition of \( L_{\text{max}}(v) \), we have \( d_{up}(v) \leq d \).

2. When any vertex \( v \) moves to a higher layer, all of \( v \)'s incident edges within its new layer are flipped towards \( v \). When any vertex \( v \) moves to a lower layer, all of \( v \)'s incident edges within its new layer are already oriented towards \( v \) by Invariant 1 and they are not flipped. That is, right after \( v \) is moved to a new layer (in either direction), all of its incident edges within its new layer are oriented towards \( v \). Thus, for all vertices \( u \neq v \), the movement of \( v \) to a new layer cannot cause \( d^-(u) \) to increase. Then since all edge flips are triggered by a vertex changing layers, the only way for \( d^-(v) \) to increase is by the insertion of an edge incident to \( v \).
Algorithm 1

procedure Insert \((u, v)\)
  add edge \((u, v)\)
  if \(u\) and \(v\) are in different layers then
    orient the edge towards the vertex in the higher layer
  else \((u\) and \(v\) are in the same layer):\n    orient the edge arbitrarily
  if \(d^+(u) > d'\) or \(d^-_L(u) > d'\) then \(\text{Rise}(u)\)
  if \(d^+(v) > d'\) or \(d^-_L(v) > d'\) then \(\text{Rise}(u)\)

procedure Delete \((u, v)\)
  remove edge \((u, v)\)
  if \(L_{\text{max}}(u) < L(u)\) then \(\text{Drop}(u)\)
  if \(L_{\text{max}}(v) < L(v)\) then \(\text{Drop}(v)\)

procedure Rise \((v)\)
  \(L_{\text{old}} \leftarrow L(v)\)
  move \(v\) up to layer \(L_{\text{max}}(v)\)
  \(S \leftarrow\) the set of out-neighbors of \(v\) in a layer between \(L_{\text{old}}\) and \(L_{\text{max}}(v)\) inclusive
  for each \(u \in S\) do
    flip edge \((u, v)\) towards \(v\)
  for each \(u \in S\) do
    if \(d^+(u) > d'\) then \(\text{Rise}(u)\)

procedure Drop \((v)\)
  \(L_{\text{old}} \leftarrow L(v)\)
  move \(v\) down to layer \(L_{\text{max}}(v)\)
  \(S \leftarrow\) the set of in-neighbors of \(v\) in any layer above \(L_{\text{max}}(v)\) and at most \(L_{\text{old}}\)
  for each \(u \in S\) do
    flip edge \((u, v)\) away from \(v\)
  \(S^+ \leftarrow\) the set of all neighbors of \(v\) in any layer above \(L_{\text{max}}(v)\) and at most \(L_{\text{old}} + 1\)
  for each \(u \in S^+\) do
    if \(L_{\text{max}}(u) < L(u)\) then \(\text{Drop}(u)\)

Now we show that Invariants 2-4 are satisfied after each edge update is processed.

Invariant 2 is violated when \(d^+(v) > d'\). This could happen as a result of a) insertion of an edge, or b) movement of a vertex \(u\) to a higher layer, which could cause \(u\)'s neighbors to violate the invariant. In both of these cases, the algorithm calls \(\text{Rise}\) on all violating vertices.

Invariant 3 is violated when \(d^-_L(v) > d'\). By Lemma 3.1, this can only happen following the insertion of an edge. In this case, the algorithm calls \(\text{Rise}\) on all violating vertices.

Invariant 4 is violated when \(L(v) > L_{\text{max}}(v)\). This could happen as a result of a) deletion of an edge, or b) movement of a vertex \(u\) from \(L_i\) to a lower layer \(L_j\), which could cause \(u\)'s neighbors in layers from \(L_{j+1}\) to \(L_{i+1}\) to violate the invariant. In both of these cases, the algorithm calls \(\text{Drop}\) on all violating vertices.
3.5 Bounding the number of edge flips

The first step towards getting a bound on the update time is to get a bound on the number of edge flips that the algorithm performs. We will show that the amortized number of edge flips per update is $O(F(n))$ (Lemma 3.7).

We choose $d' = 8(dk + D)$, so $d' = O(D + \alpha \log n)$, as required.

Let $G = (G_0, G_1, \ldots, G_M)$ be the sequence of graphs with orientation defined by our algorithm and let $G^A = (G^A_0, G^A_1, \ldots, G^A_M)$ be the sequence of graphs with orientation defined by the algorithm $A'$. That is, for all $i$, the underlying undirected graphs corresponding to $G_i$ and $G^A_i$ are identical but their orientations may differ. Given $i$, we say an edge in $G_i$ is bad if it is oriented in the opposite direction in $G_i$ and $G^A_i$. We define a potential function:

$$\phi(i) = \text{the number of bad edges.}$$

We say that a call to $\text{Rise}$ is heavy if it triggers at least $d'/2$ edge flips, ignoring recursive calls. Otherwise, we say that a call to $\text{Rise}$ is light.

**Lemma 3.2.** Every light call to $\text{Rise}$ is due to a violation of Invariant [3].

**Proof.** Suppose otherwise; that is, suppose that a light call to $\text{Rise}$ is triggered by a violation of Invariant [2] In this case, right before the call to $\text{Rise}$, $d^+(v) > d'$. By Lemma 3.1 after $v$ is moved to a new layer, $d^+(v) \leq d$. Thus, the call to $\text{Rise}$ triggers at least $d' - d > d'/2$ edge flips so it must be heavy.

We define the following parameters.

- $M = \text{the total number of edge updates}$
- $l = \text{the total number of light calls to } \text{Rise}$
- $h = \text{the total number of heavy calls to } \text{Rise}$
- $r = \text{the total number of calls to } \text{Rise}; \text{ so } r = h + l$
- $p = \text{the total number levels that vertices move down (due to calls to } \text{Drop})$
- $f = \text{the total number of flips}$
- $f_l = \text{the total number of flips triggered by light calls to } \text{Rise}$
- $f_h = \text{the total number of flips triggered by heavy calls to } \text{Rise}$
- $f_p = \text{the total number of flips triggered by calls to } \text{Drop}$

**Observation 3.4.** Using the above parameters, it is immediate to bound the total increase in $\phi$ due to the following events:

- Edge updates: $\Delta(\phi) \leq M$.
- Edge reorientations in $G^A$: $\Delta(\phi) \leq F(n)M$.
- Light calls to $\text{Rise}$: $\Delta(\phi) \leq f_l$.
- Calls to $\text{Drop}$: $\Delta(\phi) \leq f_p$.

The only event missing from the above list is heavy calls to $\text{Rise}$. We will now argue that $\phi$ decreases substantially as a result of this event. Then, we will use these substantial decreases in $\phi$ to pay for the increases in $\phi$ from the other events.

**Lemma 3.3.** The total decrease in $\phi$ over the whole computation triggered by heavy calls to $\text{Rise}$ is at least $f_h/2$. 

17
Proof. Consider a heavy call to \textsc{Rise} on vertex \( v \). Let \( S \) be the set of edges flipped by this call to \textsc{Rise}. All of the edges in \( S \) are flipped towards \( v \). Before these flips happen, \( v \) has out-degree at least \( d'/2 \) by the definition of a heavy call to \textsc{Rise}. By Lemma 3.1, after the edges in \( S \) are flipped, \( d^+(v) \leq d \). Thus, the number of edges flipped is at least \( d'/2 - d \). We will use this fact at the end of the proof.

Before the edges in \( S \) are flipped, all are out-going of \( v \). Then since \( v \) has out-degree at most \( D \) in all \( G^i_A \), at least \(|S| - D \) of these edges are bad before they are flipped. For the same reason, after these flips at most \( D \) of the flipped edges are bad. Thus, \( \phi \) decreases by at least \(|S| - 2D \) as a result of flipping the edges in \( S \). Therefore, the total decrease in \( \phi \) over the whole computation due to heavy calls to \textsc{Rise} is at least the sum of \(|S| - 2D \) over all heavy calls to \textsc{Rise}, which is at least

\[
f_h - 2Dh \geq f_h - 4Df_h/d' \text{ since each heavy call to \textsc{Rise} flips at least } d'/2 \text{ edges}
\]

\[
\geq \frac{f_h}{2} \text{ by choice of } d'.
\]

We derive bounds for \( f_l \), \( f_p \), and \( f_h \), in the following lemmas.

Lemma 3.4. \( l(d' - d) \leq M \). \( f_l \leq M \).

Proof. By Lemma 3.2 every light call to \textsc{Rise} is triggered by \( d' - d \) for some \( v \). By Lemma 3.1 right after any vertex \( v \) is moved to a new layer, \( d'_L(v) \leq d \), and while \( v \) remains in this layer, the only way for \( d'_L(v) \) to increase is by the insertion of an edge incident to \( v \). Before a light call to \textsc{Rise}(\( v \)), \( d'_L(v) \) must increase to at least \( d' \). Thus, every light call to \textsc{Rise} must be preceded by \( d' - d \) insertions of edges incident to \( v \). Conversely, the insertion of an edge can only increase the in-degree of one vertex. Thus, \( l(d' - d) \leq M \). Each light call to \textsc{Rise} flips at most \( d'/2 \) edges so, \( f_l \leq \frac{d'}{2} \). Combining these two equations we have, \( f_l \leq \frac{Md'}{2(d' - d)} \leq M \) by choice of \( d' \).

Lemma 3.5. \( f_p \leq dp \leq \frac{f_h}{4} + M \).

Proof. By Lemma 3.1 right after the call to \textsc{Drop}(\( v \)), \( d_{up}(v) \leq d \). Then since \textsc{Drop}(\( v \)) only flips edges incident to \( v \) whose other endpoint is in a layer above \( v \), any call to \textsc{Drop}(\( v \)) flips at most \( d \) edges. Thus, \( f_p \leq dp \). Furthermore, every call to \textsc{Rise}(\( v \)) moves \( v \) up by at most \( k \) layers, so \( p \leq rk \). Thus we have,

\[
f_p \leq dk \\leq dk \left( \frac{2f_h}{d'} + l \right) \text{ since each heavy call to \textsc{Rise} flips at least } d'/2 \text{ edges}
\]

\[
\leq dk \left( \frac{2f_h}{d'} + \frac{M}{d' - d} \right) \text{ by Lemma 3.4}
\]

\[
\leq f_h/4 + M \text{ by choice of } d'
\]

Lemma 3.6. \( f_h = O(F(n)M) \)
Proof. We use the potential function: \( \phi \) is initially 0 and is never negative so the total increase in \( \phi \) must be at least the total decrease in \( \phi \). Therefore, by Observation 3.4 and Lemma 3.3, \( M + F(n)M + f_i + f_p \geq f_h/2 \). Then, by Lemmas 3.4 and 3.5 we have \( M + F(n)M + M + f_h/4 + M \geq f_h/2 \), which completes the proof.

\[ \text{Lemma 3.7.} \text{ The amortized number of flips per update is } O(F(n)). \]

\text{Proof.}

\[ f = f_p + f_i + f_h \]
\[ \leq f_h/4 + M + M + f_h \text{ by Lemmas 3.4 and 3.5} \]
\[ = O(F(n)M) \text{ by Lemma 3.3} \]

3.6 Update time bound

In this section, we will show that our algorithm runs in amortized time \( O(F(n)) \) per update.

Each vertex \( v \) keeps track of the following information:

- \( L(v) \)
- \( d^+(v) \) and the set \( N^+(v) \) of \( v \)'s out-neighbors
- \( d^-_L(v) \) and the set \( N^-_L(v) \) of \( v \)'s in-neighbors in \( L(v) \).
- For each layer \( L_i \) lower than \( L(v) \), the set \( N_i(v) \) of \( v \)'s neighbors in that layer and the number \( d_i(v) \) of them.

\text{Lemma 3.8.} \text{ INSERT}(u,v) \text{ runs in time } O(1) \text{ (ignoring calls to } \text{Rise}). \]

\text{Proof.} In \text{INSERT}(u,v) we update the stored information of both \( u \) and \( v \) in constant time simply by incrementing the appropriate counters and adding to the appropriate sets. Then we compute whether either \( d^+(u) \) or \( d^-_L(u) \) exceeds \( d' \) and the same for \( v \). These comparisons take \( O(1) \) time.

\text{Lemma 3.9.} \text{ Computing whether } L_{\text{max}}(v) < L(v) \text{ takes time } O(1). \]

\text{Proof.} Then, \( L_{\text{max}}(v) < L(v) \) if and only if the degree of \( v \) to vertices in layers at least as high as the layer just below \( L(v) \) is at most \( d \) i.e. if \( d^+(v) + d^-_L(v) + d_{i-1}(v) \leq d \) where \( i \) is such that \( L_i = L(v) \). This comparison takes \( O(1) \) time.

\text{Lemma 3.10.} \text{ DELETE}(u,v) \text{ runs in time } O(1) \text{ (ignoring calls to } \text{DROP}). \]

\text{Proof.} \text{DELETE}(u,v) updates the stored information of both \( u \) and \( v \) in constant time simply by decrementing the appropriate counters and deleting from the appropriate sets. Then \text{DELETE}(u,v) computes whether \( L_{\text{max}}(v) < L(v) \) and whether \( L_{\text{max}}(u) < L(u) \). This takes \( O(1) \) time by Lemma 3.9.

\text{Lemma 3.11.} \text{ RISE}(v) \text{ runs in time } d_{ap}(v) \text{ with respect to } v \text{'s layer immediately before the call to } \text{RISE}(v) \text{ (ignoring recursive calls).} \]
Proof. It takes time $O(d_{up}(v))$ to scan the $N^+(v) \cup N^-_L(v)$, which suffices to determine $L_{max}(v)$, build the set $S$ (defined in Algorithm 1), flip the appropriate edges, and determine for each $u \in S$ whether $d^+(u) > d$.

We must also update the stored information for $v$ and all vertices in $N^+(v) \cup N^-_L(v)$. This can be done in $O(d_{up}(v))$ time by incrementing/decrementing the appropriate counters and editing the appropriate sets. Importantly, every vertex in a layer below $v$ immediately before the call to does not need to update its stored information because vertices only keep track of the exact layer of their neighbors on lower layers. Additionally, $v$ does not need to update any of its information concerning its neighbors in lower layers.

Lemma 3.12. $\text{DROP}(v)$ runs in amortized time $O(d + F(n))$ (ignoring recursive calls).

Proof. $\text{DROP}(v)$ begins by computing $L_{max}(v)$. Let $j$ be such that $L_j = L(v)$. $L_{max}(v)$ can be calculated by finding the value $t$ such that $d_{up}(v) + \sum_{i=1}^{j} d_{j-i}(v) \leq d$ but $d_{up}(v) + \sum_{i=1}^{j+1} d_{j-i}(v) > d$. The time spent doing this calculation is proportional to the number of layers that $v$ moves in this call to $\text{DROP}(v)$. Thus, the time spent on these calculations throughout the whole computation is $O(p) = O(f_h/4 + M)$ by Lemma 3.5, which is $O(F(n)M)$ by Lemma 3.3. So the amortized time to compute $L_{max}(v)$ is $O(F(n))$.

After moving $v$ to layer $L_{max}(v)$, $\text{DROP}(v)$ builds the sets $S$ and $S^+$ (defined in Algorithm 1) and flips the appropriate edges. To do this, we scan $N^+(v)$ and $N_i(v)$ for the appropriate layers $L_i$. The sets $S$ and $S^+$ contain at most $d$ vertices and the number of $N_i(v)$ that we scan is the number of layers that $v$ moves in this call to $\text{DROP}(v)$. This takes amortized time $O(d + F(n))$ from the calculations in the previous paragraph. Then, by Lemma 3.9, determining whether $L_{max}(u) < L(u)$ for each $u \in S^+$ takes $O(d)$ time (constant time for each vertex $u$).

We must also update the stored information for $v$ and all vertices in $N^+(v) \cup N^-_L(v)$ (with respect to $v$’s new layer). This can be done in $O(d)$ time by incrementing/decrementing the appropriate counters and editing the appropriate sets. Importantly, every vertex in a layer below $v$’s new layer does not need to update its stored information because vertices only keep track of the exact layer of their neighbors on lower layers. Additionally, $v$ does not need to update any of its information concerning its neighbors in lower layers.

Lemma 3.13. Given that each call to $\text{RISE}(v)$ takes time $O(d_{up}(v))$ with respect to $v$’s layer immediately before the call to $\text{RISE}(v)$ (ignoring recursive calls), $\text{RISE}(v)$ runs in amortized $O(F(n))$ time.

Proof. We claim that when $\text{RISE}(v)$ is called, $d^-_L(v) \leq d' + 1$. By Lemma 3.1, the only operation that can trigger $d^-_L(v)$ to exceed $d'$ is an edge insertion. If such an edge insertion happens, $\text{RISE}(v)$ is immediately called at which point $d^-_L(v) = d' + 1$.

All terms in the following inequalities are with respect to $v$’s layer immediately before the call to $\text{RISE}(v)$. Let be $g$ be the number of edges flipped in the call to $\text{RISE}(v)$ (ignoring recursive calls). Then $g \geq d^+(v) - d$ since the out-degree of $v$ after the edge flips is at most $d$.

$$d_{up}(v) = d^+(v) + d^-_L(v)$$
$$\leq d^+(v) + d' + 1 \quad \text{since } d^-_L(v) \leq d' + 1$$
$$\leq g + d + d' + 1 \quad \text{since } g \geq d^+(v) - d.$$
if \( \text{RISE}(v) \) is a light call, then \( d_{\text{up}}(v) = O(d') \). Thus, the total time for light calls to \( \text{RISE} \) is

\[
O(ld') = O\left(\frac{Md'}{d' - d}\right) \quad \text{by Lemma 3.4}
\]

\[
= O(M) \quad \text{by choice of } d'
\]

so the amortized time for light calls to \( \text{RISE} \) is \( O(1) \).

**Lemma 3.14.** Given that each call to \( \text{DROP} \) takes \( O(d) \) time, \( \text{DROP} \) takes amortized \( O(F(n)) \) time.

**Proof.** If each call to \( \text{DROP} \) takes time \( O(d) \) then the total time for calls to \( \text{DROP} \) is \( O(pd) \). By Lemma 3.5, \( pd \leq \frac{M}{4} + M = O(F(n)M) \) so the amortized time is \( O(F(n)) \).

### 3.7 Coloring from LDS

**Proof of Theorem 3.2.** Recall that \( A' \) is a fully dynamic algorithm that colors graphs of maximum degree \( \Delta \) using \( O(\Delta) \) colors. Further recall that such a randomized algorithm exists with \( O(1) \) amortized update time in expectation and that \( T'(\Delta, n) \leq \text{polylog}(\Delta) \) is the running time of an optimal deterministic algorithm for this problem.

By definition, the graph induced by each layer has degree at most \( 2d' \). We assign each of the \( O(\log n) \) layers a disjoint set of \( O(d') \) colors and use algorithm \( A' \) to dynamically color the graph induced by each layer independently. The total number of colors is \( O(d' \log n) = O(D + \alpha \log n) \log n) \).

When a vertex \( v \) changes layer, all of \( v \)'s incident edges to other vertices in its old layer are removed from the graph induced by its old layer. There are at most \( d_{\text{up}}(v) \) edges (with respect to \( v \)'s layer before being moved) if \( v \) moves to a higher layer and at most \( d \) such edges if \( v \) moves to a lower layer. Also, when \( v \) changes layer all of \( v \)'s incident edges to vertices in its new layer \( L(v) \) are added to the graph induced by \( L(v) \). By Lemma 3.1 there are at most \( d \) such edges. By Lemmas 3.13 and 3.14 given that each call to \( \text{RISE} \) takes time \( O(d_{\text{up}}(v)) \) and each call to \( \text{DROP} \) takes time \( O(d) \), then the amortized time for both procedures is \( O(F(n)) \). Thus, the amortized number of edges updates over all of the layers is \( O(F(n)) \). So the amortized update time is \( O(F(n) \cdot T'(2d', n)) \).

Applying the dynamic edge orientation result of [39], which gives \( D = O(\alpha \log n) \) and \( F(n) = O(1) \) completes the proof.

**Acknowledgements** The authors thank Krzysztof Onak, Baruch Schieber and Virginia Vassilevska Williams for discussions.

**References**

[1] Ittai Abraham, Shiri Chechik, and Sebastian Krinninger. Fully dynamic all-pairs shortest paths with worst-case update-time revisited. In *Proceedings of the 28th Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2017, Barcelona, Spain, January 16-19, 2017*, pages 440–452, 2017.

[2] Ittai Abraham, David Durfee, Ioannis Koutis, Sebastian Krinninger, and Richard Peng. On fully dynamic graph sparsifiers. In *Proc. of 57th FOCS*, pages 335–344, 2016.

[3] A. Andersson and Mikkel Thorup. Dynamic ordered sets with exponential search trees. *J. ACM*, 54(3):13, 2007.
[4] Srinivasa R Arikati, Anil Maheshwari, and Christos D Zaroliagis. Efficient computation of implicit representations of sparse graphs. *Discrete Applied Mathematics*, 78(1-3):1–16, 1997.

[5] Sepehr Assadi, Krzysztof Onak, Baruch Schieber, and Shay Solomon. Fully dynamic maximal independent set with sublinear update time. In *Proc. 50th STOC*, 2018 (to appear).

[6] Baruch Awerbuch, Michael Luby, Andrew V Goldberg, and Serge A Plotkin. Network decomposition and locality in distributed computation. In *FOCS, 1989.*, 30th Annual Symposium on, pages 364–369. IEEE, 1989.

[7] Nikhil Bansal, Anupam Gupta, Ravishankar Krishnaswamy, Kirk Pruhs, Kevin Schewior, and Clifford Stein. A 2-competitive algorithm for online convex optimization with switching costs. In *Proc. of APPROX-RANDOM*, pages 96–109, 2015.

[8] Luis Barba, Jean Cardinal, Matias Korman, Stefan Langerman, André van Renssen, Marcel Roeloffzen, and Sander Verdonschot. Dynamic graph coloring. In *Proceedings of the 15th International Symposium on Algorithms and Data Structures, WADS 2017, St. John’s, NL, Canada, July 31 - August 2, 2017*, pages 97–108, 2017.

[9] Leonid Barenboim and Michael Elkin. Sublogarithmic distributed mis algorithm for sparse graphs using nash-williams decomposition. *Distributed Computing*, 22(5-6):363–379, 2010.

[10] Leonid Barenboim and Michael Elkin. Distributed graph coloring: Fundamentals and recent developments. *Synthesis Lectures on Distributed Computing Theory*, 4(1):1–171, 2013.

[11] Leonid Barenboim and Tzalik Maimon. Fully-dynamic graph algorithms with sublinear time inspired by distributed computing. In *Proceedings of the International Conference on Computational Science, ICCS 2017, Zurich, Switzerland, June 12-14, 2017*, pages 89–98, 2017.

[12] Edvin Berglin and Gerth Stølting Brodal. A simple greedy algorithm for dynamic graph orientation. In *LIPIcs-Leibniz International Proceedings in Informatics*, volume 92. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2017.

[13] Aaron Bernstein, Jacob Holm, and Eva Rotenberg. Online bipartite matching with amortized $o(\log^2 n)$ replacements. In *Proc. of 28th SODA*, pages 692–711, 2018.

[14] Aaron Bernstein, Tsvi Kopelowitz, Seth Pettie, Ely Porat, and Clifford Stein. Simultaneously load balancing for every p-norm, with reassignments. In *LIPIcs-Leibniz International Proceedings in Informatics*, volume 67. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2017.

[15] Aaron Bernstein and Cliff Stein. Fully dynamic matching in bipartite graphs. In *Proceedings of the 42nd International Colloquium on Automata, Languages, and Programming, ICALP 2015, Kyoto, Japan, July 6-10, 2015, Part I*, pages 167–179, 2015.

[16] Aaron Bernstein and Cliff Stein. Faster fully dynamic matchings with small approximation ratios. In *Proceedings of the 27th Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2016, Arlington, VA, USA, January 10-12, 2016*, pages 692–711, 2016.

[17] Sayan Bhattacharya, Deeparnab Chakrabarty, Monika Henzinger, and Danupon Nanongkai. Dynamic algorithms for graph coloring. In *Proceedings of the 29th Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2018, New Orleans, LA, USA, January 7-10, 2018*, pages 1–20, 2018.
[18] Greg Bodwin and Sebastian Krinninger. Fully dynamic spanners with worst-case update time. arXiv preprint arXiv:1606.07864, 2016.

[19] Bartlomiej Bosek, Dariusz Leniowski, Piotr Sankowski, and Anna Zych. Online bipartite matching in offline time. In Proceedings of the 55th IEEE Annual Symposium on Foundations of Computer Science, FOCS 2014, Philadelphia, PA, USA, October 18-21, 2014, pages 384–393, 2014.

[20] Bartlomiej Bosek, Dariusz Leniowski, Piotr Sankowski, and Anna Zych. Shortest augmenting paths for online matchings on trees. In Proc. of 13th WAOA, pages 59–71, 2015.

[21] Bartlomiej Bosek, Dariusz Leniowski, Anna Zych, and Piotr Sankowski. The shortest augmenting paths for online matchings on trees. CoRR, abs/1704.02093, 2017.

[22] G. S. Brodal and R. Fagerberg. Dynamic representation of sparse graphs. In Proc. of 6th WADS, pages 342–351, 1999.

[23] Keren Censor-Hillel, Elad Haramaty, and Zohar S. Karnin. Optimal dynamic distributed MIS. In Proceedings of the 2016 ACM Symposium on Principles of Distributed Computing, PODC 2016, Chicago, IL, USA, July 25-28, 2016, pages 217–226, 2016.

[24] Moses Charikar and Shay Solomon. Fully dynamic almost-maximal matching: Breaking the polynomial barrier for worst-case time bounds. arXiv preprint arXiv:1711.06883, 2017.

[25] Kamalika Chaudhuri, Constantinos Daskalakis, Robert D. Kleinberg, and Henry Lin. Online bipartite perfect matching with augmentations. In Proc. of 28th INFOCOM, pages 1044–1052, 2009.

[26] Richard Cole and Uzi Vishkin. Deterministic coin tossing with applications to optimal parallel list ranking. Information and Control, 70(1):32–53, 1986.

[27] Artur Czumaj and Christian Sohler. Testing hypergraph coloring. In International Colloquium on Automata, Languages, and Programming, pages 493–505. Springer, 2001.

[28] Maximilien Danisch, Oana Balalau, and Mauro Sozio. Listing k-cliques in sparse real-world graphs. communities, 28:43, 2018.

[29] Paul Dietz and Daniel Sleator. Two algorithms for maintaining order in a list. In Proceedings of the nineteenth annual ACM symposium on Theory of computing, pages 365–372. ACM, 1987.

[30] Paul F Dietz and Rajeev Raman. Persistence, amortization and randomization. In Proceedings of the second annual ACM-SIAM symposium on Discrete algorithms, pages 78–88. Society for Industrial and Applied Mathematics, 1991.

[31] Paul F Dietz and Rajeev Raman. A constant update time finger search tree. Information Processing Letters, 52(3):147–154, 1994.

[32] Zdeněk Dvořák and Vojtěch Tuma. A dynamic data structure for counting subgraphs in sparse graphs. In Workshop on Algorithms and Data Structures, pages 304–315. Springer, 2013.

[33] Talya Eden, Reut Levi, and Dana Ron. Testing bounded arboricity. In Proceedings of the Twenty-Ninth Annual ACM-SIAM Symposium on Discrete Algorithms, pages 2081–2092. SIAM, 2018.
[34] Guy Even, Moti Medina, and Dana Ron. Deterministic stateless centralized local algorithms for bounded degree graphs. In European Symposium on Algorithms, pages 394–405. Springer, 2014.

[35] Daniele Frigioni, Alberto Marchetti-Spaccamela, and Umberto Nanni. Fully dynamic shortest paths in digraphs with arbitrary arc weights. Journal of Algorithms, 49(1):86–113, 2003.

[36] Andrew V Goldberg, Serge A Plotkin, and Gregory E Shannon. Parallel symmetry-breaking in sparse graphs. SIAM Journal on Discrete Mathematics, 1(4):434–446, 1988.

[37] Oded Goldreich, Shari Goldwasser, and Dana Ron. Property testing and its connection to learning and approximation. Journal of the ACM (JACM), 45(4):653–750, 1998.

[38] Edward F. Grove, Ming-Yang Kao, P. Krishnan, and Jeffrey Scott Vitter. Online perfect matching and mobile computing. In Proc. of 45th Wads, pages 194–205, 1995.

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

[40] Anupam Gupta, Amit Kumar, and Cliff Stein. Maintaining assignments online: Matching, scheduling, and flows. In Proc. 25th SODA, pages 468–479, 2014.

[41] Bradley Hardy, Rhyd Lewis, and Jonathan Thompson. Modifying colourings between time-steps to tackle changes in dynamic random graphs. In European Conference on Evolutionary Computation in Combinatorial Optimization, pages 186–201. Springer, 2016.

[42] Bradley Hardy, Rhyd Lewis, and Jonathan Thompson. Tackling the edge dynamic graph colouring problem with and without future adjacency information. Journal of Heuristics, pages 1–23, 2017.

[43] M. He, G. Tang, and N. Zeh. Orienting dynamic graphs, with applications to maximal matchings and adjacency queries. In Proc. 25th ISAAC, pages 128–140, 2014.

[44] Shweta Jain and C Seshadhri. A fast and provable method for estimating clique counts using turán’s theorem. In Proceedings of the 26th International Conference on World Wide Web, pages 441–449. International World Wide Web Conferences Steering Committee, 2017.

[45] Alexis Kaporis, Christos Makris, George Mavritsakis, Spyros Sioutas, Athanasios Tsakalidis, Kostas Tsichlas, and Christos Zaroliagis. Isb-tree: a new indexing scheme with efficient expected behaviour. In International Symposium on Algorithms and Computation, pages 318–327. Springer, 2005.

[46] Subhash Khot and Ashok Kumar Ponnuswami. Better inapproximability results for maxclique, chromatic number and min-3lin-deletion. In Proc. 33rd ICALP, pages 226–237, 2006.

[47] T. Kopelowitz, R. Krauthgamer, E. Porat, and Shay Solomon. Orienting fully dynamic graphs with worst-case time bounds. In Proc. 41st ICALP, pages 532–543, 2014.

[48] Łukasz Kowalik. Fast 3-coloring triangle-free planar graphs. In European Symposium on Algorithms, pages 436–447. Springer, 2004.

24
[49] Lukasz Kowalik. Adjacency queries in dynamic sparse graphs. *Information Processing Letters*, 102(5):191–195, 2007.

[50] Lukasz Kowalik and Maciej Kurowski. Oracles for bounded-length shortest paths in planar graphs. *ACM Transactions on Algorithms (TALG)*, 2(3):335–363, 2006.

[51] Christos Levcopoulos and Mark H. Overmars. A balanced search tree with $O(1)$ worst-case update time. *Acta Inf.*, 26(3):269–277, 1988.

[52] Min Chih Lin, Francisco J Soulignac, and Jayme L Szwarcfiter. Arboricity, h-index, and dynamic algorithms. *Theoretical Computer Science*, 426:75–90, 2012.

[53] Nathan Linial. Distributive graph algorithms-global solutions from local data. In *Proceedings of the 28th IEEE Annual Symposium on Foundations of Computer Science, FOCS 1987*, Los Angeles, CA, USA, October 27-29, 1987, pages 331–335, 1987.

[54] Nathan Linial. Locality in distributed graph algorithms. *SIAM J. Comput.*, 21(1):193–201, 1992.

[55] Michael Luby. Removing randomness in parallel computation without a processor penalty. *J. Comput. Syst. Sci.*, 47(2):250–286, 1993.

[56] Cara Monical and Forrest Stonedahl. Static vs. dynamic populations in genetic algorithms for coloring a dynamic graph. In *Proceedings of the 2014 Annual Conference on Genetic and Evolutionary Computation*, pages 469–476. ACM, 2014.

[57] Christian Worm Mortensen. Fully dynamic orthogonal range reporting on ram. *SIAM Journal on Computing*, 35(6):1494–1525, 2006.

[58] C St JA Nash-Williams. Decomposition of finite graphs into forests. *Journal of the London Mathematical Society*, 1(1):12–12, 1964.

[59] Ofer Neiman and Shay Solomon. Simple deterministic algorithms for fully dynamic maximal matching. In *Proceedings of the 45th Annual ACM SIGACT Symposium on Theory of Computing, STOC 2013*, Palo Alto, CA, USA, June 1-4, 2013, pages 745–754, 2013.

[60] K. Onak, B. Schieber, S. Solomon, and N. Wein. Fully dynamic mis in uniformly sparse graphs. In *Proc. 45th ICALP*, 2018.

[61] Linda Ouwerfelli and Hend Bouziri. Greedy algorithms for dynamic graph coloring. In *Communications, Computing and Control Applications (CCCA), 2011 International Conference on*, pages 1–5. IEEE, 2011.

[62] Merav Parter, David Peleg, and Shay Solomon. Local-on-average distributed tasks. In *Proceedings of the 27th Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2016*, Arlington, VA, USA, January 10-12, 2016, pages 220–239, 2016.

[63] David Peleg and Shay Solomon. Dynamic $(1+\epsilon)$-approximate matchings: A density-sensitive approach. In *Proceedings of the 27th Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2016*, Arlington, VA, USA, January 10-12, 2016, 2016.

[64] Davy Preuveneers and Yolande Berbers. Acodygra: an agent algorithm for coloring dynamic graphs. *Symbolic and Numeric Algorithms for Scientific Computing (September 2004)*, 6:381–390, 2004.
A Balls and bins bound on recent degree

We prove the portion omitted from the proof of Lemma 2.1. Lemma A.1 is easy to see based on the description of the algorithm and the balls and bins game; we include the rigorous proof for the purpose of completeness.

Lemma A.1. In the special case of the balls are bins game defined in the proof of lemma 2.1, the size of the largest bin is at least the maximum recent degree in the graph.

Proof. Let $B = b_1, \ldots, b_n$ be the set of bins in the game and let $B' = b'_1, \ldots, b'_n$ be the set of hypothetical bins such that at all times the number of balls in bin $b'_i$ is exactly the recent degree of vertex $i$. For $B'$ we say that Player 1' adds balls to $B'$ and Player 2' removes them. We will show that at all times the size of the largest bin in $B$ is at least the size of the largest bin in $B'$. We cannot simply say that for all $i$, $|b_i| \geq |b'_i|$ because Player 2 might empty different bins in $B$ than Player 2' empties in $B'$. Instead, we will show that there is a permutation $\sigma$ of $[n]$ so that
each bin in $B'$ is at least as large as its corresponding bin in $B$. We say that $B$ dominates $B'$ (with respect to $\sigma$) if for all $i$, $b_{\sigma(i)}$ contains at least as many balls than $b'_i$. We will show that $B$ indeed dominates $B'$ at all times, which completes the proof.

Initially $\sigma$ is the identity permutation and all bins are empty so $B$ trivially dominates $B'$. Suppose inductively that $B$ dominates $B'$. Only the following two types of operations could cause $B$ to stop dominating $B'$: a) balls are added to bins in $B'$, and b) balls are deleted from bins in $B$.

The first type of operation is not an issue because by definition when a ball is added to a bin $b'_i$ in $B'$ a ball is also added to bin $b_{\sigma(i)}$ in $B$. The second type of operation occurs in the deterministic game when Player 2 empties the largest bin of $B$. When this happens, Player 2' empties the largest bin of $B'$. Let $i$ be such that $b_{\sigma(i)}$ is the bin in $B$ that is emptied and let $j$ be such that $b'_j$ is the bin in $B'$ that is emptied. By the inductive hypothesis, $|b_{\sigma(j)}| \geq |b'_j|$ and by choice of $j$, $|b'_j| \geq |b'_i|$. Thus, $|b_{\sigma(j)}| \geq |b'_i|$. Then, since $|b_{\sigma(i)}| \geq |b'_j|$ (both are set to 0), we can modify $\sigma$ by switching $\sigma(i)$ and $\sigma(j)$, so that $B$ still dominates $B'$.

The second type of operation also occurs in the randomized game when Player 2 empties a random bin as defined in the game. When this happens, Player 2' also empties a random bin as defined in the algorithm. By these definitions, Players 2 and 2' choose a bin randomly over the same distribution of bins (i.e. if Player 2 chooses bin $b_{\sigma(i)}$ with probability $p$, then Player 2' chooses bin $b'_i$ with probability $p$). We can assume that Players 2 and 2' share a random coin and select matching bins. \qed