Tree Compression with Top Trees*

Philip Bille[1**], Inge Li Gørtz[1**], Gad M. Landau[2***], and Oren Weimann[2**]

[1] Technical University of Denmark, DTU Compute, {phbi,ilg}@dtu.dk
[2] University of Haifa, {oren,landau}@cs.haifa.ac.il

Abstract. We introduce a new compression scheme for labeled trees based on top trees [3]. Our compression scheme is the first to simultaneously take advantage of internal repeats in the tree (as opposed to the classical DAG compression that only exploits rooted subtree repeats) while also supporting fast navigational queries directly on the compressed representation. We show that the new compression scheme achieves close to optimal worst-case compression, can compress exponentially better than DAG compression, is never much worse than DAG compression, and supports navigational queries in logarithmic time.

1 Introduction

A labeled tree T is a rooted ordered tree with n nodes where each node has a label from an alphabet Σ. Many classical applications of trees in computer science (such as tries, dictionaries, parse trees, suffix trees, and XML databases) generate navigational queries on labeled trees (e.g, returning the label of node v, the parent of v, the depth of v, the size of v's subtree, etc.). In this paper we present new and simple compression scheme that support such queries directly on the compressed representation.

While a huge literature exists on string compression, labeled tree compression is much less studied. The simplest way to compress a tree is to serialize it using, say, preorder traversal to get a string of labels to which string compression can be applied. This approach is fast and is used in practice, but it does not support the various navigational queries. Furthermore, it does not capture possible repeats contained in the tree structure.

To get a sublinear space representation for trees with many repeated substructures (such as XML databases), one needs to define "repeated substructures" and devise an algorithm that identifies such repeats and collapses them (like Lempel-Ziv does to strings). There have been two main ways to define repeats: *subtree repeats* and the more general *tree pattern repeats* (see Fig. 1). A *subtree repeat* is an identical (both in structure and in labels) occurrence

* A draft of the full version of the paper can be found as Arxiv preprint arXiv:1304.5702

** Partially supported by the Danish Agency for Science, Technology and Innovation.

*** Partially supported by the National Science Foundation Award 0904246, Israel Science Foundation grant 347/09, Yahoo, Grant No. 2008217 from the United States-Israel Binational Science Foundation (BSF) and DFG.

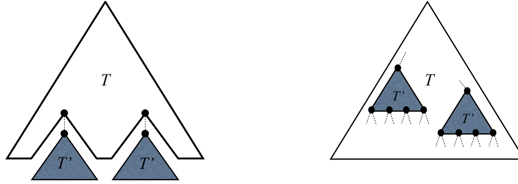

Fig. 1. A tree T with a *subtree* repeat T' (left), and a *tree pattern* repeat T' (right).

of a *rooted* subtree in T. A *tree pattern repeat* is an identical (both in structure and in labels) occurrence of any *connected subgraph* of T. Subtree repeats are used in DAG compression [8, 14] and tree patterns repeats in tree grammars [9, 10, 17–19]. In this paper we introduce *top tree compression* based top trees [3] that exploits tree pattern repeats. Compared to the existing techniques our compression scheme has the following advantages: Let T be a tree of size n with nodes labeled from an alphabet of size σ. We support navigational queries in $O(\log n)$ time (a similar result is not known for tree grammars), the compression ratio is in the worst case at least $\log_\sigma^{0.19} n$ (no such result is known for neither DAG compression or tree grammars), our scheme can compress exponentially better than DAG compression, and is never worse than DAG compression by more than a $\log n$ factor.

Previous Work Using subtree repeats, a node in the tree T that has a child with subtree T' can instead point to any other occurrence of T'. This way, it is possible to represent T as a directed acyclic graph (DAG). Over all possible DAGs that can represent T, the smallest one is unique and can be computed in $O(n)$ time [12]. Its size can be exponentially smaller than n. Using subtree repeats for compression was studied in [8,14], and a Lempel-Ziv analog of subtree repeats was suggested in [1]. It is also possible to support navigational queries [7] and path queries [8] directly on the DAG representation in logarithmic time.

The problem with subtree repeats is that we can miss many internal repeats. Consider for example the case where T is a single path of n nodes with the same label. Even though T is highly compressible (we can represent it by just storing the label and the length of the path) it does not contain a single subtree repeat and its minimal DAG is of size n.

Alternatively, *tree grammars* are capable of exploiting tree pattern repeats. Tree grammars generalize grammars from deriving strings to deriving trees and were studied in [9,10,17–19]. Compared to DAG compression a tree grammar can be exponentially smaller than the minimal DAG [17]. Unfortunately, computing a minimal tree grammar is NP-Hard [11], and all known tree grammar based compression schemes can only support navigational queries in time proportional to the height of the grammar which can be $\Omega(n)$.

Our Results. We propose a new compression scheme for labeled trees, which we call *top tree compression*. To the best of our knowledge, this is the first

compression scheme for trees that (i) takes advantage of tree pattern repeats (like tree grammars) but (ii) simultaneously supports navigational queries on the compressed representation in logarithmic time (like DAG compression). In the worst case, we show that (iii) the compression ratio of top tree compression is always at least $\log_\sigma^{0.19} n$ (compared to the information-theoretic lower bound of $\log_\sigma n$). This is in contrast to both tree grammars and DAG compression that do not have good provable worst-case compression performance. Finally, we compare the performance of top tree compression to DAG compression. We show that top tree compression (iv) can compress exponentially better than DAG compression, and (v) is never much worse than DAG compression.

With these features, top tree compression significantly improves the state-of-the-art for tree compression. Specifically, it is the first scheme to simultaneously achieve (i) and (ii) and the first scheme based on either subtree repeats or tree pattern repeats with provable good compression performance compared to worst-case (iii) or the DAG (iv).

The key idea in top tree compression is to transform the input tree T into another tree $\mathcal{T}$ such that tree pattern repeats in T become subtree repeats in $\mathcal{T}$. The transformation is based on top trees [2–4] – a data structure originally designed for dynamic (uncompressed) trees. After the transformation, we compress the new tree $\mathcal{T}$ using the classical DAG compression resulting in the *top DAG* $\mathcal{TD}$. The top DAG $\mathcal{TD}$ forms the basis for our compression scheme. We obtain our bounds on compression (iii), (iv), and (v) by analyzing the size of $\mathcal{TD}$, and we obtain efficient navigational queries (ii) by augmenting $\mathcal{TD}$ with additional data structures.

To state our bounds, let n_G denote the total size (vertices plus edges) of the graph G. We first show the following worst-case compression bound achieved by the top DAG.

Theorem 1. *Let T be any ordered tree with nodes labeled from an alphabet of size σ and let $\mathcal{TD}$ be the corresponding top DAG. Then, $n_{\mathcal{TD}} = O(n_T / \log_\sigma^{0.19} n_T)$.*

This worst-case performance of the top DAG should be compared to the optimal information-theoretic lower bound of $\Omega(n_T / \log_\sigma n_T)$. Note that with standard DAG compression the worst-case bound is $O(n_T)$ since a single path is incompressible using subtree repeats.

Secondly, we compare top DAG compression to standard DAG compression.

Theorem 2. *Let T be any ordered tree and let D and $\mathcal{TD}$ be the corresponding DAG and top DAG, respectively. For any tree T we have $n_{\mathcal{TD}} = O(\log n_T) \cdot n_D$ and there exist families of trees T such that $n_D = \Omega(n_T / \log n_T) \cdot n_{\mathcal{TD}}$.*

Thus, top DAG compression can be exponentially better than DAG compression and it is always within a logarithmic factor of DAG compression. To the best of our knowledge this is the first non-trivial bound shown for any tree compression scheme compared to the DAG.

Finally, we show how to represent the top DAG $\mathcal{TD}$ in $O(n_{\mathcal{TD}})$ space such that we can quickly answer a wide range of queries about T without decompressing.

Theorem 3. *Let T be an ordered tree with top DAG $\mathcal{TD}$. There is an $O(n_{\mathcal{TD}})$ space representation of T that supports* Access*,* Depth*,* Height*,* Size*,* Parent*,* Firstchild*,* NextSibling*,* LevelAncestor*, and* NCA *in $O(\log n_T)$ time. Furthermore, we can* Decompress *a subtree T' of T in time $O(\log n_T + |T'|)$.*

The Access, Depth, Height, Size, Parent, Firstchild, and NextSibling all take a node v in T as input and return its label, its depth, its height, the size of its subtree, its parent, its first child, and its immediate sibling, respectively. The LevelAncestor returns an ancestor at a specified distance from v, and NCA returns the nearest common ancestor to a given pair of nodes. Finally, the Decompress operation decompresses and returns any rooted subtree.

Related work (Succinct data structures) Jacobson [16] was the first to observe that the naive pointer-based tree representation using $\Theta(n \log n)$ bits is wasteful. He showed that *unlabeled* trees can be represented using $2n + o(n)$ bits and support various queries by inspection of $\Theta(\lg n)$ bits in the bit probe model. This space bound is asymptotically optimal with the information-theoretic lower bound averaged over all trees. Munro and Raman [20] showed how to achieve the same bound in the RAM model while using only constant time for queries. Such representations are called *succinct data structures*, and have been generalized to trees with higher degrees [5] and to a richer set of queries such as subtree-size queries [20] and level-ancestor queries [15]. For *labeled* trees, Ferragina et al. [13] gave a representation using $2n \log \sigma + O(n)$ bits that supports basic navigational operations, such as find the parent of node v, the i'th child of v, and any child of v with label α.

All the above bounds for space are averaged over all trees and do not take advantage of the cases where the input tree contains many repeated substructures. The focus of this paper is achieving sublinear bounds in trees with many repeated substructures (i.e., highly compressible trees).

2 Top Trees and Top DAGs

Top trees were introduced by Alstrup et al. [2–4] for maintaining an uncompressed, unordered, and unlabeled tree under link and cut operations. We extend them to ordered and labeled trees, and then introduce top DAGs for compression. Our construction is related to well-known algorithms for top tree construction, but modified for our purposes. In particular, we need to carefully order the steps of the construction to guarantee efficient compression, and we disallow some combination of cluster merges to ensure fast navigation.

Clusters Let v be a node in T with children $v_1, \ldots, v_k$ in left-to-right order. Define $T(v)$ to be the subtree induced by v and all proper descendants of v. Define $F(v)$ to be the forest induced by all proper descendants of v. For $1 \leq s \leq r \leq k$ let $T(v, v_s, v_r)$ be the tree pattern induced by the nodes $\{v\} \cup T(v_s) \cup T(v_{s+1}) \cup \cdots \cup T(v_r)$.

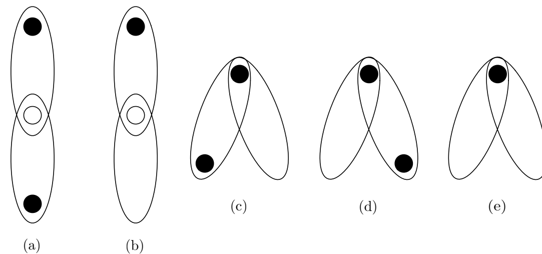

Fig. 2. Five ways of merging clusters. The $\bullet$ nodes are boundary nodes that remain boundary nodes in the merged cluster. The $\circ$ nodes are boundary nodes that become internal (non-boundary) nodes in the merged cluster.

A *cluster* with *top boundary node* v is a tree pattern of the form $T(v, v_s, v_r)$, $1 \leq s \leq r \leq k$. A *cluster* with *top boundary node* v and *bottom boundary node* u is a tree pattern of the form $T(v, v_s, v_r) \setminus F(u)$, $1 \leq s \leq r \leq k$, where u is a node in $T(v_s) \cup \cdots \cup T(v_r)$. Clusters can therefore have either one or two boundary nodes. For example, let $p(v)$ denote the parent of v then a single edge $(v, p(v))$ of T is a cluster where $p(v)$ is the top boundary node. If v is a leaf then there is no bottom boundary node, otherwise v is a bottom boundary node.

Two edge disjoint clusters A and B whose vertices overlap on a single boundary node can be *merged* if their union $C = A \cup B$ is also a cluster. There are five ways of merging clusters, as illustrated by Fig. 2. The original paper on top trees [2–4] contains more ways to merge clusters, but allowing these would lead to a violation of our definition of clusters as a tree pattern of the form $T(v, v_s, v_r) \setminus F(u)$, which we need for navigational purposes.

Top Trees A *top tree* $\mathcal{T}$ of T is a hierarchical decomposition of T into clusters. It is an ordered, rooted, and binary tree and is defined as follows.

- The nodes of $\mathcal{T}$ correspond to clusters of T.
- The root of $\mathcal{T}$ is the cluster T itself.
- The leaves of $\mathcal{T}$ correspond to the edges of T. The label of each leaf is the pair of labels of the endpoints of the edges in T.
- Each internal node of $\mathcal{T}$ is a merged cluster of its two children. The label of each internal node is the type of merge it represents (out of the five merging options). The children are ordered so that the left child is the child cluster visited first in a preorder traversal of T.

Constructing the Top Tree We now describe a greedy algorithm for constructing a top tree $\mathcal{T}$ of T that has height $O(\log n_T)$. The algorithm constructs the top tree $\mathcal{T}$ bottom-up in $O(\log n_T)$ iterations starting with the edges of T as the leaves of $\mathcal{T}$. During the construction, we maintain an auxiliary tree $\widetilde{T}$ initialized as $\widetilde{T} := T$. The edges of $\widetilde{T}$ will correspond to the nodes of $\mathcal{T}$ and to

the clusters of T. In the beginning, these clusters represent actual edges $(v, p(v))$ of T. In this case, if v is not a leaf in T then v is the bottom boundary node of the cluster and $p(v)$ is the top boundary node. If v is a leaf then there is no bottom boundary node.

In each one of the $O(\log n_T)$ iterations, a constant fraction of $\widetilde{T}$'s edges (i.e., clusters of T) are merged. Each merge is performed on two overlapping edges (u, v) and (v, w) of $\widetilde{T}$ using one of the five types of merges from Fig. 2: If v is the parent of u and the only child of w then a merge of type (a) or (b) contracts these edges in $\widetilde{T}$ into the edge (u, w). If v is the parent of both u and w, and w or u are leaves, then a merge of type (c), (d), or (e) replaces these edges in $\widetilde{T}$ with either the edge (u, v) or (v, w). In all cases, we create a new node in $\mathcal{T}$ whose two children are the clusters corresponding to (u, v) and to (v, w).

This way, we get that a single iteration shrinks the tree $\widetilde{T}$ (and the number of parentless nodes in $\mathcal{T}$) by a constant factor. The process ends when $\widetilde{T}$ is a single edge. Each iteration is performed as follows:

Step 1: Horizontal Merges. For each node $v \in \widetilde{T}$ with $k \geq 2$ children $v_1, \ldots, v_k$, for $i = 1$ to $\lfloor k/2 \rfloor$, merge the edges (v, v_{2i-1}) and (v, v_{2i}) if v_{2i-1} or v_{2i} is a leaf. If k is odd and v_k is a leaf and both v_{k-2} and v_{k-1} are non-leaves then also merge (v, v_{k-1}) and (v, v_k).

Step 2: Vertical Merges. For each maximal path $v_1, \ldots, v_p$ of nodes in $\widetilde{T}$ such that v_{i+1} is the parent of v_i and $v_2, \ldots, v_{p-1}$ have a single child: If p is even merge the following pairs of edges $\{(v_1, v_2), (v_2, v_3)\}, \{(v_3, v_4), (v_4, v_5)\}, \ldots, (v_{p-2}, v_{p-1})\}$. If p is odd merge t$\{(v_1, v_2), (v_2, v_3)\}, \{(v_3, v_4), (v_4, v_5)\}, \ldots, (v_{p-3}, v_{p-2})\}$, and if (v_{p-1}, v_p) was not merged in Step 1 then also merge $\{(v_{p-2}, v_{p-1}), (v_{p-1}, v_p)\}$.

Lemma 1. *A single iteration shrinks $\widetilde{T}$ by a factor of $c \geq 8/7$.*

Proof. Suppose that in the beginning of the iteration the tree $\widetilde{T}$ has n nodes. Any tree with n nodes has at least $n/2$ nodes with less than 2 children. Consider the edges $(v_i, p(v_i))$ of $\widetilde{T}$ where v_i has one or no children. We show that at least half of these $n/2$ edges are merged in this iteration. This will imply that $n/4$ edges of $\widetilde{T}$ are replaced with $n/8$ edges and so the size of $\widetilde{T}$ shrinks to $7n/8$. To prove it, we charge each edge $(v_i, p(v_i))$ that is not merged to a unique edge $f(v_i, p(v_i))$ that is merged.

Case 1. Suppose that v_i has no children (i.e., is a leaf). If v_i has at least one sibling and $(v_i, p(v_i))$ is not merged it is because v_i has no right sibling and its left sibling v_{i-1} has already been merged (i.e., we have just merged $(v_{i-2}, p(v_{i-2}))$ and $(v_{i-1}, p(v_{i-1}))$ in Step 1 where $p(v_i) = p(v_{i-1}) = p(v_{i-2})$). We also know that at least one of v_{i-1} and v_{i-2} must be a leaf. We set $f(v_i, p(v_i)) = (v_{i-1}, p(v_{i-1}))$ if v_{i-1} is a leaf, otherwise we set $f(v_i, p(v_i)) = (v_{i-2}, p(v_{i-2}))$.

Case 2. Suppose that v_i has no children (i.e., is a leaf) and no siblings (i.e., $p(v_i)$ has only one child). The only reason for not merging $(v_i, p(v_i))$ with $(p(v_i), p(p(v_i)))$ in Step 2 is because $(p(v_i), p(p(v_i)))$ was just merged in Step 1. In this case, we set $f(v_i, p(v_i)) = (p(v_i), p(p(v_i)))$. Notice that we haven't already charged $(p(v_i), p(p(v_i))$ in *Case 1* because $p(v_i)$ is not a leaf.

Case 3. Suppose that v_i has exactly one child $c(v_i)$ and that $(v_i, p(v_i))$ was not merged in Step 1. The only reason for not merging $(v_i, p(v_i))$ with $(c(v_i), v_i)$ in Step 2 is if $c(v_i)$ has only one child $c(c(v_i))$ and we just merged $(c(v_i), v_i)$ with $(c(c(v_i)), c(v_i))$. In this case, we set $f(v_i, p(v_i)) = (c(v_i), v_i)$. Notice that we haven't already charged $(c(v_i), v_i)$ in *Case 1* because $c(v_i)$ is not a leaf. We also haven't charged $(c(v_i), v_i)$ in *Case 2* because v_i has only one child. □

Corollary 1. *Given a tree T, the greedy top tree construction creates a top tree of size $O(n_T)$ and height $O(\log n_T)$ in $O(n_T)$ time.*

The next lemma follows from the construction of the top tree and Lemma 1.

Lemma 2. *For any node c in the top tree corresponding to a cluster C of T, the total size of all clusters corresponding to nodes in the subtree $\mathcal{T}(c)$ is $O(|C|)$.*

Top Dags The *top DAG* of T, denoted $\mathcal{TD}$, is the minimal DAG representation of the top tree $\mathcal{T}$. It can be computed in $O(n_T)$ time from $\mathcal{T}$ using the algorithm of [12]. The entire top DAG construction can thus be done in $O(n_T)$ time.

3 Compression Analysis

Worst-case Bounds for Top Dag Compression We now prove Theorem 1.

Let $\mathcal{T}$ be the top tree for T. Identical clusters in T are represented by identical complete subtrees in $\mathcal{T}$. Since identical subtrees in $\mathcal{T}$ are shared in $\mathcal{TD}$ we have the following lemma.

Lemma 3. *For any tree T, all clusters in the corresponding top DAG $\mathcal{TD}$ are distinct.*

Lemma 4. *Let T be any tree with n_T nodes labeled from an alphabet of size σ and let $\mathcal{T}$ be its top tree. The nodes of $\mathcal{T}$ correspond to at most $O(n_T / \log_\sigma^{0.19} n_T)$ distinct clusters in T.*

Proof. Consider the bottom-up construction of the top tree $\mathcal{T}$ starting with the leaves of $\mathcal{T}$ (the clusters corresponding to the edges of T). By Lemma 1 each level in the top tree reduces the number of clusters by a factor $c = 8/7$, while at most doubling the size of the current clusters. After round i we are therefore left with at most $O(n_T/c^i)$ clusters, each of size at most $2^i + 1$.

To bound the total number of distinct cluster, we partition the clusters into *small clusters* and *large clusters*. The small clusters are those created in rounds 1

to $j = \log_2(0.5 \log_{4\sigma} n_T)$ and the large clusters are those created in the remaining rounds from $j+1$ to h. The total number of large clusters is at most

$$\sum_{i=j+1}^{h} O(n_T/c^i) = O(n_T/c^{j+1}) = O(n_T/\log_\sigma^{0.19} n_T).$$

In particular, there are at most $O(n_T/\log_\sigma^{0.19} n_T)$ distinct clusters among these.

Next, we bound the total number of distinct small clusters. Each small cluster corresponds to a connected subgraph (i.e., a tree pattern) of T that is of size at most 2^j+1 and is an ordered and labeled tree. The total number of distinct ordered and labeled trees of size at most x is given by

$$\sum_{i=1}^{x} \sigma^i C_{i-1} = \sum_{i=1}^{x} \frac{\sigma^i}{i}\binom{2i-1}{i-1} = \sum_{i=1}^{x} O(4^i\sigma^i) = O\left((4\sigma)^{x+1}\right),$$

where C_i denotes the ith Catalan number. Hence, the total number of distinct small clusters is bounded by $O((4\sigma)^{2^j+2}) = O(\sigma^2\sqrt{n_T}) = O(n_T^{3/4})$. In the last equality we used the fact that $\sigma < n_T^{1/8}$. If $\sigma > n_T^{1/8}$ then the lemma trivially holds because $O(n_T/(\log_\sigma^{0.19} n_T)) = O(n_T)$. We get that the total number of distinct clusters is at most $O(n_T/\log_\sigma^{0.19} n_T + n_T^{3/4}) = O(n_T/\log_\sigma^{0.19} n_T)$. $\square$

Combining Lemma 3 and 4 we obtain Theorem 1.

Comparison to Subtree Sharing We now prove Theorem 2. To do so we first show two useful properties of top trees and top dags.

Let T be a tree with top tree $\mathcal{T}$. For any internal node z in T, we say that the subtree $T(z)$ is *represented* by a set of clusters $\{C_1, \ldots, C_\ell\}$ from $\mathcal{T}$ if $T(z) = C_1 \cup \cdots \cup C_\ell$. Since each edge in T is a cluster in $\mathcal{T}$ we can always trivially represent $T(z)$ by at most $|T(z)|-1$ clusters. We prove that there always exists a set of clusters, denoted S_z, of size $O(\log n_T)$ that represents $T(z)$.

Let z be any internal node in T and let z_1 be its leftmost child. Since z is internal we have that z is the top boundary node of the leaf cluster $L = (z, z_1)$ in $\mathcal{T}$. Let U be the smallest cluster in $\mathcal{T}$ containing all nodes of $T(z)$. We have that L is a descendant leaf of U in $\mathcal{T}$. Consider the path P of cluster in $\mathcal{T}$ from U to L. An *off-path* cluster of P is a cluster C that is not on P, but whose parent cluster is on P. We define

$$S_z = \{C \mid C \text{ is off-path cluster of } P \text{ and } C \subseteq T(z)\} \cup \{L\}$$

Since the length of P is $O(\log n_T)$ the number of clusters in S_z is $O(\log n_T)$. We need to prove that $\cup_{C\in S_z} C = T(z)$. By definition we have that all nodes in $\cup_{C\in S_z} C$ are in $T(z)$. For the other direction, we first prove the following lemma. Let $E(C)$ denote the set of edges of a cluster C.

Lemma 5. *Let C be an off-path cluster of P. Then either $E(C) \subseteq E(T(z))$ or $E(C) \cap E(T(z)) = \emptyset$.*

Proof. We will show that any cluster in $\mathcal{T}$ containing edges from both $T(z)$ and $T \setminus T(z)$ contains both $(p(z), z)$ and (z, z_1), where z_1 is the leftmost child of z and $p(z)$ is the parent of z. Let C be a cluster containing edges from both $T(z)$ and $T \setminus T(z)$. Consider the subtree $\mathcal{T}(C)$ and let C' be the smallest cluster containing edges from both $T(z)$ and $T \setminus T(z)$. Then C' must be a merge of type (a) or (b), where the higher cluster A only contains edges from $T \setminus T(z)$ and the bottom cluster, B, only contains edges from $T(z)$. Also, z is the top boundary node of B and the bottom boundary node of A. Clearly, A contains the edge $(p(z), z)$, since all clusters are connected tree patterns. A merge of type (a) or (b) is only possible when B contains all children of its top boundary node. Thus B contains the edge (z, z_1).

We have $L = (z, z_1)$ and therefore all clusters in $\mathcal{T}$ containing (z, z_1) lay on the path from L to the root. The path P is a subpath of this path, and thus no off-path clusters of P can contain (z, z_1). Therefore no off-path clusters of P can contain edges from both $T(z)$ and $T \setminus T(z)$. □

Any edge from $T(z)$ (except (z, z_1)) contained in a cluster on P must be contained in an off-path cluster of P. Lemma 5 therefore implies that $T(z) = \cup_{C \in S_z} C$ and the following corollary.

Corollary 2. *Let T be a tree with top tree $\mathcal{T}$. For any node z in T, the subtree $T(z)$ can be represented by a set of $O(\log n_T)$ clusters in $\mathcal{T}$.*

Next we prove that our bottom-up top tree construction guarantees that two identical subtrees $T(z), T(z')$ are represented by two *identical* sets of clusters $S_z, S_{z'}$. Two sets of clusters are identical (denoted $S_z = S_{z'}$) when $C \in S_z$ iff $C' \in S_z$ such that C and C' are clusters corresponding to tree patterns in T that have the same structure and labels.

Lemma 6. *Let T be a tree with top tree $\mathcal{T}$. Let $T(z)$ and $T(z')$ be identical subtrees in T and let S_z and $S_{z'}$ be the corresponding representing set of clusters in $\mathcal{T}$. Then, $S_z = S_{z'}$.*

Proof. Omitted due to lack of space.

Theorem 4. *For any tree T, $n_{\mathcal{TD}} = O(\log n_T) \cdot n_D$.*

Proof. An edge is shared in the DAG if it is in a shared subtree of T. We denote the edges in the DAG D that are shared as *red* edges, and the edges that are not shared as *blue*. Let r_D and b_D be the number of red and blue edges in the DAG D, respectively.

A cluster in the top tree $\mathcal{T}$ is *red* if it only contains red edges from D, *blue* if it only contains blue edges from D, and *purple* if it contains both. Since clusters are connected subtrees we have the property that if cluster C is red (resp. blue), then all clusters in the subtree $\mathcal{T}(C)$ are red (resp. blue). Let r, b, and p be the number of red, blue, and purple clusters in $\mathcal{T}$, respectively. Since $\mathcal{T}$ is a binary tree, where all internal nodes have 2 children and all leaves are either red or

blue, we have $p \leq r + b$. It is thus enough to bound the number of red and blue clusters.

First we bound the number of red clusters in the DAG $\mathcal{TD}$. Consider a shared subtree $T(z)$ from the DAG compression. $T(z)$ is represented by at most $O(\log n_T)$ clusters in $\mathcal{T}$, and all these contain only edges from $T(z)$. It follows from Lemma 6 that all the clusters representing $T(z)$ (and their subtrees in $\mathcal{T}$) are identical for all copies of $T(z)$. Therefore each of these will appear only once in the top DAG $\mathcal{TD}$.

From Corollary 2 we have that each of the clusters representing $T(z)$ has size at most $O(|T(z)|)$. Thus, the total size of the subtrees of the clusters representing $T(z)$ is $O(|T(z)| \log n_T)$. This is true for all shared subtrees, and thus $r = O(r_D \log n_T)$.

To bound the number of blue clusters in the top DAG, we first note that the blue clusters form rooted subtrees in the top tree. Let C be the root of such a blue subtree in $\mathcal{T}$. Then C is a connected component of blue edges in T. It follows from Corollary 2 that $|\mathcal{T}(C)| = O(|C|)$. Thus the number of blue clusters $b = O(b_D)$. The number of edges in the $\mathcal{TD}$ is thus $b + r + p \leq 2(b + r) = O(b_D + r_D \log n_T) = O(n_D \log n_T)$. ☐

Lemma 7. *There exist trees T, such that $n_D = \Omega(n_T / \log n_T) \cdot n_{\mathcal{TD}}$.*

Proof. Caterpillars and paths have $n_{\mathcal{TD}} = O(\log n_T)$, whereas $n_D = n_T$. ☐

4 Supporting Navigational Queries

In this section we give a high-level sketch of how to perform the navigational queries. All details can be found in the full version [6]. Let T be a tree with top DAG $\mathcal{TD}$. To uniquely identify nodes of T we refer to them by their preorder numbers. For a node of T with preorder number x we want to support the following queries.

$\mathsf{Access}(x)$**:** Return the label associated with node x.
$\mathsf{Decompress}(x)$**:** Return the tree $T(x)$.
$\mathsf{Parent}(x)$**:** Return the parent of node x.
$\mathsf{Depth}(x)$**:** Return the depth of node x.
$\mathsf{Height}(x)$**:** Return the height of node x.
$\mathsf{Size}(x)$**:** Return the number of nodes in $T(x)$.
$\mathsf{Firstchild}(x)$**:** Return the first child of x.
$\mathsf{NextSibling}(x)$**:** Return the sibling immediately to the right of x.
$\mathsf{LevelAncestor}(x, i)$**:** Return the ancestor of x whose distance from x is i.
$\mathsf{NCA}(x, y)$**:** Return the nearest common ancestor of the nodes x and y.

The Data Structure In order to enable the above queries, we augment the top DAG $\mathcal{TD}$ of T with some additional information. Consider a cluster C in $\mathcal{TD}$. Recall that if C is a leaf in $\mathcal{TD}$ then C is a single edge in T and C stores

the labels of this edge's endpoints. For each internal cluster C in $\mathcal{TD}$ we save information about which type of merge that was used to construct C, the height and size of the tree pattern C, and the distance between the top of C to the top boundary nodes of its child clusters. From the definition of clusters and tree patterns it follows that the preorder numbers in T of all the nodes (except possibly the root) in a cluster/tree pattern C are either a consecutive sequence $[i_1, \ldots, i_r]$ or two consecutive sequences $[i_1, \ldots, i_r], [i_s, \ldots, i_t]$, where $r < s$. In the second case the nodes with preorder numbers $[i_{r+1}, \ldots, i_{s-1}]$ are nodes in the subtree rooted at the bottom boundary node of C. We augment our data structure with information that allows us to compute the sequence(s) of preorder numbers in C in constant time.

All of our queries are based on traversals of the augmented top DAG $\mathcal{TD}$. During the traversal we identify nodes by computing preorder numbers local to the cluster that we are currently visiting. The intervals of local preorder numbers can be computed in constant time for each cluster.

Our data structures uses constant space for each cluster of $\mathcal{TD}$, and thus the total space remains $O(n_{\mathcal{TD}})$.

Navigation The navigational queries can be grouped into three types. Access and Depth are computed by a single top-down search of $\mathcal{TD}$ starting from its root and ending with the leaf cluster containing x. At each step the local preorder number of x is computed, along with some additional information in the case of Depth. The procedures Firstchild, LevelAncestor, Parent, and NCA are computed by a top-down search to find the local preorder number in a relevant cluster C, and then a bottom-up search to compute the corresponding preorder number in T. Since the depth of $\mathcal{TD}$ is $O(\log n_T)$ and the computation in each cluster of $\mathcal{TD}$ takes constant time the total time for each operation is $O(\log n_T)$.

To answer the queries Decompress, Height, Size, and NextSibling the key idea is to compute a small set of clusters representing $T(x)$ and all necessary information needed to answer the queries. The set is a subset denoted $\check{S}_x$ of the set S_x, where S_x is the set of $O(\log n_T)$ off-path clusters of P that represents $T(x)$ as defined in Section 3. Let M be the highest cluster on P that only contains edges from $T(x)$ and partition S_x into the set $\hat{S}_x$ that contains all clusters in S_x that are descendants of M and the set $\check{S}_x$ that contains the remaining clusters. The desired subset is the set $\check{S}_x$. We then show how to implement a procedure FindRepresentatives that efficiently computes $\check{S}_x$ in $O(\log n_T)$ time. This is done by a top-down traversal of $\mathcal{TD}$ to find M followed by a bottom-up traversal that exploits the structural properties of the top tree. Note that not all off-path clusters of P are in S_x. Therefore we carefully need to consider for each off-path cluster in the bottom-up traversal, if it is in S_x or not. With FindRepresentatives, the remaining procedures are straightforward to implement using the information from the clusters in $\check{S}_x$.

References

1. J. Adiego, G. Navarro, and P. de la Fuente. Lempel-Ziv compression of highly structured documents. *J. Amer. Soc. Inf. Sci. and Techn.*, 58(4):461–478, 2007.
2. S. Alstrup, J. Holm, K. de Lichtenberg, and M. Thorup. Minimizing diameters of dynamic trees. In *Proc. 24th ICALP*, pages 270–280, 1997.
3. S. Alstrup, J. Holm, K. D. Lichtenberg, and M. Thorup. Maintaining information in fully-dynamic trees with top trees. *ACM Trans. Algorithms*, 1:243–264, 2003.
4. S. Alstrup, J. Holm, and M. Thorup. Maintaining center and median in dynamic trees. In *Proc. 7th SWAT*, pages 46–56, 2000.
5. D. Benoit, E. Demaine, I. Munro, R. Raman, V. Raman, and S. Rao. Representing trees of higher degree. *Algorithmica*, 43:275–292, 2005.
6. P. Bille, I. L. Gørtz, G. M. Landau, and O. Weimann. Tree compression with top trees. *Arxiv preprint arXiv:1304.5702*, 2013.
7. P. Bille, G. Landau, R. Raman, S. Rao, K. Sadakane, and O. Weimann. Random access to grammar-compressed strings. In *Proc. 22nd SODA*, pages 373–389, 2011.
8. P. Buneman, M. Grohe, and C. Koch. Path queries on compressed XML. In *Proc. 29th VLDB*, pages 141–152, 2003.
9. G. Busatto, M. Lohrey, and S. Maneth. Grammar-based tree compression. Technical report, EPFL, 2004.
10. G. Busatto, M. Lohrey, and S. Maneth. Efficient memory representation of XML document trees. *Information Systems*, 33(4-5):456–474, 2008.
11. M. Charikar, E. Lehman, A. Lehman, D. Liu, R. Panigrahy, M. Prabhakaran, A. Sahai, and A. Shelat. The smallest grammar problem. *IEEE Trans. Inform. Theory*, 51(7):2554–2576, 2005.
12. P. J. Downey, R. Sethi, and R. E. Tarjan. Variations on the common subexpression problem. *J. ACM*, 27:758–771, 1980.
13. P. Ferragina, F. Luccio, G. Manzini, and S. Muthukrishnan. Compressing and indexing labeled trees, with applications. *J. ACM*, 57:1–33, 2009.
14. M. Frick, M. Grohe, and C. Koch. Query evaluation on compressed trees. In *Proc. 18th LICS*, pages 188–197, 2003.
15. R. Geary, R. Raman, and V. Raman. Succinct ordinal trees with level-ancestor queries. In *Proc. 15th SODA*, pages 1–10, 2004.
16. G. Jacobson. Space-efficient static trees and graphs. In *Proc. 30th FOCS*, pages 549–554, 1989.
17. M. Lohrey and S. Maneth. The complexity of tree automata and XPath on grammar-compressed trees. *Theoret. Comput. Sci.*, 363(2), 2006.
18. M. Lohrey, S. Maneth, and R. Mennicke. Tree structure compression with repair. *Arxiv preprint arXiv:1007.5406*, 2010.
19. S. Maneth and G. Busatto. Tree transducers and tree compressions. In *Proc. 7th FOSSACS*, pages 363–377, 2004.
20. J. I. Munro and V. Raman. Succinct representation of balanced parentheses and static trees. *SIAM J. Comput.*, 31(3):762–776, 2001.