Dynamic Path Counting and Reporting in Linear Space*

Meng He[1], J. Ian Munro[2], and Gelin Zhou[2]

[1] Faculty of Computer Science, Dalhousie University, Canada.
mhe@cs.dal.ca
[2] David R. Cheriton School of Computer Science, University of Waterloo, Canada.
{imunro, g5zhou}@uwaterloo.ca

Abstract. In the path reporting problem, we preprocess a tree on n nodes each of which is assigned a weight, such that given an arbitrary path and a weight range, we can report the nodes whose weights are within the range. We consider this problem in dynamic settings, and propose the first non-trivial linear-space solution that supports path reporting in $O((\lg n/\lg\lg n)^2+occ\lg n/\lg\lg n)$ time, where occ is the output size, and the insertion and deletion of a node of an arbitrary degree in $O(\lg^{2+\epsilon} n)$ amortized time, for any constant $\epsilon \in (0,1)$. Obvious solutions based on directly dynamizing solutions to the static version of this problem all require $\Omega((\lg n/\lg\lg n)^2)$ time for each node reported, and thus our query time is much faster. For the counting version of this problem, we design a structure that supports path counting in $O((\lg n/\lg\lg n)^2)$ time, and insertion and deletion in $O((\lg n/\lg\lg n)^2)$ amortized time. This matches the current best result for 2D dynamic range counting, which can be viewed as a special case of path counting.

1 Introduction

In computer science, trees are widely used in modeling and representing different types of data. In many scenarios, objects are represented by nodes and their properties are characterized by weights assigned to nodes. Researchers have studied the problems of maintaining a weighted tree, such that, given any pair of nodes, certain functions over the path between these two nodes can be computed efficiently [7, 1, 18, 17, 5, 14, 21, 15, 6]. The inquires of the values of these functions are referred to as *path queries*.

Previously, most work on path queries focus on static weighted trees, i.e., the structure and the weights of nodes remain unchanged over time. This assumption is not always realistic and it is highly inefficient to rebuild the whole data structure when handling updates. In this paper, we consider the problem of maintaining dynamic weighted trees and design data structures that support *path counting* and *path reporting* queries in linear space and efficient time. More precisely, given a query path and a query range, these types of queries return the

* This work was supported by NSERC and the Canada Research Chairs Program.

number/set of nodes on the path whose weights are in the given range. As mentioned in He et al.'s work [14, 15], these path queries generalize two-dimensional range counting and reporting queries.

Without loss of generality, we represent the input tree as an ordinal one, i.e., a rooted tree in which children of a node are ordered. Our data structures allow to change the weight of an existing node, insert a new node, or delete an existing node. These updates are referred to as `modify_weight`, `node_insert`, and `node_delete`, respectively. For `node_insert` and `node_delete`, we adopt the same powerful updating protocol as Navarro and Sadakane [20], which allows us to insert or delete a leaf, a root, or an internal node. A newly inserted internal node will become the parent of consecutive children of an existing node, and a deleted root must have only zero or one child. The deletion of a non-root node is described in Section 2.2.

It is natural to identify nodes with their preorder ranks in static ordinal trees. However, preorder ranks of nodes can change over time in dynamic trees. Thus, in our dynamic data structures, nodes are identified by immutable identifiers of sizes $O(\lg n)$ bits[3]. Unless otherwise specified, the underlying model of computation in this paper is the unit-cost word RAM model with word size $w = \Omega(\lg n)$.

Previous Work. The problems of supporting static path counting and path reporting queries have been heavily studied in recent years [14, 21, 15, 6]. Given an input tree on n nodes whose weights are drawn from $[1..\sigma]$, He et al. [15] designed succinct data structures to support path counting queries in $O(\lg \sigma / \lg \lg n + 1)$ time, and path reporting queries in $O((occ+1)(\lg \sigma / \lg \lg n + 1))$ time, where occ is the size of output. Later, Chan et al. [6] achieved more time/space tradeoffs for path reporting queries. They developed an $O(n)$-word structure with $O(\lg^{\epsilon} n + occ \cdot \lg^{\epsilon} n)$ query time, where ϵ is an arbitrary constant in $(0, 1)$; an $O(n \lg \lg n)$-word structure with $O(\lg \lg n + occ \cdot \lg \lg n)$ query time; and an $O(n \lg^{\epsilon} n)$-word structure with $O(\lg \lg n + occ)$ query time.

There are other heavily studied path query problems such as path minimum queries, and we refer to Chan et al. [6] for a recent survey on the static version of this problem. The dynamic version of the path minimum problem has also been studied extensively. Brodal et al. [5] designed a linear space data structure that supports queries and changes to the weight of a node in $O(\lg n / \lg \lg n)$ time, and handles insertions or deletions of a node with zero or one child in $O(\lg n / \lg \lg n)$ amortized time. The query time is optimal under the cell probe model provided that the update time is $O(\lg^{O(1)} n)$ [2]. For the more restricted case in which only insertions and deletions of leaves are allowed, queries can be answered in $O(1)$ time and updates can be supported in $O(1)$ amortized time [1, 17, 5].

Our Contributions. We develop efficient dynamic data structures for path counting and path reporting queries, all of which occupy $O(n)$ words. Our data structure supports path counting queries in $O((\lg n / \lg \lg n)^2)$ time, and handles changes of weights, insertions and deletions in $O((\lg n / \lg \lg n)^2)$ amortized time. This structure matches the best known result for dynamic range counting [12].

For path reporting queries, our data structure requires $O(\lg^{2+\epsilon} n)$ time for

[3] We use lg to denote the base-2 logarithm.

updates, but answers queries in $O((\lg n/\lg\lg n)^2 + occ \lg n/\lg\lg n))$ time, where occ is the output size. By slightly sacrificing the update time, our structure significantly improves the query time over the straightforward approaches that dynamize known static data structures[14, 21, 15, 6]: One could dynamize the static structure of He et al. [14] by replacing static labeled ordinal trees with dynamic unlabeled trees and dynamic bit vectors, and managing weight ranges using a red-black tree [8]. This leads to an $O(n)$-word data structure with $O((1+occ)\lg^2 n/\lg\lg n)$ query time and $O(\lg^2 n/\lg\lg n)$ update time. Alternatively, one could obtain another $O(n)$-word structure with $O(\lg^{2+\epsilon} n + occ \cdot (\lg n/\lg\lg n)^2)$ query time and $O((\lg n/\lg\lg n)^2)$ update time, by dynamizing the improved result of He et al. [15] in a similar manner. It is unclear how to dynamize the structures designed by Patil et al. [21] and Chan et al. [6] within linear space.

All of our dynamic structures presented in this paper are able to handle insertions and deletions of nodes with multiple children, which are not supported in previous dynamic data structures for path queries [1, 17, 5]. Our approach is almost completely different from He et al.'s [15] approach for static path queries. To develop our data structures, we employ a various of techniques including *topology trees*, *tree extraction*, and *balanced parentheses*. In particular, for dynamic path reporting, one key strategy is to carefully design transformations on trees that preserve certain properties, such that the idea of dynamic fractional cascading can be adapted to work on multiple datasets in which each set represents tree-structured data. This new approach may be of general interest.

Section 2 reviews the techniques used in our data structures. Section 3 describes our dynamic data structures for path reporting. Due to the page limitation, the support for path counting is deferred to the full version of this paper.

2 Preliminary

2.1 Restricted Multilevel Partition and Topology Trees

Frederickson [9–11] proposed topology trees to maintain connectivity information and minimum spanning trees of dynamic graphs, and to support operations over dynamic trees. We make use of a variant of topology trees based on a restricted partition of a binary tree $\mathcal{B}$, where the nodes of $\mathcal{B}$ are clustered into disjoint sets such that the elements in each set are nodes of a connected component of $\mathcal{B}$. Each of such components is called a *cluster*, and its *external degree* is the number of edges with exactly one endpoint being a vertex in the cluster. A *restricted partition of order* s of $\mathcal{B}$ is defined to be a partition that satisfies the following conditions: each cluster has external degree at most 3; each cluster with external degree 3 contains only one node; each cluster with external degree less than 3 has at most s nodes, and no two adjacent clusters can be combined without breaking the above conditions. Frederickson gave a linear-time algorithm that creates a restricted partition of order s for a given binary tree on n nodes, and proved that the number of clusters is $\Theta(\lceil n/s \rceil)$.

The endpoints of the edges that connect different clusters are called *boundary nodes*. We further follow the notation of He et al. [13] and define the *preorder*

segments of a cluster to be the maximal contiguous subsequences of nodes in the preorder sequence that are in the same cluster. Thus Frederickson's approach guarantees that each cluster has up to two boundary nodes and up to three preorder segments. The clusters including the root may have 3 preorder segments.

Frederickson further defined a *restricted multi-level partition* of a binary tree $\mathcal{B}$ consisting of a set of h partitions of the nodes which can be computed recursively as follows: The clusters at level 0, which are called *base clusters*, are obtained by computing a restricted partition of order s of $\mathcal{B}$. Then, to compute the level-l clusters for any level $l > 0$, we view each cluster at level $l-1$ as a node, and then compute a restricted partition of order 2 of the resulting tree. This recursion stops when the partition contains only one cluster containing all the nodes, which is the level-h cluster.

A *topology tree* $\mathcal{H}$ is defined for a restricted multi-level partition of a binary tree $\mathcal{B}$. $\mathcal{H}$ contains $h+1$ levels. Each node of $\mathcal{H}$ at level l represents a level-l cluster, and the up to two children of a node at level l each corresponds to one of the two level-$(l-1)$ clusters that this level-l cluster consists of. Additional links are maintained between each pair of adjacent nodes at the same level of $\mathcal{H}$. Frederickson proved that $h = O(\lg n)$. Topology trees were used in maintaining a dynamic forest of binary trees, to support two operations: `link` which combines two trees in the forest into one by adding an edge between the root of one binary tree and an arbitrary given node of the other that has less than two children, and `cut` which breaks one tree into two by removing an arbitrary given edge. The following lemma summaries a special case of their results to be used in our solutions, in which we say that a cluster is *modified* during updates if it is deleted or created during this update, its nodes or edges have been changed or an edge with an endpoint in the cluster has been inserted or deleted:

Lemma 1 ([10, 11]). *The topology trees of the binary trees in a given forest F on n nodes can be maintained in $O(s + \lg n)$ time for each* `link` *and* `cut`*, where s is the maximum size of base clusters. Furthermore, each* `link` *or* `cut` *modifies $O(1)$ clusters at any level of the topology trees maintained for the two binary trees updated by this operation, and once a cluster is modified, the clusters represented by the ancestors of its corresponding node in the topology tree are all modified. These topology trees have $\Theta(f + n/s)$ nodes in total, where f is the current number of trees in F, and occupy $O(S + (f + n/s)\lg n)$ bits in total, where S is the total space required to store the tree structures of base clusters.*

2.2 Tree Extraction

Tree extraction has proved to be a powerful technique in supporting various types of static path queries [14–16, 6]. This technique is based on the deletion operation defined in the context of tree edit distance [4]. To delete a non-root node u, which is a child of v, the children of u are inserted in place of u in the list of children of v, preserving the original order. Let T be an ordinal weighted tree and I be a weight range. For the sake of convenience, we add a dummy node r to be the new root of T, which has a `NULL` weight and will be the parent of the

original root. We define T_I to be the extracted tree obtained by deleting all the non-root nodes whose weights are not in I from the argumentation of T. That is, T_I only consists of the dummy root and the nodes whose weights are in I. The crucial observation is that tree extraction preserve the ancestor-descendant, preorder, and postorder relationships among the remaining nodes.

3 Dynamic Path Reporting

Let T be a dynamic tree on n weighted nodes. W.l.o.g, we assume that node weights are distinct. We construct a weight-balanced B-tree [3], W, with leaf parameter 1 and branching factor $d = \lceil \lceil \lg n \rceil^{\epsilon} \rceil$ for any positive constant ϵ less than $1/2$. When the value of d changes due to updates, we reconstruct the entire data structure and amortize the cost of rebuilding to updates. By the properties of weight-balanced B-trees, each internal node of W has at least $d/4$ and at most $4d$ children, and the only exception is the root which is allowed to have fewer children. Each leaf of W represents a weight range $[a, b)$, where a and b are weights assigned to nodes of T, and there is no node of T whose weight is between a and b. An internal node of W represents a (contiguous) range which is the union of the ranges represented by its children, where the children are sorted by the left endpoints of these weight ranges. The levels of W are numbered $0, 1, 2, \ldots, t$, starting from the leaf level, where $t = O(\lg n / \lg \lg n)$ denotes the number of the root level. The tree structure of W together with the weight range represented by each node is maintained explicitly.

For each internal node v of W, we conceptually construct a tree $T(v)$ as follows: Let $[a, b)$ denote the weight range represented by v. We construct a tree $T_{[a,b)}$ consisting of nodes of T whose weights are in $[a, b)$ using the tree extraction approach described in Section 2.2. For each node x in $T_{[a,b)}$, we then assign an integer label $i \in [1..4d]$ if the weight of x is within the weight range of the ith child of v. The resulting labeled tree is $T(v)$.

We do not store each $T(v)$ explicitly. Instead, we transform the tree structure of each $T(v)$ into a binary tree $\mathcal{B}(v)$ as in [6]: For each node x of $T(v)$ with $k > 2$ children denoted as $x_1, x_2, \cdots, x_k$, we add $k - 1$ dummy nodes $y_1, y_2, \cdots, y_{k-1}$. Then, x_1 and y_1 become the left and the right children of x, respectively. For $i = 1, 2, \ldots, k-2$, the left and the right children of y_i are set to be x_{i+1} and y_{i+1}, respectively. Finally, x_k becomes the left and only child of y_{k-1}. In $\mathcal{B}(v)$, the node corresponding to the dummy root of $T(v)$ is also considered a dummy node, and a node is called an *original node* if it is not a dummy node. We observe that this transformation preserves the preorder and postorder relationships among the original nodes in $T(v)$. Furthermore, the set of original nodes along the path between any two original nodes remains unchanged after transformation. Each original node in $\mathcal{B}(v)$ is associated with its label in $T(v)$, which is an integer in $[1..4d]$, while each dummy node is assigned with label 0.

Let F_i denote the forest containing all the binary trees created for the nodes at the ith level of W for $i > 0$, i.e., $F_i = \{\mathcal{B}(v) : v \text{ is a node at the } i\text{th level of } W\}$ for $i \in [1..t]$. Thus F_t contains only one binary tree which corresponds to the

root of W, and this tree contains all the nodes of the given tree T as original nodes. This allows us to maintain a bidirectional pointer between each node in T and its corresponding original node in F_t.

W and T are stored using standard, pointer-based representations of trees. In the rest of this section, we first present, in Section 3.1, a data structure that can be used to maintain a dynamic forest in which each node is assigned a label from an alphabet of sub-logarithmic size, to support a set of operations including *path summary* queries which is to be defined later. This structure is of independent interest and will be used to encode each F_i. We next show, in Section 3.2, how to maintain pointers between forests constructed for different levels of W, which will be used to locate appropriate nodes of these forests when answering path reporting queries. Finally we describe how to answer path reporting queries and perform updates in weighted trees in Section 3.3.

3.1 Representing Dynamic Forests with Small Labels to Support Path Summary Queries

We now describe a data structure which will be used to encode F_i in subsequent subsections. As this structure may be of independent interest, we formally describe the problem its addresses as follows. Let F be a dynamic forest of binary trees on n nodes in total, in which each node is associated with a label from the alphabet $[0..\sigma]$, where $\sigma = O(\lg^\epsilon n)$ for an arbitrary constant $\epsilon \in (0, 1/2)$. Our objective is to maintain F to support `link`, `cut` and the following operations:

- $\texttt{parent}_\alpha(x)$: return the α-parent of node x, i.e., the lowest ancestor of x that has label α, which can be x itself.
- $\texttt{LCA}(x, y)$: return the lowest common ancestor of two given nodes x and y residing in the same binary tree.
- $\texttt{pre\_succ}_\alpha(x)$: return the α-successor of x in preorder, i.e., the first α-node in preorder that succeeds x (this could be x itself).
- $\texttt{post\_pred}_\alpha(x)$: return the α-predecessor of x in postorder, i.e., the last α-node in postorder that precedes x (this could be x itself).
- $\texttt{summary}(x, y)$: given two nodes x and y residing in the same binary tree, return a bit vector of $\sigma + 1$ bits in which the αth bit is 1 iff there exists an α-node along the path from x to y. This query is called path summary.
- $\texttt{modify}(x, \alpha)$: change the value of x's label to α.

We first set $s = \lceil \frac{\lceil \lg n \rceil}{\lg \lceil \lg n \rceil} \rceil$ in Lemma 1, and use the lemma to maintain the topology trees of the binary trees in F. We call each base cluster a *micro-tree*.

We next define a subset of levels of the topology trees *marked levels*. For $i = 0, 1, \ldots$, the ith marked level of a topology tree is level $i\lfloor \epsilon \lg \lg n \rfloor$ of this topology tree. Since in a topology tree, the restricted partition at each level except level 0 is of order 2, each internal node of the topology tree has at most two children. Therefore, for $i \geq 1$, each cluster at the ith marked level contains at most $2^{\lfloor \epsilon \lg \lg n \rfloor} \leq 2^{\epsilon \lg \lg n} = \lg^\epsilon n$ clusters at the $(i-1)$st marked level. We then define the *macro-tree* for a node at the ith marked level of a topology tree,

for $i \geq 1$, to be the tree obtained by viewing each cluster at the $(i-1)$st marked level as a single node and adding an edge between two of these nodes if and only if their corresponding clusters are adjacent. As shown in the discussion above, each macro-tree is a binary tree with at most $\lg^{\epsilon} n$ nodes. A macro-tree is called a *tier-i* macro-tree if it is constructed for a node at the ith marked level. A node in a tier-i macro-tree is called a *boundary node* if its corresponding cluster contains the endpoint of an edge that has only one endpoint in this tier-i macro-tree. By the properties of restricted multi-level partitions, each macro-tree has at most two boundary nodes and at most one of them is a leaf in the macro tree.

We construct auxiliary data structures for each micro-tree and macro-tree. Our main idea is to create structures that can fit in $\frac{1}{2}\lg n$ bits (in addition to maintaining pointers such as those that can be used to map macro tree nodes to macro trees at the lower marked level), so that we can construct $o(n)$-bit lookup tables to perform operations in each micro-tree or macro tree. Operations over F are then supported by operating on a constant number of micro-trees and a constant number of macro-trees at each marked level. The proof of the following lemma is omitted due to the page limitation.

Lemma 2. *Let F be a dynamic forest of binary trees on n nodes in total, in which each node is associated with a label from the alphabet $[0..\sigma]$, where $\sigma = O(\lg^{\epsilon} n)$ for any constant $\epsilon \in (0, 1/2)$. F can be represented in $O(n \lg\lg n + f \lg n)$ bits to support* $\texttt{parent}_\alpha$, `LCA`, `summary`, $\texttt{pre\_succ}_\alpha$, $\texttt{post\_pred}_\alpha$ *and* `modify` *in $O(\lg n / \lg\lg n)$ time, and* `link` *and* `cut` *in $O(\lg^{1+\epsilon} n)$ time, where f is the current number of trees in F.*

3.2 Navigation Between Different Levels of W

As discussed previously, we use Lemma 2 to encode each F_i for $i > 0$. For each node at the ith level of W, we store a pointer to the root of its corresponding topology tree in F_i. Each tree node in F_i can be uniquely identified by a pointer to the micro-tree containing the node and its preorder rank in the micro-tree. We call this pair of pointer and preorder rank the *local id* of this node in F_i.

Since each node, x, of T appears once in F_i as an original node for each $i \in [0..t]$, x has one local id at each level of W. In our algorithm for path reporting, given the local id of x in F_i, we need find its local id in F_{i-1} and F_{i+1}. Explicitly storing the answers would require too much space. Thus, our overall strategy is to precompute, for only a subset of nodes of T, their local ids in F_{i-1} and F_{i+1}. Then, we design an algorithm to compute local ids of other nodes, by making use of the fact that both tree extraction and our way of transforming each $T(v)$ to $\mathcal{B}(v)$ preserve relative preorder among nodes of T.

We now describe our strategy in details. In F_i, we call the clusters at the first marked level of the topology trees *mini-trees*. By our discussions in Section 3.1, each mini-tree then contains at most $\lg^{\epsilon} n$ micro-trees, and has $O(\lg^{1+\epsilon} n)$ tree nodes. There is a one-to-one correspondence between mini-trees and tier-1 macro-trees, but they are conceptually different: each node in a mini-tree is a node of F_i, while a node in a tier-1 macro-tree represents a micro-tree. Because of this

one-to-one correspondence, however, we do not distinguish the pointers to a tier-1 macro-tree from a pointer to its corresponding mini-tree. We say a micro-tree is the ith micro-tree of a mini-tree (or its corresponding tier-1 macro-tree), if this micro-tree is represented by the ith node in preorder of the tier-1 macro-tree corresponding to this mini-tree.

A node in F_i can also be uniquely identified by a pointer to the mini-tree containing the node and its preorder rank in the mini-tree. Conversions between this type of identification and the local id of the node can be done in constant time (the details are omitted). Thus we consider each of these two different identifiers as a valid local id of a node in F_i in the rest of the paper. Furthermore, we consider the support of $\texttt{parent}_\alpha$ within any given mini-tree, i.e., given a node x, we are interested in finding its α-parent in the same mini-tree if it exists. We also consider the following two operators over a mini-tree:

- $\texttt{pre\_rank}_\alpha(x)$, which computes the number of α-nodes preceding x in preorder (including x itself if it is labeled α);
- $\texttt{pre\_select}_\alpha(i)$, which locates the ith α-node in preorder.

In the above definition, we allow α be set to $\bar{0}$, which matches any label that is not 0. We have the following lemma. The proof is omitted here.

Lemma 3. *With $o(n)$ additional bits, $\texttt{parent}_\alpha$, $\texttt{pre\_rank}_\alpha$ and $\texttt{pre\_select}_\alpha$ can be supported in $O(1)$ time over each mini-tree in F_i.*

We next define a set of pointers between mini-trees at different levels of W and we call these pointers *inter-level pointers*. These pointers are defined for each mini-tree μ in any F_i. Let v be the node of W such that $\mathcal{B}(v)$ contains μ. If $i < t$, then for each preorder segment of μ, we create an *up pointer* for the first original node, x, of this segment in preorder. This pointer points from x to the original node in F_{i+1} that corresponds to the same node of T. Next, if $i > 1$, for each preorder segment of μ and for each label $\alpha \in [1..4d]$, if node y is the first node in this segment in preorder that is labeled α, we store a *down pointer* from y to the original node in F_{i-1} that corresponds to the same node of T that y represents. No pointers are created for nodes labeled 0, as they are dummy nodes. So far we have created at most $3(4d+1) = O(\lg^\epsilon n)$ inter-level pointers for each mini-tree, as each mini-tree has at most three preorder segments. Finally, we create a back pointer for each up or down pointer, doubling the total number of inter-level pointers created over all the levels of W.

To store inter-level pointers physically, we maintain all the pointers that leave from mini-tree μ (again, suppose that μ is in $\mathcal{B}(v)$ which is part of F_i) in a structure called P_μ, including up and down pointers created for nodes in μ, and back pointers for some of the up and down pointers created for mini-trees at adjacent levels of W. We further categorize these pointers into at most $4d+1$ types: A type-0 pointer arrives at a mini-tree in F_{i+1}, i.e., goes to the level above, and a type-α pointer for $\alpha > 1$ arrives at a mini-tree in $B(u)$, where u is the αth child of v. Note that it is possible that an up or down pointer of μ and a back pointer from an adjacent level stored in P_μ have the same source (a node

in μ) and destination (a node in the forest for an adjacent level of W). In this case, the back pointer is not stored separately in P_μ, and hence each inter-level pointer in P_μ can be uniquely identified by its type and the preorder rank of its source node in μ. We also maintain the preorder rank of the first node of each of the (at most three) preorder segments in μ. We use the approach of Navarro and Nekrich [19, Section A.3] with trivial modifications to encode P_μ in $|P_\mu|$ words, so that given a node x in μ, we can retrieve in $O(1)$ time the closest preceding node (this can be x itself) in the preorder segment of μ containing x that has an inter-level pointer of a given type α, as well as the local id of the destination of this pointer. Insertion and deletion of inter-level pointers can also supported in $O(1)$ time. The details are deferred to the full version due to space limitation. We can now prove the following lemma:

Lemma 4. *Give the local id of a original node x in F_i, the local id of the original node in F_{i+1} (if $i < t$) or F_{i-1} (if $i > 1$) that represents the same node of T can be computed in $O(1)$ time.*

Proof. We first show how to locate the node, y, in F_{i+1} that represents the same node of T. We start to find the closest node, x', that precedes x in preorder, has a type-0 inter-level pointer, and resides in the same preorder segment, s_0, of the mini-tree containing x (x' is allowed to be x itself). The destinate node, y', of this pointer is also retrieved during the same process, and it is a node in F_{i+1}. Node x' always exists because the first original node of each preorder segment in a mini-tree has an up pointer.

If x' happens to be x itself, then y' is y which is the answer. If not, we observe that y and y' are in the same preorder segment of a mini-tree in F_{i+1}. Suppose that u is the αth node of v. It then follows that the number of α-nodes of $\mathcal{B}(v)$ that are between y' and y in preorder is equal to the number, k, of original nodes between x' and x in $\mathcal{B}(u)$. As the number of original and dummy nodes between x' and x in $\mathcal{B}(u)$ is equal to the difference between the preorder ranks of x' and x, it suffices to compute the number of dummy nodes between them, which can be computed as $\texttt{pre\_rank}_0(x) - \texttt{pre\_rank}_0(x')$ in $\mathcal{B}(u)$. By Lemma 3, this requires constant time since they are in the same mini-tree. Then, the preorder of y can be computed as $\texttt{pre\_select}_\alpha(\texttt{pre\_rank}_\alpha(y') + k)$ in $\mathcal{B}(v)$, which again requires constant time. This gives us the local id of y', and the entire process uses $O(1)$ time. The node, z, in F_{i-1} that represents the same node of T as x does can be located using a similar process. □

3.3 Supporting Path Reporting

Lemma 5. *The structures in this section can answer a path reporting query in $O((\lg n/\lg\lg n)^2 + occ \lg n/\lg\lg n)$ time, where occ is the output size.*

Proof. Let x and y be the two nodes that define the query path, and let $[p, q]$ be the query weight range. We perform a top-down traversal in W to locate its up to two leaves that represent ranges containing p and q. During this traversal, at each level, i, of W, we visit at most two nodes of W, and each node to visit at

the next level can be located using a binary search in $O(\lg d) = O(\lg \lg n)$ time, as each node has at most $4d$ children. As there are $O(\lg n / \lg \lg n)$ levels in W, the total time required to determine the nodes of W to visit is $O(\lg n)$.

For each node, v, of W visited during the above top-down traversal, we also determine the original nodes x_v and y_v in $\mathcal{B}(v)$ respectively corresponding to the lowest ancestors of x and y in T that are represented by nodes in $\mathcal{B}(v)$ (each node is considered to be its own ancestor). These nodes are located during the top-down traversal as follows. Let u denote the parent of v in W, and suppose that v is the αth child of u. Then to compute x_v, if x_u is labeled with α, then we use Lemma 4 to locate x_v in constant time. Otherwise, we first locate x_u's lowest α-parent, x', using Lemma 2 in $O(\lg n / \lg \lg n)$ time, and then compute x_v as the node corresponding to x' in $\mathcal{B}(v)$ in $O(1)$ time using Lemma 4. y_v can be computed in a similar manner. The total time required to locate all these nodes in our query algorithm is thus $O((\lg n / \lg \lg n)^2)$.

For each node, v, of W visited during the traversal, if the range of at least one of v's children is contained entirely in $[p, q]$, then we compute $z_v = \texttt{LCA}(x_v, y_v)$ in $\mathcal{B}(v)$. We also perform a path summary query using x_v and y_v as the endpoints of the query path, and let V be the bit vector returned by the query. Suppose that the children of v whose ranges are contained in $[p, q]$ are numbered $j, j+1, \ldots, k$. Since V has $O(\lg^\epsilon n)$ bits, then we can use an $o(n)$-bit table to retrieve the position of each 1 bit in $V[j..k]$ in constant time. Then for each $l \in [j..k]$ such that $V[l] = 1$, we claim that there are nodes along the path between x_v and y_v in $\mathcal{B}(v)$ that are labeled l, and these nodes correspond to nodes of T to be reported. Each node from x_v to z_v (including x_v and z_v) labeled l can be located using $\texttt{parent}_l$ over $\mathcal{B}(v)$ (when we reach a node whose preorder in $\mathcal{B}(v)$ is less than or equal to that of z_v, we have located all these nodes), and for each node found, we keep finding its local id in the level above, until we find its local id at the root level of W which immediately gives us a node in T, and we report this node of T. The nodes from y_v to z_v (including x_v but excluding z_v) labeled l can be located and have their corresponding nodes in T reported using the same approach. We observe that a constant number of `LCA` and `summary` are performed at each level of W, which require $O((\lg n / \lg \lg n)^2)$ time in total. Then, for each node reported, only $O(\lg n / \lg \lg n)$ time is spent: if we always charge each $\texttt{parent}_\alpha$ operation to the last node reported before this operation is performed, then each node is charged a constant number of times, and the process described above which finds the node of T given its local id in $\mathcal{B}(v)$ requires $O(\lg n / \lg \lg n)$ time. This completes the proof. □

Lemma 6. *The structures in this section support* `node_insert`*,* `node_delete` *and* `modify_weight` *in* $O(\lg^{2+\epsilon} n)$ *amortized time.*

Proof. We only show how to support `node_insert`; the other update operations can be handled similarly. Note that update operations may eventually change the value of $\lceil \lg n \rceil$, but this can be handled by standard techniques of dynamic data structures.

Suppose that we insert a new node h with weight w_h. The new node h is inserted as a child of x, and a set of consecutive children of x between and in-

cluding child nodes y and z become the children of h after the insertion. Here we consider the general case in which y and z exist and are different nodes; degenerate cases can be handled similarly. In the first step of our insertion algorithm, we insert the weight w_h into W by creating a new leaf for it. This may potentially cause the parent of this new leaf to split, but for now, we consider the case in which a split will not happen. The support for node splits in W is omitted due to the page limitation.

We next perform a top-down traversal of W to fix the structures created for the forest F_i at each level i of W. In our description, when we say node h (or x, etc.) in F_i, we are referring to the original node, either to be inserted to F_i or already exist in F_i, that corresponds to this node in T. At the top level, i.e., the tth level, of W, the forest F_t contains one single binary tree. As we maintain bidirectional pointers between nodes in T and nodes in F_t, we can immediately locate the nodes x, y and z in F_t. Let $x_1, x_2, \ldots$ be the dummy nodes created for x in F_t, among which x_j and x_k are the dummy nodes that are parents of y and z, respectively. We then perform a constant number of updates to F_t as follows. First we perform the `cut` operation twice to remove the edge between x_{j-1} and x_j, and the edge between x_k and $x_k + 1$. This divides F_t into three trees. We then create a tree on a new node x'_j which is a dummy node, and temporarily include this tree into F_t. Note that creating the topology tree and associated auxiliary data structures for a tree on a single node can be trivially done in constant time. We then replace the dummy node x_j by the node h to be inserted. This can be done by first performing binary searches in the ranges of the children of the root, r, of W, so that we know the correct label, α, to assign to h. We then simply call `modify` to change the label, 0, assigned to x'_j, to α using `modify`. We then perform `link` to add three edges so that x'_j becomes the right child of x_{j-1}, h becomes the left child of x'_j, and x_{k+1} becomes the right child of x'_j. It is clear that all these operations require $O(\lg^{1+\epsilon} n)$ time in total.

To update F_{t-1}, let v be the αth child of r. We observe that it suffices to update $\mathcal{B}(v)$ without making changes to any other tree in F_{t-1}. Then we claim that if x is also labeled α in $\mathcal{B}(r)$, then in $\mathcal{B}(v)$, we will also insert h as a child of the original node corresponding to x; otherwise, we insert h as a child of the original node of $\mathcal{B}(v)$ that corresponds to the node, x', in $\mathcal{B}(r)$ that is $\texttt{parent}_\alpha(x)$. If x' does not exist, then h is inserted as a child of the dummy root of $\mathcal{B}(v)$. We then observe that h will be inserted to $\mathcal{B}(v)$ as the new parent of the set of children of x, x' or the dummy root (depending on which of the above three cases applies) that are between and including the original nodes in $\mathcal{B}(v)$ that correspond to the nodes $\texttt{pre\_succ}_\alpha(y)$ and $\texttt{post\_pred}_\alpha(z)$ in $\mathcal{B}(r)$. Thus, in $O(\lg n / \lg\lg n)$ time, we have found where to insert h in F_{t-1}, and by the approach shown in the previous paragraph, we can use `link`, `cut` and `modify` to update F_{i-1} in $O(\lg^{1+\epsilon} n)$ time. This process can then be repeated at each successive level of W. Hence it requires $O(\lg^{2+\epsilon} n / \lg\lg n)$ time to update all the F_i's.

When updating the F_i's, we also update inter-level pointers. The details for that are deferred to the full version. □

The space analysis is also omitted. We thus have the final result:

Theorem 1. *Under the word RAM model with word size $w = \Omega(\lg n)$, an ordinal tree on n weighted nodes can be stored in $O(n)$ words of space, such that path reporting queries can be answered in $O((\lg n/\lg\lg n)^2 + occ \lg n/\lg\lg n)$ time, where occ is the output size, and* `modify_weight`, `node_insert` *and* `node_delete` *can be supported in $O(\lg^{2+\epsilon} n)$ amortized time for any constant $\epsilon \in (0, 1)$.*

References

1. Alstrup, S., Holm, J.: Improved algorithms for finding level ancestors in dynamic trees. In: ICALP. pp. 73–84 (2000)
2. Alstrup, S., Husfeldt, T., Rauhe, T.: Marked ancestor problems. In: FOCS. pp. 534–544 (1998)
3. Arge, L., Vitter, J.S.: Optimal external memory interval management. SIAM Journal on Computing 32(6), 1488–1508 (2003)
4. Bille, P.: A survey on tree edit distance and related problems. Theor. Comput. Sci. 337(1-3), 217–239 (2005)
5. Brodal, G.S., Davoodi, P., Rao, S.S.: Path minima queries in dynamic weighted trees. In: WADS. pp. 290–301 (2011)
6. Chan, T.M., He, M., Munro, J.I., Zhou, G.: Succinct indices for path minimum, with applications to path reporting. In: ESA. pp. 247–259 (2014)
7. Chazelle, B.: Computing on a free tree via complexity-preserving mappings. Algorithmica 2, 337–361 (1987)
8. Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C.: Introduction to Algorithms (3. ed.). MIT Press (2009)
9. Frederickson, G.N.: Data structures for on-line updating of minimum spanning trees, with applications. SIAM J. Comput. 14(4), 781–798 (1985)
10. Frederickson, G.N.: Ambivalent data structures for dynamic 2-edge-connectivity and k smallest spanning trees. SIAM J. Comput. 26(2), 484–538 (1997)
11. Frederickson, G.N.: A data structure for dynamically maintaining rooted trees. J. Algorithms 24(1), 37–65 (1997)
12. He, M., Munro, J.I.: Space efficient data structures for dynamic orthogonal range counting. Comput. Geom. 47(2), 268–281 (2014)
13. He, M., Munro, J.I., Satti, S.R.: Succinct ordinal trees based on tree covering. ACM Transactions on Algorithms 8(4), 42 (2012)
14. He, M., Munro, J.I., Zhou, G.: Path queries in weighted trees. In: ISAAC. pp. 140–149 (2011)
15. He, M., Munro, J.I., Zhou, G.: Succinct data structures for path queries. In: ESA. pp. 575–586 (2012)
16. He, M., Munro, J.I., Zhou, G.: A framework for succinct labeled ordinal trees over large alphabets. Algorithmica (2014), to appear
17. Kaplan, H., Shafrir, N.: Path minima in incremental unrooted trees. In: ESA. pp. 565–576 (2008)
18. Krizanc, D., Morin, P., Smid, M.H.M.: Range mode and range median queries on lists and trees. Nord. J. Comput. 12(1), 1–17 (2005)
19. Navarro, G., Nekrich, Y.: Optimal dynamic sequence representations. In: SODA. pp. 865–876 (2013)
20. Navarro, G., Sadakane, K.: Fully functional static and dynamic succinct trees. ACM Transactions on Algorithms 10(3), 16 (2014)
21. Patil, M., Shah, R., Thankachan, S.V.: Succinct representations of weighted trees supporting path queries. J. Discrete Algorithms 17, 103–108 (2012)