Dynamic Graph Coloring

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

1 **Départment d'Informatique, Université Libre de Bruxelles, Brussels, Belgium** `{lbarbafl,jcardin,stefan.langerman}@ulb.ac.be`
2 **School of Computer Science, Carleton University, Ottawa, Canada**
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 Electrical Engineering and Computer Science, University of Ottawa, 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 $\mathcal{C}$-colorable graph under insertion and deletion of vertices and edges. We assume that all updates keep the graph $\mathcal{C}$-colorable and that N is the maximum number of vertices in the graph at any point in time.

We present two algorithms to maintain an approximation of the optimal coloring that, together, present a trade-off 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 and recolors at most $O(d)$ vertices per update. The second maintains an $O(\mathcal{C}d)$-coloring using $O(dN^{1/d})$ recolorings per update. Both algorithms achieve the same asymptotic performance when $d = \log N$.

Moreover, we show that for any algorithm that maintains a c-coloring of a 2-colorable graph on N vertices during a sequence of m updates, there is a sequence of updates that forces the algorithm to recolor at least $\Omega(m \cdot N^{\frac{2}{c(c-1)}})$ vertices.

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*, defined as graphs undergoing local changes over time, has been the subject of study of several volumes in the past couple of decades [1, 2, 14, 23, 24, 26].

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 [12], 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 consider the achievable trade-offs between the amount of recolorings needed per change of the graph, and the overall number of colors used.

To keep our working hypotheses minimal, we use (approximate) coloring algorithms as

* Supported in part by the ELC project (MEXT KAKENHI No. 24106008).
† Directeur de Recherches du F.R.S.-FNRS.
‡ Supported in part by NSERC.

black boxes. More precisely, 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 a graph is NP-complete in general [16] and even NP-hard to approximate to within $n^{1-\epsilon}$ for any $\epsilon > 0$ [28]. However, there are many classes of graphs for which the problem is solvable or approximable in polynomial time. These include planar graphs, d-degenerate graphs, chordal graphs [9], perfect graphs [10], partial k-trees, unit disk graphs [19], and circular arc graphs [17]. Moreover, many practical applications appear to yield graphs that can be colored optimally in reasonable time [5]. Thus, our techniques provide efficient solutions when the graph is constrained to these classes, when heuristic colorings suffice, or when the cost of recoloring a vertex or using too many colors far outweighs the cost of computing even an optimal coloring.

1.1 Definitions and Results

Let $\mathcal{C}$ be a positive integer. A *$\mathcal{C}$-coloring* of a graph is a function that assigns a color in $\{1,\dots,\mathcal{C}\}$ to each vertex of the graph. A $\mathcal{C}$-coloring is *proper* if no two adjacent vertices are assigned the same color. We say that a graph is *$\mathcal{C}$-colorable* if it admits a proper $\mathcal{C}$-coloring.

Given a graph G, consider updates that add or remove edges or vertices with their adjacent edges. These operations can be alternated in any arbitrary order and may change the chromatic number of the graph. Our algorithms do not explicitly compute or use the chromatic number, but the number of colors used in the algorithms is bounded as a function of $\mathcal{C}$, which will denote the highest chromatic number of the graph at any point in time.

A *recoloring algorithm* is an algorithm that maintains a proper $(c \cdot \mathcal{C})$-coloring of a $\mathcal{C}$-colorable graph through a sequence of updates, for some constant $c \geq 1$. The naive $(1\cdot\mathcal{C})$-coloring algorithm recolors all vertices of G after each update. On the other extreme, by using n colors we certify that no recoloring is needed between updates. In this work we look for an intermediate solution that uses more than $\mathcal{C}$ colors but recolors $o(n)$ vertices after each update.

Note that we do not assume that the value $\mathcal{C}$ is known in advance, or at any point during the algorithm. In fact, the algorithms presented in this paper adapt if the chromatic number of the graph changes, and their performance can be measured against the maximum chromatic number of the graph through the sequence of updates.

In Section 2 and 3 we present two complementary recoloring algorithms that both maintain an $O(c\cdot\mathcal{C})$-coloring of a $\mathcal{C}$-colorable graph. In what follows, let N denote the maximum number of vertices of the graph at any time. Both algorithms rely on an integer parameter $d > 0$. The first, called the *small-buckets algorithm*, maintains an $O(dN^{1/d} \cdot \mathcal{C})$-coloring of the graph using at most $O(d)$ vertex recolorings per update. The second, called the *big-buckets algorithm* maintains an $O(d\cdot\mathcal{C})$-coloring using $O(dN^{1/d})$ recolorings per update (see Theorem 1 and Theorem 2 for exact bounds). Interestingly, when $d = \log N$, the two algorithms yield the same result. In particular, the de-amortized version of the small-buckets algorithm maintains a proper $O(\mathcal{C}\log N)$-coloring of a $\mathcal{C}$-colorable dynamic graph, using $O(\log N)$ vertex recolorings per update in the worst case. In addition, the amortized algorithms can be easily modified so that their bounds track the current number of vertices of the graph, instead of the maximum number.

In Section 4, we provide lower bounds. We show that for any recoloring algorithm A using c colors, there exists a specific 2-colorable graph and a sequence of m updates that forces A to perform at least $\Omega(m \cdot N^{\frac{2}{c(c-1)}})$ vertex recolorings. Our construction uses only edge insertions and deletions and maintains a forest with $O(N)$ vertices. This implies that for small c our big bucket algorithm is close to optimal: if $c = O(d\cdot\mathcal{C})$ for some positive integer

d, then the big bucket algorithm maintains a c-coloring using $O(cN^{2/c})$ vertex recolorings per update, while our lower bound shows that it cannot do better than $\Omega(N^{\frac{2}{c(c-1)}})$ amortized per update.

1.2 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 [22], Ouerfelli and Bouziri [20], and Dutot et al. [8]. A related problem of maintaining a graph-coloring in an online fashion was studied by Borowiecki and Sidorowicz [4]. 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 do not allow recolorings at every step. Near-linear lower bounds on the best achievable competitive factor have been proven by Halldórsson and Szegedy more than two decades ago [12]. They show their bound holds even when a constant fraction of vertices are recolored 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 with competitive factor coming close, or equal to this lower bound have been proposed by Lovász et al. [18], Vishwanathan [27], and Halldórsson [11].

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. [7, 6] give a good overview of those. Recent progress on dynamic connectivity [15] and approximate single-source shortest paths [13] are witnesses of the current activity in this field.

Data structure dynamization. Our bucketing algorithms are very much inspired by standard techniques for the dynamization of static data structures, pioneered by Bentley and Saxe [25, 3], and improved by Overmars and van Leeuwen [21].

2 Upper bound: Recoloring-algorithms

Throughout this paper, we assume that a recoloring algorithm has some way of computing an optimal coloring for the current graph or one of its induced subgraphs. If the running time is important, and the graph satisfies additional restrictions, this can be substituted by an appropriate approximation algorithm. In that case, the number of colors used by our algorithms increases proportionally. Before continuing to the specific strategies, we first introduce some concepts and definitions that are common to all our algorithms.

It is easy to see that deleting a vertex or edge never invalidates the coloring of the graph. As such, our algorithms do not perform any recolorings when vertices or edges are deleted. The same is true when an edge is inserted between two vertices of different color, leaving only the insertion of an edge between two vertices of the same color, and the insertion of a new vertex, connected to a given set of current vertices, as interesting cases. In our algorithms, we simplify this even further, by implementing the edge insertion case as deleting one of its endpoints and re-inserting it with its new set of adjacent edges. Therefore both the description of the algorithms and the proofs in this section consider only vertex insertions.

Our algorithms partition the vertices into a set of buckets, each of which has its own set of colors that it uses to color the vertices it contains. This set of colors is completely distinct from the sets used by other buckets. Since all our algorithms guarantee that the subgraph induced by the vertices inside each bucket is properly colored, 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 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 N – the maximum number of vertices thus far.

Using the maximum number of vertices instead of the current number of vertices simplifies the description of the algorithm and the bounds. However, it is not a significant difference in practice. Our bounds can be expressed in the number of vertices at the last reset, which can be kept 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. Standard amortization techniques can be used to show that this would cost only a constant number of additional amortized recolorings per insertion or deletion, although deamortization would be more complicated. We omit these details for the sake of simplicity.

Our algorithms depend heavily on an integer parameter $d > 0$, which represents the number of levels in our structure. By modifying d, we obtain different trade-offs between the number of colors and number of recolorings used. We express the size of the buckets in $s = \lceil N_R^{1/d} \rceil$, where N_R is the maximum number of vertices up to the last reset.

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*.

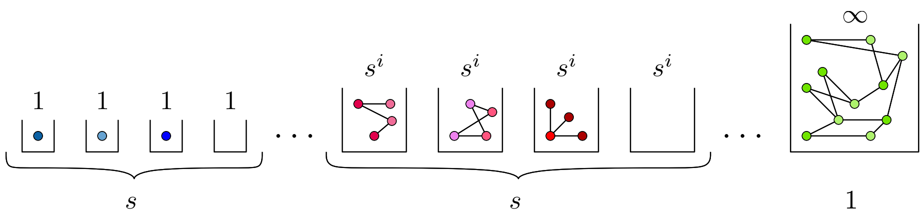

Figure 1 The small-buckets algorithm uses d levels of buckets, each with s buckets of capacity s^i, where i is the level and $s = \lceil N^{1/d} \rceil$.

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 v 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 $s \cdot s^i = s^{i+1}$ 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 1.** *The small-buckets algorithm maintains a proper $(1 + d(s-1))\mathcal{C}$-coloring of a $\mathcal{C}$-colorable graph, using at most $d+2$ amortized vertex recolorings per update.*

Proof. 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. Since any induced subgraph of a $\mathcal{C}$-colorable graph is also $\mathcal{C}$-colorable, each bucket needs at most $\mathcal{C}$ colors. Thus, the total number of colors is at most $(1 + d(s-1))\mathcal{C}$.

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 maintain the invariant that each non-empty bucket on level i has $s^i \cdot (d-i+1)$ coins.

When we merge the vertices on level i into a new bucket on level $i+1$, we pay a single coin for each vertex that changes color. Since each bucket had $s^i \cdot (d-i+1)$ coins, and we recolored at most $s \cdot s^i = s^{i+1}$ vertices, our new bucket has at least $s \cdot s^i \cdot (d-i+1) - s^{i+1} = s^{i+1} \cdot (d-(i+1)+1)$ coins left, satisfying the invariant.

When we fill up level $d-1$, we reset the structure and recolor all vertices. At this point, the buckets on level $d-1$ have a total of $s \cdot s^{d-1} \cdot (d-(d-1)+1) = 2s^d$ coins, and no more than s^d vertices. Since all new vertices are inserted on level 0, and vertices are moved to the reset bucket only during a reset, the number of vertices in the reset bucket is at most N_R. Since $s^d = \lceil N_R^{1/d} \rceil^d \geq (N_R^{1/d})^d = N_R$, we have enough coins to recolor all vertices. Thus, we require no more than $d+2$ amortized recolorings per update. ◀

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, 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*).

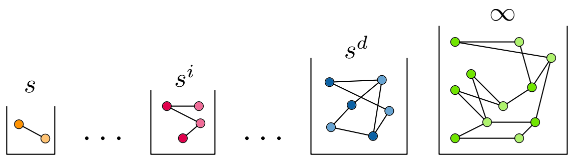

Figure 2 Besides the reset bucket, the big-buckets algorithm uses d buckets, each with capacity s^{i+1}, where i is the bucket number.

When a new vertex is inserted, we place it in the first bucket. Since this bucket may already contain other vertices, we have to recolor all its vertices, so that their induced subgraph 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.** *The big-buckets algorithm maintains a $(d+1)\mathcal{C}$-coloring of a $\mathcal{C}$-colorable graph, using at most $(d+1)s$ amortized vertex recolorings per update.*

Proof sketch. The proof uses an accounting argument similar to that of Theorem 1. In this case the proof relies on the invariant that each bucket has $P_i = \lceil k_i/s^i \rceil \cdot s^{i+1} \cdot (d-i)$ coins. Recoloring is done only on level 0 or when the bucket of the previous level is full. The coins used for recoloring and maintaining the invariant then come from the $(d+1)s$ coins provided by the new vertex insertion or by coins stored in the lower level bucket. ◀

3 De-amortization

In this section, we show how to de-amortize the algorithms presented in the previous section. Intuitively, we show how to distribute the vertex recolorings among the different buckets as each update is handled by the algorithm. We distinguish between the two different strategies.

3.1 Small-buckets algorithm

The de-amortized version of the algorithm essentially splits each operation into two steps: a *link* step, and a *move* step. During the link step we simulate the amortized version, except we do not actually perform any recolorings. Instead, whenever the amortized algorithm would move and recolor a vertex v, we create a *shadow vertex* with the new color at the intended destination, and set it as v's *target*. Then, during the move step, we pick a number of (real) vertices and move and recolor them to their target shadow vertices. Note that if a shadow vertex would be assigned its own target (this happens when multiple levels fill up as the result of the same operation), we instead update the target of the original vertex and remove the old shadow vertex. We ensure that if, at any point, we immediately execute all scheduled moves, the result is identical to the amortized version. The de-amortized version uses the same number of buckets at each level, except for an additional reset bucket. At each stage of the algorithm there will be one primary reset bucket and one secondary reset bucket. Intuitively, the primary reset bucket contains vertices with their "current" coloring, whereas the secondary reset bucket may contain vertices that have not been recolored yet, after the last reset. During a reset, the primary and secondary reset buckets change roles.

As before, we discuss only vertex insertion. The amortized algorithm places a new vertex v in an empty bucket on level 0, and then iteratively merges full levels into higher ones, possibly triggering a reset if the last level fills up. The de-amortized algorithm simulates this during the link step: we create a shadow vertex for v in an empty bucket on level 0. Then, if all other buckets on this level contain vertices that do not have a target (which is equivalent to being full in the amortized version), we create a shadow vertex for each vertex on level 0 (including v's shadow vertex) in an empty bucket on level 1, and color the shadow vertices with the new bucket's colors so that their induced graph is properly colored. We repeat this as long as all buckets on the current level contain a vertex without a target.

If this triggers a reset, we immediately discard all shadow vertices, and create a new shadow vertex in the (empty) secondary reset bucket for each real vertex. We then compute a proper coloring of the entire graph and assign these colors to the shadow vertices. At this point, the primary and secondary reset buckets switch roles. As in the amortized algorithm, we also recompute the value of s, based on the maximum number of vertices thus far. Note that this cannot decrease s. If s increases, we add additional empty buckets and increase the capacity of the current buckets.

All of this happens during the link step. During the move step, we perform the actual recolorings. In particular, we move the modified vertex v into its new bucket at level 0 (or higher, if level 0 filled up) and give it the new color. Then, we move and recolor one vertex from each level to its target. Specifically, we look for buckets without shadow vertices, all of whose vertices have targets. Among those buckets, we pick the bucket with the least number of vertices and move and recolor one of its vertices to its target. Finally, we check the secondary reset bucket for vertices with targets and move and recolor one if found.

Analysis We show correctness by arguing that an empty bucket is available when needed.

▶ **Lemma 3.** *Let t be the number of updates since the last time level i was full, or the last reset, whichever is more recent. Then there is a bucket on level $i + 1$ that does not contain any shadow vertices and contains at most $s^{i+1} - t$ real vertices, each with a target.*

Proof sketch. This follows from our strategy in the moving step. Recall that in each level we move a real vertex to its shadow counterpart from the bucket that contains the fewest real vertices. As we do not add shadow vertices to non-empty buckets the lemma follows. ◀

Combined with the observation that it requires s^{i+1} updates to fill level i (see Lemma 13 in the appendix) we conclude that there is always an empty bucket of level $i > 0$ when needed. In a similar way we can show that there is always an empty bucket at level 0 and an empty reset bucket when a reset occurs (see Lemma 12 and 14 in the appendix).

▶ **Lemma 4.** *When level i fills up, there is an empty bucket on level $i+1$ (for $0 \leq i < d-1$); there is always an empty bucket on level 0; and when a reset occurs, one of the reset buckets is empty.*

Since we have an additional reset bucket, the number of colors is increased by $\mathcal{C}$ compared to the amortized algorithm. During the move step, we recolor at most one vertex that goes to level 0, one from each of the d levels, and one from the secondary reset bucket. Thus, the number of recolorings per operation is the same.

▶ **Theorem 5.** *The de-amortized small-buckets algorithm maintains a proper $(2+d(s-1))\mathcal{C}$-coloring of a $\mathcal{C}$-colorable graph, using at most $d + 2$ vertex recolorings per update.*

3.2 Big-buckets algorithm

The de-amortization technique we use for the big-buckets algorithm is similar to that for the small-buckets algorithm. Each update is split into a link step, in which we link real vertices to shadow vertices in the buckets the amortized algorithm would place them in, and a move step, in which we move and recolor a limited number of vertices to their linked targets.

The main difference is that we need to double all buckets, instead of just the reset bucket. In addition to the primary and secondary reset buckets, each level has a primary bucket, which contains a mix of shadow and real vertices without targets, and a secondary bucket, which contains only real vertices with a target. Thus, if all scheduled moves were executed at once, all vertices would end up in the primary buckets and the secondary buckets would be empty. Intuitively, the primary buckets contain the vertices with their current coloring, while the secondary buckets hold old vertices that have not been recolored yet. The primary and secondary buckets switch roles each time the previous level fills up and adds its vertices to this level. The algorithm guarantees that the secondary bucket is empty at this point. As usual, each bucket has its own set of unique colors.

We maintain a slightly modified *high point invariant*: on level i, the primary bucket never contains more than $s^{i+1} - s^i$ vertices (both real and shadow).

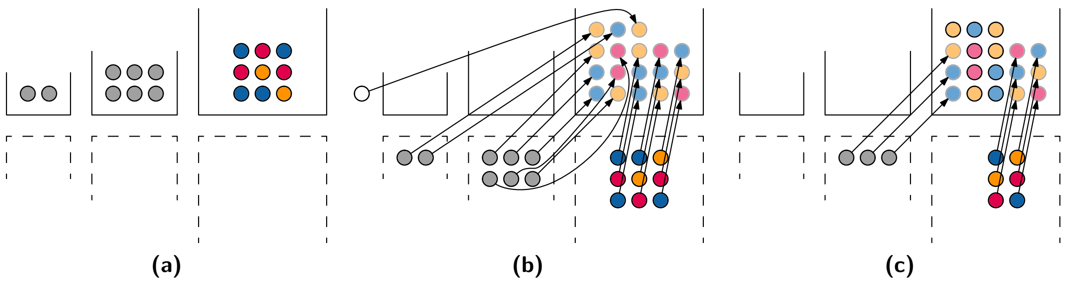

Figure 3 One vertex insertion in the deamortized big-buckets algorithm. (a) Before the update. (b) During the link step, the new vertex causes the first two levels to fill up, linking all vertices to shadow vertices in the secondary third bucket. This swaps the roles of these bucket pairs. (c) The move step moves and recolors up to s vertices from each level.

When a vertex v is inserted, the amortized big-bucket algorithm tries to place it in bucket 0. If that violates the highpoint-invariant, the bucket is emptied into the bucket one level higher. This is repeated until we reach a level i where the high-point invariant is not violated. If such an i does not exist, a reset is triggered.

The de-amortized algorithm simulates this as follows. During the link step, we find the first level i where we can insert the new vertex and all vertices on lower levels without violating the high-point invariant. We link all these vertices, along with the vertices in the primary bucket of level i, to new shadow vertices in the secondary bucket of level i and compute a coloring for them, see Fig. 3. We then switch the roles of the primary and secondary buckets for all levels involved. During the move step, we move and recolor v to its target. In addition, we move and recolor up to s vertices from each level's secondary bucket and the secondary reset bucket to their linked targets.

If we cannot find a level to insert the new vertex without violating the high-point invariant, we reset. This involves discarding all current shadow vertices and linking all vertices to new shadow vertices in the secondary reset bucket, computing a coloring for these vertices, and switching the primary and secondary reset buckets. We also recompute s and update bucket sizes accordingly.

Analysis For correctness, the only thing we need to show is that, when we link vertices to a secondary bucket on level i, that bucket is empty. A similar argument shows that the secondary reset bucket is empty whenever the high point invariant would fail for level $d-1$.

▶ **Lemma 6.** *When an update causes us to create new shadow vertices in the secondary bucket of level i, that bucket is empty.*

Proof. Since the buckets on level 0 contain fewer than s vertices, the move step after each update empties the secondary bucket. For $i > 0$, recall that we only create new shadow vertices if the high point invariant would have been violated at level $i-1$, which means that there are at least $s^i - s^{i-1} + 1$ real vertices in the levels before i. Moreover, none of these vertices have a target, since each level's secondary bucket is empty by induction, and the primary bucket only contains vertices without a target. Recall that, whenever we link vertices to the secondary bucket of level i, we link all of the vertices on the lower levels. Thus, these at least $s^i - s^{i-1} + 1$ vertices were inserted after the secondary bucket was last filled. Since each move step moves s vertices from the secondary bucket into the primary bucket, and since the buckets on level i contain no more than $s^{i+1} - s^i$ vertices by the high point invariant, the secondary bucket will be empty before the current insertion. ◀

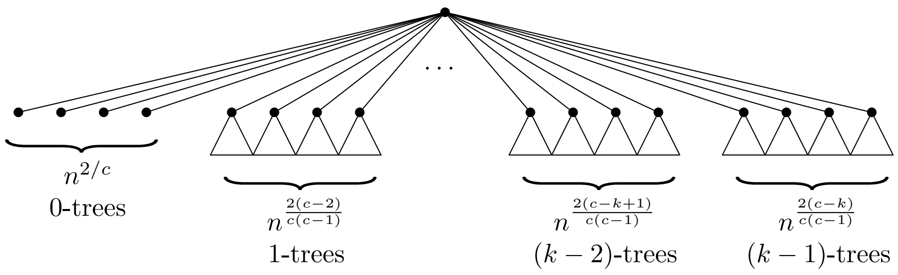
...
$n^{2/c}$
0-trees
$n^{\frac{2(c-2)}{c(c-1)}}$
1-trees
$n^{\frac{2(c-k+1)}{c(c-1)}}$
$(k-2)$-trees
$n^{\frac{2(c-k)}{c(c-1)}}$
$(k-1)$-trees

Figure 4 A k-tree.

The number of colors used is doubled compared to the amortized version, but the number of recolorings per operation is the same: 1 for the inserted vertex, at most $s-1$ for level 0, and at most s for every other level and the reset bucket.

▶ **Theorem 7.** *The de-amortized big-buckets algorithm maintains a $2(d+1)\mathcal{C}$-coloring of a $\mathcal{C}$-colorable graph, using at most $(d+1)s$ vertex recolorings per update.*

4 Lower bounds

In this section, we provide a lower bound construction for an arbitrary (but constant) number of colors c. We say that a vertex is *c-colored* if it has a color in $[c] = \{1,\dots,c\}$. For simplicity of description, we assume in this section that a recoloring algorithm only recolors vertices when there is an edge insertion and not when an edge is deleted as edge deletions do not invalidate the coloring.

As a very broad intuition, our lower bound construction will proceed as follows. We give a construction scheme for an adversary whose goal is to build a so-called c-tree by incrementally constructing k-trees from $k=1$. During this process the adversary monitors the colors of the trees and when certain conditions are violated he will reset the construction back to a previous state. We then provide an argument that charged these “wasted” construction steps to recolorings that the algorithm must have performed to violate the conditions on the k-trees. Next we first provide and details on the construction of k-trees and then proceed to define the color conditions that cause resets of the construction.

4.1 On k-trees

A *0-tree* is a a single node, and for each $1 \le k \le c$, 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 with root r among them. 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 $0 \le j \le k-1$, 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 the colors of 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 it 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.

The idea behind the construction is to construct a constant number of $(c-1)$-trees with the same colors blocked. If we build enough of these (at least $c+1$) there would be at least two with the same color assigned to them. We can then connect their roots with an edge to force the algorithm to invalidate at least one k-tree in one of these $(c-1)$-trees. This is our basic mechanism for forcing the algorithm to recolor vertices.

4.2 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 we say that 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 8.** *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 9.** *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. ◀

4.3 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 9, 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 9, 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 10.** *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.*

4.4 Forcing recolorings

If A stops triggering resets, then at some point we reach a $(c-1)$-configuration. Assuming we choose α sufficiently large, this configuration contains at least $T_{c-1} \geq 2(c+1)$ trees and $c+1$ valid ones. From these valid trees there must be at least 2 that are assigned the same color. By adding an edge between these two the algorithm must invalidate one of the two trees. We can then remove the edge we added and keep picking two $(c-1)$-trees of the same color until we no longer have a valid $(c-1)$-configuration. This triggers a reset and we continue construction again. This guarantees that we can keep forcing A to invalidate trees and trigger resets and the following theorem follows.

▶ **Theorem 11.** *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 Journal on Computing*, 44(1):88–113, 2015.

2 S. Baswana, S. Khurana, and S. Sarkar. Fully dynamic randomized algorithms for graph spanners. *ACM Transactions on Algorithms (TALG)*, 8(4):35, 2012.

3 J. L. Bentley and J. B. Saxe. Decomposable searching problems I: static-to-dynamic transformation. *J. Algorithms*, 1(4):301–358, 1980.

4 P. Borowiecki and E. Sidorowicz. Dynamic coloring of graphs. *Fundamenta Informaticae*, 114(2):105–128, 2012.

5 O. Coudert. Exact coloring of real-life graphs is easy. In *Proceedings of the 34th annual Design Automation Conference*, pages 121–126. ACM, 1997.

6 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.

7 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.

8 A. Dutot, F. Guinand, D. Olivier, and Y. Pigné. On the decentralized dynamic graph-coloring problem. In *Workshop on Complex Systems and Self-Organization Modelling (Cossom 2007), satellite workshop of European Simulation and Modelling Conference (ESM'2007)*, 2007.

9 M. C. Golumbic. *Algorithmic graph theory and perfect graphs*, volume 57 of *Annals of Discrete Mathematics*. Elsevier Science B.V., Amsterdam, second edition, 2004.

10 M. Grötschel, L. Lovász, and A. Schrijver. The ellipsoid method and its consequences in combinatorial optimization. *Combinatorica*, 1(2):169–197, 1981.

11 M. M. Halldórsson. Parallel and on-line graph coloring. *J. Algorithms*, 23(2):265–280, 1997.

12 M. M. Halldórsson and M. Szegedy. Lower bounds for on-line graph coloring. *Theoretical Computer Science*, 130(1):163 – 174, 1994.

13 M. Henzinger, S. Krinninger, and D. Nanongkai. A subquadratic-time algorithm for decremental single-source shortest paths. In *Proceedings of the Twenty-Fifth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2014, Portland, Oregon, USA, January 5-7, 2014*, pages 1053–1072, 2014.

14 J. Holm, K. De Lichtenberg, and M. Thorup. Poly-logarithmic deterministic fully-dynamic algorithms for connectivity, minimum spanning tree, 2-edge, and biconnectivity. *Journal of the ACM (JACM)*, 48(4):723–760, 2001.

15 B. M. Kapron, V. King, and B. Mountjoy. Dynamic graph connectivity in polylogarithmic worst case time. In *Proceedings of the Twenty-Fourth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2013, New Orleans, Louisiana, USA, January 6-8, 2013*, pages 1131–1142, 2013.

16 R. M. Karp. Reducibility among combinatorial problems. In *Complexity of computer computations*, pages 85–103. Plenum, New York, 1972.

17 V. Kumar. Approximating circular arc colouring and bandwidth allocation in all-optical ring networks. In *Approximation algorithms for combinatorial optimization*, volume 1444 of *Lecture Notes in Comput. Sci.*, pages 147–158. Springer, Berlin, 1998.

18 L. Lovász, M. E. Saks, and W. T. Trotter. An on-line graph coloring algorithm with sublinear performance ratio. *Discrete Mathematics*, 75(1-3):319–325, 1989.

19 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.

20 L. Ouerfelli and H. Bouziri. Greedy algorithms for dynamic graph coloring. In *International Conference on Communications, Computing and Control Applications (CCCA)*, pages 1–5, 2011.

21 M. H. Overmars and J. van Leeuwen. Worst-case optimal insertion and deletion methods for decomposable searching problems. *Inf. Process. Lett.*, 12(4):168–173, 1981.

22 D. Preuveneers and Y. Berbers. ACODYGRA: an agent algorithm for coloring dynamic graphs. In *Symbolic and Numeric Algorithms for Scientific Computing*, pages 381–390, 2004.

23 L. Roditty and U. Zwick. Improved dynamic reachability algorithms for directed graphs. In *Foundations of Computer Science, 2002. Proceedings. The 43rd Annual IEEE Symposium on*, pages 679–688. IEEE, 2002.

24 L. Roditty and U. Zwick. Dynamic approximate all-pairs shortest paths in undirected graphs. In *Foundations of Computer Science, 2004. Proceedings. 45th Annual IEEE Symposium on*, pages 499–508. IEEE, 2004.

25 J. B. Saxe and J. L. Bentley. Transforming static data structures to dynamic structures (abridged version). In *20th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 148–168, 1979.

26 M. Thorup. Fully-dynamic min-cut. *Combinatorica*, 27(1):91–127, 2007.

27 S. Vishwanathan. Randomized online graph coloring. *J. Algorithms*, 13(4):657–669, 1992.

28 D. Zuckerman. Linear degree extractors and the inapproximability of max clique and chromatic number. *Theory of Computing*, 3:103–128, 2007.

A Omitted proofs

▶ **Theorem 2.** *The big-buckets algorithm maintains a $(d+1)\mathcal{C}$-coloring of a $\mathcal{C}$-colorable graph, using at most $(d+1)s$ amortized vertex recolorings per update.*

Proof. The bound on the number of colors follows directly from the fact that we use d buckets in addition to the reset bucket. Hence, we use at most $(d+1)\mathcal{C}$ colors at any point in time. We proceed to analyze the number of recolorings per update.

As in the small-buckets algorithm, we give coins to each bucket that we then use to pay for recolorings. In particular, we ensure that bucket i always has $P_i = \lceil k_i/s^i \rceil \cdot s^{i+1} \cdot (d-i)$ coins, where k_i is the number of vertices in bucket i.

Consider what happens when we place a vertex into bucket 0. Initially, the bucket has $P_0 = \lceil k_0/s^0 \rceil \cdot s^1 \cdot (d-0) = k_0 sd$ coins. As a result of the insertion, we need to recolor all k_0+1 vertices, and the invariant requires that the bucket has $(k_0+1)sd$ coins afterwards. By the high point invariant, we have that $1+k_0 \leq s$, so we can bound the number of coins we need to pay per update by $k_0+1+sd \leq (d+1)s$.

Recall that this insertion may trigger a promotion of all vertices in bucket 0 to bucket 1, and that this could propagate until the high point invariant is satisfied again. When we merge bucket i into bucket $i+1$, we need to recolor all vertices in these two buckets. This will be paid for by the coins stored in the smaller bucket. At this point, the high point invariant gives us that bucket i contains at least $k_i \geq s^{i+1} - s^i + 1$ vertices. Thus, since $\lceil k_i/s^i \rceil \geq \lceil (s^{i+1} - s^i + 1)/s_i \rceil = s$, bucket i has at least $P_i = \lceil k_i/s^i \rceil \cdot s^{i+1} \cdot (d-i) \geq s^{i+2} \cdot (d-i)$ coins.

At the same time, bucket $i+1$ has $P_{i+1} = \lceil k_{i+1}/s^{i+1} \rceil \cdot s^{i+2} \cdot (d-i-1)$ coins, and needs to gain at most $s^{i+2} \cdot (d-i-1)$ coins, as it gains at most s^{i+1} vertices. This leaves $s^{i+2} \cdot (d-i) - s^{i+2} \cdot (d-i-1) = s^{i+2}$ coins to pay for the recoloring. Since bucket $i+1$ contained no more than $s^{i+2} - s^{i+1}$ vertices by the high-point invariant, and we added at most s^{i+1} new ones, this suffices to recolor all vertices involved and maintain the coin invariant.

Finally, we perform a reset when bucket $d-1$ overflows. In that case, bucket $d-1$ contains at least $s^d - s^{d-1} + 1$ vertices and therefore has at least $\lceil (s^d - s^{d-1} + 1)/s^{d-1} \rceil \cdot s^d \cdot (d-d+1) = s^{d+1}$ coins. Since the reset bucket contains at most $N_R \leq s^d$ vertices, we need to recolor at most $2s^d$ vertices. As $s = \lceil N_R^{1/d} \rceil \geq 2$ if $N_R \geq 2$, we have enough coins to pay for all these recolorings. Therefore we can maintain the coloring with $(d+1)s$ amortized recolorings per update. ◀

▶ **Lemma 12.** *After every update, there is at least one empty bucket on level* 0.

Proof. Suppose that there is an empty bucket on level 0 before the operation. This is certainly true initially, as all levels start off empty. If, after the link step, there is a vertex on level 0 with a target, that bucket will be empty after the operation, and the property is maintained. If there is no vertex with a target, then there must be at least one more empty bucket, as we would have merged this level into level 1 otherwise. ◀

▶ **Lemma 13.** *At least s^{i+1} updates are required for level i to fill up again after a reset or after it has filled up.*

Proof. Recall that a level is called full when each bucket contains a vertex without a target. When a level fills up, or during a reset, each vertex on the level is assigned a target. For level 0, each update adds a vertex without a target in a new bucket. Thus, the level fills up after exactly s updates. For level $i > 0$, the only way to create a vertex without a target is for level $i-1$ to fill up. By induction, this happens only every s^i updates, and each occurrence creates these vertices in only one bucket. Since there are s buckets on level i, it takes at least s^{i+1} updates for it to fill up. ◀

▶ **Lemma 14.** *When a reset happens, one of the reset buckets is empty.*

Proof. Initially, the entire point set is contained in one of the reset buckets, and the other is empty. Since only a reset can create shadow vertices in a reset bucket, we have at least one empty reset bucket at the first reset.

When a subsequent reset happens, we have performed at least $s^d \geq N_R$ updates in order to fill up level $d-1$. Since for each insertion we move one vertex from the first reset bucket to the second reset bucket and the first reset bucket contains at most N_R vertices, this bucket will be empty when the next reset happens. ◀

▶ **Lemma 8.** *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 $1 \leq i < j < k$.*

Proof. The proof goes by induction on k. For $k = 0$ the results holds trivially. Assume the result holds for $k-1$.

When constructing F_{k-1} from F_k, we know that each $(k-1)$-tree in F_k is assigned the same color c_{k-1}. Moreover, by the induction hypothesis, for each $1 \leq j < k-1$, each core j-tree of a valid $(k-1)$-tree in F_{k-1} had color c_j assigned to it. Thus, each core j-tree of a valid k-tree also has color c_j assigned to it.

We now show that $c_i \neq c_j$ for each $i < j$. Let τ_j be a core j-tree of a valid k-tree in F_k with color c_j. Since every core j-tree of a valid k-tree is also valid, τ_j is a valid j-tree. Therefore, there is a $(j-1)$-child, say r, of τ_k of color c_j. Let τ_{j-1} be the $(j-1)$-subtree of τ_j rooted at r. Since τ_{j-1} has color c_{j-1} assigned to it by the first part of this lemma, we know that its root cannot have color c_{j-1}. Therefore, $c_j \neq c_{j-1}$ and hence, we can assume that $i < j-1$.

By construction and since $i < j-1$, we know that r is also the root of its core i-tree, say τ_i. Because τ_i is valid and has color c_i, it must have an $(i-1)$-child v of color c_i. Since r is the root of τ_i, r and v are adjacent. Because r has color c_j while v has color c_i, and since F_k is c-colored, we conclude that $c_i \neq c_j$. ◀

▶ **Theorem 11.** *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.*

Proof. Use the construction described in this section until m updates have been performed. Let $m' \leq m$ be the number of edge insertions during this sequence of m updates. Notice that $m' \geq m/2$ as an edge can only be deleted if it was first inserted and we start with a graph having no edges.

During the construction, A can be charged with $\Omega(n^{\frac{2}{c(c-1)}})$ vertex recolorings per wasted edge insertion by Lemma 10. Because the graph in our construction consists of at most $O(n)$ edges at all times, at most $O(n)$ of the performed edge insertions are non-wasted. Since every other edge insertion is wasted during a reset, we know that A recolored $\Omega((m'-n) \cdot n^{\frac{2}{c(c-1)}})$ vertices. Because $m' \geq m/2$ and since $m = \Omega(n)$, our results follows. ◀