Tree Path Majority Data Structures*

Travis Gagie[a], Meng He[a], Gonzalo Navarro[b,c,d,*], Carlos Ochoa[b,d]

[a] *Faculty of Computer Science, Dalhousie University, Halifax, Canada*
[b] *Center of Biotechnology and Bioengineering (CeBiB), Chile*
[c] *Millennium Institute for Foundational Research on Data (IMFD), Chile*
[d] *Department of Computer Science, University of Chile, Santiago, Chile*

Abstract

We present the first solution to finding τ-majorities on tree paths. Given a tree of n nodes, each with a label from $[1..\sigma]$, and a fixed threshold $0 < \tau < 1$, such a query gives two nodes u and v and asks for all the labels that appear more than $\tau \cdot |P_{uv}|$ times in the path P_{uv} from u to v, where $|P_{uv}|$ denotes the number of nodes in P_{uv}. Note that the answer to any query is of size up to $1/\tau$. On a w-bit RAM, we obtain a linear-space data structure with $O((1/\tau) \lg \lg_w \sigma)$ query time, which is worst-case optimal for polylogarithmic-sized alphabets. We also describe two succinct-space solutions with query time $O((1/\tau) \lg^* n \lg \lg_w \sigma)$. One uses $2nH+4n+o(n)(H+1)$ bits, where $H \leq \lg \sigma$ is the entropy of the label distribution; the other uses $nH+O(n)+o(nH)$ bits. By using just $o(n \lg \sigma)$ extra bits, our succinct structures allow τ to be specified at query time. We obtain analogous results to find a τ-minority, that is, an element that appears between 1 and $\tau \cdot |P_{uv}|$ times in P_{uv}.

Keywords: Majorities on trees, Succinct data structures

1. Introduction

Finding frequent elements in subsets of a multiset is a fundamental operation for data analysis and data mining [2, 3]. When the sets have a certain

*An early partial version of this article appeared in *Proc. ISAAC 2018* [1].

*Corresponding author

Email addresses: travis.gagie@mail.udp.cl (Travis Gagie), mhe@cs.dal.ca (Meng He), gnavarro@dcc.uchile.cl (Gonzalo Navarro), cochoa@dcc.uchile.cl (Carlos Ochoa)

structure, it is possible to preprocess the multiset to build data structures that efficiently find the frequent elements in any subset.

The best studied multiset structure is the sequence, where the subsets that can be queried are ranges (i.e., contiguous subsequences) of the sequence. Applications of this case include time sequences, linear-versioned structures, and one-dimensional models, for example. Data structures for finding the *mode* (i.e., the most frequent element) in a range require time $O(\sqrt{n/\lg n})$, and it is unlikely that this can be done much better within reasonable extra space [4]. Instead, listing all the elements whose relative frequency in a range is over some fraction τ (called the τ-*majorities* of the range) is feasible within linear space and $O(1/\tau)$ time, which is worst-case optimal [5]. Mode and τ-majority queries on higher-dimensional arrays have also been studied [6, 4].

In this paper we focus on finding frequent elements when the subsets that can be queried are the labels on paths from one given node to another in a labeled tree. For example, given a minimum spanning tree of a graph, we might be interested in frequent node types on the path between two nodes. Path mode or τ-majority queries on multi-labeled trees could be useful when handling the tree of versions of a document or a piece of software, or a phylogenetic tree (which is essentially a tree of versions of a genome). If each node stores a list of the sections (i.e., chapters, modules, genes) on which its version differs from its parent's, then we can efficiently query which sections are changed most frequently between two given versions.

There has been relatively little previous work on finding frequent elements on tree paths. Krizanc *et al.* [7] considered path mode queries, obtaining $O(\sqrt{n}\lg n)$ query time. This was recently improved by Durocher *et al.* [8], who obtained $O(\sqrt{n/w}\lg\lg n)$ time on a RAM machine of $w = \Omega(\lg n)$ bits. As in the special case of sequences, these times are not likely to improve much. No previous work has considered the problem of finding path τ-majority queries, which is more tractable than finding the path mode. This is our focus.

We present the first data structures to support path τ-majority queries on trees of n nodes, with labels in $[1..\sigma]$, on a RAM machine. We first obtain a data structure using $O(n\lg n)$ space and $O((1/\tau)\lg\lg_w\sigma)$ time (Theorem 3). Building on this result, we manage to reduce the space to $O(n)$ without affecting the query time (Theorem 7). We then show that our linear-space data structure can be further compressed, to either $2nH + 4n + o(n)(H+1)$

bits or $nH + O(n) + o(nH)$ bits, where $H \leq \lg \sigma$ is the entropy of the distribution of the labels in T, while increasing the query time of the linear-space data structure only slightly, to $O((1/\tau) \lg^* n \lg \lg_w \sigma)$ (Theorems 8 and 9). Finally, we extend the succinct results so as to allow τ to be specified at query time, at the cost of just $o(n \lg \sigma)$ further bits of space (Theorems 11 and 12).

Durocher *et al.* [8] also considered queries that look for the least frequent elements and τ-minorities on paths. In Theorem 14, we slightly improve their query time to $O((1/\tau) \lg \lg_w \sigma)$ within linear space, and in Theorem 15 we show how to compress the data structure to fit in succinct space, with only a very slight increase in query time.

Finally, we describe how to adapt our results to multi-labeled trees and to path queries on functions, and discuss some open problems.

An early partial version of this paper appeared in *Proc. ISAAC 2018* [1]. This version includes an improved complexity for the linear-space version (so the super-linear space version of the conference paper becomes obsolete), which is also simplified. It also includes new results for τ specified at query time, for τ-minorities, and for extensions to path queries on functions. Finally, we have improved the writing and added more detail, fixed some minor errors.

2. Preliminaries

2.1. Definitions

We deal with *rooted ordinal trees* (or just *trees*) T. Further, our trees are *labeled*, that is, each node u of T has an integer label $\texttt{label}(u) \in [1..\sigma]$. We assume that, if our main tree has n nodes, then $\sigma = O(n)$; if not, we can remap the labels to a range of size at most n without altering the semantics of the queries of interest in this paper.

The *path* between nodes u and v in a tree T is the (only) sequence of nodes $P_{uv} = \langle u = z_1, z_2, \ldots, z_{k-1}, z_k = v \rangle$ such that there is an edge in T between each pair z_i and z_{i+1}, for $1 \leq i < k$. The length of the path is $|P_{uv}| = k$; for example, the length of the path P_{uu} is 1. Any path from u to v goes from u to the lowest common ancestor of u and v, and then from there it goes to v (if u is an ancestor of v or vice versa, one of these two subpaths is empty).

Given a real number $0 < \tau < 1$, a *τ-majority* of the path P_{uv} is any label that appears (strictly) more than $\tau \cdot |P_{uv}|$ times among the labels of the nodes

in P_{uv}. The *path τ-majority problem* is, given u and v, list all the τ-majorities in the path P_{uv}. Note that there can be up to $\lfloor 1/\tau \rfloor$ such τ-majorities.

Our results hold in the RAM model of computation, assuming a computer word of $w = \Omega(\lg n)$ bits, supporting the standard operations.

Our logarithms are to the base 2 by default. By $\lg^{[k]} n$ we mean the function that applies logarithm k times to n, i.e., $\lg^{[0]} n = n$ and $\lg^{[k]} n = \lg(\lg^{[k-1]} n)$. By $\lg^* n$ we denote the iterated logarithm, i.e., the minimum k such that $\lg^{[k]} n \leq 1$.

2.2. Sequence representations

A bitvector $B[1..n]$ can be represented within $n+o(n)$ bits so that the following operations take constant time: $\texttt{access}(B, i)$ returns $B[i]$, $\texttt{rank}_b(B, i)$ returns the number of times bit b appears in $B[1..i]$, and $\texttt{select}_b(B, j)$ returns the position of the jth occurrence of b in B [9]. If B has m 1s, then it can be represented within $m \lg(n/m) + O(m)$ bits while retaining the same operation times [10]. Note the space is $o(n)$ bits if $m = o(n)$. Those structures can be built in linear time.

Analogous operations are defined on sequences $S[1..n]$ over alphabets $[1..\sigma]$. For example, one can represent S within $nH + o(n)(H + 1)$ bits, where $H \leq \lg \sigma$ is the entropy of the distribution of symbols in S, so that `rank` takes time $O(\lg \lg_w \sigma)$, `access` takes time $O(1)$, and `select` takes any time in $\omega(1)$ [11, Thm. 8]. The construction takes linear time. While this `rank` time is optimal, we can answer *partial rank* queries in $O(1)$ time, $\texttt{prank}(S, i) = \texttt{rank}_{S[i]}(S, i)$, by adding $O(n(1 + \lg H))$ bits on top of a representation giving constant-time `access` [12, Sec. 3]. This construction requires linear randomized time.

2.3. Range τ-majorities on sequences

A special version of the path τ-majority queries on trees is range τ-majority queries on sequences $S[1..n]$, which have been studied in greater depth. Given i and j, the problem is to return all the distinct symbols that appear more than $\tau \cdot (j - i + 1)$ times in $S[i..j]$. The most recent result on this problem [13, 5] is a linear-space data structure, built in $O(n \lg n)$ time, that answers queries in the worst-case optimal time, $O(1/\tau)$.

For our succinct representations, we also use a data structure [5, Thm. 5.2] that requires $nH+o(n)(H+1)$ bits, and can answer range τ-majority queries in any time in $(1/\tau) \cdot \omega(1)$. The structure is built on the sequence representation mentioned above [11, Thm. 8], and thus it includes its support for

`access`, `rank`, and `select` queries on the sequence. To obtain the given times for τ-majorities, the structure includes the support for partial rank queries [12, Sec. 3], and therefore its construction time is randomized. In this paper, however, it will be sufficient to obtain $O((1/\tau)\lg\lg_w\sigma)$ time, and therefore we can replace their `prank` queries by general `rank` operations. These take time $O(\lg\lg_w\sigma)$ instead of $O(1)$, but can be built in linear time.[1] Therefore, this slightly slower structure can also be built in $O(n\lg n)$ deterministic time.

When a set has no structure, we can find its τ-majorities in linear time. Misra and Gries [14] proposed an optimal solution that computes all τ-majorities using $O(n\lg(1/\tau))$ comparisons. When implemented on a word RAM over an integer alphabet of size σ, the running time becomes $O(n)$ [3].

2.4. Tree operations

For tree nodes u and v, we define the operations `root` (the tree root), `parent`(u) (the parent of node u), `depth`(u) (the depth of node u, 0 being the depth of the root), `preorder`(u) (the rank of u in a preorder traversal of T), `postorder`(u) (the rank of u in a postorder traversal of T), `subtreesize`(u) (the number of nodes descending from u, including u), `anc`(u,d) (the ancestor of u at depth d), and `lca`(u,v) (the lowest common ancestor of u and v). All those operations can be supported in constant time and linear space on a static tree after a linear-time preprocessing, trivially with the exceptions of `anc` [15] and `lca` [16].

A less classical query is `labelanc`(u,ℓ), which returns the nearest ancestor of u (possibly u itself) labeled ℓ (note that the label of u need not be ℓ). If u has no ancestor labeled ℓ, `labelanc`(u,ℓ) returns *null*. This operation can be solved in time $O(\lg\lg_w\sigma)$ using linear space and preprocessing time [17, 18, 8].

2.5. Succinct tree representations

A tree T of n nodes can be represented as a sequence $P[1..2n]$ of parentheses (i.e., a bit sequence). In particular, we consider the balanced parentheses representation, where we traverse T in depth-first order, writing an opening parenthesis when reaching a node and a closing one when leaving its subtree. A node is identified with the position $P[i]$ of its opening parenthesis. By

[1] In fact, their structure [5] can be considerably simplified if one can spend the time of a general `rank` query per returned majority.

using $2n + o(n)$ bits, all the tree operations defined in Section 2.4 (except those on labels) can be supported in constant time [19].

This representation also supports `access`, `rank` and `select` on the bitvector of parentheses, and the operations $\texttt{close}(P, i)$ (the position of the parenthesis closing the one that opens at $P[i]$), $\texttt{open}(P, i)$ (the position of the parenthesis opening the one that closes at $P[i]$), and $\texttt{enclose}(P, i)$ (the position of the rightmost opening parenthesis whose corresponding parenthesis pair encloses $P[i]$; when P represents a tree, this parenthesis represents the parent of the node to which $P[i]$).

Labeled trees can be represented within $nH + 2n + o(n)(H + 1)$ bits by adding the sequence $S[1..n]$ of the node labels in preorder, so that $\texttt{label}(i) = \texttt{access}(S, \texttt{preorder}(i))$.

3. An $O(n \lg n)$-Space Solution

In this section we design a data structure answering path τ-majority queries on a tree of n nodes using $O(n \lg n)$ space and $O((1/\tau) \lg \lg_w \sigma)$ time. This introduces the basic ideas to obtain our final results.

We start by *marking* $O(\tau n)$ tree nodes, in a way that any node has a marked ancestor at distance $O(1/\tau)$. A simple way to obtain these bounds is to mark every node whose height is $\geq \lceil 1/\tau \rceil$ and whose depth is a multiple of $\lceil 1/\tau \rceil$. Therefore, every marked node is the nearest marked ancestor of at least $\lceil 1/\tau \rceil - 1$ distinct non-marked nodes, which guarantees that there are $\leq \tau n$ marked nodes. On the other hand, any node is at distance at most $2\lceil 1/\tau \rceil - 1$ from its nearest marked ancestor.

For each marked node x, we will consider prefixes $P_i(x)$ of the labels in the path from x to the root, of length $1 + 2^i$, that is,

$$P_i(x) = \langle \texttt{label}(x), \texttt{label}(\texttt{parent}(x)), \texttt{label}(\texttt{parent}^2(x)), \ldots, \texttt{label}(\texttt{parent}^{2^i}(x)) \rangle$$

(terminating the sequence at the root if we reach it). For each $0 \leq i \leq \lceil \lg \texttt{depth}(x) \rceil$, we store $C_i(x)$, the set of $(\tau/2)$-majorities in $P_i(x)$. Note that $|C_i(x)| \leq 2/\tau$ for any x and i.

By successive applications of the next lemma we have that, to find all the τ-majorities in the path from u to v, we can partition the path into several subpaths and then consider just the τ-majorities in each subpath.

Lemma 1. *Let u and v be two tree nodes, and let z be an intermediate node in the path. Then, a τ-majority in the path from u to v is a τ-majority in*

the path from u to z (including z) or a τ-majority in the path from z to v (excluding z), or in both.

Proof. Let d_{uz} be the distance from u to z (counting z) and d_{zv} be the distance from z to v (not counting z). Then the path from u to v is of length $d = d_{uz} + d_{zv}$. If a label ℓ occurs at most $\tau \cdot d_{uz}$ times in the path from u to z and at most $\tau \cdot d_{zv}$ times in the path from z to v, then it occurs at most $\tau(d_{uz} + d_{zv}) = \tau \cdot d$ times in the path from u to v. □

Let us now show that the candidates we record for marked nodes are sufficient to find path τ-majorities towards their ancestors.

Lemma 2. *Let x be a marked node. All the τ-majorities in the path from x to a proper ancestor z are included in $C_i(x)$ for some suitable i.*

Proof. Let $d_{xz} = \texttt{depth}(x) - \texttt{depth}(z)$ be the distance from x to z (i.e., the length of the path from x to z minus 1). Let $i = \lceil \lg d_{xz} \rceil$. The prefix $P_i(x)$ contains all the nodes in an upward path of length $1 + 2^i$ starting at x, where $d_{xz} \leq 2^i < 2d_{xz}$. Therefore, $P_i(x)$ contains node z, but its length is $|P_i(x)| < 1 + 2d_{xz}$. Therefore, any τ-majority in the path from x to z appears more than $\tau \cdot (1 + d_{xz}) > (\tau/2) \cdot (1 + 2d_{xz}) > (\tau/2) \cdot |P_i(x)|$ times, and thus it is a $(\tau/2)$-majority recorded in $C_i(x)$. □

3.1. Queries

With the properties above, we can find a candidate set of size $O(1/\tau)$ for the path τ-majorities between arbitrary tree nodes u and v. Let $z = \texttt{lca}(u, v)$. If $v \neq z$, let us also define $z' = \texttt{anc}(v, \texttt{depth}(z) + 1)$, that is, the child of z in the path to v. The path is then split into at most four subpaths, each of which can be empty:

1. The nodes from u to its nearest marked ancestor, x, not including x. If x does not exist or is a proper ancestor of z, then this subpath contains the nodes from u to z. The length of this path is less than $2\lceil 1/\tau \rceil$ by the definition of marked nodes, and it is empty if $u = x$.
2. The nodes from v to its nearest marked ancestor, y, not including y. If y does not exist or is an ancestor of z, then this subpath contains the nodes from v to z'. The length of this path is again less than $2\lceil 1/\tau \rceil$, and it is empty if $v = y$ or $v = z$.
3. The nodes from x to z. This path exists only if x exists and descends from z.

4. The nodes from y to z'. This path exists only if y exists and descends from z'.

By Lemma 1, any τ-majority in the path from u to v must be a τ-majority in some of these four paths. For the paths 1 and 2, we consider all their up to $2\lceil 1/\tau \rceil - 1$ nodes as candidates. For the paths 3 and 4, we use Lemma 2 to find suitable values i and j so that $C_i(x)$ and $C_j(y)$, both of size at most $2/\tau$, contain all the possible τ-majorities in those paths. In total, we obtain a set of at most $8/\tau + O(1)$ candidates that contain all the τ-majorities in the path from u to v.

In order to verify whether a candidate is indeed a τ-majority, we follow the technique of Durocher et al. [8]. Every tree node u will store $\texttt{count}(u)$, the number of times its label occurs in the path from u to the root. We also make use of the operation $\texttt{labelanc}(u, \ell)$. If u has no ancestor labeled ℓ, this operation returns *null*, and we define $\texttt{count}(null) = 0$. Therefore, the number of times label ℓ occurs in the path from u to an ancestor z of u (including z) can be computed as $\texttt{count}(\texttt{labelanc}(u, \ell)) - \texttt{count}(\texttt{labelanc}(\texttt{parent}(z), \ell))$. Each of our candidates can then be checked by counting their occurrences in the path from u to v using

$$(\texttt{count}(\texttt{labelanc}(u, \ell)) - \texttt{count}(\texttt{labelanc}(\texttt{parent}(z), \ell)))$$
$$+ (\texttt{count}(\texttt{labelanc}(v, \ell)) - \texttt{count}(\texttt{labelanc}(z, \ell))).$$

The time to perform query `labelanc` is $O(\lg \lg_w \sigma)$ using a linear-space data structure on the tree [17, 18, 8], and therefore we find all the path τ-majorities in time $O((1/\tau) \lg \lg_w \sigma)$.

The space of our data structure is dominated by the $O(\lg n)$ candidate sets $C_i(x)$ we store for the marked nodes x. These amount to $O((1/\tau) \lg n)$ space per marked node, of which there are $O(\tau n)$. Thus, we spend $O(n \lg n)$ space in total.

Theorem 3. *Let T be a tree of n nodes with labels in $[1..\sigma]$, and $0 < \tau < 1$. On a RAM machine of w-bit words, we can build an $O(n \lg n)$ space data structure that answers path τ-majority queries in time $O((1/\tau) \lg \lg_w \sigma)$.*

3.2. Construction

The construction of the data structure is easily carried out in linear time (including the fields `count` and the data structure to support `labelanc` [8]), except for the candidate sets $C_i(x)$ of the marked nodes x. We can compute the sets $C_i(x)$ for all i in total time $O(\texttt{depth}(x))$ using the linear-time

algorithm of Misra and Gries [14] because we compute $(\tau/2)$-majorities of doubling-length prefixes $P_i(x)$. This amounts to time $O(mt)$ on a tree of t nodes and m marked nodes. In our case, where $t = n$ and $m \leq \tau n$, this is $O(\tau n^2)$.

To reduce this time, we proceed as follows. First we build all the data structure components except the sets $C_i(x)$. We then decompose the tree into heavy paths [20] in linear time, and collect the labels along the heavy paths to form a set of sequences. On the sequences, we build in $O(t \lg t)$ time the range τ-majority data structure [13, 5] that answers queries in time $O(1/\tau)$. The prefix $P_i(x)$ for any marked node x then spans $O(\lg t)$ sequence ranges, corresponding to the heavy paths intersected by $P_i(x)$. We can then compute $C_i(x)$ by collecting and checking the $O(1/\tau)$ $(\tau/2)$-majorities from each of those $O(\lg t)$ ranges.

Let the path from x to the root be formed by $O(\lg t)$ heavy path segments $\pi_1, \ldots, \pi_k$. We first compute the $O(1/\tau)$ $(\tau/2)$-majorities in the sequences corresponding to each prefix $\pi_1, \ldots, \pi_k$: For each π_j, we (1) compute its $2/\tau$ majorities on the corresponding sequence in time $O(1/\tau)$, (2) add them to the set of $2/\tau$ majorities already computed for $\pi_1, \ldots, \pi_{j-1}$, and (3) check the exact frequencies of all the $4/\tau$ candidates in the path $\pi_1, \ldots, \pi_j$ in time $O((1/\tau) \lg \lg_w \sigma)$, using the structures already computed on the tree. All the $(\tau/2)$-majorities for $\pi_1, \ldots, \pi_j$ are then found.

Each prefix $P_i(x)$ is formed by some prefix $\pi_1, \ldots, \pi_{j-1}$ plus a prefix of π_j. We can then carry out a process similar to the one to compute the majorities of $\pi_1, \ldots, \pi_j$, but using only the proper prefix of π_j. The $O(\lg t)$ sets $C_i(x)$ are then computed in total time $O((1/\tau) \lg t \lg \lg_w \sigma)$. Added over the m marked nodes, we obtain $O((1/\tau)m \lg t \lg \lg_w \sigma)$ construction time.

Lemma 4. *On a tree of t nodes, m of which are marked, all the candidate sets $C_i(x)$ can be built in time $O((1/\tau)m \lg t \lg \lg_w \sigma)$.*

The construction time in our case, where $t = n$ and $m \leq \tau n$, is the following.

Corollary 5. *The data structure of Theorem 3 can be built in $O(n \lg n \lg \lg_w \sigma)$ time.*

4. A Linear-Space Solution

We can reduce the space of our data structure by stratifying our tree. First, let us create a separate structure to handle unary paths, that is, formed

by nodes with only one child. The labels of upward maximal unary paths are laid out in a sequence, and the sequences of the labels of all the unary paths in T are concatenated into a single sequence, S, of length at most n. On S we build the linear-space data structure that solves range τ-majority queries in time $O(1/\tau)$ [13, 5]. Each node in a unary path of T points to its position in S. Each node also stores a pointer to its nearest branching ancestor (i.e., one with more than one child).

The stratification then proceeds as follows. We say that a tree node is *large* if it has more than $(1/\tau)\lg n$ descendant nodes (counting itself); other nodes are *small*. Then the subset of the large nodes, which is closed by `parent`, induces a subtree T' of T with the same root and containing at most $\tau n/\lg n$ leaves, because for each leaf in T' there are at least $(1/\tau)\lg n - 1$ distinct nodes of T not in T'. Further, $T - T'$ is a forest of trees $\{F_j\}$, each of size at most $(1/\tau)\lg n$.

We will use for T' a structure similar to the one from Section 3, with some changes to ensure linear space. Note that T' may have $\Theta(n)$ nodes, but since it has at most $\tau n/\lg n$ leaves, T' has only $O(\tau n/\lg n)$ branching nodes. We modify the marking scheme, so that we mark precisely the branching nodes in T'. Spending $O((1/\tau)\lg n)$ space for the candidate sets $C_i(x)$ over all branching nodes of T' adds up to $O(n)$ space.

The procedure to solve path τ-majority queries on T' is then as follows. We split the path from u to v into four subpaths, exactly as in Section 3. The subpaths of type 1 and 2 can now be of arbitrary length, but they are unary, thus we obtain their (up to) $1/\tau$ candidates in time $O(1/\tau)$ from the corresponding range of S. Finally, we check all the $O(1/\tau)$ candidates in time $O((1/\tau)\lg\lg_w \sigma)$ as in Section 3.

The nodes u and v may, however, belong to some small tree F_j, which is of size $|F_j| \leq (1/\tau)\lg n$. We preprocess all those trees F_j in a way analogous to Section 3, using its same marking scheme to ensure that at most $\tau|F_j|$ nodes x are marked. The definition of the prefix $P_i(x)$, and consequently of their $(\tau/2)$-majorities $C_i(x)$, however, is slightly modified: $P'_i(x)$ is the sequence of the labels of the first $1 + 2^i/\tau$ nodes in the path from x to the root of its small subtree F_j, that is,

$$P'_i(x) = \langle \texttt{label}(x), \texttt{label}(\texttt{parent}(x)), \texttt{label}(\texttt{parent}^2(x)), \ldots, \texttt{label}(\texttt{parent}^{2^i/\tau}(x)) \rangle$$

terminating the sequence at the root of F_j if we reach it. Let $\texttt{depth}_{F_j}(x)$ be the depth of x within F_j. For each $0 \leq i \leq \lceil \lg(\tau \cdot \texttt{depth}_{F_j}(x)) \rceil \leq \lceil \lg(\tau|F_j|) \rceil \leq \lceil \lg\lg n \rceil$, we store $C'_i(x)$, the set of $(\tau/2)$-majorities in $P'_i(x)$.

The sizes $|C'_i(x)|$ are still at most $2/\tau$ for any x and i. Lemma 2 applies with $C'_i(x)$ as well, as we show next.

Lemma 6. *Let x be a marked node in a small tree F_j. All the τ-majorities in the path from x to a proper ancestor z in F_j at distance $d_{xz} > 1/(2\tau)$ are included in $C'_i(x)$ for some suitable i.*

Proof. Let $d_{xz} = \texttt{depth}(x) - \texttt{depth}(z)$ be the distance from x to z (i.e., the length of the path from x to z minus 1). Let $i = \lceil \lg(\tau \cdot d_{xz}) \rceil \geq 0$. The path $P'_i(x)$ contains all the nodes in an upward path of length $1 + 2^i/\tau$ starting at x, where $d_{xz} \leq 2^i/\tau < 2d_{xz}$. Therefore, $P'_i(x)$ contains node z, but its length is $|P'_i(x)| < 1 + 2d_{xz}$. Therefore, any τ-majority in the path from x to z appears more than $\tau \cdot (1 + d_{xz}) > (\tau/2) \cdot (1 + 2d_{xz}) > (\tau/2) \cdot |P'_i(x)|$ times, and thus it is a $(\tau/2)$-majority recorded in $C'_i(x)$. □

Note that, if $d_{xz} \leq 1/(2\tau)$, we do not need to use any $C'_i(x)$; we can simply collect all the $O(1/\tau)$ elements in the path from x to z.

If the $O(\lg \lg n)$ candidate sets $C'_i(x)$, for a marked node x, were stored as in Section 3, they would require $O((1/\tau) \lg \sigma \lg \lg n)$ bits. Instead of storing the candidate labels ℓ directly, however, we will store $\texttt{depth}(y)$, where y is the nearest ancestor of x with label ℓ. We can then recover $\ell = \texttt{label}(\texttt{anc}(x, \texttt{depth}(y)))$ in constant time. Since the depths in F_j are also $O((1/\tau) \lg n)$, we need only $O(\lg((1/\tau) \lg n))$ bits per candidate. Further, by sorting the candidates by their $\texttt{depth}(y)$ value, we can encode only the differences between consecutive depths using γ-codes [21]. Encoding k increasing numbers in $[1..t]$ with this method requires $O(k \lg(t/k))$ bits; therefore we can encode our $O(1/\tau)$ candidates using $O((1/\tau) \lg \lg n)$ bits in total. Added over all the $O(\lg \lg n)$ values of i, the candidates $C_i(x)$ require $O((1/\tau)(\lg \lg n)^2)$ bits per marked node. Added over all the $O(\tau|F_j|)$ marked nodes of F_j, this amounts to $O(|F_j|(\lg \lg n)^2)$ bits of space, and added over all the small trees F_j, this yields $O(n(\lg \lg n)^2)$ bits, or $o(n)$ words, in total. The other pointers of F_j, as well as node labels, can be represented normally, as they are $O(n)$ in total.

To solve a general path τ-majority query from u to v, we compute $z = \texttt{lca}(u, v)$ and process the path from u to z as follows:

- If u (and thus z) belongs to T', then we proceed on T' as explained.
- If z (and thus u) belongs to some small tree F_j, then we proceed on F_j as in Section 3, collecting $O(1/\tau)$ candidates in our path from u to

its nearest marked ancestor x, and then other $O(1/\tau)$ candidates from the corresponding set $C'_i(x)$.

- If u is in some F_j and z is in T', then let u' be the root of F_j (we have enough space to store a pointer to u' for each node u), whose parent is a leaf in T'. Then we collect $O(1/\tau)$ candidates in the path from u to u' using the mechanism of F_j, and then other $O(1/\tau)$ candidates in the path from the parent of u' to z using the mechanism of T'.

Other $O(1/\tau)$ candidates are collected analogously in the path from v to z', where z' is the child of z in the path to v, that is, $z' = \texttt{anc}(v, \texttt{depth}(z) + 1)$. Finally, all the candidates are checked as in Section 3, each in time $O(\lg \lg_w \sigma)$.

The time to build the structures on T', using the technique of Lemma 4, is $O(n \lg \lg_w \sigma)$ because T' has $t = O(n)$ nodes and $m = O(\tau n / \lg n)$ marked nodes. For the small trees F_j, we can use the $O(mt)$-time method described in the first paragraph of Section 3.2. Since on F_j it holds that $t = |F_j| \leq (1/\tau) \lg n$ and $m \leq \tau \cdot |F_j|$, the construction time is $O(|F_j| \lg n)$, which adds up to $O(n \lg n)$. Note that we also need $O(n \lg n)$ time to build the range majority data structure on S.

Theorem 7. *Let T be a tree of n nodes with labels in $[1..\sigma]$, and $0 < \tau < 1$. On a RAM machine of w-bit words, we can build in $O(n \lg n)$ time an $O(n)$ space data structure that answers path τ-majority queries in time $O((1/\tau) \lg \lg_w \sigma)$.*

Example. Figure 1 shows an example tree T where we have defined that a node is large if it has more than 7 descendant nodes (including itself). The large nodes of T form T', which has gray background. The branching nodes of T' are circled; those are the sampled nodes of T'. We have chosen the path between a small node u and a large node v. The node u is then within a small subtree F_j, rooted at u'. The path P_{uv} between u and v is split into several subpaths: (1) from u to u', which is handled within the subtree F_j (possibly with a mix of brute force and the use of a set $C'_i(\cdot)$); (2) from the parent of u' to its nearest marked ancestor x in T' (excluding x), which is a unary path and thus handled with a range query on the sequence S (not drawn); (3) from x to $z = \texttt{lca}(u, v)$, which is handled with a set $C_i(x)$ for some i; (4) from v to $z' = \texttt{anc}(v, \texttt{depth}(z) + 1)$, which is handled with a set $C_i(v)$ for some i, because v belongs to T' and is a sampled node.

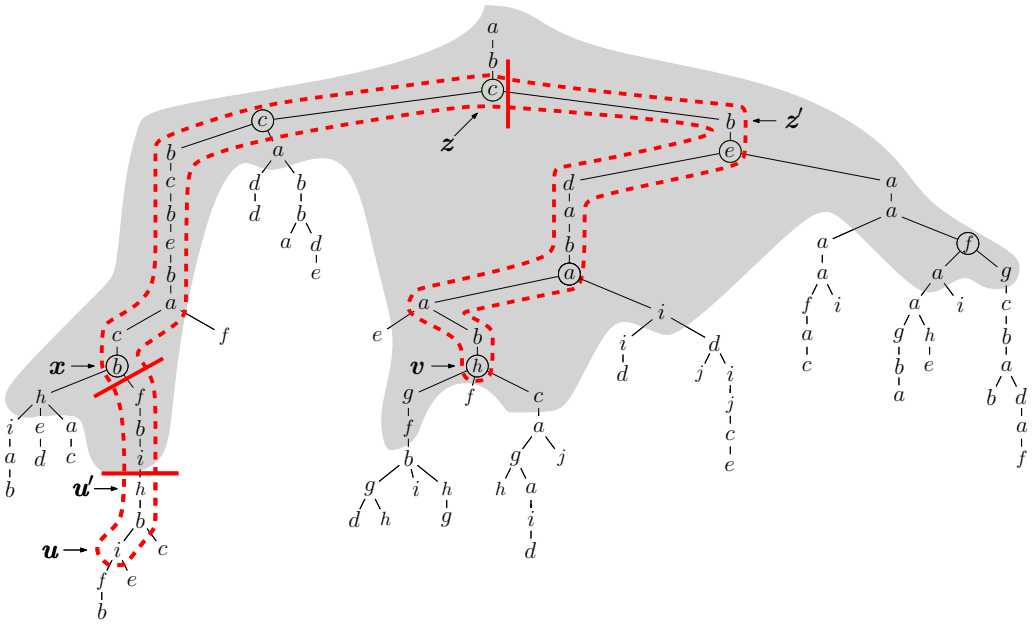

Figure 1: An example tree T where the labels are the letters of each node. The top tree T' of large nodes (with more than 7 descendants) has gray background. The path between two chosen nodes u and v is highlighted in dashed lines; note that u belongs to a small subtree F_j rooted at u', whereas v belongs to T'. We also show the nodes $z = \texttt{lca}(u, v)$ and $z' = \texttt{anc}(v, \texttt{depth}(z) + 1)$. Finally, we show the nearest branching ancestor x or u'.

Figure 2 shows the nodes included in each prefix $P_i(x)$ in the path from x to the root of T', for $i = 0$ to 4.

Assume $\tau = 1/3$. The subpaths (1) and (2) do not yield any candidate to τ-majority, since no label appears in more than a third of the subpath nodes. Instead, $C_4(x) = \{b, c\}$ (since b and c are the $(\tau/2 = 1/6)$-majorities in the path $P_4(x)$ from x to the root of T') and $C_3(v) = \{a, b\}$ (since a and b are the $(\tau/2 = 1/6)$-majorities in the path $P_3(v)$ from v to z'). We thus check the candidates a, b, and c, and report only b, because it appears $9 > \tau \cdot |P_{uv}| = (1/3) \cdot 25 = 8.\overline{3}$ times in P_{uv}.

5. A Succinct Space Solution

To obtain a succinct-space structure from Theorem 7, we increase the thresholds that define the large nodes in Section 4 and generalize the stratification to several levels. Let us say that the original tree T is of level 0. We now define the large nodes as those whose subtree size is larger than

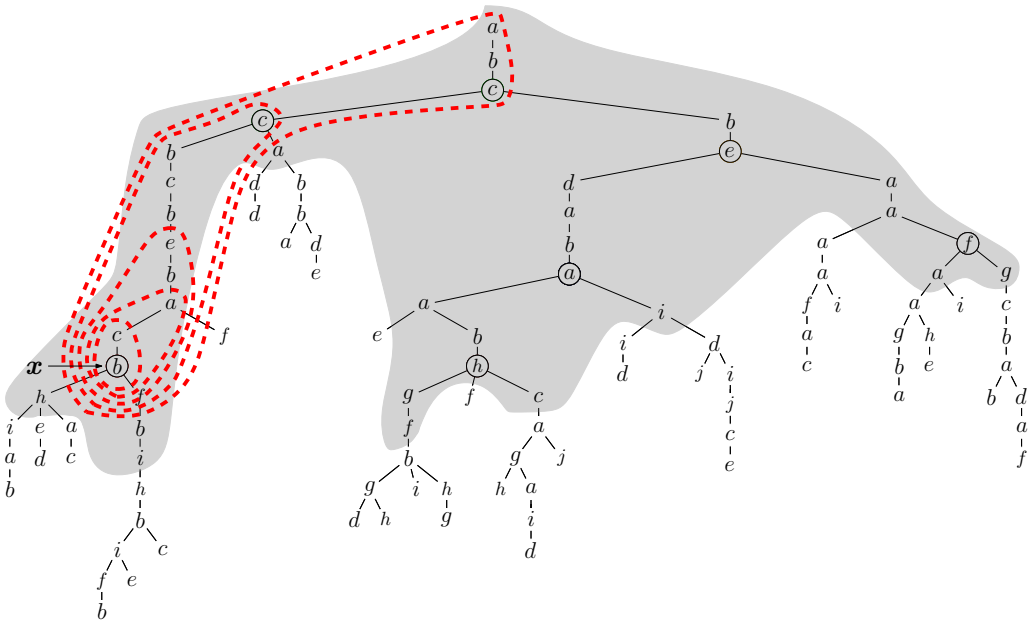

Figure 2: The precomputed prefixes $P_i(x)$, for $0 \le i \le 4$, on the tree T and node x of Figure 1. The corresponding $(\tau/2 = 1/6)$-majorities for the prefixes $P_i(x)$ are $C_0(x) = \{b, c\}$, $C_1(x) = \{a, b, c\}$, $C_2(x) = \{a, b, c, e\}$, $C_3(x) = \{b, c\}$, and $C_4(x) = \{b, c\}$.

$(1/\tau)(\lg n)^3$; these form the nodes corresponding to T' in Section 4. The small trees F_j of Section 4, which here are of size $\le (1/\tau)(\lg n)^3$, are said to be of level 1. We recursively apply the same stratification on the small trees F_j. On those, we define large nodes as those whose subtree size is larger than $(1/\tau)(\lg \lg n)^3$; the resulting small trees are said to be of level 2. We iterate this process κ times. In general, the trees of level $1 \le k \le \kappa$ are of size at most $(1/\tau)(\lg^{[k]} n)^3$. The large nodes of the trees of level $0 \le k < \kappa$ are those whose subtree size exceeds $(1/\tau)(\lg^{[k+1]} n)^3$. The smallest trees, of level κ, are of size $(1/\tau)(\lg^{[\kappa]} n)^3$ and are not further decomposed.

Level 0 can be handled exactly as T' in Section 4. In this case, since T' has $O(\tau n/\lg^3 n)$ branching nodes, the space for the sets $C_i(x)$ amounts to only $O(n/\lg n) = o(n)$ bits. In all the other levels, except the last one, we sample the branching nodes (as done for T' in Section 4), but build on them the sets $C'_i(x)$ (as done for the subtrees F_j in Section 4). A tree F of level $1 \le k < \kappa$ has $t \le (1/\tau)(\lg^{[k]} n)^3$ nodes and $m \le \tau \cdot |F|/(\lg^{[k+1]} n)^3$ branching nodes. The representation of a set $C'_i(x)$ in such a tree F, using the described differential encoding, takes $O((1/\tau)\lg((\lg^{[k]} n)^3)) = O((1/\tau)\lg^{[k+1]} n)$ bits. Added over

all the branching nodes, we obtain $O(|F|/(\lg^{[k+1]} n)^2)$ bits. Since every node belongs to one tree F, the total space amounts to $O(n/(\lg^{[\kappa]} n)^2)$ bits.

We aim to use about $\lg^* n$ levels. This will introduce a slowdown factor of the same order in query times, but in exchange the smallest trees will be small enough that they can be traversed by brute force, within the same penalty factor as well. We must carefully choose κ so as to also obtain $o(n)$ bits of space for all the sets $C'_i(x)$. Thus we set $\kappa = 1 + \lg^* n - \lg^{**} n$, so that there are $\kappa = O(\lg^* n)$ levels, and the last-level subtrees are of size $O((1/\tau)(\lg^{[\kappa]} n)^3) = O((1/\tau)(\lg^{[1+\lg^* n - \lg^{**} n]} n)^3) = O((1/\tau)(\lg \lg^* n)^3) = o((1/\tau)\lg^* n)$. Still, there are $O(\tau n/(\lg^{[\kappa]} n)^3) = O(\tau n/(\lg^{[1+\lg^* n - \lg^{**} n]} n)^3) = O(\tau n/(\lg \lg^* n)^3) = o(n)$ subtrees in the last level, and the space of all the sets $C'_i(x)$ is bounded by $O(n/(\lg^{[\kappa]} n)^2) = O(n/(\lg^{[1+\lg^* n - \lg^{**} n]} n)^2) = O(n/(\lg \lg^* n)^2) = o(n)$ bits.

The general process to solve a path τ-majority query from u to v is then as follows. We compute $z = \texttt{lca}(u, v)$ and split the path from u to z into $k - k' + 1$ subpaths, where k' and k (note $k' \leq k \leq \kappa$) are the levels of the subtree where z and u belong, respectively. Let us call u_i the root of the subtree of level i that is an ancestor of u, except that we call $u_{k'} = z$. For uniformity, the sets $C_i(x)$ of level 0 are called $C'_i(x)$ as well.

1. If $k = \kappa$, then u belongs to one of the smallest subtrees. We then collect the $o((1/\tau)\lg^* n)$ node labels in the path from u to u_κ one by one and include them in the set of candidates. We then move to the parent of that root, setting $u \leftarrow \texttt{parent}(u_\kappa)$ and $k \leftarrow \kappa - 1$.
2. At levels $k' \leq k < \kappa$, if u is a branching node, we collect the $2/\tau$ candidates from the corresponding set $C'_i(u)$, where i is sufficient to cover u_k ($C'_i(u)$ will not store candidates beyond the subtree root). We then set $u \leftarrow \texttt{parent}(u_k)$ and $k \leftarrow k - 1$.
3. At levels $k' \leq k < \kappa$, if u is not a branching node, let x be lowest between $\texttt{parent}(z)$ and the nearest branching ancestor of u. Let also p be the position of u in S. Then we find the $O(1/\tau)$ τ-majorities in $S[p..p + \texttt{depth}(u) - \texttt{depth}(x) - 1]$ in time $O((1/\tau)\lg \lg_w \sigma)$; see Section 5.1. We then continue from $u \leftarrow x$ and $k \leftarrow k(x)$, where $k(x)$ is the level of the subtree where x belongs. Note that $k(x)$ can be equal to k, but it can also be any other level less than k.
4. We stop when $u = \texttt{parent}(z)$.

A similar procedure is followed to collect the candidates from v to z', where again $z' = \texttt{anc}(v, \texttt{depth}(z) + 1)$ is the child of z in the path to v.

In total, since each path has at most one case 2 and one case 3 per level k, we collect at most $2\kappa = O(\lg^* n)$ candidate sets of size $O(1/\tau)$, plus two of size $o((1/\tau)\lg^* n)$. The total cost to verify all the candidates is then $O((1/\tau)\lg^* n \lg\lg_w \sigma)$.

The construction time, using Lemma 4 on level 0, is $O(n \lg\lg_w \sigma)$ as in Section 4. Applied on level 1, the lemma yields $O((n/(\lg\lg n)^3)\lg((1/\tau)\lg n)$ $\lg\lg_w \sigma) = o(n \lg n)$ construction time. For higher levels, we use the basic quadratic method described in the first paragraph of Section 3.2: a subtree F of level $2 \le k < \kappa$ is built in time $O(mt) = O(|F|(\lg^{[k]} n)^3/(\lg^{[k+1]} n)^3)$, which adds up to $O(n(\lg^{[k]} n)^3/(\lg^{[k+1]} n)^3)$ time for level k. This is maximized at level $k = 2$, yielding time $O(n(\lg\lg n/\lg\lg\lg n)^3) = o(n \lg n)$. All these costs are dominated by the $O(n \lg n)$ time to build the range majority data structure on S, which also absorbs the time to sort all the sets $C_i(x)$ by decreasing frequency.

We still need, however, to use succinct space for all the other linear-space components of the structure. The topology of the whole tree T can be represented using a sequence P of balanced parentheses in $2n + o(n)$ bits, supporting in constant time all the standard tree traversal operations we use [19]. We assume that opening and closing parentheses are represented with 1s and 0s in P, respectively. Let us now focus on the less standard operations needed.

5.1. Counting labels in paths

In Section 3, we count the number of times a label ℓ occurs in the path from u to the root by means of a query `labelanc` and by storing `count` fields in the nodes. In Section 4, we use in addition a string S to support range majority queries on the unary paths.

To solve `labelanc` queries, we use the representation of Durocher *et al.* [8, Lem. 7], which uses $nH+2n+o(n)(H+1)$ bits in addition to the $2n+o(n)$ bits of the tree topology. This representation includes a string $S[1..n]$ where all the labels of T are written in preorder; any implementation of S supporting `access`, `rank`, and `select` in time $O(\lg\lg_w \sigma)$ can be used (e.g., [11]). This string can also play the role of the one we call S in Section 4, because the labels of unary paths are contiguous in S, and any node v can access its label from $S[\texttt{preorder}(v)]$.

On top of this string we must also answer range τ-majority queries in time $O((1/\tau)\lg\lg_w \sigma)$. We can use the slow variant of the succinct structure described in Section 2.3, which requires only $o(n)(H+1)$ additional bits and

also supports `access` in $O(1)$ time and `rank` and `select` in time $O(\lg \lg_w \sigma)$. This variant of the structure is built in $O(n \lg n)$ time.

In addition to supporting operation `labelanc`, we need to store or compute the `count` fields. Durocher *et al.* [8] also require this field, but find no succinct way to represent it. We now show a way to obtain this value within succinct space.

The sequence S lists the labels of T in preorder, that is, aligned with the opening parentheses of P. Assume we have another sequence $S'[1..n]$ where the labels of T are listed in postorder (i.e., aligned with the closing parentheses of P). Since the opened parentheses not yet closed in $P[1..i]$ are precisely node i and its ancestors, we can compute the number of times a label ℓ appears in the path from $P[i]$ to the root as

$$\mathtt{rank}_\ell(S, \mathtt{rank}_1(P, i)) - \mathtt{rank}_\ell(S', \mathtt{rank}_0(P, i)).$$

Therefore, we can support this operation with $nH + o(n)(H+1)$ additional bits. Note that, with this representation, we do not need the operation `labelanc`, since we do not need that $P[i]$ itself is labeled ℓ.

If we do use operation `labelanc`, however, we can ensure that $P[i]$ is labeled ℓ, and another solution is possible based on partial rank queries. Let $o = \mathtt{rank}_\ell(S, \mathtt{rank}_1(P, i))$ and $c = \mathtt{rank}_\ell(S', \mathtt{rank}_0(P, i))$ be the numbers of opening and closing parentheses up to $P[i]$, respectively, so that we want to compute $o - c$. Since $P[i]$ is labeled ℓ, it holds that $S[\mathtt{rank}_1(P, i))] = \ell$, and thus $o = \mathtt{prank}(S, \mathtt{rank}_1(P, i))$. To compute c, we do not store S', but rather $S''[1..2n]$, so that $S''[i]$ is the label of the node whose opening or closing parenthesis is at $P[i]$ (i.e., S'' is formed by interleaving S and S'). Then, $\mathtt{prank}(S'', i) = o + c$; therefore the answer we seek is $o - c = 2 \cdot \mathtt{prank}(S, \mathtt{rank}_1(P, i)) - \mathtt{prank}(S'', i)$.

We use the structure for constant-time partial rank queries [12, Sec. 3] that requires $O(n) + o(nH)$ bits on top of a sequence that can be accessed in $O(1)$ time. We can build it on S and also on S'', though we do not explicitly represent S'': any access to S'' is simulated in constant time with $S''[i] = S[\mathtt{rank}_1(P, i)]$ if $P[i] = 1$, and $S''[i] = S[\mathtt{rank}_1(P, \mathtt{open}(P, i))]$ otherwise. This partial rank structure is built in $O(n)$ randomized time and in $O(n \lg n)$ time with high probability.[2]

[2]It involves building perfect hash functions, which succeeds with constant probability p in time $O(n)$. Repeating $c \lg n$ times, the failure probability is $1 - O(1/n^{c/\lg(1/p)})$.

5.2. Other data structures

The other fields stored at tree nodes, which we must now compute within succinct space, are the following:

Pointers to candidate sets $C'_i(x)$. All the branching nodes in all subtrees except those of level κ are marked, and there are $O(n/(\lg^{[\kappa]} n)^3) = o(n)$ such nodes. We can then mark their preorder ranks with 1s in a bitvector $M[1..n]$. Since M has $o(n)$ 1s, it can be represented within $o(n)$ bits [10] while supporting constant-time `rank` and `select` operations. We can then find out when a node i is marked (iff $M[\texttt{preorder}(i)] = 1$), and if it is, its rank among all the marked nodes, $r = \texttt{rank}_1(M, \texttt{preorder}(i))$. The $C'_i(x)$ sets of all the marked nodes x of any level can be written down in a contiguous memory area of total size $o(n)$ bits, sorted by the preorder rank of x. A bitvector C of length $o(n)$ marks the starting position of each new node x in this memory area. Then the area for marked node i starts at $p = \texttt{select}_1(C, r)$. A second bitvector D can mark the starting position of each $C'_j(x)$ in the memory area of each node x, and thus we access the specific set $C'_j(x)$ from position $\texttt{select}_1(D, \texttt{rank}_1(D, p-1) + j)$.

Pointers to subtree roots. We store an additional bitvector $B[1..2n]$, parallel to the parentheses bitvector $P[1..2n]$. In B, we mark with 1s the positions of the opening and closing parentheses that are roots of subtrees of any level. As there are $O(n/(\lg^{[\kappa]} n)^3) = o(n)$ such nodes, B can be represented within $o(n)$ bits while supporting constant-time `rank` and `select` operations. We also store the sequence of $o(n)$ parentheses P' corresponding to those in P marked with 1s in B. The nearest subtree root containing node $P[i]$ is obtained by finding the nearest position to the left that is marked in B, i.e., $j = \mathit{select}_1(B, r)$ with $r = \texttt{rank}_1(B, i)$, and then considering the corresponding position $P'[r]$. If it is an opening parenthesis, then the nearest subtree root is the node whose parenthesis opens in $P[j]$. Otherwise, it is the one opening at $P[j']$, where $j' = \texttt{select}_1(B, \texttt{enclose}(P', \texttt{open}(P', r)))$ (see [22, Sec. 4.1]).

Finding the nearest branching ancestor. A unary path looks like a sequence of opening parentheses followed by a sequence of closing parentheses. The nearest branching ancestor of $P[i]$ is obtained in constant time by finding the nearest closing parenthesis to the left, $l = \texttt{select}_0(\texttt{rank}_0(P, i))$, and the nearest opening parenthesis to the right, $r = \texttt{select}_1(\texttt{rank}_1(\texttt{close}(P, i)) + 1)$. Then the answer is the larger between $\texttt{enclose}(P, \texttt{open}(P, l))$ and $\texttt{enclose}(P, r)$.

Determining the subtree level of a node. We can compute $s = \texttt{subtreesize}(i)$ of a node $P[i]$ in constant time, so we can determine the corresponding level: if $s > (1/\tau)\lg^3 n$, it is level 1. Otherwise, we look up $\tau \cdot s$ in a precomputed table of size $O(\lg^3 n)$ that stores the level corresponding to each possible size.

Therefore, depending on whether we represent both S and S' or use partial rank structures, we obtain two results within succinct space.

Theorem 8. *Let T be a tree of n nodes with labels in $[1..\sigma]$, and $0 < \tau < 1$. On a RAM machine of w-bit words, we can build in $O(n \lg n)$ time a data structure using $2nH + 4n + o(n)(H+1)$ bits, where $H \leq \lg \sigma$ is the entropy of the distribution of the node labels, that answers path τ-majority queries in time $O((1/\tau)\lg^* n \lg \lg_w \sigma)$.*

Theorem 9. *Let T be a tree of n nodes with labels in $[1..\sigma]$, and $0 < \tau < 1$. On a RAM machine of w-bit words, we can build in $O(n \lg n)$ time (w.h.p.) a data structure using $nH + O(n) + o(nH)$ bits, where $H \leq \lg \sigma$ is the entropy of the distribution of the node labels, that answers path τ-majority queries in time $O((1/\tau)\lg^* n \lg \lg_w \sigma)$.*

We can also retain the same complexity of the linear-space version by using a constant number κ of levels, at the cost of using a slightly superlinear number of bits. In this case, we do not use brute force on the last-level trees, but rather combine the marking scheme of Section 3 with the storage format of the sets $C'_i(x)$. In this case, level κ requires $O(n(\lg^{[\kappa+1]} n)^2)$ bits and its $O(1/\tau)$ candidates are obtained as in Section 3. Next we write $O(n(\lg^{[\kappa+1]} n)^2) \subset O(n \lg^{[\kappa]} n)$ for simplicity.

Theorem 10. *Let T be a tree of n nodes with labels in $[1..\sigma]$, and $0 < \tau < 1$. On a RAM machine of w-bit words, for any constant κ, we can build in $O(n \lg n)$ time (w.h.p.) a data structure using $nH + O(n \lg^{[\kappa]} n) + o(nH)$ bits, where $H \leq \lg \sigma$ is the entropy of the distribution of the node labels, that answers path τ-majority queries in time $O((1/\tau)\lg \lg_w \sigma)$.*

We note that, within this space, all the typical tree navigation functionality, as well as access to labels, is supported.

6. Variable τ

Up to now, the value of τ is known at index construction time and cannot be changed later (we can obviously query for some $\tau' \geq \tau$ by using τ' when

verifying the candidates, but the time is still proportional to $1/\tau$). We aim at a structure that is independent of τ and can receive it together with the query nodes u and v, and answer in time proportional to $1/\tau$.

Note that, if $\tau = O(1/\sigma)$, we can simply test all the candidates of the alphabet in P_{uv} in time $O(\sigma \lg \lg_w \sigma) = O((1/\tau) \lg \lg_w \sigma)$. Therefore, we only care about values $\tau > 2/\sigma$.

Our solution builds on the succinct-space structure of Section 5. We build one copy of the data structure for each value $\tau' = 1/2^r$, $r \in [1..\lfloor \lg \sigma \rfloor]$. Then, given τ at query time, we use the structure with the value $\tau' = 1/2^{\lceil \lg(1/\tau) \rceil}$. This will return $O(1/\tau') = O(1/\tau)$ candidates, since $\tau' \geq \tau/2$ (the candidates are then checked with the exact τ value).

This solution increases the space by a factor of $\lg \sigma$. Note, however, that in the succinct solutions of Theorems 8 to 10, the space component $O(nH) + 4n + o(n)$ is due to the tree topology and the sequence S, which do not depend on the value of τ'. In particular, the representation of S is used to perform τ'-majority queries on the unary paths, but it allows τ' be specified at query time [5].

The structures that do depend on τ' (i.e., the information on levels and all the candidate sets $C'_i(x)$) require only $o(n)$ bits in Theorems 8 and 9, and $O(n \lg^{[\kappa]} n)$ bits in Theorem 10. These spaces stay succinct or near-succinct even after our space increase. We then obtain results close to those of Theorem 8 to 10, now for any τ specified at query time.

The construction time of the structure is $O(n(\lg \sigma / \lg \lg n)^2 + n \lg n)$, which includes the time to build $\lg \sigma$ copies of the $C'_i(x)$ structures.

Theorem 11. *Let T be a tree of n nodes with labels in $[1..\sigma]$. On a RAM machine of w-bit words, we can build in $O(n(\lg \sigma / \lg \lg n)^2 + n \lg n)$ time a data structure using $2nH + 4n + o(n \lg \sigma)$ bits, where $H \leq \lg \sigma$ is the entropy of the distribution of the node labels, that answers path τ-majority queries for any $0 < \tau < 1$, in time $O((1/\tau) \lg^* n \lg \lg_w \sigma)$.*

Theorem 12. *Let T be a tree of n nodes with labels in $[1..\sigma]$. On a RAM machine of w-bit words, we can build in $O(n(\lg \sigma / \lg \lg n)^2 + n \lg n)$ time (w.h.p.) a data structure using $nH + O(n) + o(n \lg \sigma)$ bits, where $H \leq \lg \sigma$ is the entropy of the distribution of the node labels, that answers path τ-majority queries for any $0 < \tau < 1$, in time $O((1/\tau) \lg^* n \lg \lg_w \sigma)$.*

Theorem 13. *Let T be a tree of n nodes with labels in $[1..\sigma]$. On a RAM machine of w-bit words, for any constant κ, we can build in $O(n(\lg \sigma / \lg \lg n)^2 +$*

$n \lg n$) time (w.h.p.) a data structure using $O(n \lg \sigma \lg^{\lceil \kappa \rceil} n)$ bits, that answers path τ-majority queries for any $0 < \tau < 1$, in time $O((1/\tau) \lg \lg_w \sigma)$.

7. Path τ-Minorities

A path τ-minority query asks for a τ-minority in a given path P_{uv}, that is, a label that appears at least once and at most $\tau \cdot |P_{uv}|$ times in this path. If we try $A = 1 + \lfloor 1/\tau \rfloor$ distinct elements in the path from u to v, then one of them will turn out to be a τ-minority. With this idea, we extend the technique of Chan *et al.* [23] to tree paths. To find a τ-minority, we will find A distinct labels (or all the labels, if there are not that many) in the path P_{uz}, where $z = \texttt{lca}(u, v)$, and check their frequency in P_{uv}. We then run an analogous process on the path P_{vz}. We will stop as soon as we find a label that is not a τ-majority. We describe the process on P_{uz}, as P_{vz} is analogous. Note that we need to know τ only at query time.

To find A distinct labels, we will simulate on P_{uz} the algorithm of Muthukrishnan [24], which finds A distinct elements in any range of an array E. In his algorithm, Muthukrishnan defines the array C where $C[i] = \max\{j < i,\ E[j] = E[i]\}$ ($C[i]$ is set to 0 if such a value does not exist) and builds on C a range minimum query (RMQ) data structure; a range minimum query asks for the minimum element in a given subrange of the array. Then he finds A (or all the) distinct elements in any range $E[i..j]$ via $O(A)$ RMQs.

In our case, we store for each node u the field

$$\texttt{prevlabel}(u) = \texttt{depth}(\texttt{labelanc}(\texttt{parent}(u), \texttt{label}(u)))\,,$$

which is the depth of the nearest ancestor of u with its same label (and -1 if there is none). Then we conceptually define E and C over P_{uz}, where $E[i] = \texttt{label}(\texttt{anc}(u, \texttt{depth}(z) - 1 + i))$ and $C[i] = 1 + \texttt{prevlabel}(\texttt{anc}(u, \texttt{depth}(z) - 1 + i))$. Note that we do not store E or C explicitly, but each entry of E or C can be computed in constant time using these formulas. To solve RMQs on C, we also build the linear-space data structure of Chazelle [25], which can return the minimum-weight node in any path of a weighted tree in constant time. This data structure is constructed over the tree T, for which we assign $\texttt{prevlabel}(u)$ as the weight of each node u. With all these structures, we can run Muthukrishnan's algorithm and obtain A distinct labels of P_{uz}. This yields our first result, which slightly reduces the $O((1/\tau) \lg \lg n)$ time (within linear space) of Durocher *et al.* [8]. Note that the `prevlabel` fields are easily computed in $O(n)$ time in a DFS traversal.

Theorem 14. *Let T be a tree of n nodes with labels in $[1..\sigma]$. On a RAM machine of w-bit words, we can build an $O(n)$ space data structure that answers path τ-minority queries for any $0 < \tau < 1$, in time $O((1/\tau)\lg\lg_w \sigma)$. The structure is built in linear time.*

It is likely that the result of Durocher *et al.* [8] can be improved to match ours, by just using a faster predecessor data structure. We can, however, make our solution succinct by using our tree representation of $2n + o(n)$ bits [19]. Instead of storing field `prevlabel`, we compute it on the fly with the given formula. Using the structures of Durocher *et al.* [8, Lem. 7], we can compute `labelanc` in time $O(\lg\lg_w \sigma)$. Their structure uses $2n + o(n)$ bits in addition to the topology of T and the representation of S.

The structure for RMQs, on the other hand, can be replaced by the one of Chan *et al.* [26], which uses $2n+o(n)$ further bits and answers RMQs with $O(\alpha(n))$ queries `prevlabel`(u), where α is the inverse Ackermann function. Therefore, we can spot the A candidates in time $O(A \cdot \alpha(n)\lg\lg_w \sigma)$ and then verify them in time $O(A \cdot \lg\lg_w \sigma)$. This yields the first result for path α-minority queries within succinct space.

Theorem 15. *Let T be a tree of n nodes with labels in $[1..\sigma]$. On a RAM machine of w-bit words, we can build in $O(n)$ time a data structure using $nH + 6n + o(n)(H + 1)$ bits, where $H \leq \lg\sigma$ is the entropy of the distribution of the node labels, that answers path τ-minority queries for any $0 < \tau < 1$, in time $O((1/\tau)\alpha(n)\lg\lg_w \sigma)$, where α is the inverse Ackermann function.*

8. Extensions

8.1. Multi-labeled trees

As mentioned in the Introduction, many applications of these results require that the trees are multi-labeled, that is, each node holds several labels. We can easily accommodate multi-labeled trees T in our data structure, by building a new tree T^* where each node u of T with $m(u)$ labels $\ell_1, \ldots, \ell_{m(u)}$ is replaced by an upward path of nodes $u_1, \ldots, u_{m(u)}$, each u_i holding the label ℓ_i and being the only child of u_{i+1} (and $u_{m(u)}$ being a child of v_1, where v is the parent of u in T). Path queries from u to v in T are then transformed into path queries from u_1 to v_1 in T^*, except when u (v) is an ancestor of v (u), in which case we replace u (v) by $u_{m(u)}$ ($v_{m(v)}$) in the query. All our complexities then hold on T^*, which is of size $n = |T^*| = \sum_{u \in T} m(u)$.

8.2. Queries on functions

Gagie et al. [27] consider path queries over a structure more general than trees. Let $f : [1..n] \to [1..n]$ be a function and $\ell : [1..n] \to [1..\sigma]$ an assignment of labels to the domain elements. The function defines a directed graph where nodes $v(i)$ are associated with the domain elements i and the edges lead from $v(i)$ to $v(f(i))$. The general form of these graphs is a set of cycles with trees sprouting from the cycle nodes (arrows point upwards, toward the cycles). We are interested in the so-called "positive path queries": given i and $0 \leq k_1 \leq k_2$, the path contains all the distinct elements $\{f^k(i),\ k_1 \leq i \leq k_2\}$. Our tree paths are then a particular case of positive path queries. They consider several queries on the labels of the path, and give general results like the following theorem.

Theorem 16. *[27] Let $f : [1..n] \to [1..n]$ be a function and $\ell : [1..n] \to [1..\sigma]$ an assignment of labels to the domain elements. Let there be a tree representation that computes in constant time the mapping between nodes and preorders, ancestor queries, depths of nodes, leftmost leaves of nodes, and lowest common ancestors, and in addition it solves a certain decomposable path query on n-node trees with labels in $[1..\sigma]$ in $T(n,\sigma)$ time, using in total $S(n,\sigma)$ bits of space. Then, there exists a data structure using $n \lg n + O(n) + S(n,\sigma)$ bits that answers the same query on the positive paths of f in time $O(\lg n / \lg\lg n) + T(n,\sigma)$. There exists another data structure using $n \lg n(1 + 1/t) + O(n) + S(n,\sigma)$ bits that answers the query in time $O(t) + T(n,\sigma)$, for any $t > 0$.*

Our results in this article allow, for the first time, using this result to answer τ-majority queries on the positive paths of functions. Although τ-majority queries are not decomposable (i.e., we cannot answer the query from the results on a partition of the path into subpaths), we can obtain a set of $O(1/\tau)$ candidates, with their frequencies, in each subpath of the partition. Lemma 1 shows that this is sufficient to find all the τ-majorities. In the result of Theorem 16, the query is partitioned into a constant number of subpaths; therefore we can use the result of Theorem 16 as if the query were decomposable. For example, combining it with Theorems 7 and 9, we obtain the following results.

Theorem 17. *Let $f : [1..n] \to [1..n]$ be a function and $\ell : [1..n] \to [1..\sigma]$ an assignment of labels to the domain elements. Then, there exists a data*

structure using $O(n)$ space that answers τ-majority queries on the positive paths of f in time $O((1/\tau)\lg\lg_w\sigma)$, on a w-bit word machine.

Theorem 18. *Let $f : [1..n] \to [1..n]$ be a function and $\ell : [1..n] \to [1..\sigma]$ an assignment of labels to the domain elements. Then, there exists a data structure using $n\lg n + O(n) + nH + o(nH)$ bits of space, where $H \le \lg\sigma$ is the entropy of the distribution of the values in ℓ, that answers τ-majority queries on the positive paths of f in time $O(\lg n/\lg\lg n + (1/\tau)\lg^* n\lg\lg_w\sigma)$, on a w-bit word machine.*

In some applications, however, the semantics of a positive path collecting the *unique* elements may be inappropriate. Instead, the adequate meaning may be the sequence $f^{[k_1..k_2]}(i) = \langle f^{k_1}(i), f^{k_1+1}(i), \ldots, f^{k_2}(i)\rangle$, of length exactly $k_2 - k_1 + 1$; this may include the same element several times if we enter a cycle, $f^k(i) = f^{k'}(i)$ for some $k_1 \le k < k' \le k_2$. Therefore we want the τ-majorities of the sequence $\langle \ell(f^{k_1}(i)), \ell(f^{k_1+1}(i)), \ldots, \ell(f^{k_2}(i))\rangle$.

We can still obtain the results of Theorems 17 and 18 under these semantics. The data structure of Gagie et al. [27, Sec 3.1] converts the cycle of the function graph into a tree: it cuts each cycle at an arbitrary edge $v_c \to v_1$ and writes the cycle as the leftmost path of a tree; each such node then has its sprouting subtree. They show how the data structure of Theorem 16 can compute in constant time the node $v = v(i)$, the node v' from where the tree of v sprouts from its cycle, the first and last nodes v_1 and v_c of the cycle, the length c of the cycle and the distance d between v and v'. With this information, we can easily separate the path $f^{[k_1..k_2]}(i)$ into three subpaths, each of which can be empty: (1) d_1 elements from $v(f^{k_1}(i))$ to v'; (2) $c \cdot r$ elements along r complete traversals of the cycle, from v' to its cycle predecessor; and (3) d_2 elements from v' to $v(f^{k_2}(i))$. The path then has length $k_2 - k_1 + 1 = d_1 + cr + d_2$, and by Lemma 1, any τ-majority in $f^{[k_1..k_2]}(i)$ must be a τ-majority in (1), (2), or (3). Further, the τ-majorities in (2) are exactly the τ-majorities in a single traversal of the cycle, that is, the path from v_1 to v_c. Thus we collect the $O(1/\tau)$ candidates along the tree paths of length d_1 (1), c (2), and d_2 (3), and each candidate x appearing x_1 times in (1), x_2 times in (2), and x_3 times in (3), is reported iff $x_1 + r\cdot x_2 + x_3 > \tau\cdot(k_2 - k_1 + 1)$.

9. Conclusions

We have presented the first data structures that can efficiently find the τ-majorities on the path between any two given nodes in a tree. Our data

structures use linear space, and even succinct space, whereas our query times are close to optimal, by a factor near log-logarithmic. We also obtained analogous results for path τ-minorities.

Our query time for path τ-majorities and τ-minorities in linear space, $O((1/\tau)\lg\lg_w \sigma)$, is over the optimal time $O(1/\tau)$ that can be obtained for the analogous range queries on sequences [5]. It is open whether we can obtain optimal time on trees within linear (or even near-linear) space. Another important open problem is how to support insertions and deletions of nodes in T while answering these queries, as achieved on sequences [28].

Acknowledgments

TG was funded by Fondecyt grant 1171058, Chile. MH was funded by NSERC, Canada. GN was funded by basal funds FB0001, Conicyt, Chile; Millennium Institute for Foundational Research on Data (IMFD), Chile; and Fondecyt grant 1170048, Chile. CO was funded by basal funds FB0001, Conicyt, Chile, and Fondecyt grant 1170048, Chile.

References

References

[1] T. Gagie, M. He, G. Navarro, Tree path majority data structures, in: Proc. 29th International Symposium on Algorithms and Computation (ISAAC), LIPICs 123, 2018, p. article 68.

[2] M. Fang, N. Shivakumar, H. Garcia-Molina, R. Motwani, J. D. Ullman, Computing iceberg queries efficiently, in: Proc. 24th VLDB, 1998, pp. 299–310.

[3] E. D. Demaine, A. López-Ortiz, J. I. Munro, Frequency estimation of internet packet streams with limited space, in: Proc. 10th ESA, 2002, pp. 348–360.

[4] T. M. Chan, S. Durocher, K. G. Larsen, J. Morrison, B. T. Wilkinson, Linear-space data structures for range mode query in arrays, Theor. Comp. Syst. 55 (4) (2014) 719–741.

[5] D. Belazzougui, T. Gagie, J. I. Munro, G. Navarro, Y. Nekrich, Range majorities and minorities in arrays, CoRR abs/1606.04495 (2016).

[6] T. Gagie, M. He, J. I. Munro, P. K. Nicholson, Finding frequent elements in compressed 2d arrays and strings, in: Proc. 18th SPIRE, 2011, pp. 295–300.

[7] D. Krizanc, P. Morin, M. H. M. Smid, Range mode and range median queries on lists and trees, Nordic J. Comp. 12 (1) (2005) 1–17.

[8] S. Durocher, R. Shah, M. Skala, S. V. Thankachan, Linear-space data structures for range frequency queries on arrays and trees, Algorithmica 74 (1) (2016) 344–366.

[9] D. R. Clark, Compact PAT trees, Ph.D. thesis, University of Waterloo, Canada (1996).

[10] R. Raman, V. Raman, S. S. Rao, Succinct indexable dictionaries with applications to encoding k-ary trees, prefix sums and multisets, ACM Trans. Alg. 3 (4) (2007) article 43.

[11] D. Belazzougui, G. Navarro, Optimal lower and upper bounds for representing sequences, ACM Trans. Alg. 11 (4) (2015) article 31.

[12] D. Belazzougui, G. Navarro, Alphabet-independent compressed text indexing, ACM Trans. Alg. 10 (4) (2014) article 23.

[13] D. Belazzougui, T. Gagie, G. Navarro, Better space bounds for parameterized range majority and minority, in: Proc. 12th WADS, 2013, pp. 121–132.

[14] J. Misra, D. Gries, Finding repeated elements, Sci. Comp. Prog. 2 (2) (1982) 143–152.

[15] M. Bender, M. Farach-Colton, The level ancestor problem simplified, Theor. Comp. Sci. 321 (1) (2004) 5–12.

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

[17] M. He, J. I. Munro, G. Zhou, A framework for succinct labeled ordinal trees over large alphabets, Algorithmica 70 (4) (2014) 696–717.

[18] D. Tsur, Succinct representation of labeled trees, Theor. Comp. Sci. 562 (2014) 320–329.

[19] G. Navarro, K. Sadakane, Fully-functional static and dynamic succinct trees, ACM Trans. Alg. 10 (3) (2014) article 16.

[20] D. Sleator, R. E. Tarjan, A data structure for dynamic trees, J. Comp. Sys. Sci. 26 (3) (1983) 362–391.

[21] T. C. Bell, J. Cleary, I. H. Witten, Text Compression, Prentice Hall, 1990.

[22] L. Russo, G. Navarro, A. Oliveira, Fully-compressed suffix trees, ACM Trans. Alg. 7 (4) (2011) article 53.

[23] T. M. Chan, S. Durocher, M. Skala, B. T. Wilkinson, Linear-space data structures for range minority query in arrays, Algorithmica 72 (4) (2015) 901–913.

[24] S. Muthukrishnan, Efficient algorithms for document retrieval problems, in: Proc. 13th SODA, 2002, pp. 657–666.

[25] B. Chazelle, Computing on a free tree via complexity-preserving mappings, Algorithmica 2 (1) (1987) 337–361.

[26] T. M. Chan, M. He, J. I. Munro, G. Zhou, Succinct indices for path minimum, with applications, Algorithmica 78 (2) (2017) 453–491.

[27] T. Gagie, M. He, G. Navarro, Path queries on functions, Theoretical Computer Science 770 (2019) 34–50.

[28] T. Gagie, M. He, G. Navarro, Compressed dynamic range majority and minority data structures, CoRR abs/1611.01835 (2018).