The Nearest Colored Node in a Tree*

Paweł Gawrychowski† Gad M. Landau‡ Shay Mozes§ Oren Weimann¶

Abstract

We start a systematic study of data structures for the nearest colored node problem on trees. Given a tree with colored nodes and weighted edges, we want to answer queries (v, c) asking for the nearest node to node v that has color c. This is a natural generalization of the well-known nearest marked ancestor problem. We give an $O(n)$-space $O(\log\log n)$-query solution and show that this is optimal. We also consider the dynamic case where updates can change a node's color and show that in $O(n)$ space we can support both updates and queries in $O(\log n)$ time. We complement this by showing that $O(\text{polylog}\, n)$ update time implies $\Omega(\frac{\log n}{\log\log n})$ query time. Finally, we consider the case where updates can change the edges of the tree (link-cut operations). There is a known (top-tree based) solution that requires update time that is roughly linear in the number of colors. We show that this solution is probably optimal by showing that a strictly sublinear update time implies a strictly subcubic time algorithm for the classical all pairs shortest paths problem on a general graph. We also consider versions where the tree is rooted, and the query asks for the nearest ancestor/descendant of node v that has color c, and present efficient data structures for both variants in the static and the dynamic setting.

Dedicated to Professor Costas S. Iliopoulos, on the occasion of his 60^{th} *birthday.*

1 Introduction

We consider a number of problems on trees with colored nodes. Each of these problems can be either static, meaning the color of every node of a tree T on n nodes is fixed, or dynamic, meaning that an update can change a node's color (but the tree itself does not change). The edges of T may have arbitrary nonnegative lengths and $\mathsf{dist}(u, v)$ denotes the total length of the unique path connecting u and v. Depending on the version of the problem, given a node u and a color c we are interested in:

The nearest colored ancestor: the first node v on the u-to-root path that has color c.

The nearest colored descendant: the node v of color c such that the v-to-root path goes through u and the distance from u to v is as small as possible.

The nearest colored node: the node v of color c such that the distance from u to v is as small as possible.

*A preliminary version of this paper appeared in CPM 2016.

†University of Haifa, gawry@mimuw.edu.pl. Partially supported by Israel Science Foundation grant 794/13.

‡University of Haifa and New York University, landau@cs.haifa.ac.il. Partially supported by ISF grant 571/14 and BSF grant 2014028.

§IDC Herzliya, smozes@idc.ac.il. Partially supported by Israel Science Foundation grant 794/13.

¶University of Haifa, oren@cs.haifa.ac.il, Partially supported by Israel Science Foundation grant 794/13.

In the static case, if the number of colors is k, then there is a trivial solution for all three problems with $O(nk)$-space and $O(1)$-query (in fact, for $k \leq \log n$ there is an $O(n)$-space and $O(1)$-query solution [8]). In the word RAM model with word size w, the nearest colored ancestor problem can be solved in $O(n + nk/w)$-space and $O(1)$-query [7]. For an arbitrary number of colors, a lower bound of $\Omega(\log\log n)$-query for any $O(n \operatorname{polylog} n)$-space solution to each of these problems (in fact, even on strings) follows from a simple reduction from the well known predecessor problem. For the nearest colored ancestor problem, a tight $O(n)$-space $O(\log\log n)$-query solution was given by Muthukrishnan and Müller [17]. We show how to achieve these same bounds for the other two problems (for completeness, we also describe a simple nearest colored ancestor solution). To achieve this, as was done in [17], for every color c we construct a separate tree $T(c)$. If there are total s nodes of color c then $T(c)$ is only of size $O(s)$ but (after augmenting it with appropriate additional data) it captures for all n nodes of the original tree their nearest node of color c.

In the dynamic case, the nearest colored ancestor problem has been studied by Alstrup-Husfeldt-Rauhe [3] who gave a solution with $O(n)$-space, $O(\frac{\log n}{\log\log n})$-query, and $O(\log\log n)$-update. They also gave a lower bound stating that $O(\operatorname{polylog} n)$-update requires $\Omega(\frac{\log n}{\log\log n})$-query. This holds even when the number of colors is only two (then a node is either marked or unmarked and the problem is known as the marked ancestor problem). We show that this lower bound (with the same statement and only two colors) extends to both the nearest colored node and the nearest colored descendant. For upper bounds, we show that the nearest colored node problem can be solved with $O(n)$-space, $O(\log n)$-update, and $O(\log n)$-query. Our solution can be seen as a variant of the centroid decomposition tweaked to guarantee some properties of top-trees.

The original top-trees of Alstrup-Holm-de Lichtenberg-Thorup [2] were designed for only two colors (i.e., for the nearest marked node problem). They achieve $O(\log n)$ query and update and also support updates that insert and delete edges (i.e., maintain a forest under link-cut operations). The straightforward generalization of top-trees from two to k colors increases the space dramatically to $O(nk)$. We believe it is possible to improve this to $O(n)$ using similar ideas to those we present here. However, because we do not allow link-cut operations, compared to top-trees our solution is simpler. Moreover, our query time can be improved to (optimal) $O(\frac{\log n}{\log\log n})$ at the cost of increasing the update time by a $\log^{\epsilon} n$ factor and the space by a $\log^{1+\epsilon} n$ factor. Whether such an improvement is possible with top trees remains open. We note that in both the $O(nk)$ and the $O(n)$ space solutions with top-trees, while queries and color-changes require $O(\log n)$ time the time for link/cut is $O(k \cdot \log n)$. This can be significant since k can be as large as n (we emphasise that our solution does not support link/cut at all). We show that $\tilde{O}(k)$ is probably optimal by showing that $O(k^{1-\varepsilon})$ query and update time implies an $O(n^{3-\varepsilon})$ solution for the classical all pairs shortest paths problem on a general graph with n vertices. The non existence of such an algorithm has recently been widely used as an assumption with various consequences [22].

Finally, for the nearest colored descendant problem, we give a solution with $O(\frac{\log n}{\log\log n})$-query and $O(\log^{2/3+\epsilon} n)$-update by reducing the problem to 3-sided emptiness queries on points in the plane. We then show that the $O(\operatorname{polylog} n)$-update $\Omega(\frac{\log n}{\log\log n})$-query lower bound of [3] also applies to the nearest colored descendant problem by giving a reduction from nearest colored ancestor to nearest colored descendant.

Related work. The approximate version of the nearest colored node problem (where we settle for approximate distances) has recently been studied (as the vertex-to-label distance query problem) in general graphs [9, 13, 15] and in planar graphs [1, 14, 16]. In fact, the query-time in [16] is dominated by a $O(\log\log n)$ nearest colored node query on a string (which we now know is optimal).

Preliminaries. A *predecessor* structure is a data structure that stores a set of n integers $S \subseteq [0, U]$, so that given $x \in [0, U]$ we can determine the largest $y \in S$ such that $y \leq x$. It is known [18] that for $U = n^2$ any predecessor structure of $O(n \operatorname{polylog} n)$-space requires $\Omega(\log \log n)$-query, and that linear-size structures with such query-time exist [19, 21].

A *Range Minimum Query* (RMQ) structure on an array $A[1, \ldots, n]$ is a data structure for answering queries $\min\{A[i], \ldots, A[j]\}$. When the array A is static, RMQ can be optimally solved in $O(n)$-space and $O(1)$ query [5, 6, 12]. In the dynamic case, we allow updates that change the value of array elements. When the query range is restricted to be a suffix $A[i, \ldots, n]$ we refer to the problem as the Suffix Minimum Query (SMQ) problem.

A *Lowest Common Ancestor* (LCA) structure on a rooted tree T is a data structure for finding the common ancestor of two nodes u, v with the largest distance from the root. For static trees, LCA is equivalent to RMQ and thus can be solved in $O(n)$-space and $O(1)$-query.

A *perfect hash* structure stores a collection of n integers S. Given x we can determine if $x \in S$ and return its associated data. There exists $O(n)$-space, $O(1)$-query perfect hash structure [11], which can be made dynamic with $O(1)$-update (expected amortized) [10].

2 Static Upper Bounds

We root the tree at node 1 and assign pre- and post-order number $\mathsf{pre}(u), \mathsf{post}(u) \in [1, 2n]$ to every node u. All these numbers are distinct, $[\mathsf{pre}(u), \mathsf{post}(u)]$ is a laminar family of intervals, and u is an ancestor of v if and only if $\mathsf{pre}(v) \in (\mathsf{pre}(u), \mathsf{post}(u))$. We order edges outgoing from every node according to the preorder numbers of the corresponding nodes.

We assume the colors are represented by integers in $[1, n]$. We will construct a separate additional structure for every possible color c. The size of the additional structure will be always proportional to the number of nodes of color c, which sums up to $O(n)$ over all colors c. Below we describe the details of the additional structure for every version of the problem.

Nearest colored descendant. Let $v_1, v_2, \ldots, v_s$ be all nodes of color c sorted so that $\mathsf{pre}(v_1) < \mathsf{pre}(v_2) < \ldots < \mathsf{pre}(v_s)$. We insert the preorder numbers of all these nodes into a predecessor structure, so that given an interval $[x, y]$ we can determine the range $v_i, v_{i+1}, \ldots, v_j$ consisting of all nodes with preorder numbers from $[x, y]$ in $O(\log \log n)$ time. Additionally, we construct an array $D[1..s]$, where $D[i] = \mathsf{dist}(1, v_i)$. The array is augmented with an RMQ structure. To answer a query, we use the predecessor structure to locate the range consisting of nodes v such that $\mathsf{pre}(v) \in [\mathsf{pre}(u), \mathsf{post}(u)]$. Then, if the range is nonempty, a range minimum query allows us to retrieve the nearest descendant of u with color c. The total query time is hence $O(\log \log n)$.

Nearest colored ancestor. Let $v_1, v_2, \ldots, v_s$ be all nodes of color c. We insert all their pre- and postorder numbers into a predecessor structure. Additionally, for every i we store (in an array) the nearest ancestor with the same color for the node v_i (or null if such ancestor does not exist). To answer a query, we use the predecessor structure to locate the predecessor of $\mathsf{pre}(u)$. There are two cases:

1. The predecessor is $\mathsf{pre}(v_i)$ for some i. Because $[\mathsf{pre}(v), \mathsf{post}(v)]$ create a laminar family, either $\mathsf{pre}(u) \in [\mathsf{pre}(v_i), \mathsf{post}(v_i)]$ and v_i is the answer, or u has no ancestor of color c.

2. The predecessor is $\mathsf{post}(v_i)$ for some i. Consider an ancestor u' of u with the same color. Then $\mathsf{pre}(u') < \mathsf{post}(v_i)$, so u' is also an ancestor of v_i. Similarly, consider an ancestor v' of v_i with

the same color, then $\mathsf{post}(v') > \mathsf{pre}(u)$ so v' is also an ancestor of u. Therefore, the nearest ancestor of color c is the same for u and v_i, hence we can return the answer stored for v_i.

The query time is hence again $O(\log \log n)$.

Nearest colored node. We define the subtree induced by color c, denoted $T(c)$, as follows. Let $v_1, v_2, \ldots, v_s$ be all nodes of color c. $T(c)$ consists of all nodes v_i together with the lowest common ancestor of every pair of nodes v_i and v_j. The parent of $u \in T(c)$ is defined as the nearest ancestor v of $u \in T$ such that $v \in T(c)$ as well; if there is no such node, u is the root of $T(c)$ (there is at most one such node). Thus, an edge $(u, v) \in T(c)$ corresponds to a path from u to v in T.

Lemma 2.1. *$T(c)$ consists of at most $2s - 1$ nodes and can be constructed in $O(s)$ time assuming that we are given a list of all nodes of color c sorted according to their preorder numbers and a constant time LCA built for T.*

Proof. Let $v_1, v_2, \ldots, v_s$ be the given list of nodes of color c. By assumption, $\mathsf{pre}(v_1) < \mathsf{pre}(v_2) < \ldots < \mathsf{pre}(v_s)$. We claim that $T(c)$ consists of all nodes v_i and the lowest common ancestor of every v_i and v_{i+1}. To prove this, consider two nodes v_i and v_j such that $i < j$ such that their lowest common ancestor u is different than v_i and v_j. Then, u is a proper ancestor of v_i and v_j, and furthermore v_i is a descendant of u_a and v_j a descendant of u_b, where $a < b$ and $u_1, u_2, \ldots, u_\ell$ is an ordered list of the children of u. v_i can be replaced by the node $v_{i'}$ of color c with the largest preorder number in the subtree rooted at u_a. Then the lowest common ancestor of $v_{i'}$ and $v_{i'+1}$ is still u, so it is indeed enough to include only the lowest common ancestor of such pairs of nodes and the bound of $2s - 1$ follows.

To construct $T(c)$ we need to determine its set of nodes and edges. Determining the nodes is easy by the above reasoning. To determine the edges, we use a method similar to constructing the Cartesian tree of a sequence: we scan $v_1, v_2, \ldots, v_s$ from the left to right while maintaining the subtree induced by $v_1, v_2, \ldots, v_i$. We keep the rightmost path of the current subtree on a stack, with the bottommost edge on the top. To process the next v_{i+1}, we first calculate its lowest common ancestor with v_i, denoted x. Then, we pop from the stack all edges (u, v) such that u and v are both below (or equal to) x in T. Finally, we possibly split the edge on the top of the stack into two and push a new edge onto the stack. The amortized complexity of every step is constant, so the total time is $O(s)$. □

The first part of the additional structure is the nearest node of color c stored for every node of $T(c)$. Given a node u, we need to determine its nearest ancestor u' such that $u' \in T(c)$ or u' lies strictly inside some path corresponding to an edge of $T(c)$. In the latter case, we want to retrieve the endpoints of that edge. This is enough to find the answer, as any path from u to a node of color c must necessarily go through u' (because u' is the lowest ancestor of u such that the subtree rooted at u' contains at least one node of color c, and a simple path from u to a node of color c must go up as long as the subtree rooted at the current node does not contain any node of color c), and then either $u' \in T(c)$ and we have the answer for u' or the path continues towards one of the endpoints of the edge of $T(c)$ strictly containing u' (because the subtrees hanging off the inside of a path corresponding to an edge of $T(c)$ do not contain any nodes of color c). Hence after having determined u' we need only constant time to return the answer.

To determine u', we use the structure for the nearest colored ancestor constructed for a subset of $O(s)$ marked nodes of T. These marked nodes are all nodes of T corresponding to the nodes of $T(c)$, and additionally, for every path $u_1 \to u_2 \to \ldots \to u_\ell$ corresponding to an edge of $T(c)$, the node u_2 (where u_1 is closer to the root than u_ℓ and $\ell \geq 2$). For every marked node of the second type we

store the endpoints (u_1, u_ℓ) of its corresponding edge of $T(c)$. Then, locating the nearest marked ancestor of u allows us to determine that the sought nearest ancestor u' is a node of $T(c)$, or find the edge of $T(c)$ strictly containing u'. By plugging in the aforementioned structure for the nearest colored ancestor, we obtain the answer in $O(\log\log n)$ time with a structure of size $O(s)$.

This concludes the description of our static solution. Before moving on to the dynamic case, we note that the above solution can be easily extended to the case where every node $v \in T$ has an associated set of colors $\mathsf{C}(v)$ and instead of looking for a node of color c we look for a node v such that $c \in \mathsf{C}(v)$.

3 Dynamic Upper Bounds

In the dynamic setting, we allow updates to change a node's color. To be even more general, we assume that every node $v \in T$ has an associated (dynamically changing) set of colors $\mathsf{C}(v)$, and an update can either insert or remove a color c from the current set $\mathsf{C}(v)$.

Nearest colored descendant. As in the static case, we construct a separate structure for every possible color c. We also maintain a mapping from the set of colors to their corresponding structures. Let $v_1, v_2, \ldots, v_s$ be all nodes of color c. We create a set of points of the form $(\mathsf{pre}(v_i), \mathsf{dist}(1, v_i))$. Then, a nearest colored descendant query can be answered by locating the point with the smallest y-coordinate in the slab $[\mathsf{pre}(u), \mathsf{post}(u)] \times (-\infty, \infty)$. We store the points in a fully dynamic 3-sided emptiness structure of Wilkinson [20]. The structure answers a 3-sided emptiness query by locating the point with the smallest y-coordinate in a slab $[x_1, x_2] \times (-\infty, \infty)$ in $O(\frac{\log n}{\log\log n})$ time and can be updated by inserting and removing points in $O(\log^{2/3+\epsilon} n)$ time, with both the update and the query time being amortized. Consequently, we obtain the same bounds for the nearest colored descendant.

Nearest colored ancestor. This has been considered by Alstrup-Husfeldt-Rauhe [3]. The query time is $O(\frac{\log n}{\log\log n})$ and the update time $O(\log\log n)$. While not explicitly stated in the paper, the total space is linear.

Nearest colored node. Our data structure is based on a variant of the centroid decomposition. That is, we recursively decompose the tree into smaller and smaller pieces by successively removing nodes. The difference compared to the standard centroid decomposition is that each of the obtained smaller trees has up to two appropriately defined boundary nodes (similarly to the decomposition used in top-trees).[1]

The basis of our recursive decomposition is the following well-known fact.

Fact 1. *In any tree T on n nodes there exists a node $c \in T$ such that $T \setminus \{c\}$ is a collection of trees of size at most $\frac{n}{2}$ each.*

We apply it recursively. The input to a single step of the recursion is a tree T on n nodes with at most two distinguished *boundary* nodes. We use Fact 1 to find node $c_1 \in T$ such that $T \setminus \{c_1\}$ is a collection of smaller trees $T_1, T_2, \ldots$. Each neighbor of c_1 in the original tree becomes a boundary node in its smaller tree T_i. A boundary node $u \in T$ such that $u \neq c_1$ is also a boundary node in its smaller tree T_i. Because T contains at most two boundary nodes, at most one smaller tree T_i contains three boundary nodes, while all other smaller trees contain at most two boundary nodes. If

[1] Using standard centroid decomposition leads to update time of $O(\log^2 n)$, compared to $O(\log n)$ when controlling the number of boundary nodes.

such T_i containing three boundary nodes u_1, u_2, u_3 exists, we further partition it into even smaller trees $T'_1, T'_2, \ldots$. This is done by finding a node $c_2 \in T_i$ which, informally speaking, separates all u_1, u_2, u_3 from each other. Formally speaking, we take c_2 to be any node belonging to all three paths $u_1 - u_2, u_1 - u_3, u_2 - u_3$ (intersection of such three paths is always nonempty). Then, $T_i \setminus \{c_2\}$ is a collection of trees $T'_1, T'_2, \ldots$ such that each T'_j contains at most one of the nodes u_1, u_2, u_3. Finally, each neighbor of c_2 in T_i becomes a boundary node in its smaller tree T'_j; see Figure 1.

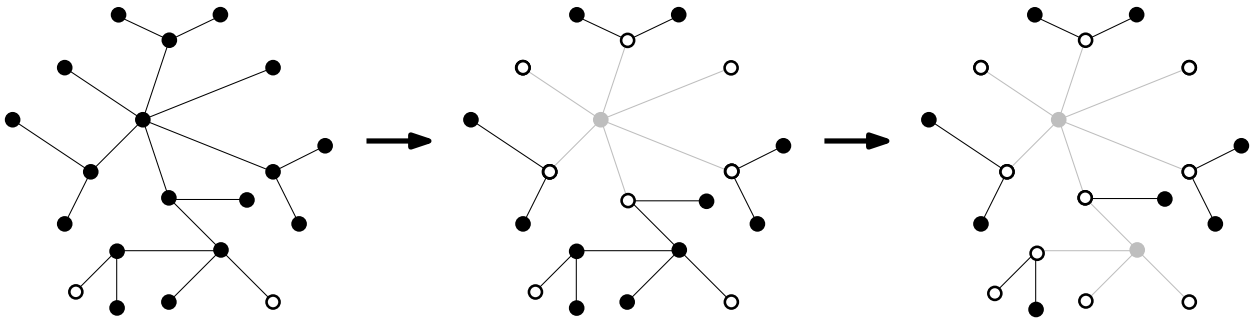

Figure 1: Schematic depiction of a single step of our centroid decomposition. After removing the grayed out node and its adjacent edges we obtain 6 pieces. One of them contains three boundary nodes and hence needs to be further partitioned into 4 smaller pieces.

Lemma 3.1. *Given a tree T on n nodes with at most two boundary nodes b_1, b_2, we can find two nodes $c_1, c_2 \in T$, called the centroids of T, such that $T \setminus \{c_1, c_2\}$ is a collection of trees $T_1, T_2, \ldots$ with the property that each T_i consists of at most $\frac{n}{2}$ nodes and contains at most two boundary nodes, which are defined as nodes corresponding to the original boundary nodes of T or nodes adjacent to c_1 or c_2 in T.*

Let T_0 denote the original input tree. We apply Lemma 3.1 to T_0 recursively until the tree is empty. The resulting recursive decomposition of T_0 can be described by a decomposition tree $\mathcal{T}$ as followed. Each node of $\mathcal{T}$ corresponds to a subtree of T_0. The root r of $\mathcal{T}$ corresponds to T_0. The children of a node $u \in \mathcal{T}$, whose corresponding subtree of T_0 is T, are the recursively defined decomposition trees of the smaller trees T_i obtained by removing the centroid nodes from T with Lemma 3.1. For a node $u \in \mathcal{T}$ whose corresponding subtree is T we define $\mathsf{C}(u)$ to be $\mathsf{C}(c_1) \cup \mathsf{C}(c_2)$, where c_1 and c_2 are the centroids of T. Because the size of the tree decreases by a factor of two in every step, the depth of $\mathcal{T}$ is at most $\log n$. We will sometimes abuse notation and say that a tree T in the decomposition is the parent of T' if the node of $\mathcal{T}$ whose corresponding tree is T is the parent of the node of $\mathcal{T}$ whose corresponding tree is T'. This concludes the description of our recursive decomposition.

We now describe the information maintained in order to implement dynamic nearest colored node queries. We will henceforth assume that the degree of every node in T is at most 3. This can be achieved by standard ternarization with zero length edges. For every tree T in the decomposition, every boundary node b of T, and every color c such that $c \in \mathsf{C}(v)$ for some $v \in T$, we store the node of T with color c that is nearest to b. Observe that, since the degree is bounded, this information can be used to compute in constant time the nearest node with color c to each centroid c_i of T, by considering the nearest nodes with color c to each of the adjacent (to c_1 or to c_2) boundary nodes of the children T_i of T in $\mathcal{T}$.

For every node $v \in T_0$ we store a pointer to the unique node of $\mathcal{T}$ in which v is a centroid. We also preprocess the original tree T_0 so that the distance between any two nodes can be calculated in

constant time: we root the tree at node 1, construct an LCA structure, and store $\mathsf{dist}(1, v)$ for every $v \in T_0$. Such preprocessing actually allows us to compute the distance between any two nodes in any of the smaller trees in the decomposition.

Queries. Given a tree T in the decomposition, a node $v \in T$ and a color c, we need to find the node $u \in T$ with color c that is nearest to v.

Let c_i $(i = 1, 2)$ be the centroids of T. Either some c_i lies on the v-to-u path in T, or v and u belong to the same child T_i of T. In the former case u is the closest node to c_i in T with color c. Note that this information is already stored. In the latter case, the query is reduced to a query in T_i.

It follows that, in order to find the closest node to v with color c in T_0, it suffices to consider the closest nodes with color c to each of the centroids of each of the trees on the path in $\mathcal{T}$ from the node of $\mathcal{T}$ in which v is a centroid to the root of $\mathcal{T}$. There are $O(\log n)$ such centroids, and each of them can be checked in constant time using the stored information.

Updates. Consider adding or removing color c from $\mathsf{C}(v)$. We implement the updates in a bottom-up fashion along the same path used for the query. Subtrees on this path are the only ones in the decomposition containing v, so only their information should be updated.

Repairing the information for the boundary nodes of a subtree T along the path in $\mathcal{T}$ is done in a similar manner to that of the query. For each boundary node b_i of T $(i = 1, 2)$, we need to find the nearest node $u \in T$ with color c. Let c_j $(j = 1, 2)$ be the centroids of T. Let T_i denote the child of T in $\mathcal{T}$ that contains b_i. Either b_i and u both belong to T_i, or $u = c_j$ for some j, or u is in some other child T_ℓ of T and some c_j lies on the b_i-to-u path in T. In all cases we can use the information stored at the children of T to correctly determine the information stored at T. If $b_i, u \in T_i$ then b_i is a boundary node of T_i, so we use the information stored for T_i. If $u = c_j$ then we verify that $c \in \mathsf{C}(c_j)$. Finally, in the last case, the closest node to b_i with color c in T_ℓ is also the closest node to the boundary node of T_ℓ adjacent (in T) to c_j, so we use the information stored for T_ℓ.

Summary. To summarize, both the query and the update time is $O(\log n)$. The space is $O(\log n \cdot \sum_{v \in T_0} |\mathsf{C}(v)|)$, because every $c \in C(v)$ contributes constant space at every level.

Decreasing the space. The space can be reduced to $O(n + \sum_{v \in T_0} |\mathsf{C}(v)|)$. Let T be a tree in the decomposition. Recall that for each boundary node $u \in T$ and color c such that $c \in \mathsf{C}(v)$ for some $v \in T$ we maintain the nearest node of T with color c. Hence, every $c \in \mathsf{C}(v)$ might contribute constant space at every tree T such that $v \in T$. Now we describe how this can be avoided by maintaining, for every color c, a separate structure of size proportional to the number of nodes with color c.

Recall that we extend the colors of nodes in the original tree T_0 to color sets of nodes of the decomposition tree $\mathcal{T}$. For a node $u \in \mathcal{T}$ that is associated with subtree T of T_0 we define u's color set to be the union of the color sets of the centroids of T. For every color c we maintain the subtree of $\mathcal{T}$ induced by color c (cf. Section 2 for definition of induced), denoted $\mathcal{T}(c)$. Before we describe how these subtrees can be efficiently maintained, we describe how to use $\mathcal{T}(c)$ instead of $\mathcal{T}$ to perform queries and updates.

Consider a query (v, c) and let u be the node of $\mathcal{T}$ in which v is a centroid. The query traverses the ancestors of u. At each such ancestor $u' \in \mathcal{T}$, we iterate through the centroids c_i $(i = 1, 2)$ and consider their nearest node with color c as candidate for the answer. The nearest node is either the centroid itself, or the nearest node with color c to a boundary node of a child $u'' \in \mathcal{T}$ of u'. In the former case, $u' \in \mathcal{T}(c)$. In the latter case, $u' \notin \mathcal{T}(c)$. If also $u'' \notin \mathcal{T}(c)$ then, by

definition of $\mathcal{T}(c)$, $c \notin \mathsf{C}(u'')$ and u'' has at most one child $u''' \in \mathcal{T}$ containing nodes with color c in its corresponding subtree of T. Hence instead of iterating through the boundary nodes of u'' we can iterate through the boundary nodes of u'''. By repeating this reasoning, u'' can be replaced by its highest descendant belonging to $\mathcal{T}(c)$ (such highest descendant is uniquely determined, unless the subtree of T corresponding to u'' has no nodes with color c). Consequently, the queries can be modified to operate on $\mathcal{T}(c)$ instead of $\mathcal{T}$: we locate the first ancestor $u' \in \mathcal{T}$ of u such that $u' \in \mathcal{T}(c)$ (if there is none, we take the root of $\mathcal{T}(c)$ as u'), and then iterate through all ancestors of u' in $\mathcal{T}(c)$. For each such ancestor u'', we consider as candidates for the answer its centroid nodes c_i ($i = 1, 2$) and also the nearest node with color c to every boundary node of each child of u'' in $\mathcal{T}(c)$. The same reasoning allows us to recalculate, upon an update, the information stored at $u \in \mathcal{T}(c)$ using the information stored at all of its children in $\mathcal{T}(c)$.

By Lemma 2.1, the size of the subtree induced by color c is at most $2|\{v \in T : c \in \mathsf{C}(v)\}| - 1$. Summing over all colors we obtain that the total size of all induced subtrees is $2\sum_{v \in T_0} |\mathsf{C}(v)|$. We still need to show how to maintain them and also how to efficiently locate the first ancestor $u' \in \mathcal{T}$ of u such that $u' \in \mathcal{T}(c)$. The latter is implemented with a nearest colored ancestor structure. We only describe how to update $\mathcal{T}(c)$ after adding c to some $\mathsf{C}(v)$, where $v \in T_0$, and do not change $\mathcal{T}(c)$ after removing c (so our trees will be in fact larger than necessary). Whenever the total size of all maintained subtrees exceeds $4\sum_{v \in T_0} |\mathsf{C}(v)|$, we rebuild the whole structure. This does not increase the amortized complexity of an update and can be deamortized using the standard approach of maintaining two copies of the structure.

After adding c to some $\mathsf{C}(v)$, where $v \in T_0$, we might also need to include c in $\mathsf{C}(u)$ for some $u \in \mathcal{T}$, thus changing $\mathcal{T}(c)$. Inspecting the proof of Lemma 2.1 we see that the change consists of two parts: we need to include u in $\mathcal{T}(c)$, and in particular insert it onto the sorted list of nodes of $\mathcal{T}$ of color c. Then, we might also need to include the lowest common ancestor of u and its predecessor on the list, and also the lowest common ancestor of u and its successor there. We implement the list with a balanced search tree, so that all these new nodes can be generated in $O(\log n)$ time. We also need to generate new edges (or, more precisely, split some existing edges into two and possibly attach a new edge to the new middle node). This is easy to do if we are able to efficiently find the edge of $\mathcal{T}(c)$ corresponding to a path containing a given node $u \in \mathcal{T}$. To this end, we also maintain a list of all nodes of $\mathcal{T}(c)$ sorted according to their preorder numbers in $\mathcal{T}$. Then binary searching over the list gives us the highest descendant of u belonging to $\mathcal{T}(c)$. By implementing the list with a balanced search tree we can hence find such an edge in $O(\log n)$ time. Thus, the update and the query time is still $O(\log n)$ and the space linear.

Decreasing the query time. The query time can be decreased to $O(\frac{\log n}{\log \log n})$, which is optimal, at the cost of increasing the update time to $O(\log^{1+\epsilon} n)$ and the space to $O(\log^{1+\epsilon} n \sum_{v \in T} |\mathsf{C}(v)|)$.

For trees of constant degree, Lemma 3.1 decomposes T into a constant number of trees, each of size at most $\frac{n}{2}$, by removing at most two nodes. By iterating the lemma $\epsilon \log \log n$ times we obtain the following.

Lemma 3.2. *Given a tree T on n nodes with at most two boundary nodes, we can find $O(\log^{\epsilon} n)$ centroid nodes $c_1, c_2, \ldots \in T$ such that $T \setminus \{c_1, c_2, \ldots\}$ is a collection of trees $T_1, T_2, \ldots$ with the property that each T_i consists of at most $\frac{n}{\log^{\epsilon} n}$ nodes and contains at most two boundary nodes, which are defined as nodes corresponding to the original boundary nodes of T or nodes adjacent to any c_i in T.*

We apply Lemma 3.2 recursively. Now the depth of the recursion is $O(\frac{\log n}{\log \log n})$.

Note that, because the number of centroids c_i and trees T_i is no longer constant, it is no longer true that the nearest node to centroid c_i with color c in T can be computed in $O(1)$ time from the

information stored for boundary nodes of the T_is. Therefore, to implement query (v, c) in $O(\frac{\log n}{\log \log n})$ time, we maintain explicitly, for each centroid node c_i, its nearest node of T with color c. This allows us to process the case when $v = c_i$ in constant time. If v is not a centroid of T, then $v \in T_j$ for some j. We recurse on T_j. The only remaining possibility is that the sought node u does not belong to T_j. In such case, the path from v to u must go through one of the boundary nodes of T_j. Each of these boundary nodes is adjacent to a constant number of the centroid nodes c_i of T (because of the constant degree assumption). We iterate through every such centroid c_i and consider its nearest node with color c as a candidate for the answer in constant total time.

Implementing updates is again done in a bottom-up fashion. However, now we also need to recalculate the nearest node with color c to every centroid node c_i. Recalculating the nearest node with color c (to either a boundary or a centroid node) takes now $O(\log^\epsilon n)$ time, because we need to consider boundary nodes of up to $O(\log^\epsilon n)$ subtrees T_i and also $O(\log^\epsilon n)$ centroid nodes. Hence the total update time at every level of recursion is $O(\log^{2\epsilon} n)$. By adjusting ϵ we get that the total update time is $O(\log^{1+\epsilon} n)$.

4 Lower Bounds

Static nearest colored node, descendant, and ancestor. First we consider the static nearest colored node. In such case, there is a lower bound stating that $O(n \operatorname{polylog} n)$ space requires $\Omega(\log \log n)$ query time. In fact, the lower bound already applies for paths, and follows easily from Belazzougui and Navarro [4]: they show (via reduction from predecessor [18]) that any data structure that uses $O(n \operatorname{polylog} n)$ space to represent a string S of length n over alphabet $\{1, \ldots, n\}$ must use time $\Omega(\log \log n)$ to answer rank queries. A $\mathsf{rank}_\sigma(i)$ query asks for the number of times the letter σ appears in $S[1, \ldots, i]$. The reduction to nearest colored node is trivial: each letter corresponds to a color, we create a path on n nodes where the color of the i-th node is $S[i]$, and additionally the node stores $\mathsf{rank}_{S[i]}(i)$. Then, to calculate an arbitrary $\mathsf{rank}_\sigma(i)$, we consider the i-th node and find its nearest node of color σ. Then, if that nearest node is on the left of i, we return its stored answer, and otherwise we return its stored answer decreased by one. This also shows that one cannot beat $O(\log \log n)$ time with a structure of size $O(n \operatorname{polylog} n)$ for the static nearest colored descendant and ancestor.

In all dynamic problems, the lower bounds hold even if we have only two colors, that is, every node is marked or not.

Dynamic nearest marked node and ancestor. We next show that the following lower bound of Alstrup-Husfeldt-Rauhe [3] for marked ancestor also applies to dynamic nearest marked node. Notice that Theorem 4.1 implies that any $O(\operatorname{polylog} n)$ update time requires $\Omega(\frac{\log n}{\log \log n})$ query time. In the marked ancestor problem, the query is to detect if a node has a marked ancestor, and an update marks or unmarks a node, so we immediately obtain a lower bound for the dynamic nearest marked ancestor.

Theorem 4.1 ([3]). *For the marked ancestor problem, if t_u is the update time and t_q is the query time then*

$$t_q = \Omega\left(\frac{\log n}{\log t_u + \log \log n}\right)$$

The lower bound holds under amortization and randomization.

The proof of Theorem 4.1 uses a (probabilistic) sequence of operations (mark/unmark/marked ancestor query) on an unweighted complete tree T on n leaves and out-degree ≥ 2. To show that the

bounds of Theorem 4.1 also apply to dynamic nearest marked node, we add edge weights to T that increase exponentially with depth: edges outgoing from a node at depth d has weight 2^d. This way, if a node has a marked ancestor, then its nearest marked node is necessarily the nearest marked ancestor (because in the worst case the distance to the nearest marked ancestor is $2^0 + 2^1 + \ldots + 2^{d-1}$, while the distance to any proper descendant is at least 2^d). Hence the marked ancestor problem reduces to nearest marked node. In fact, it is possible to achieve a reduction without using weights by replacing each weight W with a path of W nodes. Since T is balanced, this will increase the space of T to be $O(n^2)$ which is fine since the bound of Theorem 4.1 is independent of space.

Dynamic nearest marked descendant. We next show that the bounds of Theorem 4.1 also apply to the case of nearest marked *descendant*. This requires three simple reductions:

1. dynamic existential marked ancestor $\rightarrow$ planar dominance emptiness.

   Dynamic existential marked ancestor is a simpler variant of the dynamic marked ancestor problem where a query does not need to find the nearest marked ancestor but only to report if there exists a marked ancestor. In fact, the proof [3] of the lower bound of Theorem 4.1 is for the dynamic existential marked ancestor problem. In the planar dominance emptiness problem, we need to maintain a set $S \subseteq [n]^2$ of points in the plane under insertions and deletions, such that given a query point (x, y) we can determine if there exits a point (x', y') in S that dominates (x, y) (i.e., $x' \geq x$ and $y' \geq y$). As shown in [3], since we can assume the input tree is balanced, there is a very simple reduction obtained by embedding the tree nodes as points in the plane where node (x', y') is an ancestor of node (x, y) iff $x' \geq x$ and $y' \geq y$.

2. planar dominance emptiness $\rightarrow$ dynamic SMQ.

   In the dynamic SMQ problem we are given an array $A[1, \ldots, n]$ where each entry $A[i]$ is in $\{1, \ldots, n\}$. An update (i, j) changes the value of $A[i]$ to be j, and a suffix maximum query SMQ(i) returns the maximum value in $A[i, \ldots, n]$. The reduction is as follow: For each x in $\{1, \ldots, n\}$ we set $A[x]$ to be the largest y s.t $(x, y) \in S$ (or zero if there is no $(x, y) \in S$). It is easy to see that a dominance query (x, y) in S reduces to checking whether SMQ$(x) > y$. Upon an insertion or deletion of a point (x, y) we need to update $A[x]$. For this we need to maintain for every x the maximum y s.t. $(x, y) \in S$. This can be done in $O(\log \log n)$ time and linear space using a predecessor structure for each x.

3. dynamic SMQ $\rightarrow$ dynamic nearest marked descendant.

   The reduction is as follows: Given an array A, we build a tree T of size n^2. The tree is composed of a spine $v_1, \ldots, v_n$ where each v_i has two children: the spine node v_{i+1} and the unique path $v_{i,n} \rightarrow v_{i,n-1} \rightarrow \cdots \rightarrow v_{i,1}$. The weight of each spine edge (v_i, v_{i+1}) is 1 and the weight of each non-spine edge $(v_{i,j}, v_{i,j-1})$ is n (again, we could replace weight n with n weight-1 edges, which increases $|T|$ to n^3). In each path $v_{i,n} \rightarrow v_{i,n-1} \rightarrow \cdots \rightarrow v_{i,1}$ there is exactly one marked node: If $A[i] = j$ then the marked node is $v_{i,j}$. It is easy to see that SMQ(i) indeed corresponds to the nearest marked descendant of v_i.

Dynamic nearest colored node and descendant with link-cut operations. Recall that, to support insertion and deletion of edges (i.e., maintain a forest under link and cut operations), the (top-tree based) solution of Alstrup-Holm-de Lichtenberg-Thorup [2] can be extended from two colors to k colors at the cost of increasing the update time to $\tilde{O}(k)$. We show that this is probably optimal. Namely, we prove (via a simple reduction) that a solution with $O(k^{1-\varepsilon})$ query and update

time implies an $O(n^{3-\varepsilon})$ solution for the classical All Pairs Shortest Paths (APSP) problem on a general graph with n vertices.

Vassilevska Williams and Williams [22] introduced this approach and showed *subcubic equivalence* between APSP and a list of seven other problems, including: deciding if a graph has a triangle whose total length is negative, min-plus matrix multiplication, deciding if a given matrix defines a metric, and the replacement paths problem. Namely, they proved that either all these problems have an $O(n^{3-\varepsilon})$ solution or none of them does.

It is well known that in APSP we can assume w.l.o.g that the graph is tripartite. That is, it has $3n$ vertices partitioned into three sets A, B, C each of size n. The edges have lengths $\ell(\cdot)$ and are all in $A \times B \cup B \times C$. The problem is to determine for every pair $(a, c) \in A \times C$ the value $\min_{b \in B}(\ell(a, b) + \ell(b, c))$.

We now describe the reduction: Given a tripartite graph $A = \{a_1, \ldots, a_n\}$, $B = \{b_1, \ldots, b_n\}$, $C = \{c_1, \ldots, c_n\}$ we pick vertex a_1 in A and make it the root of the tree. We set its children to be $b_1, b_2, \ldots, b_n$ where the edge (a_1, b_j) has the same length $\ell(a_1, b_j)$ as in the tripartite graph. Each b_j has n children. The kth child has color c_k, and the corresponding edge has length $\ell(b_j, c_k)$. We get a tree that is of size $O(n^2)$, and has depth two. We then ask the n queries (a_1, c_k) where c_k is a color. This completes the handling of a_1. I.e., for every $c_k \in C$ we have found $\min_{b \in B}(\ell(a_1, b) + \ell(b, c_k))$. We next want to do the same for a_2. To this end we do n updates: for each i we change the root-to-b_j edge so that its length becomes $\ell(a_2, b_j)$. We then ask n queries, and so on.

Overall we do n^2 updates and n^2 queries on a tree that is of size $N = n^2$, and $k = \sqrt{N}$ colors. Assuming that APSP cannot be solved in $O(n^{3-\varepsilon})$ time, we get that, for dynamic nearest colored node on a tree of size N with link-cut operations, the query or the update must take $\Omega(\sqrt{N}) = \Omega(k)$. Note that, the updates in this reduction do not alter the topology of the tree, but only the edge lengths. Hence, the lower bound applies even to a dynamic nearest colored node problem with just edge-weight updates (and no link or cut updates).

References

[1] I. Abraham, S. Chechik, R. Krauthgamer, and U. Wieder. Approximate nearest neighbor search in metrics of planar graphs. In *18th APPROX/RANDOM*, pages 20–42, 2015.

[2] S. Alstrup, J. Holm, K. de Lichtenberg, and M. Thorup. Maintaining information in fully dynamic trees with top trees. *ACM Transactions on Algorithms (TALG)*, 1(2):243–264, 2005.

[3] S. Alstrup, T. Husfeldt, and T. Rauhe. Marked ancestor problems. *Technical Report DIKU 98-8, Dept. Comput. Sc., Univ. Copenhagen*, 1998. (Some of the results needed from here are not included in the FOCSâĂŹ98 extended abstract).

[4] B. Belazzougui and G. Navarro. Optimal lower and upper bounds for representing sequences. *ACM Transactions on Algorithms (TALG)*, 11(4):1–21, 2010.

[5] M. Bender, M. Farach-Colton, G. Pemmasani, S. Skiena, and P. Sumazin. Lowest common ancestors in trees and directed acyclic graphs. *Journal of Algorithms*, 57(2):75–94, 2005.

[6] O. Berkman and U. Vishkin. Recursive star-tree parallel data structure. *SIAM Journal on Computing*, 22(2):221–242, 1993.

[7] P. Bille, P. H. Cording, and I. L. Gørtz. Compressed subsequence matching and packed tree coloring. *Algorithmica*, 77(2):336–348, 2017.

[8] P. Bille, G. Landau, R. Raman, S. Rao, K. Sadakane, and O. Weimann. Random access to grammar-compressed strings and trees. *SIAM Journal on Computing (SICOMP)*, 44(3):513–539, 2015.

[9] S. Chechik. Improved distance oracles and spanners for vertex-labeled graphs. In *20th ESA*, pages 325–336, 2012.

[10] M. Dietzfelbinger, A. Karlin, K. Mehlhorn, F. M. auf der Heide, H. Rohnert, and R. Tarjan. Dynamic perfect hashing: Upper and lower bounds. *SIAM J. Comput.*, 23(4):738–761, 1994.

[11] M. Fredman, J. Komlós, and E. Szemerédi. Storing a sparse table with $o(1)$ worst case access time. *J. ACM*, 31(3):538–544, 1984.

[12] D. Harel and R. E. Tarjan. Fast algorithms for finding nearest common ancestors. *SIAM Journal on Computing*, 13(2):338–355, 1984.

[13] D. Hermelin, A. Levy, O. Weimann, and R. Yuster. Distance oracles for vertex-labeled graphs. In *38th ICALP*, pages 490–501, 2011.

[14] M. Li, C. C. Ma, and L. Ning. $(1 + \epsilon)$-distance oracles for vertex-labeled planar graphs. In *10th TAMC*, pages 42–51, 2013.

[15] J. Łącki, J. Oćwieja, M. Pilipczuk, P. Sankowski, and A. Zych. The power of dynamic distance oracles: Efficient dynamic algorithms for the steiner tree. In *47th STOC*, pages 11–20, 2015.

[16] S. Mozes and E. Skop. Efficient vertex-label distance oracles for planar graphs. In *13th WAOA*, pages 97–109, 2015.

[17] S. Muthukrishnan and M. Müller. Time and space efficient method-lookup for object-oriented programs (extended abstract). In *7th SODA*, pages 42–51, 1996.

[18] M. Pătraşcu and M. Thorup. Time-space trade-offs for predecessor search. In *38th STOC*, pages 232–240, 2006.

[19] P. van Emde Boas, R. Kaas, and E. Zijlstra. Design and implementation of an efficient priority queue. *Mathematical Systems Theory*, 10:99–127, 1977. Announced by van Emde Boas at FOCS 1975.

[20] B. Wilkinson. Amortized bounds for dynamic orthogonal range reporting. In *22nd ESA*, pages 842–856, 2014.

[21] D. Willard. Log-logarithmic worst-case range queries are possible in space $\theta(n)$. *Inf. Process. Lett.*, 17(2):81–84, 1983.

[22] V. V. Williams and R. Williams. Subcubic equivalences between path, matrix and triangle problems. In *51st FOCS*, pages 645–654, 2010.