Information and Computation 243 (2015) 166–177

Contents lists available at ScienceDirect

Information and Computation

www.elsevier.com/locate/yinco

Tree compression with top trees

Philip Bille [a,1], Inge Li Gørtz [a,2], Gad M. Landau [b,3], Oren Weimann [b,*,4]

[a] DTU Compute, Technical University of Denmark, Denmark
[b] Department of Computer Science, University of Haifa, Israel

ARTICLE INFO

Article history:
Received 13 August 2013
Available online 6 January 2015

Keywords:
Tree compression
Top trees
Pattern matching

ABSTRACT

We introduce a new compression scheme for labeled trees based on top trees. 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, where each node has a label from an alphabet Σ. Labeled trees appear in computer science as tries, dictionaries, parse trees, suffix trees, XML trees, etc. In this paper, we study compression schemes for labeled trees that take advantage of *repeated substructures* and support navigational queries, such as returning the label of a node v, the parent of v, the depth of v, the size v's subtrees, etc., directly on the compressed representation. We consider the following two basic types of repeated substructures (see Fig. 1). The first type is used in DAG compression [1,2] and the second in tree grammars [3–7].

Subtree repeat. A *rooted subtree* is a subgraph of T consisting of a node and all its descendants. A *subtree repeat* is an identical (both in structure and in labels) occurrence of a rooted subtree in T.

Tree pattern repeat. A *tree pattern* is any connected subgraph of T. A *tree pattern repeat* is an identical (both in structure and in labels) occurrence of a tree pattern in T.

In this paper, we introduce a simple new compression scheme, called *top tree compression*, 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

[*] Corresponding author.
E-mail addresses: phbi@dtu.dk (P. Bille), inge@dtu.dk (I.L. Gørtz), landau@cs.haifa.ac.il (G.M. Landau), oren@cs.haifa.ac.il (O. Weimann).

[1] Partially supported by the Danish Agency for Science, Technology and Innovation.

[2] Partially supported by the Danish Agency for Science, Technology and Innovation.

[3] 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.

[4] Partially supported by the Israel Science Foundation grant 794/13.

http://dx.doi.org/10.1016/j.ic.2014.12.012
0890-5401/© 2015 Elsevier Inc. All rights reserved.

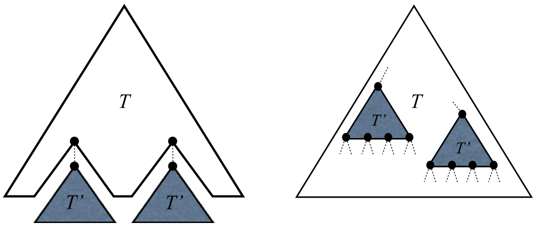

Fig. 1. A tree T with a *subtree* repeat T' (left), and a *tree pattern* repeat T' (right).

either DAG compression or tree grammars), our scheme can compress exponentially better than DAG compression, and the compression ratio is never worse than DAG compression by more than a $\log n$ factor.

1.1. Previous work

The previous work on tree compression can be described by three major approaches: using subtree repeats, using tree pattern repeats, and using succinct data structures. Below we briefly discuss these approaches and the existing tree compression schemes. Extensive practical work has recently been done on all these tree compression schemes (see e.g., the recent survey of Sakr [8]).

DAG compression. 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 [9]. Its size can be exponentially smaller than n. DAG representation of trees are broadly used for identifying and sharing common subexpressions, e.g., in programming languages [10] and binary decision diagrams [11]. Compression based on DAGs has also been studied more recently in [1,2,12] and a Lempel–Ziv analog of subtree repeats was suggested in [13]. It is possible to support navigational queries [14] and path queries [1] 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 path length) it does not contain a single subtree repeat and its minimal DAG is of size n.

Tree grammars. Alternatively, *tree grammars* are capable of exploiting tree pattern repeats. Tree grammars generalize grammars from deriving strings to deriving trees and were studied in [3–7]. Compared to DAG compression, a tree grammar can be exponentially smaller than the minimal DAG [6]. Unfortunately, computing a minimal tree grammar is NP-Hard [15], 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 $\Omega(n)$.

Succinct data structures. A different approach to tree compression is *succinct data structures* that compactly encode trees. 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. Munro and Raman [17] 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 include a richer set of queries such as subtree-size queries [17,18] and level-ancestor queries [19]. For *labeled* trees, Ferragina et al. [20] 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. Ferragina et al. also introduced the notion of k'th order *tree entropy* H_k in a restricted model. In this model, used by popular XML compressors [21,22], the label of a node is a function of the labels of all its ancestors. For such a tree T, Ferragina et al. gave a representation requiring at most $nH_k(T) + 2.01n + o(n)$ bits. Note that the above space bounds do not guarantee a compact representation when the input contains many subtree repeats or tree pattern repeats. In particular, the total space is never $o(n)$ bits.

1.2. 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 bound of $\log_\sigma n$). This is in contrast to both tree grammars and DAG compression that have not yet been proven to have worst-case compression performance comparable to the information-theoretic bound. 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 worse than DAG compression by more than a $\log n$ factor.

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 [23–25] – 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 assume a standard word RAM model of computation with logarithmic word size. All space complexities refer to the number of words used by the data structure. 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 information-theoretic lower bound of $\Omega(n_T / \log_\sigma n_T)$. This lower bound applies already for strings (so it clearly holds for labeled trees). It is obtained by simply noticing that there are $\Omega(\sigma^{n_T})$ string of length n_T over an alphabet of size σ, implying a lower bound of $\Omega(n_T \log \sigma)$ bits or $\Omega(n_T / \log_\sigma n_T)$ words. Note that with standard DAG compression the worst-case bound is $\Theta(n_T)$ 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 (since it's possible that $n_D = O(\log n_T)$) 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 operations Access, Depth, Height, Size, Parent, Firstchild, and NextSibling all take a node v in T as input[5] and return its label, its depth, its height, the size of its subtree, its parent, its first child, and its sibling to the right, 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.

2. Top trees and top DAGs

Top trees were introduced by Alstrup et al. [23–25] 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.

2.1. 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 $1 \leqslant s \leqslant r \leqslant k$ let $T(v, v_s, v_r)$ be the tree pattern induced by the nodes $\{v\} \cup T(v_s) \cup T(v_{s+1}) \cup \cdots \cup T(v_r)$.

A *cluster* with *top boundary node* v is a tree pattern of the form $T(v, v_s, v_r)$, $1 \leqslant s \leqslant r \leqslant k$. A *cluster* with *top boundary node* v and *bottom boundary node* u is a tree pattern of the form $T(v, v_s, v_r) \setminus F(u)$, $1 \leqslant s \leqslant r \leqslant k$, 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. Nodes that are not boundary nodes are called *internal* nodes.

[5] The nodes of T are uniquely identified by their preorder numbers. See Section 4.

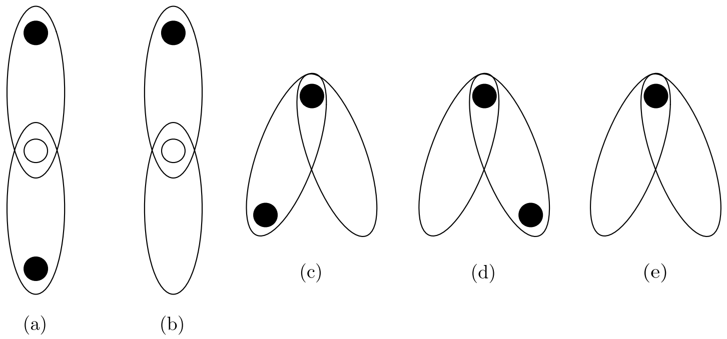

Fig. 2. Five ways of merging clusters. The • nodes are boundary nodes that remain boundary nodes in the merged cluster. The ◦ nodes are boundary nodes that become internal (non-boundary) nodes in the merged cluster. Note that in the last four merges at least one of the merged clusters has a top boundary node but no 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 in Fig. 2. Merges of type (a) and (b) can be done if the common boundary node is not a boundary node of any other cluster except A and B. Merges of type (c), (d), and (e) can be done only if at least one of A or B does not have a bottom boundary node. The original paper on top trees [23–25] 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.

2.2. Top trees

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

- The nodes of $\mathcal{T}$ correspond to clusters of T.
- The root of $\mathcal{T}$ corresponds to 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 its corresponding edge (u, v) in T. The two labels are ordered so that the label of the parent appears before the label of the child.
- Each internal node of $\mathcal{T}$ corresponds to the 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.

2.3. 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, $\mathcal{T}$ is a forest, and we maintain an auxiliary rooted ordered 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. The internal nodes of $\widetilde{T}$ will correspond to boundary nodes of clusters in T, and the leaves of $\widetilde{T}$ will correspond to a subset of the leaves 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).

We prove below that a single iteration shrinks the tree $\widetilde{T}$ (and the number of roots 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 $k \geqslant 2$ 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 the following pairs of edges $\{(v_1, v_2), (v_2, v_3)\}, \{(v_3, v_4), (v_4, v_5)\}, \ldots, \{(v_{p-3}, v_{p-2})\}$, 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 $c \geqslant 8/7$.*

Proof. Suppose that in the beginning of the iteration the tree $\widetilde{T}$ has n nodes. Any tree with n nodes has at least $(n+1)/2$ 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. □

Since each iteration can be done in linear time and shrinks $\widetilde{T}$ by a factor > 1 we obtain the following.

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 number of nodes in the subtree $\mathcal{T}(c)$ is $O(|C|)$.*

2.4. 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 $O(n_{\mathcal{T}})$ time from $\mathcal{T}$ using the algorithm of [9]. The entire top DAG construction can thus be done in $O(n_T)$ time.

3. Compression analysis

3.1. Worst-case bounds for top DAG compression

We now prove Theorem 1. Let T be an ordered tree with n_T nodes labeled from an alphabet of size σ, let $\mathcal{T}$ be its top tree and $\mathcal{TD}$ be its top DAG. We call two rooted subtrees of $\mathcal{T}$ *identical* if they have the same structure and labels, otherwise they are called *distinct*. To show that the size of $\mathcal{TD}$ is at most $O(n_T/\log_\sigma^{0.19} n_T)$ is suffices to show that $\mathcal{T}$ has only $O(n_T/\log_\sigma^{0.19} n_T)$ distinct rooted subtrees.

Recall that each node in the top tree $\mathcal{T}$ corresponds to a cluster in T. A leaf of $\mathcal{T}$ corresponds to a cluster of a single edge of T and is labeled by this edges endpoints (so there are $O(\sigma^2)$ possible labels). An internal node is labeled by the type of merge that formed it (there are five merging options so there are five possible labels).

The bottom-up construction of $\mathcal{T}$ starts with the leaves of $\mathcal{T}$. 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 (the size of a cluster is the number of nodes in the corresponding tree pattern). 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 rooted subtrees, we partition the clusters into *small clusters* and *large clusters*. The small clusters are those created in rounds 1 to $j = \log_2(0.5\log_{4\sigma^2}(n_T)) = O(\log_2 \log_\sigma n_T)$ 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

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

In particular, there are at most $O(n_T/\log_\sigma^{0.19} n_T)$ nodes of $\mathcal{T}$ that correspond to large clusters. So clearly there are at most $O(n_T/\log_\sigma^{0.19} n_T)$ distinct subtrees rooted at these nodes.

Next, we bound the total number of distinct subtrees of $\mathcal{T}$ rooted at nodes corresponding to small clusters. Each such subtree is of size at most $2^j + 1$ and is a binary tree whose nodes have labels from an alphabet of size at most $\sigma^2 + 5$. The total number of distinct labeled binary trees of size at most x is given by

$$\sum_{i=1}^{x} (\sigma^2+5)^i \cdot C_{i-1} = \sum_{i=1}^{x} O\big((\sigma^2+5)^i \cdot 4^i\big) = O\big((4\sigma^2)^{x+1}\big),$$

where C_i denotes the ith Catalan number. Since $x = 2^j + 1$, this number is bounded by $O((4\sigma^2)^{2^j+2}) = O(\sigma^4\sqrt{n_T}) = O(n_T^{3/4})$. In the last equality we assumed that $\sigma < n_T^{1/16}$. If $\sigma > n_T^{1/16}$ 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 subtrees of $\mathcal{T}$ rooted at small clusters is therefore also $O(n_T/\log_\sigma^{0.19} n_T)$. This completes the proof of Theorem 1.

3.2. 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 $\{C_1, \dots, C_\ell\}$ from $\mathcal{T}$ if $T(z) = C_1 \cup \dots \cup C_\ell$. Here $G = X_1 \cup \dots \cup X_\ell$ denotes the graph with node set $V(G) = \bigcup_{i=1,\dots,k} V(X_i)$ and edge set $E(G) = \bigcup_{i=1,\dots,k} E(X_i)$. 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 $L_z = (z, z_1)$ in $\mathcal{T}$. Let U be the smallest cluster in $\mathcal{T}$ containing all nodes of $T(z)$. We have that L_z is a descendant leaf of U in $\mathcal{T}$. Consider the path P_z in $\mathcal{T}$ from U to L_z. An *off-path* cluster of P_z is a cluster C that is not on P_z, but whose parent cluster is on P_z. We define

$$S_z = \big\{C \;\big|\; C \text{ is off-path cluster of } P_z \text{ and the tree pattern } C \text{ is a subtree of } T(z)\big\} \cup \{L_z\}.$$

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

Lemma 3. *Let C be an off-path cluster of P_z. 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 in $\mathcal{T}(C)$ containing edges from both $T(z)$ and $T \setminus T(z)$. That is, C' is the cluster found by descending down from C towards a child with both types of edges as long as such a child exists. 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). It follows that C' (and therefore C since it is an ancestor of C') contains both $(p(z), z)$ and (z, z_1).

We have $L_z = (z, z_1)$ and therefore all clusters in $\mathcal{T}$ containing (z, z_1) lie on the path from L_z to the root. The path P_z 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 3 therefore implies that $T(z) = \bigcup_{C \in S_z} C$ and the following corollary.

Corollary 2. *Let T be a tree with top tree $\mathcal{T}$. For any internal 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 $S_z = S_{z'}$) if there is a 1–1 correspondence between the clusters in S_z and $S_{z'}$, such that two clusters mapped to each other are identical tree patterns in T (have the same structure and labels).

Lemma 4. *Let T be a tree with top tree $\mathcal{T}$. Let $T(z)$ and $T(z')$ be identical subtrees in T. Then, $S_z = S_{z'}$.*

Proof. Consider the tree $\widetilde{T}$ at some iteration of the construction of the top tree. We will say that an edge e in $\widetilde{T}$ *belongs to* $T(z)$ (resp. $T(z')$) if the cluster corresponding to e only contains edges from $T(z)$ (resp. $T(z')$) in the original tree. Let L_z be the cluster in $\widetilde{T}$ containing the edge $L = (z, z_1)$, where z_1 is the leftmost child of z. Define $L_{z'}$ similarly.

We will say that a cluster $C \neq L_z$ is added to S_z in the iteration where its parent on P_z is created, and we say that L_z is added to S_z right before the first round. Similarly for clusters in S_z.

We will show that new clusters only are added to S_z (resp. $S_{z'}$) if L_z (resp. $L_{z'}$) is merged with an edge belonging to $T(z)$ (resp. $T(z')$), and that these merges are identical for the two subtrees in each iteration.

Recall that U is the smallest cluster in $\mathcal{T}$ containing all nodes of $T(z)$ and that P is the path of clusters in $\mathcal{T}$ from U to L. By definition, all clusters on the path P contain L. This implies that new off-path clusters are only constructed when L_z (resp. $L_{z'}$) is merged. Merges of identical edges belonging to $T(z)$ and $T(z')$ are the same in the two subtrees of $\widetilde{T}$, since we merge first horizontally, and then vertically bottom-up. By the same argument if L_z is merged with an edge belonging to $T(z)$ then $L_{z'}$ is merged with the corresponding edge from $T(z')$. For a merge with an edge belonging to $T(z)$ (resp. $T(z')$) and an edge not belonging to $T(z)$ (resp. $T(z')$), one of the edges must be L_z (resp. $L_{z'}$). If L_z is merged in this iteration, but $L_{z'}$ is not, then L_z is merged with an edge not belonging to $T(z)$ (and vice versa). Thus, after the iteration all edges belonging to $T(z)$ in $\widetilde{T}$ are identical to the edges belonging to $T(z')$ in $\widetilde{T}$.

New off-path clusters are only constructed when L_z (resp. $L_{z'}$) are merged. It only adds new clusters to S_z (resp. $S_{z'}$) if it is a merge with an edge belonging to $T(z)$ (resp. $T(z')$). Since these merges are identical for the two subtrees in each iteration, and L_z is merged with an edge belonging to $T(z)$ iff $L_{z'}$ is merged with the corresponding edge belonging to $T(z')$, we have $S_z = S_{z'}$. □

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

Proof. Denote an edge in the DAG as shared 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 the top DAG $\mathcal{TD}$, respectively.

First we bound the number of red clusters in the top 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 $T(z)$. Thus all the clusters in S_z are red. It follows from Lemma 4 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 $\mathcal{TD}$.

The clusters representing $T(z)$ are edge-disjoint connected subtrees of $T(z)$. It follows from Lemma 2 that $|\mathcal{T}(C)| = O(|C|)$ for each cluster in S_z. Therefore the total size of the subtrees of the clusters representing $T(z)$ in $\mathcal{T}$ is $O(|T(z)|)$. As argued above these are only represented once in the top DAG $\mathcal{TD}$. Thus the number of red clusters $r = O(r_D)$.

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 Lemma 2 that $|\mathcal{T}(C)| = O(|C|)$. Thus the number of blue clusters $b = O(b_D)$.

It remains to bound the number p of purple clusters (clusters containing both shared and non-shared edges). The number of purple clusters in the top DAG $\mathcal{TD}$ is bounded by the number of purple clusters in the top tree $\mathcal{T}$. For any purple cluster we have that all its ancestors in $\mathcal{T}$ are also purple. Consider the set P of purple clusters in $\mathcal{T}$ that have no purple descendants. Each of the clusters in P have a blue leaf cluster in its subtree. These blue leaf clusters are all distinct, and since the corresponding edges are not shared in the DAG D, we have $|P| \leqslant b_D$. Each cluster in P is the endpoint of a purple path from the root (and the union of these paths contains all purple clusters in $\mathcal{T}$). Since the height of $\mathcal{T}$ is $O(\log n_T)$ the number of nodes on each path is at most $O(\log n_T)$. It follows that the number of purple clusters in $\mathcal{T}$ (and thus also in $\mathcal{TD}$) is at most $|P| \cdot O(\log n_T) = O(b_D \log n_T)$.

The number of edges in the $\mathcal{TD}$ is thus $b + r + p = O(b_D + r_D + b_D \log n_T) = O(n_D \log n_T)$. □

Lemma 5. *There exist trees T, such that $n_D = \Omega(n_T / \log n_T) \cdot n_{\mathcal{TD}}$.*

Proof. Caterpillars and paths (where all nodes have identical labels) have $n_{\mathcal{TD}} = O(\log n_T)$, whereas $n_D = n_T$ (see Fig. 3). □

4. Supporting navigational queries

In this section we prove Theorem 3. 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.

Access(x)**:** Return the label associated with node x.
Decompress(x)**:** Return the tree $T(x)$.

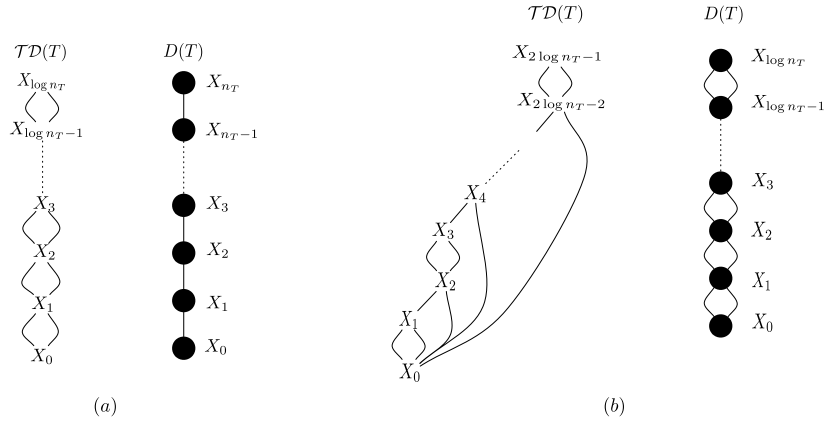

Fig. 3. A top DAG $\mathcal{TD}$ and a DAG $D(T)$ of (a) a path and (b) a complete binary tree. All labels are identical. On a path (and also a caterpillar and a star) the size of $\mathcal{TD}$ is $O(\log n_T)$ whereas the size of $D(T)$ is $O(n_T)$. On a complete binary tree (b) both $\mathcal{TD}$ and $D(T)$ are of size $O(\log n_T)$.

Parent(x): Return the parent of node x.
Depth(x): Return the depth of node x.
Height(x): Return the height of node x.
Size(x): Return the number of nodes in $T(x)$.
Firstchild(x): Return the first child of x.
NextSibling(x): Return the sibling immediately to the right of x.
LevelAncestor(x, i): Return the ancestor of x whose distance from x is i.
NCA(x, y): Return the nearest common ancestor of the nodes x and y.

4.1. 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. Otherwise, C is a cluster of T obtained by merging two clusters: the cluster A corresponding to C's left child and the cluster B corresponding to C's right child. Consider a preorder traversal of C. Let $\ell(B)$ denote the first node visited in this traversal that is also a node in B. Let $r(B)$ (resp. $r(A)$) denote the last node visited that is also a node in B (resp. in A). We augment each cluster C with:

- The integers $r(A)$, $\ell(B)$, and $r(B)$.
- The type of merge that was applied to A and B to form C. If C is a leaf cluster then the labels of its corresponding edge's endpoints in T.
- The height and size of C (i.e., of the tree pattern C in T).
- The distance from the top boundary node of C to the top boundary nodes of A and B.

Since we use constant space for each cluster of $\mathcal{TD}$, the total space remains $O(n_{\mathcal{TD}})$.

Local preorder numbers. 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. Specifically, let u be a node in the cluster C. Define the *local preorder number of* u, denoted u_C, to be the position of u in a preorder traversal of C. The following lemma states that in $O(1)$ time we can compute u_A and u_B from u_C and vise versa.

Lemma 6. *Let c be an internal node of $\mathcal{TD}$ that corresponds to the cluster C of T obtained by merging the cluster A (corresponding to c's left child) and the cluster B (corresponding to c's right child). For any node u in C, given u_C we can tell in constant time if u is in A (and obtain u_A) in B (and obtain u_B) or in both. Similarly, if u is in A or in B we can obtain u_C in constant time from u_A or u_B.*

Proof. If C is a merge of A and B of type (a) or (b) then

- $u_C = 1$ iff u is the top boundary node of A and C and $u_A = 1$.
- $u_C \in [2, \ell(B) - 1]$ iff u is an internal node of A and $u_A < l(B)$. In this case $u_A = u_C$.
- $u_C = \ell(B)$ iff u is the shared boundary node of A and B, $u_A = \ell(B)$, and $u_B = 1$.
- $u_C \in [\ell(B) + 1, r(B)]$ iff u is an internal node in B. In this case $u_B = u_C - \ell(B) + 1$.
- $u_C \in [r(B) + 1, r(A)]$ iff u is an internal node in A and $u_A > l(B)$. In this case $u_A = u_C - r(B) + \ell(B)$.

Otherwise, if C is a merge of A and B of type (c), (d), or (e) then

- $u_C = 1$ iff u is the shared boundary node of A, B, and C and $u_A = u_B = 1$.
- $u_C \in [2, r(A)]$ iff u is an internal node in A. In this case $u_A = u_C$.
- $u_C \in [r(A) + 1, r(B)]$ iff u is an internal node in B. In this case $u_B = u_C - r(A) + 1$. □

4.2. *Implementation of the procedures*

We now show how to implement the queries using local preorder numbers in top-down and bottom-up traversals of $\mathcal{TD}$.

4.2.1. *Access and depth*

The queries Access(x) and Depth(x) ask for the label and depth of the node whose preorder number in T is x. They are both performed by a single top-down search of $\mathcal{TD}$ starting from its root and ending with the leaf cluster containing x. Since the depth of $\mathcal{TD}$ is $O(\log n_T)$ the total time is $O(\log n_T)$.

Access. At each cluster C on the top-down search we compute the local preorder number x_C. Initially, the root cluster corresponds to the entire T so we set $x_T = x$. Let C be a cluster on the way. If C is a leaf cluster we return the label of the top boundary node if $x_C = 1$ and the label of the single internal node if $x_C = 2$. If on the other hand C is an internal cluster with child clusters A and B, we continue the search in the child cluster containing x_C. We compute the new local preorder number according to Lemma 6. If x_C is the shared boundary node between A and B we continue the search in either A or B.

Depth. The only difference between Depth(x) and Access(x) is that during the top-down search we also sum the distances between the top boundary nodes of the visited clusters. Let d be this distance. At the leaf cluster at the end of the search we return d if $x_C = 1$ and $d + 1$ if $x_C = 2$. Since the distances are stored the total time remains $O(\log n_T)$.

4.2.2. *Firstchild, level ancestor, parent, and NCA*

We answer these queries 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.

Firstchild. We compute Firstchild(x) in two steps.

Step 1: Top-down Search. We do a top-down search to find the first cluster with top boundary node x. We use local preorder numbers as in the algorithm for Access. Let C be a cluster in the search. If $x_C = 1$ we stop the search. Otherwise we know that $x_C > 1$. If C is a leaf cluster we stop and report that x does not have a first child since it is a leaf in T. If on the other hand C is an internal cluster with child clusters A and B, we continue the search in the child cluster containing x_C. If x_C is the shared boundary node between A and B we always continue the search in B. This ensures that we continue to the cluster containing the children of x (recall that B is the deeper cluster in merges of type (a) and (b)). Combined with the condition that we stop the search in the first cluster C where x is the top boundary node (and therefore the last merge before we stop must be of type (a) or (b)), this implies that all children of x are in C.

Step 2: Bottom-up Search. Let C be the cluster found in Step 1. Since all children of x are in C, the node with local preorder number 2 in C is the first child of x. We do a bottom-up search from C to the root cluster to compute the preorder number in T of the node with $x_C = 2$.

Level Ancestor and Parent. Notice that Parent(x) can be computed as LevelAncestor(x, 1). Since LevelAncestor(x, 0) $= x$ we focus on LevelAncestor(x, i) for $i \geqslant 1$. This is done in three steps:

Step 1: Compute Depth. Compute the depth of LevelAncestor(x, i) as $d =$ Depth(x) $- i$.

Step 2: Top-down Search. We do a top-down search to find the cluster with top boundary node y of depth d such that x is a descendant of y (we will show that such a cluster exists). During the search we maintain the depth of the current top boundary node as in the algorithm for Depth. At each cluster C in the search we also compute a local preorder number x'_C to guide the search. The idea is that x'_C either corresponds to x or to an ancestor of x within C. Initially, for the root cluster T we set $x'_T = x$. Let C be an internal cluster in the search with top boundary node v and with children A and B. If the depth of v is d we stop the search. Otherwise, we proceed as follows.

1. If C is of type (a) or (b), x'_C is in B, and the shared boundary node of A and B has depth $> d$, we continue the search in A and set x'_A to be the bottom boundary of A.
2. In all other cases, we continue the search in the child cluster containing x'_C, and compute the new local preorder number for x'_C.

Note that if the shared boundary node in case 1 has depth d we continue the search in B. Combined with the assumption that $i > 0$, it inductively follows that y becomes the top boundary node at some cluster during the top-down search. Hence, at some cluster in the top-down search the depth of the top boundary node is d.

Step 3: Bottom-up Search. Let C be the cluster whose top boundary node v has depth d found in Step 2. We do a bottom-up search to compute the preorder number of v in T. Finally, we report the result as y.

Nearest Common Ancestor. We compute $\mathsf{NCA}(x, y)$ in the following steps. We assume w.l.o.g. that $x \neq y$ in the following since $\mathsf{NCA}(x, x) = x$.

Step 1: Top-down Search. We do a top-down search to find the first cluster, whose top boundary node is $\mathrm{nca}(x, y)$ (this cluster always exists since $x \neq y$). At each cluster C in the search we compute local preorder numbers x'_C and y'_C. The idea is that x'_C and y'_C are either x or y or ancestors of x and y and their depth is at least the depth of $\mathrm{nca}(x, y)$. Initially, for the root cluster T we set $x'_T = x$ and $y'_T = y$. Let C be a cluster visited during the search. If C is a leaf cluster we stop the search. Otherwise, C is an internal cluster with children A and B. We proceed as follows.

1. If x'_C and y'_C are in the same child cluster, we continue the search in that cluster, and compute new local preorder numbers for x'_C and y'_C.
2. If C is of type (a) or (b) and x'_C and y'_C are in different child clusters we continue the search in A. We update the local preorder number of the node in B to be the bottom boundary of A.
3. If C is of type (c), (d), or (e) and x'_C and y'_C are in different child clusters we stop the search.

Step 2: Bottom-up Search. Let C be the cluster computed in Step 1. We do a bottom-up search to compute the preorder number of the top boundary node of C in the entire tree T, and return the result.

4.2.3. Decompress, height, size, and next sibling

To answer these queries, the key idea is to compute a small set of clusters representing $T(x)$. This set will be a subset of the set S_x defined in Section 3.2 and will contain all the relevant information.

We need the following definitions. Let u be a node in T. We say that u is on the *spine path* in a cluster C if u is the top boundary node in C, or u is on the path from the top boundary node in C to the bottom boundary node in C. Since clusters are connected subtrees we immediately have the following.

Lemma 7. *Let $C = A \cup B$ be a cluster with left child A and right child B. A node u in T is on the spine path of C iff one of the following cases are true:*

- *C is of type* (c) *and u is on the spine path in A.*
- *C is of type* (d) *and u is on the spine path in B.*
- *C is of type* (a) *and u is on the spine path in A or B.*
- *u is the top boundary node of C.*

Let x be any internal node in T. As in Section 3.2, let L be the leftmost leaf cluster in $\mathcal{TD}$ such that x is the top boundary node and let P be the path of clusters from the smallest cluster U containing all nodes of $T(x)$ to L. We also define M to be the highest cluster on P that has x as the top boundary node, i.e., M is the highest cluster on P that only contains edges from $T(x)$. Recall that S_x is the set of $O(\log n_T)$ off-path clusters of P that represent $T(x)$. We partition S_x into the set $\widehat{S}_x$ that contains all clusters in S_x that are descendants of M and the set $\check{S}_x$ that contains the remaining clusters. We characterize these sets as follows.

Lemma 8. *Let B be an off-path cluster of P with parent C and sibling A. Then*

1. *B is in $\widehat{S}_x$ iff B is a descendant of M.*
2. *B is in $\check{S}_x$ iff C is a merge of type* (a) *or* (b), *B is the right child of C, and x is on the spine path of A.*

Proof. For the first property, first note that if B is in $\widehat{S}_x$ it is by definition a descendant of M. Conversely, if B is a descendant of M, we have that $E(B) \subseteq E(M) \subseteq E(T(x))$. By definition of $\widehat{S}_x$, we have that B is in $\widehat{S}_x$.

Next consider property 2. Suppose that B is in $\check{S}_x$. Then, by Lemma 3 and the definition of S_x we have that $E(B) \subseteq E(T(x))$. Furthermore, since C is a proper ancestor of M, C contains edges from both $T(x)$ and $T \setminus T(x)$, and therefore A must also contain edges from both T and $T \setminus T(x)$.

Assume for contradiction that C is of type (c), (d), or (e). Then, the top boundary node v of C is also the top boundary node in A and B. Since $x \neq v$ by definition of M, we have by Lemma 3 that $E(B) \cap E(T(x)) = \emptyset$ and thus B cannot be in $\check{S}_x$.

Hence, assume that C is of type (a) or (b). Assume for contradiction that B is the left child of C. Since all clusters on P contain $E(L)$ and C contains edges from both $T(x)$ and $T \setminus T(x)$, we have that the top boundary node of B is a proper ancestor of x. Hence, B cannot be in $\check{S}_x$.

Finally, if B is of type (a) or (b) and is the right child of C, then $E(B) \subseteq E(T(x))$ iff the top boundary node v of B is a descendant of x. But v is a descendant of x iff x is on the spine path of A. Hence, B is in $\breve{S}_x$ iff x is on the spine path of A. □

In the following we show how to efficiently compute $\breve{S}_x$ using the procedure FindRepresentatives. We then use FindRepresentatives to implement the remaining procedures.

FindRepresentatives. Procedure FindRepresentatives(x) computes the set $\breve{S}_x$ and cluster M in two steps.

Step 1: Top-down Search. We do a top-down search to find the cluster M, i.e., the highest cluster on P that has x as the top boundary node. If no such node exists, then x is a leaf node in T.

Step 2: Bottom-up Search. We do a bottom-up search from M and add clusters according to Lemma 8 as follows. Initially, set $S = \emptyset$. Let A be a cluster on the path with sibling B and parent C.

1. If C is of type (a) or (b) and A is the left child of C, add B to S.
2. If one of the following conditions are true, stop the traversal:
   - C is of type (c) and A is the right child of C.
   - C is of type (d) and A is the left child of C.
   - C is of type (e) or (b).

Note that, as long as we continue the bottom-up search and consider clusters on the path, we have that x is on the spine path of these clusters. This is because we continue the bottom-up search according to the cases of Lemma 7. It follows from Lemma 8 that the clusters we add to S are exactly the clusters in the set representing $T(x)$. The total time is $O(\log n_T)$.

Decompress. To compute Decompress(x), we use FindRepresentatives(x) to compute the sets of cluster $\breve{S}_x$ and M. We construct $T(x)$ from $\breve{S}_x$ (and M) and the path P computed during the traversal of $\mathcal{TD}$. First, we decompress all clusters in $\breve{S}_x$ (and M) by unfolding their subDAG and constructing their corresponding subtree of T. We then combine these subtrees using the merge information stored for each cluster in P.

In total we use $O(\log n_T)$ time for FindRepresentatives(x) and computing the path P. The total time to decompress a cluster $\mathcal{TD}$ by unfolding is linear in its size. Hence, the total time used is $O(\log n_T + |T(x)|)$.

Height. First we compute the set of clusters $\breve{S}_x$ and cluster M using FindRepresentatives(x). Define the *local height* of a cluster to be the length of the path from the top boundary node to the bottom boundary node if it is an internal cluster, and the height of the cluster if it is a leaf cluster. We compute the height of $T(x)$ as the sum of the local heights of all clusters in $\breve{S}_x$ plus the height of M. This correctly computes the height since all clusters in $\breve{S}_x$ are merged with their siblings by type (a) or (b). Since the height and the distance from top boundary node to bottom boundary node for each cluster in $\mathcal{TD}$ is stored we use $O(\log n_T)$ time in total.

Size. Similar to height. We sum the sizes of clusters in $\breve{S}_x$ and M and subtract $|\breve{S}_x|$ (to exclude shared boundary nodes). This also uses $O(\log n_T)$ time.

Nextsibling. We compute NextSibling(x) directly from Size(x) since NextSibling(x) $= x +$ Size(x).

5. Conclusion and open problems

We have presented the new top tree compression scheme, and shown that it 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. We conclude with some open problems.

- Surprisingly, top tree compression is the first compression scheme for trees that achieves any provable non-trivial compression guarantee compared to the classical DAG compression. We wonder how other tree compression schemes compare to DAG compression and if it is possible to construct a tree compression scheme that exploits tree pattern repeats and always compresses better than a logarithmic factor of the DAG compression.
- Pattern matching in compressed *strings* is a well-studied and well-developed area with numerous results, see e.g., the surveys [26–28]. Pattern matching in compressed trees (especially within tree compression schemes that exploit tree pattern repeats) is a wide open area.
- We wonder if top tree compression is practical. In preliminary experiments we have compared our top DAG compression with standard DAG compression on typical XML datasets that were previously used in papers on DAG compression. The experiments match our theoretical expectations, i.e., that most trees compress better with top tree compression, and only balanced trees compress slightly better with standard DAG compression.

Acknowledgment

We would like to thank the anonymous reviewer for the important and helpful comments.

References

[1] P. Buneman, M. Grohe, C. Koch, Path queries on compressed XML, in: Proc. 29th VLDB, 2003, pp. 141–152.

[2] M. Frick, M. Grohe, C. Koch, Query evaluation on compressed trees, in: Proc. 18th LICS, 2003, pp. 188–197.

[3] G. Busatto, M. Lohrey, S. Maneth, Grammar-based tree compression, Tech. rep., EPFL, 2004.

[4] G. Busatto, M. Lohrey, S. Maneth, Efficient memory representation of XML document trees, Inf. Syst. 33 (4–5) (2008) 456–474.

[5] M. Lohrey, S. Maneth, R. Mennicke, Tree structure compression with repair, Arxiv preprint, arXiv:1007.5406.

[6] M. Lohrey, S. Maneth, The complexity of tree automata and XPath on grammar-compressed trees, Theor. Comput. Sci. 363 (2) (2006) 196–210.

[7] S. Maneth, G. Busatto, Tree transducers and tree compressions, in: Proc. 7th FOSSACS, 2004, pp. 363–377.

[8] S. Sakr, XML compression techniques: a survey and comparison, J. Comput. Syst. Sci. 75 (5) (2009) 303–322.

[9] P.J. Downey, R. Sethi, R.E. Tarjan, Variations on the common subexpression problem, J. ACM 27 (1980) 758–771.

[10] S.S. Muchnick, Advanced Compiler Design Implementation, Morgan Kaufmann, 1997.

[11] C. Meinel, T. Theobald, Algorithms and Data Structures in VLSI Design: OBDD-Foundations and Applications, Springer, 1998.

[12] M. Lohrey, S. Maneth, E. Noeth, XML compression via DAGs, in: Proceedings of the 16th International Conference on Database Theory, ACM, 2013, pp. 69–80.

[13] J. Adiego, G. Navarro, P. de la Fuente, Lempel–Ziv compression of highly structured documents, J. Am. Soc. Inf. Sci. Technol. 58 (4) (2007) 461–478.

[14] P. Bille, G. Landau, R. Raman, S. Rao, K. Sadakane, O. Weimann, Random access to grammar-compressed strings, in: Proc. 22nd SODA, 2011, pp. 373–389.

[15] M. Charikar, E. Lehman, A. Lehman, D. Liu, R. Panigrahy, M. Prabhakaran, A. Sahai, A. Shelat, The smallest grammar problem, IEEE Trans. Inf. Theory 51 (7) (2005) 2554–2576.

[16] G. Jacobson, Space-efficient static trees and graphs, in: Proc. 30th FOCS, 1989, pp. 549–554.

[17] J.I. Munro, V. Raman, Succinct representation of balanced parentheses and static trees, SIAM J. Comput. 31 (3) (2001) 762–776.

[18] D. Benoit, E. Demaine, I. Munro, R. Raman, V. Raman, S. Rao, Representing trees of higher degree, Algorithmica 43 (2005) 275–292.

[19] R. Geary, R. Raman, V. Raman, Succinct ordinal trees with level-ancestor queries, in: Proc. 15th SODA, 2004, pp. 1–10.

[20] P. Ferragina, F. Luccio, G. Manzini, S. Muthukrishnan, Compressing and indexing labeled trees, with applications, J. ACM 57 (2009) 1–33.

[21] J. Cheney, Compressing XML with multiplexed hierarchical PPM models, in: Proc. 11th IEEE Data Compression Conf., DCC, 2001, pp. 163–172.

[22] H. Liefke, D. Suciu, Xmill: an efficient compressor for XML data, in: Proc. 2000 ACM SIGMOD Int. Conf. on Manag. of Data, SIGMOD, 2000, pp. 153–164.

[23] S. Alstrup, J. Holm, K. de Lichtenberg, M. Thorup, Minimizing diameters of dynamic trees, in: Proc. 24th ICALP, 1997, pp. 270–280.

[24] S. Alstrup, J. Holm, M. Thorup, Maintaining center and median in dynamic trees, in: Proc. 7th SWAT, 2000, pp. 46–56.

[25] S. Alstrup, J. Holm, K.D. Lichtenberg, M. Thorup, Maintaining information in fully-dynamic trees with top trees, ACM Trans. Algorithms 1 (2003) 243–264.

[26] L. Gasieniec, M. Karpinski, W. Plandowski, W. Rytter, Efficient algorithms for Lempel–Ziv encoding, in: Proc. 4th SWAT, 1996, pp. 392–403.

[27] W. Rytter, Grammar compression, LZ-encodings, and string algorithms with implicit input, in: Proc. 31st ICALP, 2004, pp. 15–27.

[28] M. Lohrey, Algorithmics on SLP-compressed strings: a survey, Groups Complex. Cryptol. 4 (2) (2012) 241–299.