Dynamic Graph Coloring*

Luis Barba[1], Jean Cardinal[2], Matias Korman[3], Stefan Langerman[2], André van Renssen[4,5], Marcel Roeloffzen[4,5], and Sander Verdonschot[6]

[1] Dept. of Computer Science, ETH Zürich, Switzerland. luis.barba@inf.ethz.ch
[2] Département d'Informatique, Université Libre de Bruxelles, Brussels, Belgium. {jcardin,stefan.langerman}@ulb.ac.be
[3] Tohoku University, Sendai, Japan. mati@dais.is.tohoku.ac.jp
[4] National Institute of Informatics, Tokyo, Japan. {andre,marcel}@nii.ac.jp
[5] JST, ERATO, Kawarabayashi Large Graph Project.
[6] School of Computer Science, Carleton University, Ottawa, Canada. sander@cg.scs.carleton.ca

Abstract. In this paper we study the number of vertex recolorings that an algorithm needs to perform in order to maintain a proper coloring of a graph under insertion and deletion of vertices and edges. We present two algorithms that achieve different trade-offs between the number of recolorings and the number of colors used. For any $d > 0$, the first algorithm maintains a proper $O(\mathcal{C}dN^{1/d})$-coloring while recoloring at most $O(d)$ vertices per update, where $\mathcal{C}$ and N are the maximum chromatic number and maximum number of vertices, respectively. The second algorithm reverses the trade-off, maintaining an $O(\mathcal{C}d)$-coloring with $O(dN^{1/d})$ recolorings per update. We also present a lower bound, showing that any algorithm that maintains a c-coloring of a 2-colorable graph on N vertices must recolor at least $\Omega(N^{\frac{2}{c(c-1)}})$ vertices per update, for any constant $c \geq 2$.

1 Introduction

It is hard to underestimate the importance of the graph coloring problem in computer science and combinatorics. The problem is certainly among the most studied questions in those fields, and countless applications and variants have been tackled since it was first posed for the special case of maps in the mid-nineteenth century. Similarly, the maintenance of some structures in *dynamic graphs* has been the subject of study of several volumes in the past couple of decades [1,2,11,18,19,20]. In this setting, an algorithmic graph problem is modelled in the dynamic environment as follows. There is an online sequence of insertion and deletion of edges or vertices, and our goal is to maintain the solution of the graph problem after each update. A trivial way to maintain this solution is to run the best static algorithm for this problem after each update; however, this

* M. K. was partially supported by MEXT KAKENHI grant Nos. 12H00855, and 17K12635. A. v. R. and M. R. were supported by JST ERATO Grant Number JPMJER1305, Japan. L. B. was supported by the ETH Postdoctoral Fellowship. S. V. was partially supported by NSERC and the Carleton-Fields postdoctoral award.

is clearly not optimal. A dynamic graph algorithm seeks to maintain some clever data structure for the underlying problem such that the time taken to update the solution is much smaller than that of the best static algorithm.

In this paper, we study the problem of maintaining a coloring in a dynamic graph undergoing insertions and deletions of both vertices and edges. At first sight, this may seem to be a hopeless task, since there exist near-linear lower bounds on the competitive factor of online graph coloring algorithms [9], a restricted case of the dynamic setting. In order to break through this barrier, we allow a "fair" number of *vertex recolorings* per update. We focus on the combinatorial aspect of the problem – the trade-off between the number of colors used versus the number of recolorings per update. We present a strong general lower bound and two simple algorithms that provide complementary trade-offs.

Definitions and Results. Let $\mathcal{C}$ be a positive integer. A *$\mathcal{C}$-coloring* of a graph G is a function that assigns a color in $\{1, \ldots, \mathcal{C}\}$ to each vertex of G. A $\mathcal{C}$-coloring is *proper* if no two adjacent vertices are assigned the same color. We say that G is *$\mathcal{C}$-colorable* if it admits a proper $\mathcal{C}$-coloring, and we call the smallest such $\mathcal{C}$ the *chromatic number* of G.

A *recoloring algorithm* is an algorithm that maintains a proper coloring of a simple graph while that graph undergoes a sequence of updates. Each update adds or removes either an edge or a vertex with a set of incident edges. We say that a recoloring algorithm is *c-competitive* if it uses at most $c \cdot \mathcal{C}_{max}$ colors, where $\mathcal{C}_{max}$ is the maximum chromatic number of the graph during the updates.

For example, an algorithm that computes the optimal coloring after every update is 1-competitive, but may recolor every vertex for every update. At the other extreme, we can give each vertex a unique color, resulting in a linear competitive factor for an algorithm that recolors at most 1 vertex per update. In this paper, we investigate intermediate solutions that use more than $\mathcal{C}$ colors but recolor a sublinear number of vertices per update. Note that we do not assume that the value $\mathcal{C}$ is known in advance, or at any point during the algorithm.

In Section 2, we present two complementary recoloring algorithms: an $O(dN^{1/d})$-competitive algorithm with an amortized $O(d)$ recolorings per update, and an $O(d)$-competitive algorithm with an amortized $O(dN^{1/d})$ recolorings per update, where d is a positive integer parameter and N is the maximum number of vertices in the graph during a sequence of updates. Interestingly, for $d = \Theta(\log N)$, both are $O(\log N)$-competitive with an amortized $O(\log N)$ vertex recolorings per update. Using standard techniques, the algorithms can be made sensitive to the current (instead of the maximum) number of vertices in the graph.

We provide lower bounds in Section 3. In particular, we show that for any recoloring algorithm A using c colors, there exists a specific 2-colorable graph on N vertices and a sequence of m edge insertions and deletions that forces A to perform at least $\Omega(m \cdot N^{\frac{2}{c(c-1)}})$ vertex recolorings. Thus, any x-competitive recoloring algorithm performs in average at least $\Omega(N^{\frac{1}{x(2x-1)}})$ recolorings per update.

To allow us to focus on the combinatorial aspects, we assume that we have access to an algorithm that, at any time, can color the current graph (or an induced subgraph) using few colors. Of course, finding an optimal coloring of an

n-vertex graph is NP-complete in general [13] and even NP-hard to approximate to within $n^{1-\epsilon}$ for any $\epsilon > 0$ [22]. Still, this assumption is not as strong as it sounds. Most practical instances can be colored efficiently [4], and for several important classes of graphs the problem is solvable or approximable in polynomial time, including bipartite graphs, planar graphs, k-degenerate graphs, and unit disk graphs [15].

Related results. *Dynamic graph coloring.* The problem of maintaining a coloring of a graph that evolves over time has been tackled before, but to our knowledge, only from the points of view of heuristics and experimental results. This includes for instance results from Preuveneers and Berbers [17], Ouerfelli and Bouziri [16], and Dutot et al. [7]. A related problem of maintaining a graph-coloring in an online fashion was studied by Borowiecki and Sidorowicz [3]. In that problem, vertices lose their color, and the algorithm is asked to recolor them.

Online graph coloring. The online version of the problem is closely related to our setting, except that most variants of the online problem only allow the coloring of new vertices, which then cannot be recolored later. Near-linear lower bounds on the best achievable competitive factor have been proven by Halldórsson and Szegedy more than two decades ago [9]. They show their bound holds even when the model is relaxed to allow a constant fraction of the vertices to change color over the whole sequence. This, however, does not contradict our results. We allow our algorithms to recolor all vertices at some point, but we bound only the number of recolorings *per update.* Algorithms for online coloring with competitive factor coming close, or equal to this lower bound have been proposed by Lovász et al. [14], Vishwanathan [21], and Halldórsson [8].

Dynamic graphs. Several techniques have been used for the maintenance of other structures in dynamic graphs, such as spanning trees, transitive closure, and shortest paths. Surveys by Demetrescu et al. [5,6] give a good overview of those. Recent progress on dynamic connectivity [12] and approximate single-source shortest paths [10] are witnesses of the current activity in this field.

2 Upper bound: Recoloring-algorithms

For the description of our algorithms we consider only inserting a vertex with its incident edges. Deletions cannot invalidate the coloring and edge insertions can be done by removing and adding one of the vertices with the appropriate edges.

Our algorithms partition the vertices into a set of *buckets*, each of which has its own distinct set of colors. All our algorithms guarantee that the subgraph induced by the vertices inside each bucket is properly colored and this implies that the entire graph is properly colored at all times.

The algorithms differ in the number of buckets they use and the size (maximum number of vertices) of each bucket. Typically, there is a sequence of buckets of increasing size, and one *reset bucket* that can contain arbitrarily many vertices and that holds vertices whose color has not changed for a while. Initially, the size of each bucket depends on the number of vertices in the input graph. As vertices are inserted and deleted, the current number of vertices changes. When certain

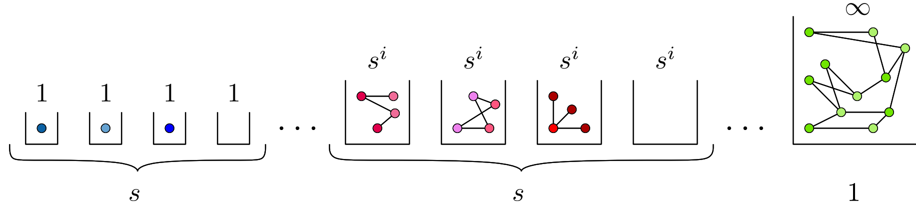

Fig. 1: The small-buckets algorithm uses d levels, each with s buckets of capacity s^i, where i is the level, $s = \lceil N_R^{1/d} \rceil$, and N_R is the number of vertices during the last reset.

buckets are full, we *reset* everything, to ensure that we can accommodate the new number of vertices. This involves emptying all buckets into the reset bucket, computing a proper coloring of the entire graph, and recomputing the sizes of the buckets in terms of the current number of vertices.

We refer to the number of vertices during the most recent reset as N_R, and we express the size of the buckets in $s = \lceil N_R^{1/d} \rceil$, where $d > 0$ is an integer parameter that allows us to achieve different trade-offs between the number of colors and number of recolorings used. Since $s = O(N^{1/d})$, where N is the maximum number of vertices thus far, we state our bounds in terms of N. Note that it is also possible to keep N_R within a constant factor of the current number of vertices by triggering a reset whenever the current number of vertices becomes too small or too large. We omit these details for the sake of simplicity.

2.1 Small-buckets algorithm

Our first algorithm, called the *small-buckets algorithm*, uses a lot of colors, but needs very few recolorings. In addition to the reset bucket, the algorithm uses ds buckets, grouped into d *levels* of s buckets each. All buckets on level i, for $0 \leq i < d$, have capacity s^i (see Fig. 1). Initially, the reset bucket contains all vertices, and all other buckets are empty. Throughout the execution of the algorithm, we ensure that every level always has at least one empty bucket. We call this the *space invariant*.

When a new vertex is inserted, we place it in any empty bucket on level 0. The space invariant guarantees the existence of this bucket. Since this bucket has a unique set of colors, assigning one of them to the new vertex establishes a proper coloring. Of course, if this was the last empty bucket on level 0, filling it violates the space invariant. In that case, we gather up all s vertices on this level, place them in the first empty bucket on level 1 (which has capacity s and must exist by the space invariant), and compute a new coloring of their induced graph using the set of colors of the new bucket. If this was the last free bucket on level 1, we move all its vertices to the next level and repeat this procedure. In general, if we filled the last free bucket on level i, we gather up all at most $s \cdot s^i = s^{i+1}$ vertices on this level, place them in an empty bucket on level $i + 1$ (which exists by the space invariant), and recolor their induced graph with the new colors. If we fill up the last level $(d - 1)$, we reset the structure, emptying each bucket into

the reset bucket and recoloring the whole graph.

Theorem 2.1. *For any integer $d > 0$, the small-buckets algorithm is an $O(dN^{1/d})$-competitive recoloring algorithm that uses at most $O(d)$ amortized vertex recolorings per update.*

Proof. (sketch) The total number of colors is bounded by the maximum number of non-empty buckets $(1 + d(s-1))$, multiplied by the maximum number of colors used by any bucket. Let $\mathcal{C}$ be the maximum chromatic number of the graph. Since any induced subgraph of a $\mathcal{C}$-colorable graph is also $\mathcal{C}$-colorable, each bucket requires at most $\mathcal{C}$ colors. Thus, the total number of colors is at most $(1 + d(s-1))\mathcal{C}$, and the algorithm is $O(dN^{1/d})$-competitive.

To analyze the number of recolorings, we use a simple charging scheme that places coins in the buckets and pays one coin for each recoloring. Whenever we place a vertex in a bucket on level 0, we give $d + 2$ coins to that bucket. One of these coins is immediately used to pay for the vertex's new color, leaving $d + 1$ coins. In general, we can maintain the invariant that each non-empty bucket on level i has $s^i \cdot (d - i + 1)$ coins from which the result follows. □

2.2 Big-buckets algorithm

Our second algorithm, called the *big-buckets algorithm*, is similar to the small-buckets algorithm, except it merges all buckets on the same level into a single larger bucket. Specifically, the algorithm uses d buckets in addition to the reset bucket. These buckets are numbered sequentially from 0 to $d - 1$, with bucket i having capacity s^{i+1}, see Fig. 2. Since we use far fewer buckets, an upper bound on the total number of colors drops significantly, to $(d + 1)\mathcal{C}$. Of course, as we will see later, we pay for this in the number recolorings. Similar to the space invariant in the small-buckets algorithm, the big-buckets algorithm maintains the *high point invariant*: bucket i always contains at most $s^{i+1} - s^i$ vertices (its *high point*).

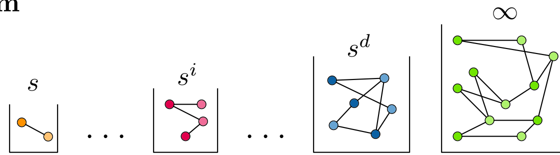

Fig. 2: Besides the reset bucket, the big-buckets algorithm uses d buckets, each with capacity s^{i+1}, where i is the bucket number.

When a new vertex is inserted, we place it in the first bucket. Since this bucket may already contain other vertices, we recolor all its vertices, so that the subgraph induced by these vertices remains properly colored. This revalidates the coloring, but may violate the high point invariant. If we filled bucket i beyond its high point, we move all its vertices to bucket $i + 1$ and compute a new coloring for this bucket. We repeat this until the high point invariant is satisfied, or we fill bucket $d - 1$ past its high point. In the latter case we reset, adding all vertices to the reset bucket and computing a new coloring for the entire graph.

Theorem 2.2. *For any integer $d > 0$, the big-buckets algorithm is an $O(d)$-competitive recoloring algorithm that uses at most $O(dN^{1/d})$ amortized vertex recolorings per update.*

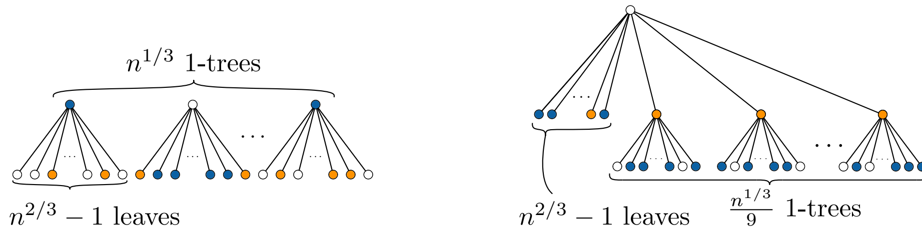

Fig. 3: (left) A 1-configuration is any forest that has many 1-trees as induced subgraphs. (right) A 2-tree is constructed by connecting the roots of many 1-trees.

3 Lower bound

In this section we prove a lower bound on the amortized number of recolorings for any algorithm that maintains a c-coloring of a 2-colorable graph, for any constant $c \geq 2$. We say that a vertex is *c-colored* if it has a color in $[c] = \{1, \ldots, c\}$. For simplicity of description, we assume that a recoloring algorithm only recolors vertices when an edge is inserted and not when an edge is deleted, as edge deletions do not invalidate the coloring. This assumption causes no loss of generality, as we can delay the recolorings an algorithm would perform in response to an edge deletion until the next edge insertion.

The proof for the lower bound consists of several parts. We begin with a specific initial configuration and present a strategy for an adversary that constructs a large configuration with a specific colouring and then repeatedly performs costly operations in this configuration. In light of this strategy, a recoloring algorithm has a few choices: it can allow the configuration to be built and perform the recolorings required, it can destroy the configuration by recoloring parts of it instead of performing the operations, or it can prevent the configuration from being built in the first place by recoloring parts of the building blocks. We show that all these options require a large number of amortized recolorings.

3.1 Maintaining a 3-coloring

To make the general lower bound easier to understand, we first show that to maintain a 3-coloring, we need at least $\Omega(n^{1/3})$ recolorings on average per update.

Lemma 3.1. *For any sufficiently large n and any $m \geq 2n^{1/3}$, there exists a forest with n vertices, such that for any recoloring algorithm A, there exists a sequence of m updates that forces A to perform $\Omega(m \cdot n^{1/3})$ vertex recolorings to maintain a 3-coloring throughout this sequence.*

Proof. Let A be any recoloring algorithm that maintains a 3-coloring of a forest under updates. We use an adversarial strategy to choose a sequence of updates on a specific forest with n nodes that forces A to recolor "many" vertices. We start by describing the initial forest structure.

A *1-tree* is a rooted (star) tree with a distinguished vertex as its root and

$n^{2/3} - 1$ leaf nodes attached to it. Initially, our forest consists of $n^{1/3}$ pairwise disjoint 1-trees, which account for all n vertices in our forest. The sequence of updates we construct never performs a cut operation among the edges of a 1-tree. Thus, the forest remains a *1-configuration*: a forest of rooted trees with the $n^{1/3}$ independent 1-trees as induced subgraphs; see Fig. 3 (left). We require that the induced subtrees are not *upside down*, that is, the root of the 1-tree should be closer to the root of the full tree than its children. Intuitively, a 1-configuration is simply a collection of our initial 1-trees linked together into larger trees.

Let F be a 1-configuration. We assume that A has already chosen an initial 3-coloring of F. We assign a color to each 1-tree as follows. Since each 1-tree is properly 3-colored, the leaves cannot have the same color as the root. Thus, a 1-tree T always has at least $\frac{n^{2/3}-1}{2}$ leaves of some color C, and C is different from the color of the root. We assign the color C to T. In this way, each 1-tree is assigned one of the three colors. We say that a 1-tree with assigned color C becomes *invalid* if it has no children of color C left. Notice that to invalidate a 1-tree, algorithm A needs to recolor at least $\frac{n^{2/3}-1}{2}$ of its leaves. Since the coloring uses only three colors, there are at least $\frac{n^{1/3}}{3}$ 1-trees with the same assigned color, say X. In the remainder, we focus solely on these 1-trees.

A *2-tree* is a tree obtained by merging $\frac{n^{1/3}}{9}$ 1-trees with assigned color X, as follows. First, we cut the edge connecting the root of each 1-tree to its parent, if it has one. Next, we pick a distinguished 1-tree with root r, and connect the root of each of the other $\frac{n^{1/3}}{9} - 1$ 1-trees to r. In this way, we obtain a 2-tree whose root r has $n^{2/3} - 1$ leaf children from the 1-tree of r, and $\frac{n^{1/3}}{9} - 1$ new children that are the roots of other 1-trees; see Fig. 3 (right) for an illustration. This construction requires $\frac{n^{1/3}}{9} - 1$ edge insertions and at most $\frac{n^{1/3}}{9}$ edge deletions (if every 1-tree root had another parent in the 1-configuration).

We build 3 such 2-trees in total. This requires at most $6(\frac{n^{1/3}}{9}) = \frac{2n^{1/3}}{3}$ updates. If none of our 1-trees became invalid, then since our construction involves only 1-trees with the same assigned color X, no 2-tree can have a root with color X. Further, since the algorithm maintains a 3-coloring, there must be at least two 2-trees whose roots have the same color. We can now perform a *matching link*, by connecting the roots of these two trees by an edge (in general, we may need to perform a cut first). To maintain a 3-coloring after a matching link, A must recolor the root of one of the 2-trees and either recolor all its non-leaf children or invalidate a 1-tree. If no 1-tree has become invalidated, this requires at least $\frac{n^{1/3}}{9}$ recolorings, and we again have two 2-trees whose roots have the same color. Thus, we can perform another matching link between them. We keep doing this until we either performed $\frac{n^{1/3}}{6}$ matching links, or a 1-tree is invalidated.

Therefore, after at most $n^{1/3}$ updates ($\frac{2n^{1/3}}{3}$ for the construction of the 2-trees, and $\frac{n^{1/3}}{3}$ for the matching links), we either have an invalid 1-tree, in which case A recolored at least $\frac{n^{2/3}-1}{2}$ nodes, or we performed $\frac{n^{1/3}}{6}$ matching links, which forced at least $\frac{n^{1/3}}{6} \cdot \frac{n^{1/3}}{9} = \frac{n^{2/3}}{54}$ recolorings. In either case, we forced A to perform at least $\Omega(n^{2/3})$ vertex recolorings, using at most $n^{1/3}$ updates.

Since no edge of a 1-tree was cut, we still have a valid 1-configuration, where the process can be restarted. Consequently, for any $m \geq 2n^{1/3}$, there exists a sequence of m updates that starts with a 1-configuration and forces A to perform $\lfloor \frac{m}{n^{1/3}} \rfloor \Omega(n^{2/3}) = \Omega(m \cdot n^{1/3})$ vertex recolorings. □

3.2 On k-trees

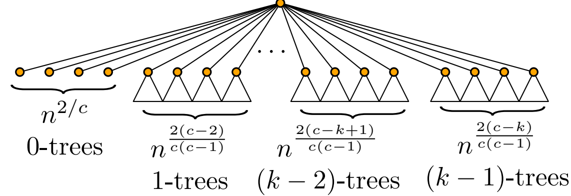

Fig. 4: A k-tree is constructed by connecting the roots of a large number of $(k-1)$-trees.

We are now ready to describe a general lower bound for any number of colors c. The general approach is the same as when using 3 colors: We construct trees of height up to $c+1$, each excluding a different color for the root of the merged trees. By now connecting two such trees, we force the algorithm A to recolor the desired number of vertices.

A *0-tree* is a single node, and for each $1 \leq k \leq c$, a *k-tree* is a tree obtained recursively by merging $2 \cdot n^{\frac{2(c-k)}{c(c-1)}}$ $(k-1)$-trees as follows: Pick a $(k-1)$-tree and let r be its root. Then, for each of the $2 \cdot n^{\frac{2(c-k)}{c(c-1)}} - 1$ remaining $(k-1)$-trees, connect their root to r with an edge; see Fig. 4 for an illustration.

As a result, for each $0 \leq j \leq k-1$, a k-tree T consists of a root r with $2 \cdot n^{\frac{2(c-j-1)}{c(c-1)}} - 1$ j-trees, called the *j-subtrees* of T, whose root hangs from r. The root of a j-subtree of T is called a *j-child* of T. By construction, r is also the root of a j-tree which we call the *core* j-tree of T.

Whenever a k-tree is constructed, it is assigned a color that is present among a "large" fraction of its $(k-1)$-children. Indeed, whenever a k-tree is assigned a color c_k, we guarantee that it has at least $\left\lceil \frac{2}{c} \cdot n^{\frac{2(c-k)}{c(c-1)}} \right\rceil$ $(k-1)$-children of color c_k. We describe later how to choose the color that is assigned to a k-tree.

We say that a k-tree that was assigned color c_k has a *color violation* if its root no longer has a $(k-1)$-child with color c_k. We say that a k-tree T becomes *invalid* if either (1) it has a color violation or (2) if a core j-tree of T has a color violation for some $1 \leq j < k$; otherwise we say that T is *valid*.

Observation 1 *To obtain a color violation in a k-tree constructed by the above procedure, A needs to recolor at least $\left\lceil \frac{2}{c} \cdot n^{\frac{2(c-k)}{c(c-1)}} \right\rceil$ vertices.*

Notice that a valid c-colored k-tree of color c_k cannot have a root with color c_k. Formally, color c_k is *blocked* for the root of a k-tree if this root has a child with color c_k. In particular, the color assigned to a k-tree and the colors assigned to its core j-trees for $1 \leq j \leq k-1$ are blocked as long as the tree is valid.

3.3 On k-configurations

A 0-configuration is a set F_0 of c-colored nodes, where $|F_0| = T_0 = \alpha n$, for some sufficiently large constant α which will be specified later. For $1 \leq k < c$, a

k-configuration is a set F_k of T_k k-trees, where

$$T_k = \frac{\alpha}{(4c)^k} \cdot n^{1-\sum_{i=1}^{k} \frac{2(c-i)}{c(c-1)}}.$$

Note that the trees of a k-configuration may be part of m-trees for $m > k$. If at least $\frac{T_k}{2}$ k-trees in a k-configuration are valid, then the configuration is *valid.*

For our construction, we let the initial configuration F_0 be an arbitrary c-colored 0-configuration in which each vertex is c-colored. To construct a k-configuration F_k from a valid $(k-1)$-configuration F_{k-1}, consider the at least $\frac{T_{k-1}}{2}$ valid $(k-1)$-trees from F_{k-1}. Recall that the trees of F_{k-1} may be part of larger trees, but since we consider edge deletions as "free" operations we can separate the trees. Since each of these trees has a color assigned, among them at least $\frac{T_{k-1}}{2c}$ have the same color assigned to them. Let c_{k-1} denote this color.

Because each k-tree consists of $2 \cdot n^{\frac{2(c-k)}{c(c-1)}}$ $(k-1)$-trees, to obtain F_k we merge $\frac{T_{k-1}}{2c}$ $(k-1)$-trees of color c_{k-1} into T_k k-trees, where

$$T_k = \frac{T_{k-1}}{2c} \cdot \frac{1}{2 \cdot n^{\frac{2(c-k)}{c(c-1)}}} = \frac{\alpha}{(4c)^k} \cdot n^{1-\sum_{i=1}^{k} \frac{2(c-i)}{c(c-1)}}.$$

Once the k-configuration F_k is constructed, we perform a *color assignment* to each k-tree in F_k as follows: For a k-tree τ of F_k whose root has $2 \cdot n^{\frac{2(c-k)}{c(c-1)}} - 1$ c-colored $(k-1)$-children, we assign τ a color that is shared by at least $\left\lfloor \frac{2}{c} \cdot n^{\frac{2(c-k)}{c(c-1)}} - 1 \right\rfloor$ of these $(k-1)$-children. Therefore, τ has at least $\left\lfloor \frac{2}{c} \cdot n^{\frac{2(c-k)}{c(c-1)}} \right\rfloor$ children of its assigned color. After these color assignments, if each $(k-1)$-tree used is valid, then each of the T_k k-trees of F_k is also valid. Thus, F_k is a valid configuration. Moreover, for F_k to become invalid, A would need to invalidate at least $\frac{T_k}{2}$ of its k-trees. Since we use $(k-1)$-trees with the same assigned color to construct k-trees, we can conclude the following about the use of colors in any k-tree.

Lemma 3.2. *Let F_k be a valid k-configuration. For each $1 \leq j < k$, each core j-tree of a valid k-tree of F_k has color c_j assigned to it. Moreover, $c_i \neq c_j$ for each $1 \leq i < j < k$.*

We also provide bounds on the number of updates needed to construct a k-configuration.

Lemma 3.3. *Using $\Theta(\sum_{i=j}^{k} T_i) = \Theta(T_j)$ edge insertions, we can construct a k-configuration from a valid j-configuration.*

Proof. To merge $\frac{T_{k-1}}{2c}$ $(k-1)$-trees to into T_k k-trees, we need $\Theta(T_{k-1})$ edge insertions. Thus, in total, to construct a k-configuration from a j-configuration, we need $\Theta(\sum_{i=j}^{k} T_i) = \Theta(T_j)$ edge insertions. □

3.4 Reset phase

Throughout the construction of a k-configuration, the recoloring-algorithm A may recolor several vertices which could lead to invalid subtrees in F_j for any $1 \leq j < k$. Because A may invalidate some trees from F_j while constructing F_k from F_{k-1}, one of two things can happen. If F_j is a valid j-configuration for each $1 \leq j \leq k$, then we continue and try to construct a $(k+1)$-configuration from F_k. Otherwise a *reset* is triggered as follows.

Let $1 \leq j < k$ be an integer such that F_i is a valid i-configuration for each $0 \leq i \leq j-1$, but F_j is not valid. Since F_j was a valid j-configuration with at least T_j valid j-trees when it was first constructed, we know that in the process of constructing F_k from F_j, at least $\frac{T_j}{2}$ j-trees where invalidated by A. We distinguish two ways in which a tree can be invalid:

(1) the tree has a color violation, but all its $j-1$-subtrees are valid and no core i-tree for $1 \leq i \leq j-1$ has a color violation; or
(2) A core i-tree has a color violation for $1 \leq i \leq j-1$, or the tree has a color violation and at least one of its $(j-1)$-subtrees is invalid.

In case (1) the algorithm A has to perform fewer recolorings, but the tree can be made valid again with a color reassignment, whereas in case (2) the j-tree has to be rebuild.

Let Y_0, Y_1 and Y_2 respectively be the set of j-trees of F_j that are either valid, or are invalid by case (1) or (2) respectively. Because at least $\frac{T_j}{2}$ j-trees were invalidated, we know that $|Y_1| + |Y_2| > \frac{T_j}{2}$. Moreover, for each tree in Y_1, A recolored at least $\frac{2}{c} \cdot n^{\frac{2(c-j)}{c(c-1)}} - 1$ vertices to create the color violation on this j-tree by Observation 1. For each tree in Y_2 however, A created a color violation in some i-tree for $i < j$. Therefore, for each tree in Y_2, by Observation 1, the number of vertices that A recolored is at least $\frac{2}{c} \cdot n^{\frac{2(c-i)}{c(c-1)}} - 1 > \frac{2}{c} \cdot n^{\frac{2(c-j+1)}{c(c-1)}} - 1$.

Case 1: $|Y_1| > |Y_2|$. Recall that each j-tree in Y_1 has only valid $(j-1)$-subtrees by the definition of Y_1. Therefore, each j-tree in Y_1 can be made valid again by performing a color assignment on it while performing no update. In this way, we obtain $|Y_0| + |Y_1| > \frac{T_j}{2}$ valid j-trees, i.e., F_j becomes a valid j-configuration contained in F_k. Notice that when a color assignment is performed on a j-tree, vertex recolorings previously performed on its $(j-1)$-children cannot be counted again towards invalidating this tree.

Since we have a valid j-configuration instead of a valid k-configuration, we "wasted" some edge insertions. We say that the insertion of each edge in F_k that is not an edge of F_j is a *wasted* edge insertion. By Lemma 3.3, to construct F_k from F_j we used $\Theta(T_j)$ edge insertions. That is, $\Theta(T_j)$ edge insertions became wasted. However, while we wasted $\Theta(T_j)$ edge insertions, we also forced A to perform $\Omega(|Y_1| \cdot n^{\frac{2(c-j)}{c(c-1)}}) = \Omega(T_j \cdot n^{\frac{2(c-j)}{c(c-1)}})$ vertex recolorings. Since $1 \leq j < k \leq c-1$, we know that $n^{\frac{2(c-j)}{c(c-1)}} \geq n^{\frac{2}{c(c-1)}}$. Therefore, we can charge A with $\Omega(n^{\frac{2}{c(c-1)}})$ vertex recolorings per wasted edge insertion. Finally, we remove each edge corresponding to a wasted edge insertion, i.e., we remove all the edges used to construct F_k

from F_j. Since we assumed that A performs no recoloring on edge deletions, we are left with a valid j-configuration F_j.

Case 2: $|Y_2| > |Y_1|$. In this case $|Y_2| > \frac{T_j}{4}$. Recall that F_{j-1} is a valid $(j-1)$-configuration by our choice of j. In this case, we say that the insertion of each edge in F_k that is not an edge of F_{j-1} is a *wasted* edge insertion. By Lemma 3.3, we constructed F_k from F_{j-1} using $\Theta(T_{j-1})$ wasted edge insertions. However, while we wasted $\Theta(T_{j-1})$ edge insertions, we also forced A to perform $\Omega(|Y_2| \cdot n^{\frac{2(c-j+1)}{c(c-1)}}) = \Omega(T_j \cdot n^{\frac{2(c-j+1)}{c(c-1)}})$ vertex recolorings. That is, we can charge A with $\Omega(\frac{T_j}{T_{j-1}} \cdot n^{\frac{2(c-j+1)}{c(c-1)}})$ vertex recolorings per wasted edge insertions. Since $\frac{T_{j-1}}{T_j} = 4c \cdot n^{\frac{2(c-j)}{c(c-1)}}$, we conclude that A was charged $\Omega(n^{\frac{2}{c(c-1)}})$ vertex recolorings per wasted edge insertion. Finally, we remove each edge corresponding to a wasted edge insertion, i.e., we go back to the valid $(j-1)$-configuration F_{j-1} as before.

Regardless of the case, we know that during a reset consisting of a sequence of h wasted edge insertions, we charged A with the recoloring of $\Omega(h \cdot n^{\frac{2}{c(c-1)}})$ vertices. Notice that each edge insertion is counted as wasted at most once as the edge that it corresponds to is deleted during the reset phase. A vertex recoloring may be counted more than once. However, a vertex recoloring on a vertex v can count towards invalidating any of the trees it belongs to. Recall though that v belongs to at most one i-tree for each $0 \leq i \leq c$. Moreover, two things can happen during a reset phase that count the recoloring of v towards the invalidation of a j-tree containing it: either (1) a color assignment is performed on this j-tree or (2) this j-tree is destroyed by removing its edges corresponding to wasted edge insertions. In the former case, we know that v needs to be recolored again in order to contribute to invalidating this j-tree. In the latter case, the tree is destroyed and hence, the recoloring of v cannot be counted again towards invalidating it. Therefore, the recoloring of a vertex can be counted towards invalidating any j-tree at most c times throughout the entire construction. Since c is assumed to be a constant, we obtain the following result.

Lemma 3.4. *After a reset phase in which h edge insertions become wasted, we can charge A with $\Omega(h \cdot n^{\frac{2}{c(c-1)}})$ vertex recolorings. Moreover, A will be charged at most $O(1)$ times for each recoloring.*

If A stops triggering resets, then at some point we reach a $(c-1)$-configuration with at least $T_{c-1} \geq 2(c+1)$ trees and $c+1$ valid ones. By linking together two such trees with the same color we can force algorithm A to trigger a reset.

Theorem 3.5. *Let c be a constant. For any sufficiently large integers n and α depending only on c, and any $m = \Omega(n)$ sufficiently large, there exists a forest F with αn vertices, such that for any recoloring algorithm A, there exists a sequence of m updates that forces A to perform $\Omega(m \cdot n^{\frac{2}{c(c-1)}})$ vertex recolorings to maintain a c-coloring of F.*

References

1. S. Baswana, M. Gupta, and S. Sen. Fully dynamic maximal matching in $O(\log n)$ update time. *SIAM J. on Comp.*, 44(1):88–113, 2015.
2. S. Baswana, S. Khurana, and S. Sarkar. Fully dynamic randomized algorithms for graph spanners. *ACM Trans. on Alg.*, 8(4):35, 2012.
3. P. Borowiecki and E. Sidorowicz. Dynamic coloring of graphs. *Fundamenta Informaticae*, 114(2):105–128, 2012.
4. O. Coudert. Exact coloring of real-life graphs is easy. In *Proc. 34th Design Autom. Conf.*, pages 121–126. ACM, 1997.
5. C. Demetrescu, D. Eppstein, Z. Galil, and G. F. Italiano. Dynamic graph algorithms. In M. J. Atallah and M. Blanton, editors, *Algorithms and Theory of Computation Handbook*. Chapman & Hall/CRC, 2010.
6. C. Demetrescu, I. Finocchi, and P. Italiano. Dynamic graphs. In D. Mehta and S. Sahni, editors, *Handbook on Data Structures and Applications*, Computer and Information Science. CRC Press, 2005.
7. A. Dutot, F. Guinand, D. Olivier, and Y. Pigné. On the decentralized dynamic graph-coloring problem. In *Proc. Worksh. Compl. Sys. and Self-Org. Mod.*, 2007.
8. M. M. Halldórsson. Parallel and on-line graph coloring. *J. Alg.*, 23(2):265–280, 1997.
9. M. M. Halldrsson and M. Szegedy. Lower bounds for on-line graph coloring. *Theo. Comp. Sci.*, 130(1):163 – 174, 1994.
10. M. Henzinger, S. Krinninger, and D. Nanongkai. A subquadratic-time algorithm for decremental single-source shortest paths. In *Proc. 25th ACM-SIAM Symp. on Discr. Alg.*, pages 1053–1072, 2014.
11. J. Holm, K. De Lichtenberg, and M. Thorup. Poly-logarithmic deterministic fully-dynamic algorithms for connectivity, minimum spanning tree, 2-edge, and biconnectivity. *J. ACM*, 48(4):723–760, 2001.
12. B. M. Kapron, V. King, and B. Mountjoy. Dynamic graph connectivity in polylogarithmic worst case time. In *Proc. 24th ACM-SIAM Symp. on Discr. Alg.*, pages 1131–1142, 2013.
13. R. M. Karp. Reducibility among combinatorial problems. In *Complexity of computer computations*, pages 85–103. Plenum, New York, 1972.
14. L. Lovász, M. E. Saks, and W. T. Trotter. An on-line graph coloring algorithm with sublinear performance ratio. *Discr. Math.*, 75(1-3):319–325, 1989.
15. M. V. Marathe, H. Breu, H. B. Hunt, III, S. S. Ravi, and D. J. Rosenkrantz. Simple heuristics for unit disk graphs. *Networks*, 25(2):59–68, 1995.
16. L. Ouerfelli and H. Bouziri. Greedy algorithms for dynamic graph coloring. In *Proc. Int. Conf. on Comm., Comp. and Control App.*, pages 1–5, 2011.
17. D. Preuveneers and Y. Berbers. ACODYGRA: an agent algorithm for coloring dynamic graphs. In *Symb. Num. Alg. Sci. Comp.*, pages 381–390, 2004.
18. L. Roditty and U. Zwick. Improved dynamic reachability algorithms for directed graphs. In *Proc. 43rd IEEE Sym. Found. Comp. Sci.*, pages 679–688, 2002.
19. L. Roditty and U. Zwick. Dynamic approximate all-pairs shortest paths in undirected graphs. In *Proc. 45th IEEE Sym. Found. Comp. Sci.*, pages 499–508, 2004.
20. M. Thorup. Fully-dynamic min-cut. *Combinatorica*, 27(1):91–127, 2007.
21. S. Vishwanathan. Randomized online graph coloring. *J. Alg.*, 13(4):657–669, 1992.
22. D. Zuckerman. Linear degree extractors and the inapproximability of max clique and chromatic number. *Theory Comp.*, 3:103–128, 2007.