Tree Compression with Top Trees Revisited

Lorenz Hübschle-Schneider
huebschle@kit.edu
Institute of Theoretical Informatics
Karlsruhe Institute of Technology
Germany

Rajeev Raman
r.raman@leicester.ac.uk
Department of Computer Science
University of Leicester
United Kingdom

Abstract

We revisit tree compression with top trees (Bille et al. [3]), and present several improvements to the compressor and its analysis. By significantly reducing the amount of information stored and guiding the compression step using a RePair-inspired heuristic, we obtain a fast compressor achieving good compression ratios, addressing an open problem posed by [3]. We show how, with relatively small overhead, the compressed file can be converted into an in-memory representation that supports basic navigation operations in worst-case logarithmic time without decompression. We also show a much improved worst-case bound on the size of the output of top-tree compression (answering an open question posed in a talk on this algorithm by Weimann in 2012).

1 Introduction

Labelled trees are one of the most frequently used nonlinear data structures in computer science, appearing in the form of suffix trees, XML files, tries, and dictionaries, to name but a few prominent examples. These trees are frequently very large, prompting a need for compression for on-disk storage. Ideally, one would like specialized tree compressors to certainly get much better compression ratios than general-purpose compressors such as `bzip2` or `gzip`, but also for the compression to be fast; as Ferragina et al. note [11, p4:25]. [1]

In fact, it is also frequently necessary to hold such trees in main memory and perform complex navigations to query or mine them. However, common in-memory representations use pointer data structures that have significant overhead—e.g. for XML files, standard DOM[2] representations are typically 8-16 times larger than the (already large) XML file [23, 25]. To process such large trees, it is essential to have compressed in-memory representations that *directly* support rapid navigation and queries, without partial or full decompression.

Before we describe previous work, and compare it with ours, we give some definitions. A *labelled tree* is an ordered, rooted tree whose nodes have labels from an alphabet Σ of size $|\Sigma| = \sigma$. We consider the following kinds of redundancy in the tree structure. *Subtree repeats* are repeated occurrences of *rooted subtrees*, i.e. a node and all of its descendants, identical in structure and labels. *Tree pattern repeats* or *internal repeats* are repeated occurrences of *tree patterns*, i.e. connected subgraphs of the tree, identical in structure as well as labels.

[1] Their remark is about XML tree compressors but applies to general ones as well.

[2] *Document Object Model*, a common interface for interacting with XML documents

1.1 Previous Work

Nearly all existing compression methods for labelled trees follow one of three major approaches: *transform-based compressors* that transform the tree's structure, e.g. into its minimal DAG, *grammar-based compressors* that compute a tree grammar, and–although not compression–*succinct representations* of the tree.

Transform-Based Compressors. We can replace subtree repeats by edges to a single shared instance of the subtree and obtain a smaller Directed Acyclic Graph (DAG) representing the tree. The smallest of these, called the *minimal DAG*, is unique and can be computed in linear time [10]. Navigation and path queries can be supported in logarithmic time [4, 5]. While its size can be exponentially smaller than the tree, no compression is achieved in the worst case (a chain of nodes with the same label is its own minimal DAG, even though it is highly repetitive). Since DAG minimization only compresses repeated subtrees, it misses many internal repeats, and is thus insufficient in many cases.

Bille et al. introduced tree compression with top trees [3], which this paper builds upon. Their method exploits both repeated subtrees and tree structure repeats, and can compress exponentially better than DAG minimization. They give a $\log_\sigma^{0.19} n$ worst-case compression ratio for a tree of size n labelled from an alphabet of size σ for their algorithm. They show that navigation and a number of other operations are supported in $O(\log n)$ time directly on the compressed representation. However, they do not give any practical evaluation, and indeed state as an open question whether top-tree compression has practical value.

Tree Grammars. A popular approach to exploit the redundancy of tree patterns is to represent the tree using a formal grammar that generates the input tree, generalizing grammar compression from strings to trees [6, 7, 15, 17–19]. These can be exponentially smaller than the minimal DAG [17]. Since it is NP-Hard to compute the smallest grammar [8], efficient heuristics are required.

One very simple yet efficient heuristic method is RePair [16]. A string compressor, it can be applied to a parentheses bitstring representation of the tree. The output grammars produced by RePair can support a variety of navigational operations and random access, in time logarithmic in the input tree size, after additional processing [4]. These methods, however, appear to require significant engineering effort before their practicality can be assessed.

TreeRePair [18] is a generalization of RePair from strings to trees. It achieves the best grammar compression ratios currently known. However, navigating TreeRePair's grammars in sublinear time with respect to their depth, which can be linear in their size [3], is an open problem. For relatively small documents (where the output of TreeRePair fits in cache), the navigation speed for simple tree traversals is about 5 times slower than succinct representations [18].

Several other popular grammar compressors exist for trees. Among them, BPLEX [6,7] is probably best-known, but is much slower than TreeRePair. The TtoG algorithm is the first to achieve a good theoretical approximation ratio [15], but has not been evaluated in practice.

Succinct Representations. Another approach is to represent the tree using near-optimal space without applying compression methods to its structure, a technique called *succinct data structures*. Unlabelled trees can be represented using $2n + o(n)$ bits [14] and support queries in constant time [22]. There are a few $n \log \sigma + \mathcal{O}(n)$ bit-representations for labelled trees, most notably that by Ferragina et al. [11], which also yields a compressor, XBZip. While XBZip has good performance on XML files

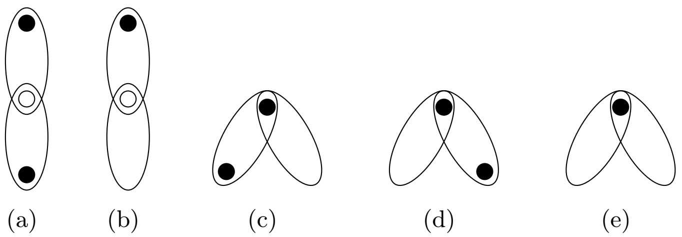

Figure 1: Five kinds of cluster merges in top trees. Solid nodes are boundary nodes, hollow ones are boundary nodes that become internal. Source of this graphic and more details: [3, Section 2.1].

in their entirety, including text, attributes etc., evidence suggests that it does not beat TreeRePair on pure labelled trees. As the authors admit, it is also slow.

1.2 Our Results

Our primary aim in this paper is to address the question of Bille et al. [3] regarding the practicality of the top tree approach, but we make some theoretical contributions as well. We first give some terminology and notation.

A *top tree* [1] is a hierarchical decomposition of a tree into *clusters*, which represent subgraphs of the original tree. Leaf clusters correspond to single edges, and inner clusters represent the union of the subgraphs represented by their two children. Clusters are formed in one of five ways, called *merge types*, shown in Figure 1. A cluster can have one or two *boundary nodes*, a top- and optionally a bottom boundary node, where other clusters can be attached by merging. A top tree's minimal DAG is referred to as a *top DAG*. For further details on the fundamentals of tree compression with top trees, refer to [3]. Throughout this paper, let T be any ordered, labelled tree with n_T nodes, and let Σ denote the label alphabet with $\sigma := |\Sigma|$. Let $\mathcal{T}$ be the top tree and $\mathcal{TD}$ the top DAG corresponding to T, and $n_{\mathcal{TD}}$ the total size (nodes plus edges) of $\mathcal{TD}$. We assume a standard word RAM model with logarithmic word size, and measure space complexity in terms of the number of words used. Then:

Theorem 1 *The size of the top DAG is* $n_{\mathcal{TD}} = \mathcal{O}\Big(\frac{n_T}{\log_\sigma n_T} \cdot \log\log_\sigma n_T\Big)$.

This is only a factor of $\mathcal{O}(\log\log_\sigma n_T)$ away from the information-theoretic lower bound, and greatly improves the bound of $\mathcal{O}\big(n/\log_\sigma^{0.19} n\big)$ obtained by Bille et al. and answers an open question posed in a talk by Weimann.

Next, we show that if only basic navigation is to be performed, the amount of information that needs to be stored can be greatly reduced, relative to the original representation [3], without affecting the asymptotic running time.

Theorem 2 *We can support navigation with the operations Is Leaf, Is Last Child, First Child, Next Sibling, and Parent in $\mathcal{O}(\log n_T)$ time, full decompression in time $\mathcal{O}(n_T)$ on a representation of size $\mathcal{O}(n_{\mathcal{TD}})$ storing only the top DAG's structure, the merge types of inner nodes (an integer from $[1..5]$), and leaves' labels.*

We believe this approach will have low overhead and fast running times in practice for in-memory navigation without decompression, and sketch how one would approach an implementation.

Furthermore, we introduce the notion of *combiners* that determine the order in which clusters are merged during top tree construction. Combiners aim to improve the compressibility of the top tree, resulting in a smaller top DAG. We present one such combiner that applies the basic idea of RePair [16] to top tree compression, prioritizing merges that produce subtree repeats in the top tree, in Section 3. We give a relatively naive encoding of the top tree, primarily using Huffman codes, and evaluate its compression performance. Although the output of the modified top tree compressor is up to 50 % larger than the state-of-the-art TreeRePair, it is about six times faster. We believe that the compression gap can be narrowed while maintaining the speed gap.

2 Top Trees Revisited

2.1 DAG Design Decisions

The original top tree compression paper [3] did not try to minimize the amount of information that actually needs to be stored. Instead, the focus was on implementing a wide variety of navigation operations in logarithmic time while maintaining $\mathcal{O}(n_{\mathcal{TD}})$ space *asymptotically*. Here, we reduce the amount of additional information stored about the clusters to obtain good compression ratios.

Instead of storing the labels of both endpoints of a leaf cluster's corresponding edge, we store only the child's label, not the parent's. In addition to reducing storage requirements, this reduces the top tree's alphabet size from $\sigma^2 + 5$ to $\sigma + 5$, as each cluster has either one label or a merge type. This increases the likelihood of identical subtrees in the top tree, improving compression. Note that this change implies that there is exactly one leaf cluster in the top DAG for each distinct label in the input. To code the root, we perform a merge of type (a) (see Section 1.2 and Figure 1) between a dummy edge leading to the root and the last remaining edge after all other merges have completed.

With these modifications, we reduce the amount of information stored with the clusters to the bare minimum required for decompression, i.e. leaf clusters' labels and inner clusters' merge types.

Lastly, we speed up compression by directly constructing the top DAG during the merge process. We initialize it with all distinct leaves, and maintain a mapping from cluster IDs to node IDs in $\mathcal{TD}$, as well as a hash map mapping DAG nodes to their node IDs. When two edges are merged into a new cluster, we look up its children in the DAG and only need to add a new node to $\mathcal{TD}$ if this is its first occurrence. Otherwise, we simply update the cluster-to-node mapping.

2.2 Navigation

We now explain how to navigate the top DAG with our reduced information set. We support full decompression in time $\mathcal{O}(n_T)$, as well as operations to move around the tree in time proportional to the height of the top DAG, i.e. $\mathcal{O}(\log n_T)$. These are: determining whether the current node is a leaf or its parent's last child, and moving to its first child, next sibling, and parent. Accessing a node's label is possible in constant time given its node number in the top DAG.

Proof (Theorem 2). As a node in a DAG can be the child of any number of other nodes, it does not have a unique parent. Thus, to allow us to move back to a node's parent in the DAG, we need to maintain a stack of parent cluster IDs along with a bit to indicate whether we descended into the left or right child. We refer to this as the *DAG stack*, and update it whenever we move around in $\mathcal{TD}$ with the operations below. Similarly, we also maintain a *tree stack* containing the DAG stack of each ancestor of the current node in the (original) tree.

Decompression: We traverse the top DAG in pre-order, undoing the merge operations to reconstruct the tree. We begin with n_T isolated nodes, and then add back the edges and labels as we traverse the top DAG. As this requires constant time per cluster and edge, we can decompress the top DAG in $\mathcal{O}(n_T)$ time.

Label Access: Since only leaf clusters store labels, and these are coded as the very first clusters in the top DAG (cf. Section 2.4), their node indices come before all other nodes'. Therefore, a leaf's label index i is its node number in the top DAG. We access the label array in the position following the $(i-1)$th null byte, which we can find with a $\mathsf{Select}_0(i-1)$ operation, and decode the label string until we reach another null byte or the end.

Is Leaf: A node is a leaf iff it is no cluster's top boundary node. Moving up through the DAG stack, if we reach a cluster of type (a) or (b) from the *left* child, the node is not a leaf (the left child of such a cluster is the *upper* one in Figure 1). If, at any point before encountering such a cluster, we exhaust the DAG stack or reach a cluster of type (b) or (c) from the right, type (d) from the left, or type (e) from either side, the node is a leaf. This can again be seen in Figure 1.

Is Last Child: We move up the DAG stack until we reach a cluster of type (c), (d), or (e) from its left child. Upon encountering a cluster of type (a) or (b) from the right, or emptying the DAG stack completely, we abort as the upward search lead us to the node's parent or exhausted the tree, respectively.

First Child and Next Sibling: First, we check whether the node is a leaf (First Child) or its parent's last child (Next Sibling), and abort if it is. First Child then pushes a copy of the DAG stack onto the tree stack. Next, we re-use the upward search performed by the previous check, removing the elements visited by the search from the DAG stack, up until the cluster with which the search ended. We descend into its right child and keep following the left child until we reach a leaf.

Parent: Since First Child pushes the DAG stack onto the tree stack, we simply reset the DAG stack to the tree stack's top element, which is removed. □

We note here that the tree stack could, in theory, grow to a size of $\mathcal{O}(n_T \log n_T)$, as the tree can have linear height and the logarithmically sized DAG stack is pushed onto it in each First Child operation. However, we argue that due to the low depth of common labelled trees, especially XML files, this stack will remain small in practice. Even when pessimistically assuming a *very* large tree with a height of 80 nodes, with a top tree of height 50, the tree stack will comfortably fit into 32 kB when using 64-bit node IDs. Our preliminary experiments confirm this.

To improve the worst-case tree stack size in theory, we can instead keep a log of movements in the top DAG, which is limited in size to the distance travelled therein. We expect this to be significantly less than $\mathcal{O}(n_T \log n_T)$ in expectation.

2.3 Worst-Case Top DAG size

Bille et al. show that a tree's top tree has at most $\mathcal{O}\big(n_T / \log_\sigma^{0.19} n_T\big)$ distinct clusters [3]. This bound, however, is an artifact of the proof. By modifying the definition of a *small cluster* in the compression analysis and carefully exploiting the properties of top trees, we are able to show a new, tighter, bound, which directly translates to an improvement on the worst-case compression ratio. Before we can prove Theorem 1, we need to show the following essential lemmata. Let $s(v)$ be the size of v's subtree, and $p(v)$ denote its parent.

Lemma 3 *Let T be any ordered labelled tree of size n_T, and let $\mathcal{T}$ be its top tree. For any node v of $\mathcal{T}$, the height of its subtree is at most $\lfloor \log_{8/7} s(v) \rfloor$.*

Proof. Consider the incremental construction process of a top tree $\mathcal{T}$. During the merge process, the algorithm builds up a tree by joining clusters into larger clusters. We start with a forest of $n_T + 1$ nodes, each representing an edge of T. Every merge operation joins two clusters and thus reduces the number of connected components in $\mathcal{T}$ by one. These connected components are subtrees of the final top tree. We can thus think of them as the top trees for tree patterns of the input tree.

Note that a subtree of the top tree is not the top tree of a rooted subtree for two reasons. For one, it might represent some, but not all, siblings of a node. This is due to horizontal merges operating on pairs of edges to consecutive siblings. Thus, a cluster could, for example, represent a node and the subtrees of the first two of its five children. Secondly, if the cluster has a bottom boundary node w (drawn as a filled node at the bottom in Figure 1), the subtree of T that is rooted at w is *not* contained in the cluster. Thus, the cluster does not correspond to a subtree of T, but rather a tree pattern, i.e. a connected subgraph.

Therefore, a subtree of a top tree is the top tree of a tree pattern of T, and the same bounds apply to its height. As each iteration of merges in the top tree construction reduces the number of strongly connected components by a factor of $c \geq 8/7$ [3], there are at most $\lceil \log_{8/7} n_T \rceil$ iterations, each of which increases the height of the top tree by exactly 1. Being a full binary tree with $n_T + 1$ leaves, the top tree has $2n_T$ edges. Thus, the height of any top tree of size n is bounded by $\lceil \log_{8/7} \frac{n}{2} \rceil < \lfloor \log_{8/7} n \rfloor \approx 5.2 \log n$. By the above, this also applies to subtrees of top trees. □

Lemma 4 *Let T be any ordered labelled tree of size n_T, let $\mathcal{T}$ be its top tree, and t be an integer. Then $\mathcal{T}$ contains at most $\mathcal{O}((n_T/t) \cdot \log t)$ nodes v so that $s(v) \leq t$ and $s(p(v)) > t$.*

Proof. We will call any node v of the top tree a *light* node iff $s(v) \leq t$, otherwise we refer to it as *heavy*. With this terminology, we are looking to bound the number of light nodes whose parent is heavy.

As $\mathcal{T}$ is a full binary tree, there are four cases to distinguish. We are not interested in the children of light nodes, nor are we interested in heavy nodes with two heavy children. This leaves us with two interesting cases:

1. A heavy node u with two light children v and w. Then, $s(v) + s(w) \geq t$. Thus, there are at most $2n_T/t = \mathcal{O}(n_T/t)$ light nodes with a heavy parent and a light sibling.

2. A heavy node with one light and one heavy child. We will consider this case in the remainder of the proof.

Consider any heavy node v. We say that v is in *class i* iff $s(v) \in \left[2^i, 2^{i+1} - 1\right]$. Observe that only classes $i \geq \lfloor \log_2 t \rfloor$ can contain heavy nodes, and that the highest non-empty class is $\lfloor \log_2 n_T \rfloor$. Let a *top class i node* be a node in class i whose parent is in class $j > i$, and a *bottom class i node* one for which both children are in classes lower than i. We now make two propositions:

Proposition 1 A node u of class i can have at most one child in class i.

Proof: Assume both children v, w of u are in class i. Then, the subtree size of u is $s(u) = 1 + s(v) + s(w) \geq 1 + 2^i + 2^i > 2^{i+1}$, and thus by definition u is not in class i.

Proposition 2 Let v be a top class i node. There are at most $\mathcal{O}(i)$ light nodes in the subtree of v that are children of class i nodes.

Proof: By Proposition 1, there exists exactly one path of class i nodes in the subtree of v. This path begins at v and ends at the bottom class i node of the subtree of v, which we refer to as w. There are no other class i nodes in the subtree of v. Being a full binary tree, the height of w's subtree fulfills $h(w) \geq \log_2 s(w) \geq \log_2 2^i = i$. We now use Lemma 3 to obtain an upper bound on $h(v)$ of $h(v) \leq \lfloor \log_{8/7} s(v) \rfloor \leq \frac{i+1}{\log_2 8/7} \approx 5.2 \cdot (i+1)$. Thus, the path from the top class i node to the bottom class i node has a length of $l \leq h(v) - h(w) = \mathcal{O}(i)$. Each node on the path can have at most one light child by Proposition 1. Thus, there are at most $\mathcal{O}(i)$ light nodes that are children of class i nodes in the subtree of a top class i node.

Combining Proposition 2 with the observation that the number of top class i nodes is clearly at most $n_T/2^i$, we obtain a bound on the number of class i nodes with one heavy and one light child of $n_T/2^i \cdot \mathcal{O}(i)$. We then sum over all classes containing heavy nodes to obtain the total number of heavy nodes with one light child,

$$\sum_{i=\lfloor \log_2 t \rfloor}^{\lfloor \log_2 n_T \rfloor} \frac{n_T}{2^i} \cdot \mathcal{O}(i) = \mathcal{O}\Big(\frac{n_T}{t} \cdot \log t\Big)$$

Thus, there are at most $\mathcal{O}(n_T/t \cdot \log t) + 2n_T/t = \mathcal{O}(n_T/t \cdot \log t)$ light nodes whose parent is heavy. This concludes the proof.

□

Proof (Theorem 1). We define a *small cluster* as one whose subtree contains at most $2^j + 1$ nodes and set $j = \log_2 (0.5 \log_{4\sigma} n_T)$. We call a small cluster *maximal* if its parent's subtree exceeds the size limit of a small cluster. A cluster that is not small is called a *large* cluster. Note that this is a special case of our distinction between light and heavy nodes in the proof of Lemma 4.

As each of the n_T leaves of the top tree is contained in exactly one maximal small cluster, and the top tree is a full binary tree, there is exactly one large cluster less than there are maximal small clusters. Thus, it suffices to show that there are at most $\mathcal{O}((n_T \cdot \log\log_\sigma n_T)/\log_\sigma n_T)$ maximal small clusters, and that the total number of distinct small clusters does not exceed said bound.

Recall that each inner node of the top tree is labelled with one of the five merge types, and that each leaf stores the label of its edge's child node, as described in Section 2.1. Therefore, the top tree is labelled with an alphabet of size $\sigma + 5 = \mathcal{O}(\sigma)$.

To bound the total number of distinct small clusters, we consider the number of distinct labelled trees of size at most x, which is $\mathcal{O}((4\sigma)^{x+1})$, and can be rewritten as $\mathcal{O}(\sigma^2 \sqrt{n_T})$ by setting $x = 2^j + 1$ [3]. If $\sigma < n_T^{1/8}$, this further reduces to $\mathcal{O}(n_T^{3/4})$. Otherwise, the theorem holds trivially as $\log_\sigma n_T = \mathcal{O}(1)$.

We now bound the number of maximal small clusters with Lemma 4 by choosing the threshold $t = 2^j + 1 = 0.5 \log_{4\sigma} n_T + 1 = \mathcal{O}(\log_\sigma n_T)$. As a maximal small cluster is a light node whose parent is heavy, we can use Lemma 4 to bound the number of maximal small clusters by $\mathcal{O}((n_T \cdot \log t)/t) = \mathcal{O}((n_T \cdot \log\log_\sigma n_T)/\log_\sigma n_T)$. This concludes the proof. □

2.4 Encoding

In the top DAG, we need to be able to access a cluster's left and right child, as well as its merge type for inner clusters or the child node's label for the edge that it refers to for leaf clusters. To realize this interface, we decompose the top DAG into a binary *core* tree and a pointer array. The core tree is defined by removing all incoming edges from each node, except for the one coming from the node with lowest pre-order number. All other occurrences are replaced by a dummy leaf node storing the pre-order number of the referenced node. Leaves in the top DAG are assigned new numbers as label pointers, which are smaller than the IDs of all inner nodes. All references to leaves, including the dummy nodes, are coded in an array of *pointers*, ordered by the pre-order number of the originating node. Similarly, the inner nodes' merge types are stored in an array in pre-order. Lastly, the core tree itself can be encoded using two bits per inner node, indicating whether the left and right children are inner nodes in the core tree.

Using this representation, all that is required for efficient navigation is an entropy coder providing constant-time random access to node pointers and merge types, and a data structure providing rank and select for the core tree and label strings. All of these building blocks can be treated as black boxes, and are well-studied and readily available, e.g. [24] and the excellent SDSL [12] library.

Simple Encoding To obtain file size results with reasonable effort, we now describe a very simple encoding that does not lend itself to navigation as easily. We compress the core tree bitstring and merge types using blocked Huffman coding. The pointer array and null byte-separated concatenated label string are encoded using a Huffman code. The Huffman trees are coded like the core tree above. The symbols are encoded using a fixed length and concatenated. Lastly, we store the sizes of the four Huffman code segments as a file header.

3 Heuristic Combiners

As described in the original paper [3], the construction of the top tree *exposes* internal repetitions. However, it does not attempt to maximize the size or number of identical subtrees in the top tree, i.e. its compressibility. Instead, the merge process sweeps through the tree linearly from left to right and bottom to top. This is a straight-forward cluster combining strategy that fulfills all the requirements for constructing a top tree, but does not attempt to maximize compression performance. We therefore replace the standard combining strategy with heuristic methods that try to increase compressibility of the top tree. Here, we present one such combiner that applies the basic idea of RePair to the horizontal merge step of top tree compression. (In preliminary experiments, it proved detrimental to apply the heuristic to vertical merges, and we limit ourselves to the horizontal merge step, but note that this is not a general restriction on combiners.)

We hash all clusters in the top tree as they are created. The hash value combines the cluster's label, merge type, and the hashes of its left and right children if these exist. As the edges in the auxiliary tree correspond to clusters in the top tree during its construction, we assign the cluster's hashes to the corresponding edges. Defining a digram as two edges whose clusters can be merged with one of the five merge types from Figure 1, we can apply the idea of RePair, identifying the edges by their hash values. In descending order of digram frequency, we merge all non-overlapping occurrences, updating the remaining edges' hash values to those of the newly created clusters.

Since this procedure does not necessarily merge a constant fraction of the edges in each iteration,

Table 1: XML corpus used for our experiments. File sizes are given for stripped documents, i.e. after removing whitespace and tags' attributes and contents.

File name	size (MB)	# nodes	height
1998statistics	0.60	28 306	6
dblp	338.87	20 925 865	6
enwiki-latest-p	229.78	14 018 880	5
factor12	359.36	20 047 329	12
factor4	119.88	6 688 651	12
factor4.8	143.80	8 023 477	12
factor7	209.68	11 697 881	12
JST-gene.chr1	5.79	173 529	7
JST-snp.chr1	27.31	803 596	8
nasa	8.43	476 646	8
NCBI-gene.chr1	35.30	1 065 787	7
proteins	365.12	21 305 818	7
SwissProt	45.25	2 977 031	5
treebank-e	25.92	2 437 666	36
uwm	1.30	66 729	5
wiki	42.29	2 679 553	5

we may need to additionally apply the normal horizontal merge algorithm if too few edges were merged by the heuristic. The constant upon which this decision is based thus becomes a tuning parameter. Note that we need to ensure that every edge is merged at most once per iteration.

4 Evaluation

We now present an experimental evaluation of top tree compression. In this section, we demonstrate its qualities as a fast and efficient compressor, compare it against other compressors, and show the effectiveness of our RePair-inspired combiner.

Experimental Setup All algorithms were implemented in C++11 and compiled with the GNU C++ compiler `g++` in version 4.9.2 using optimization level `fast` and profile-guided optimizations. The experiments were conducted on a commodity PC with an Intel Core i7-4790T CPU and 16 GB of DDR3 RAM, running Debian Linux from the `sid` channel. We used `gzip 1.6-4` and `bzip2 1.0.6-7` from the standard package repositories. Default compression settings were used for all compressors, except the `-9` flag for gzip. All input and output files were located in main memory using a `tmpfs` RAM disk to eliminate I/O delays.

XML corpus We evaluated the compressor and our heuristic improvement on a corpus of common XML files [9,13,21,26], listed in Table 1. In our experiments, we give file sizes for our simple encoding, which represent pessimistic results that can serve as an upper bound of what to expect from a more optimized encoding. We give these file sizes to demonstrate that even a simple encoding yields good results with regard to file size, speed, and ease of navigation (see Section 2.2).

Results We use a minimum merge ratio of $c = 1.26$ for the horizontal merge step using our RePair-inspired heuristic combiner in all our experiments. This is the result of an extensive evaluation which showed that values $c \in [1.2, 1.27]$ work very well on a broad range of XML documents. We observed that values close to 1 can improve compression by up to 10 % on some files, while causing a deterioration by a similar proportion on others. Thus, while better choices of c exist for individual files, we chose a fixed value for all files to provide a fair comparison, similar to the choice of 4 as the maximum rank of the grammar in TreeRePair [18].

We use a parenthesis bitstring encoding of the input tree as a baseline to measure compression ratios. The unique label strings are concatenated, separated by null bytes. Indices into this array

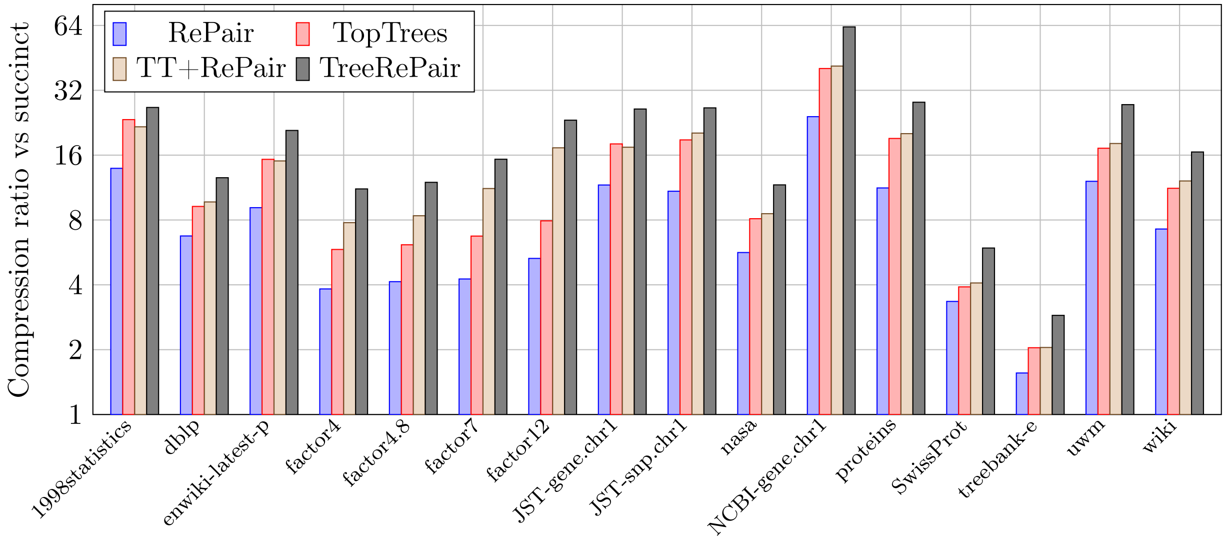

Figure 2: Comparison of compression ratios, measured by comparing file sizes against a succinct encoding of the input file (higher is better)

are stored as fixed-length numbers of $\lceil \log_2 \#\text{labels} \rceil$ bits. TreeRePair[3], which has been carefully optimized to produce very small output files, serves us as a benchmark. We are, however, reluctant to compare tree compression with top trees to TreeRePair directly, as our methods have not been optimized to the same degree.

In Figure 2 we give a compression ratios relative to the succinct encoding. We evaluated our implementation of top tree compression using the combining strategy from [3] as well as our RePair-inspired combiner. We also give the file sizes achieved by TreeRePair and those of RePair on a parentheses bitstring representation of the input tree and the concatenated nullbyte-separated label string (note that no deduplication is performed here, as this is up to the compressor). We represent RePair's grammar production rules as a sequence of integers with an implicit left-hand side and encode this representation using a Huffman code. Figure 2 shows that top tree compression consistently outperforms RePair already, but does not achieve the same level of compression as TreeRePair at this stage. We can also clearly see the impact of our RePair-inspired heuristic combiner, which improves compression on nearly all files in our corpus and is studied in more detail in the next paragraph. Table 3 gives the exact numbers for the output file sizes, supplementing them with results for general-purpose compressors.

RePair Combiner. Figure 3 compares the two versions of top tree compression, using TreeRePair as a benchmark. The RePair combiner's effect is clearly visible, reducing the maximum disparity in compression relative to TreeRePair from a file 2.93 times the size (`factor12`) to one that is 52 % larger (`NCBI-gene.chr1`). This constitutes nearly a four-fold decrease in overhead (from 1.93 to 0.52). On average, files are 1.39 times the size of TreeRePair's, down from a factor of 1.64 before. On our corpus, using the heuristic combiner reduced file sizes by 10.9 % on average, with the median being a 5.0 % improvement compared to classical top tree compression. Reduced compression performance was observed on few files only, particularly smaller ones, while larger files tended to fare better.

[3] `https://code.google.com/p/treerepair`

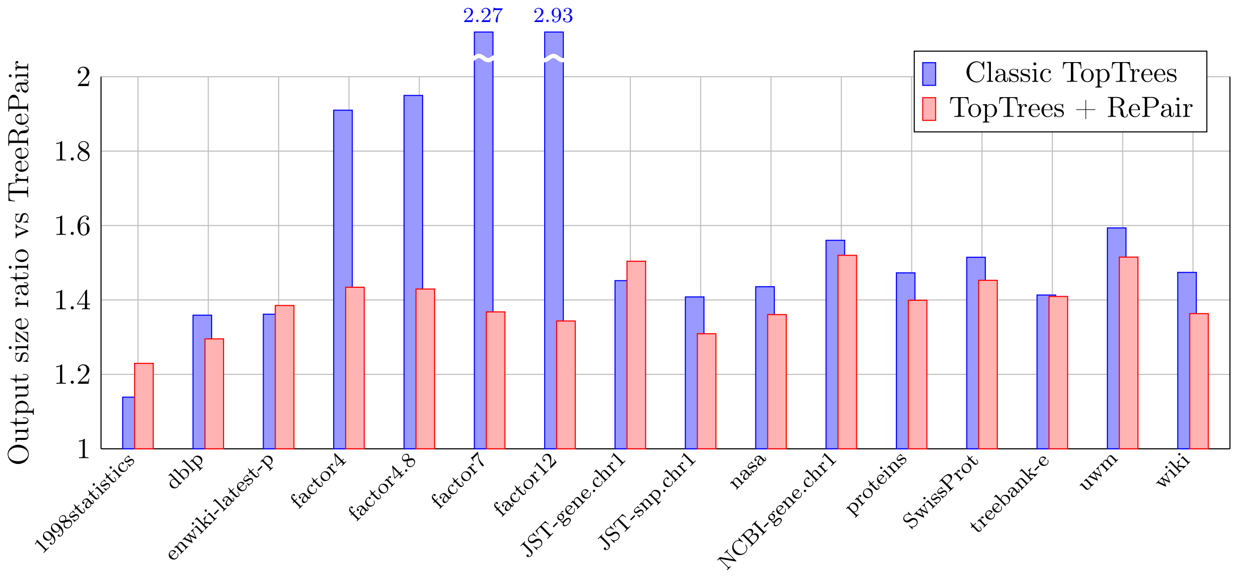

Figure 3: Comparison of output file sizes produced by top tree compression with and without the RePair combiner, measured against TreeRePair file sizes (lower is better)

Speed. Using our RePair-inspired combiner increases the running time of the top tree creation stage, doubling it on average. Our implementation of classical top tree compression was 10.5 times faster than TreeRePair on average over the corpus from Table 1, and still 6.2 times faster when using our RePair combiner. Detailed running time measurements are given in Table 2. In particular, classical top tree compression takes only twice as long as `gzip -9` on average, and 3.3 times when using our RePair combiner (TreeRePair: 21.2). In contrast, `bzip2` is 15.4 times *slower* than top tree compression on average, and 9.7 times when using our RePair combiner. This strikingly demonstrates the method's qualities as a fast compressor.

Performance on Random Trees. We examine random trees to show that tree compression with top trees is a very versatile method, and that it does not rely on typical characteristics of XML files. By definition, random trees do not compress well. Thus, we can use them to approximate worst-case behaviour. We generate trees uniformly at random using a method developed by Atkinson and Sack [2]—note that the method is not limited to binary trees. For the generated trees, we compare the average number of edges $n_{\mathcal{TD}}$ in the Top DAG to the information-theoretic lower bound of $\Omega(n_T / \log_\sigma n_T)$ for a tree of size n_T. The results of this are shown in Figure 4 for $\sigma = 2$. We can see that apart from some oscillation, the values are in a very small range between 0.0889 and 0.0904 and do not show an overall tendency to grow or shrink, except for the amplitude of oscillation. This suggests that tree compression with top trees performs asymptotically optimal on random trees.

The oscillation or zig-zag behaviour exhibited in Figure 4 poses a riddle. The period duration doubles with each quarter of a period, exhibiting exponential growth, while the wave's amplitude appears to grow by a constant amount per quarter period. We do not have a definitive explanation for the causes of this behaviour. However, we can speculate about possible contributing factors. For one, consider the subtrees that could be shareable in the top tree. Their height, and therefore number, grows logarithmically with the height of the top tree, which in turn grows logarithmically

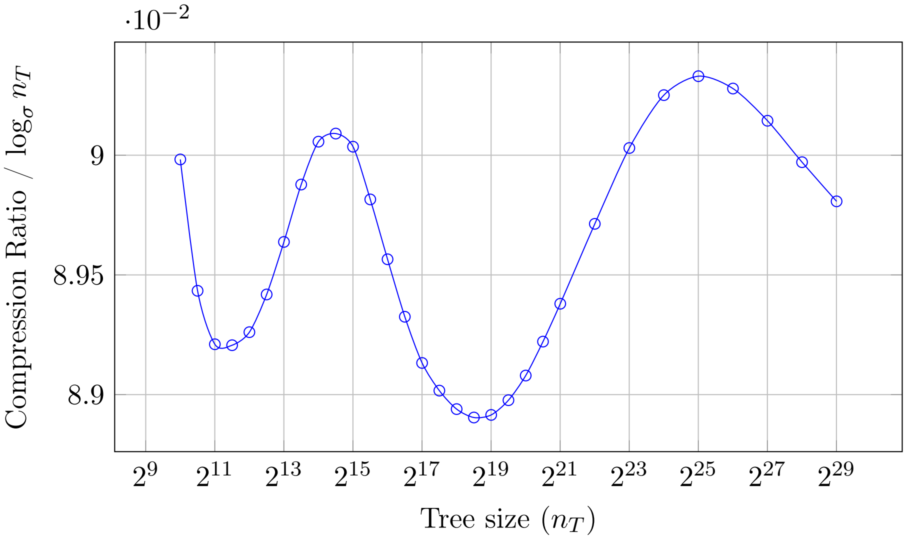

Figure 4: Compression ratio in terms of the average number of edges, divided by information-theoretical compression ratio bound $\log_\sigma n_T$, over 1000 random trees of size $n_T = 2^{10}$ to 2^{29} with $\sigma = 2$. All values are contained in a small range, suggesting asymptotically optimal—$\mathcal{O}(n_T/\log_\sigma n_T)$—compression.

with respect to the input tree's size. Thus, the number of potentially shareable subtrees grows proportional to $\log \log n_T$. As the potential for DAG compression grows with the number of shareable subtrees, we would expect a sawtooth-like pattern in the compression ratios, spiking whenever the shareable subtree height increases. This could contribute to the exponential growth in period duration. Further investigation beyond the scope of this paper would be required to account for the smoothness and amplitude of the curve.

5 Conclusions

We have demonstrated that tree compression with top trees is viable, and suggested several enhancements to improve the degree of compression achieved. Using the notion of combiners, we demonstrated that significant improvements can be obtained by carefully choosing the order in which clusters are merged during top tree creation. We showed that the worst-case compression ratio is within a $\log \log_\sigma n_T$ factor of the information-theoretical bound, and experiments with random trees suggest that actual behaviour is asymptotically optimal. Further, we gave efficient methods to navigate the compressed representation, and described how the top DAG can be encoded to support efficient navigation without prior decompression.

We thus conclude that tree compression with top trees is a very promising compressor for labelled trees, and has several key advantages over other compressors that make it worth pursuing. It is our belief that its great flexibility, efficient navigation, high speed, simplicity, and provable bounds should not be discarded easily. While further careful optimizations are required to close the compression ratio gap, tree compression with top trees is already a good and fast compressor with many advantages.

Future Work We expect that significant potential for improvement lies in more sophisticated combiners. The requirements for combiners give us a lot of space to devise better merging algorithms. Combiners might also be used to improve locality in the top tree in addition to compression performance, leading to better navigation performance. Moreover, additional compression improvements should be achievable with carefully engineered output representations. Since the vast majority of total running time is currently spent on the construction of the top DAG, using more advanced encodings may improve compression without losing speed. One starting point to replace our relatively naïve representation could be a decomposition of the top DAG into two spanning trees [20].

Table 2: Running times in seconds, median over ten iterations

File name	**TopTrees**	**TT+RePair**	**TreeRePair**	**RePair**	**gzip -9**	**bzip2**
1998statistics	0.00	0.01	0.05	0.04	0.00	0.13
dblp	6.00	11.21	45.72	39.57	2.46	74.77
enwiki-latest-p	3.92	7.14	32.98	28.33	1.29	49.12
factor12	7.16	11.54	109.47	54.19	4.48	81.86
factor4.8	2.82	4.70	46.22	21.61	1.79	33.09
factor4	2.40	3.92	39.47	17.75	1.49	28.21
factor7	4.21	6.84	67.83	31.55	2.61	48.60
JST-gene.chr1	0.04	0.06	0.38	0.54	0.03	1.27
JST-snp.chr1	0.24	0.38	2.12	3.40	0.17	6.33
nasa	0.15	0.23	0.94	0.85	0.06	1.86
NCBI-gene.chr1	0.31	0.51	2.25	3.33	0.20	7.97
proteins	6.92	11.88	50.17	53.27	2.41	81.92
SwissProt	1.13	2.13	12.35	5.74	0.50	11.15
treebank-e	1.35	1.99	12.70	4.00	2.80	3.62
uwm	0.01	0.02	0.11	0.09	0.00	0.28
wiki	0.78	1.17	5.59	4.28	0.22	9.21

Table 3: Compressed file sizes in Bytes

File name	**Succinct**	**TopTrees**	**TT+RePair**	**TreeRePair**	**RePair**	**gzip -9**	**bzip2**
1998statistics	18 426	788	851	692	1 327	4 080	1 301
dblp	13 740 160	1 486 208	1 416 538	1 093 533	2 037 878	2 476 347	1 116 311
enwiki-latest-p	7 901 904	516 638	525 532	379 410	866 161	1 490 278	544 606
factor12	16 402 888	2 069 437	948 167	705 740	3 092 194	6 342 947	2 913 894
factor4.8	6 565 499	1 070 045	784 519	548 853	1 587 043	2 542 773	1 168 654
factor4	5 473 158	937 660	704 105	490 945	1 429 872	2 119 269	973 463
factor7	9 571 503	1 421 376	855 063	625 094	2 248 370	3 702 132	1 700 043
JST-gene.chr1	96 159	5 332	5 523	3 672	8 273	33 316	7 027
JST-snp.chr1	547 594	29 084	27 039	20 654	50 347	194 862	49 857
nasa	341 161	42 077	39 883	29 310	60 394	83 231	34 404
NCBI-gene.chr1	721 803	17 880	17 418	11 459	29 912	199 308	47 901
proteins	17 315 832	905 613	860 366	614 892	1 537 249	3 214 663	1 141 697
SwissProt	2 343 730	598 960	574 466	395 417	699 757	829 119	398 197
treebank-e	2 396 061	1 173 463	1 170 304	830 324	1 537 334	1 858 722	1 032 303
uwm	37 491	2 177	2 070	1 366	3 101	7 539	2 102
wiki	1 242 418	110 686	102 371	75 090	171 075	247 898	93 858

References

[1] Stephen Alstrup, Jacob Holm, Kristian De Lichtenberg, and Mikkel Thorup. Maintaining information in fully dynamic trees with top trees. *ACM TALG*, 1(2):243–264, 2005.

[2] Michael D Atkinson and J-R Sack. Generating binary trees at random. *Information Processing Letters*, 41(1):21–23, 1992.

[3] Philip Bille, Inge Li Gørtz, Gad M. Landau, and Oren Weimann. Tree compression with top trees. *Information and Computation*, 2015.

[4] Philip Bille, Gad M Landau, Rajeev Raman, Kunihiko Sadakane, Srinivasa Rao Satti, and Oren Weimann. Random access to grammar-compressed strings. In *Proc. SODA*, pages 373–389. SIAM, 2011.

[5] Peter Buneman, Martin Grohe, and Christoph Koch. Path queries on compressed XML. In *Proc. 29th VLDB*, pages 141–152. VLDB Endowment, 2003.

[6] Giorgio Busatto, Markus Lohrey, and Sebastian Maneth. Grammar-based tree compression. Technical Report EPFL-REPORT-52615, École Polytechnique Fédérale de Lausanne, 2004.

[7] Giorgio Busatto, Markus Lohrey, and Sebastian Maneth. Efficient memory representation of XML documents. In *Proc. DBPL*, pages 199–216. Springer, 2005.

[8] Moses Charikar, Eric Lehman, Ding Liu, Rina Panigrahy, Manoj Prabhakaran, Amit Sahai, and Abhi Shelat. The smallest grammar problem. *IEEE Trans Inf Theory*, 51(7):2554–2576, 2005.

[9] O'Neil Davion Delpratt. *Space efficient in-memory representation of XML documents*. PhD thesis, University of Leicester, 2009. Supervisor: Rajeev Raman.

[10] Peter J Downey, Ravi Sethi, and Robert Endre Tarjan. Variations on the common subexpression problem. *Journal of the ACM (JACM)*, 27(4):758–771, 1980.

[11] Paolo Ferragina, Fabrizio Luccio, Giovanni Manzini, and S Muthukrishnan. Compressing and indexing labeled trees, with applications. *Journal of the ACM (JACM)*, 57(1):4, 2009.

[12] Simon Gog, Timo Beller, Alistair Moffat, and Matthias Petri. From theory to practice: Plug and play with succinct data structures. In *Proc. 13th SEA*, pages 326–337, 2014.

[13] Mika Hirakawa, Toshihiro Tanaka, Yoichi Hashimoto, Masako Kuroda, Toshihisa Takagi, and Yusuke Nakamura. JSNP: a database of common gene variations in the Japanese population. *Nucleic Acids Research*, 30(1):158–162, 2002.

[14] Guy Jacobson. Space-efficient static trees and graphs. In *Proc. 30th FOCS*, pages 549–554. IEEE, 1989.

[15] Artur Jez and Markus Lohrey. Approximation of smallest linear tree grammar. *CoRR*, abs/1309.4958, 2013.

[16] N Jesper Larsson and Alistair Moffat. Off-line dictionary-based compression. *Proceedings of the IEEE*, 88(11):1722–1732, Nov 2000.

[17] Markus Lohrey and Sebastian Maneth. The complexity of tree automata and XPath on grammar-compressed trees. *Theoretical Computer Science*, 363(2):196–210, 2006.

[18] Markus Lohrey, Sebastian Maneth, and Roy Mennicke. XML tree structure compression using RePair. *Information Systems*, 38(8):1150–1167, 2013.

[19] Sebastian Maneth and Giorgio Busatto. Tree transducers and tree compressions. In *FoSSaCS*, pages 363–377. Springer, 2004.

[20] Shirou Maruyama, Masaya Nakahara, Naoya Kishiue, and Hiroshi Sakamoto. ESP-index: A compressed index based on edit-sensitive parsing. *Journal of Discrete Algorithms*, 18:100–112, 2013.

[21] Gerome Miklau. University of Washington XML Repository. `http://www.cs.washington.edu/research/xmldatasets`.

[22] J Ian Munro and Venkatesh Raman. Succinct representation of balanced parentheses and static trees. *SIAM Journal on Computing*, 31(3):762–776, 2001.

[23] A. Poyias. XXML: Handling extra-large XML documents. 2013.

[24] Mihai Pătraşcu. Succincter. In *Proc. 49th FOCS*, pages 305–313. IEEE, 2008.

[25] Fangju Wang, Jing Li, and Hooman Homayounfar. A space efficient XML DOM parser. *Data & Knowledge Engineering*, 60(1):185–207, 2007.

[26] Wikimedia. enwiki dump. `http://dumps.wikimedia.org/enwiki/`.