Annotated Trees and their Applications to XML Compression

Tomasz Müldner¹, Jan Krzysztof Miziołek² and Tyler Corbin¹
¹Jodrey School of Computer Science, Acadia University, Wolfville, B4P 2A9 NS, Canada
²Faculty of Artes Liberales, University of Warsaw, Warsaw, Poland

Keywords: XML, Tree Compression using Annotated Trees, Permutation-based XML Compression.

Abstract: Permutation based XML-conscious compressors permute the input document to improve the compression ratio and support efficiency of operations, such as queries or updates. One such compressor, XSAQCT, uses the properties of the permuted document, called an annotated tree, to these operations. This paper provides the formal background for the definition of an of D. It also provides an algorithm for creating an annotated tree for the XML document and its reverse algorithm, and discusses a measure of compressibility using an annotated tree. The theoretical and algorithm approaches are followed by the experimental results showing compressibility of annotated trees and a general analysis of semi-structured data and XML compression.

1 INTRODUCTION

A tree is one of the most important and popular structures in computing, used to represent the relations between nodes. Therefore, there has been considerable research on succinct representation of trees while allowing various operations on these trees to be efficiently performed, see (Chen and Reif, 1996), (Bille et al., 2013), (Jacobson, 1989) and (Benoit et al., 1999). The eXtensible Markup Language, XML (XML, 2013), is one of the most popular data formats for the serialization of tree data structures and for the storage of relational data. Since XML documents are hierarchal and acyclic in nature, there have been numerous attempts to apply techniques used for general tree compression to XML, see e.g., (Busatto et al., 2005), (Ferragina et al., 2009) and (Busatto et al., 2008). A specific subset of tree-compressors designed specifically for XML (or semi-structured data in general) are called XML-conscious compressors, e.g., XQueC (Arion et al., 2007). These compressors typically parse the XML, and either sequentially build a model during compression or build a model and then compress the contents. XML-conscious permutation-based compressors permute the input document during a pre-order traversal and apply a partitioning strategy to group content nodes into a series of data containers compressed using a general-purpose compressor (a back-end compressor). For example, XML compressor XSAQCT is a permuting and streamable XML compressor, see (Müldner et al., 2009). The compression starts with SAX-based parsing to permute the input into a compressed form of the XML tree, called an annotated tree. Then it is followed by storing data values in data containers and compressing data using a back-end compressors (such as GZIP (GZIP, 2013) or BZIP2 (bzip2, 2013)).

The annotated tree in XSAQCT can be considered to be a high level index, which was proved to be useful for various applications, e.g., updates, online rather than offline compression see (Corbin et al., 2013), and parallelization of the implementation. However, the formal definition of this mapping was never provided nor was the proof that it can be reversed. These questions are very important because without answering them, the compression process used by XSAQCT is not known to be lossless. This paper fills in these gaps, by providing a formal definition of a tree and an annotated tree, and the mapping τ from a labelled tree to the annotated tree, as well as the proof that τ is injective and can be inverted. An algorithm to create an annotated tree and its reverse are provided, followed by the discussion of the compressibility measure of this approach along with experimental results and a general analysis of XML compression.

Contributions. There are several novel contributions of this paper: (1) The theoretical part, i.e., the formal background for the definition of an annotated tree and a proof that the mapping from a tree to the
an annotated tree is injective, and therefore, the annotated tree for the labelled tree $D$ provides a faithful representation of $D$: (2) An application of of the annotated tree methodology with respect to XML, i.e., a formal proof that XSAQCT’s compression process is lossless; (3) The algorithmic approach, i.e., the algorithm which inputs an arbitrary (cyclic or not) tree and outputs an annotated tree, and the “inverse” algorithm for XML, which inputs an annotated tree for the XML document, and outputs this document; (4) Quantification of text trees and a discussion of mutual information; (5) A compressibility measure defining the cost of using annotated trees and results of testing using an especially designed XML suite, showing high compressibility; and (6) General analysis of XML compression.

This paper is organized as follows. Section 2 introduces the formal background for this paper, including a definition of a tree and an annotated tree. Section 3 provides a description of trees with text elements. Section 4 provides algorithms implementing various mappings and shows time complexity of these algorithms. Section 5 provides results of testing and a general analysis, and finally Section 6 provides conclusions and describes future work.

2 TREES AND ANNOTATED TREES

2.1 Labeled Trees

Definition 1. Let $\Sigma$ be an alphabet, called the label alphabet. A labeled tree is an ordered tree with nodes labeled by strings from $\Sigma$ and having arbitrary degrees (i.e., number of children).

In what follows, by a tree we mean a labeled tree, and by Trees we denote the set of all trees. We use the letter $D$ to denote a tree, with nodes denoted by lower-case letters $x, y, u, v$ (with indices when needed) and by $x(a)$ we denote a node $x$ labeled with $a$. Let $\text{Label}(x)$ denote the label of the node $x$, $\text{Height}(D)$ denote the height of $D$, $\text{Nodes}(D)$ denote the set of all nodes of $D$, and $D_i$ be the tree consisting of all nodes and edges in $D$ at levels 1, . . . , $i$, where 1 $\leq i \leq \text{height}(D)$. Example of a tree is shown in Figure 1.

Definition 2. A path in a tree is defined to be of the form $/x_1/x_2/ . . . /x_k$, where $x_1$ is the root of $D$ and for $1 \leq i < k$, $x_{i+1}$ is a child of $x_i$.

We use a lower-case letter $p$ (with indices when needed) to denote a path and $\text{Paths}(D)$ to denote the set of all paths in $D$. For the path $p = /x_1/x_2/ . . . /x_k$, let $\text{Length}(p)$ be the number of nodes in the path $p$.

Figure 1: Example of a tree $D$. Let $A$ partial relation $\sim$ on nodes and edges in $D$, $\text{Label}$ and $\text{Length}$.

\[ \text{Last}(p) \text{ be the last element in } x_k, \text{ Label}(x_k) \text{ be the label of } p, \text{ and for } \text{Length}(p) > 1, \text{ let } p_1 \text{ be the path except the last element.} \]

Definition 3. Two paths in the tree $D$, $p_1 = /x_1/x_2/ . . . /x_k$ and $p_2 = /y_1/y_2/ . . . /y_k$ are called similar if $k = n$, $x_1 = y_1$ is the root of $D$ and for $1 \leq i < k$, $\text{Label}(x_i) = \text{Label}(y_i)$.

The similarity relation is an equivalence relation and we denote by $[p]$ the equivalence class of $p$, by $\text{Similar}(D)$ the quotient set of this relation, i.e., the set of all different sets of similar paths. For $p \in \text{Paths}(D)$ let $\text{Length}([p]) = \text{Length}(p)$ be the length of the equivalence class, $\text{Label}([p]) = \text{Label}(p)$ be the label of the class (it is easy to see that both definitions of the length and the label of an equivalence class are well defined, i.e., independent of the choice of the path $p$ from the equivalence class). Finally, for $q \in \text{Similar}(D)$ let $|q|$ be the number of paths in $q$, and $\text{Last}(q)$ be the sequence of nodes that are last elements of all paths in this class, ordered from left to right. Clearly, $|\text{Last}(q)| = |q|$.

In Figure 1, $\text{Similar}(D) = \{ [[/x]], [[/x/y]], [[/x/y_1/u_1]], [[/x/y_1/u_3]], [[/x/y_1/u_1/v_1]], [[/x/y_1/u_1/v_2]], [[/x/y_1/u_1/v_4]], [[/x/y_1/u_1/v_1/w_1]], [[/x/y_1/u_1/v_3/w_2]] \}$ and for $p = /x/y_1, [p] = \{ /x/y_1, /x/y_2 \}$: \text{Length}([p]) = 2, $\text{Label}(p) = b$, $\text{Last}([p]) = \langle y_1, y_2 \rangle$, and $|p| = 2$.

Definition 4. A partial relation $\prec$ in the set $\text{Similar}(D)$ is defined as follows: $[p_1] \prec [p_2]$ iff $\text{Length}([p_1]) = \text{Length}([p_2]) > 1$, $[p_1] \neq [p_2]$, $\text{Label}([p_1])$ is different from $\text{Label}([p_2])$, and there exist paths $p_1 \in [p_1]$ and $p_2 \in [p_2]$ such that the node $\text{Last}(p_1)$ is the left sibling of the node $\text{Last}(p_2)$. The tree $D$ has a cycle if there exist $q_1, q_2 \in \text{Similar}(D)$ such that $q_1 \prec q_2$ and $q_2 \prec q_1$. $D$ is acyclic if it does not have a cycle.

For $D$ in Figure 1, $[[/x/y_1/u_1/v_1]] \prec [[/x/y_1/u_1/v_2]]$, while $[[/x/y_1/u_1]]$ and $[[/x/y_1/u_3]]$ are
not in relation $\prec$. There would be a cycle in $D$ if there was a node $u_0(e)$ between nodes $u_1$ and $u_2$. By *Acyclic* we denote the set of all acyclic trees.

When it does not lead to confusion, a tree can be represented using a *simplified notation* (used e.g. for XML), by omitting node names, e.g., replacing $y_i(b)$ by $b_i$. In this notation, an equivalence class $[p]$ will be denoted using the label $\text{Label}(p)$, in upper case, e.g., for $D$ from Figure 1, $[x/y_1/u_1/v_1]$ will be denoted by $E$.

### 2.2 Annotated Trees and g-trees

A tree $D$ can permutate to create an annotated tree with nodes represented by equivalence classes of the similarity relation. In the worst case, if for each $q \in \text{Similar}(D)$, $|q| = 1$ then the size of $D$ would be the same as the size of the corresponding annotated tree. However, typically XML documents are regular, i.e., for majority of paths $q \in \text{Similar}(D)$, $|q| \gg 1$ and the annotated tree provides a compressed representation of $D$. For a single tree $D$, there may be more than one annotated tree such that each annotated tree can be mapped back to $D$. To formalize this idea, in this section we define annotated trees and annotated g-trees. Then we define two mappings, an injective mapping from the set of trees to the set of annotated g-trees and an injective mapping from the set of annotated trees to set of subsets of annotated trees. Finally, we define annotated text trees and a mapping from text trees to the annotated text trees. In this section we consider only acyclic trees, but in Section 4 we provide algorithms for all types of trees. In what follows, by a *dag* we mean an acyclic digraph.

**Definition 5.** An *annotated tree* is an ordered tree with nodes additionally labeled by annotations (sequences of non-negative integers). An *annotated g-tree* is an unordered tree $A$ (i.e., children are not ordered) such that (1) nodes of $A$ are dags; (2) each dag $G \in \text{Nodes}(A)$ except the root has its nodes annotated; and (3) for every node $H \in \text{Nodes}(A)$ and for every child $G$ of $H$ there exists exactly one node $n \in \text{Nodes}(H)$, called the source of $G$, and different children of $H$ have different sources.

Nodes in the annotated tree are denoted by uppercase letters (with indices where appropriate), e.g., $X(n)[\alpha_1, \ldots, \alpha_j]$ denotes a node $X$ labeled with the label $n$ and annotation $[\alpha_1, \ldots, \alpha_j]$ or in a simplified notation it is denoted as the node $N(\alpha_1, \ldots, \alpha_j)$. Let $\text{AnnotationSum}(X) = \sum_{i=1}^{j} \alpha_i$ be the sum of all annotations of the node $X$. By *Annotated* we denote the set of all annotated trees. Two examples of annotated trees (using a simplified notation) are shown in Figure 2 and Figure 3. As we will explain it later, both these trees represent the same tree $D$ from Figure 1. Example of an annotated g-tree is given in Figure 4 (the source of a node is shown using a dashed arrow). By *Annotated* $- G$ we denote the set of all annotated g-trees and by a *chain* we mean a rooted dag such that each node except the root has the in-degree one and each node except one designated node called the sink has the out-degree one; the sink has the out-degree zero. The reason for defining g-trees is that a dag $G$, which is a chain, will have its nodes representing children of the source (in left-to-right order) of $G$. If a dag $G$ is not a chain then $G$ needs to be topologically sorted for our usage. For example, in Figure 4 the topological sorting of the dag containing nodes $E, F$ and $G$ may produce the chain $E, F, G$ and these three nodes can be made children of the source of this graph, i.e., the node $C$.

### 2.3 Tree Isomorphism

Since we will show that the mapping from the set of trees to the set of g-trees is injective, we need to define the concept of "identical" or isomorphic trees, which differ only in names of the corresponding nodes. We use a similar concept for annotated trees and g-trees.

**Definition 6.** Two trees $D_1$ and $D_2$ are *isomorphic* *iff* they have the same height and for each level $i, 1 \leq i \leq h$, the sequence of all nodes $<n_1, \ldots, n_j>$ (in left-to-right order) in $D_1$ at level $i$ and the sequence of all
nodes \(<m_1, \ldots, m_r>\) (in left-to-right order) in \(D_2\) at level \(i, k = j\), and for \(1 \leq r \leq k\), nodes \(n_t\) and \(m_r\) have the same degree and label.

2.4 Mapping Trees to Sets of Annotated Trees

We define the mapping \(\tau : \text{Acyclic} \rightarrow \text{Annotated} \rightarrow G\) in two steps; first for nodes and the tree structure, and then for annotations of nodes. Nodes in all DAGs are represented by equivalence classes of the similarity relation and written as \([p] (\text{Label}(p)) \{\alpha_1, \ldots, \alpha_j\}\) or using a simplified notation \(\text{Label}(p) \{\alpha_1, \ldots, \alpha_j\}\). If \(x\) is a node in \(D\), which is not a root, then \(x\) denotes a (unique) path \(p \in \text{Paths}(D)\) which ends in \(x\).

**Definition 7.** Definition of mapping \(\tau : \text{Acyclic} \rightarrow \text{Annotated} \rightarrow G\). Let \(D \in \text{Acyclic}.

- **Mapping labeled nodes and the tree structure**
  1. The root \(r\) of \(D\) is mapped to the root of \(\tau(D)\) defined as a graph consisting of a single node \([r]\). \(\text{Label}(r)\).
  2. For any level \(i\) \(\geq 1\) of \(D\) and equivalence class \(q \in \text{Sim}(D)\) of length \(i - 1\), let \(N_{q,i}\) denote the set of nodes \(x\) in \(D\) at level \(i\) such that \([p_x] \in q\). Clearly, the sets \(\{N_{q,i}\} \in q\) form a disjoint coverage of the set of all nodes in \(D\) at level \(i\). Each set \(N_{q,i}\) is mapped to \(\tau\) to the single graph \(G \in \text{Nodes}(\tau(D))\) at level \(i\) \(G = \{[p_x]\}\text{Label}(p_x)\) where the source of the graph \(G\) is the node \(q\). For any graph \(G \in \text{Nodes}(\tau(D))\) and two nodes \(q_1, q_2 \in G\), there is an edge \(q_1 \rightarrow q_2\) in \(G\) if \(q_1 < q_2\) (see Definition 4). Since \(D\) is assumed to have no cycles, \(G\) is acyclic. The node \(H \in \text{Nodes}(\tau(D))\) is the parent of the node \(G\) if the source of \(G\) belongs to the dag \(\text{Nodes}(H)\).

From the definition of sets \(\tau(N_{q,i})\), \(H\) is the unique parent of \(G\).

- **Annotations are defined by induction on the height of \(\tau(D)\)**
  1. The annotation of the root is \([1]\)
  2. Assume that annotations are defined up to the level \(i\), \(1 \leq i < \text{Height}(D)\) and for any equivalence class \(q\) of length \(i\), \([q]\) \(\text{AnnSum}(q)\), i.e., the sum of all annotations for the node \(q\) is equal to the number of last elements in all paths in this class. Consider a dag \(G\) at level \(i + 1\) with the source \(X\), and a node \(Y \in \text{Nodes}(G)\). Let \(Y = \{[p] \{r\} \{\alpha_1, \ldots, \alpha_j\}\}, X = \{[p]\}\{m\}\) and \(k = \text{AnnSum}(X)\). First, we set the number \(j\) of annotations in \(Y\) to be \(k\). From the inductive assumption it follows that the sequence \(\text{Last}([p])\) has \(k\) elements \(s_1, \ldots, s_k\). For \(1 \leq j \leq k\), we define \(\alpha_j\) to be the number of children in \(D\) of the node \(s_j\) which are labeled with \(r\). It is easy to see that \(\text{Last}([p]) = \text{AnnSum}(Y)\).

Let \(\text{Trees}_\tau\) denote the image \(\tau(\text{Acyclic}) \subset \text{Annotated} \rightarrow G\). From the Definition 7, it follows that any g-tree \(A \in \text{Trees}_\tau\), \(A = \tau(D)\) has the following properties:

1. \(\text{Height}(D) = \text{Height}(A)\)
2. For \(1 \leq i \leq \text{Height}(D)\), \(A\) and \(D\) have identical sets of labels at level \(i\).
3. For a dag \(G \in \text{Nodes}(A)\), let \(X\) be the source of \(G\). If \(X = [q]\) and \(k = \text{AnnSum}(Y)\). Then for each node \(Y \in \text{Nodes}(G)\), \(Y = \{[p]\} \{m\}\) \(\{\alpha_1, \ldots, \alpha_j\}\), there exists \(1 \leq i \leq k\) such that \(\alpha_i > 0\) \(j = k\), and \([q] = \{p\}\).
4. If a node \(q\) is not a source of any dag, then all nodes in the sequence \(\text{Last}(q)\) are leaves in the tree \(D\).

For the tree \(D\) from Figure 4, the g-tree \(\tau(D)\) is shown in Figure 4. From Property 1 it follows that the height of a g-tree for the document \(D\) is the same as that of \(D\); however, typically these trees have different width and therefore represent a compressed form (see more in Section 5).

**Theorem 1.** The mapping \(\tau : \text{Acyclic} \rightarrow \text{Trees}_\tau\) is injective.

**Proof.** We will define the mapping \(\tau^{-1} : \text{Trees}_\tau \rightarrow \text{Acyclic}\) such that \(\tau^{-1} \circ \tau = \tau \circ \tau^{-1}\) is an identity mapping. i.e., it maps a tree to an isomorphic tree. Let \(A \in \text{Trees}_\tau\). Then \(\tau^{-1}\) is defined inductively on levels \(i\) of \(A\):

1. For \(i = 1\), let the root of \(A\) be a graph \(B\) consisting of a single node \(X(a)[1]\). Then \(\tau^{-1}(G) = x(a)\).
2. Assume that for all levels \(i\), \(1 \leq i < \text{Height}(A)\), \(D_i \rightarrow \tau^{-1}(A_i) \in \text{Acyclic}\). We define \(\tau^{-1} : A_{i+1} \rightarrow D_{i+1}\). Let \(G\) be a dag in \(A\) at level \(i + 1\), and \(\text{Nodes}(G) = \{Y_1, \ldots, Y_m\}\), where for \(j, 1 \leq j \leq m\), \(Y_j = [q] \{r\} \{\alpha_1, \ldots, \alpha_j\}\), the node \(X = \{[p]\}\{m\}\) be the source of \(G\), \(k = \text{AnnSum}(X)\), and \(\text{Last}([p])\) be the tuple \(s_1, \ldots, s_k\). For \(h, 1 \leq h \leq k\) we now define \(\tau^{-1}(G)\) as nodes in \(D\) at level \(i + 1\) that are children of each node \(s_h\). First, for \(j, 1 \leq j \leq m\), let \(N_{h,j}\) be the set of nodes in \(D\) at level \(i + 1\) consisting of \(\alpha_j\) nodes in \(D\) labeled by \(r_j\). We define a partial order between these sets as follows: for \(j_1\) and \(j_2\), \(1 \leq j_1, j_2 \leq m\), \(N_{h, j_1} \prec N_{h, j_2}\) iff in \(G\) there is
an arc between \([g_i,j] and [g_j,j]\). Finally, children of \(s_k\) are nodes from sets \(N_{h,1},\ldots,N_{h,m}\) defined as follows: (1) within each set \(N_{h,j}\), nodes are arbitrarily ordered and appear one after another; (2) for any two sets \(N_{h,j}\) and \(N_{h,j}\), all nodes from \(N_{h,j}\) appear to the left of all nodes in \(N_{h,j}\) if \(N_{h,j}\prec N_{h,j}\); and (3) for any set \(N_{h,j}\) which is not in the relation \(\prec\), we arbitrarily place children from this set to appear after all nodes from sets which are in this relation with another set.

Clearly the tree \(\tau^{-1}(A)\) is acyclic and for \(D \in \text{Acyclic}\), \(\tau^{-1}\circ \tau(D) = D\), for \(A \in \text{Trees}_t\), \(\tau^{-1}\circ \tau(A) = A\).

Next, we define the mapping \(\gamma : \text{Trees}_t \rightarrow 2^{\text{Annotated}_t}\). Let \(\text{TS}(G)\) be the set of all topological sortings of a dag \(G\), and for \(G_1,\ldots,G_n\), \(P(G_1,\ldots,G_n)\) be a Cartesian product \(\times_{a \in \text{TS}(G)}\). If \(\gamma(g_1,\ldots,g_n) \in P(G_1,\ldots,G_n)\), then each graph \(g\) represents a topologically sorted graph \(G\).

**Definition 8.** Mapping \(\gamma\) is defined as follows: For \(T \in \text{Trees}_t\), \(\gamma(T) = \{\gamma(g_1,\ldots,g_n) : g_1,\ldots,g_n \in P(G_1,\ldots,G_n)\}\), where \(\gamma(g_1,\ldots,g_n)(T)\) is the \(g\)-tree with all graphs \(G_1,\ldots,G_n\) replaced respectively by graphs \(g_1,\ldots,g_n\), and having the same arcs between dags (as well as sources) as in the tree \(T\).

Let \(\text{Trees}_\tau\) denote the image \(\gamma(\text{Trees}_t) \subset 2^{\text{Annotated}_t}\).

**Theorem 2.** The mapping \(\gamma : \text{Trees}_t \rightarrow \text{Trees}_\tau\) is injective.

**Proof.** We will define the mapping \(\gamma^{-1} : \text{Trees}_\tau \rightarrow \text{Trees}_t\) s.t. \(\gamma^{-1} \circ \gamma = I\) and \(\gamma \circ \gamma^{-1}\) are identity mappings. It is sufficient to show that each topologically sorted dag \(g\) with annotated nodes can be uniquely mapped to the dag \(G\). Two different nodes \(X[\alpha_1,\ldots,\alpha_m]\) and \(Y[\beta_1,\ldots,\beta_n]\) from the dag \(g\) will be called dependant if there exists \(m\), \(1 \leq m \leq k\) such that \(\alpha_m > 0\) and \(\beta_m > 0\). Now, let us define \(\gamma^{-1}(g)\) to be the graph consisting of the same nodes as in the graph \(g\) and with the same source, but with arcs defined as follows: there is an arc \(X \rightarrow Y\) iff the node \(X\) appears in the topological sort used in \(g\); before the node \(Y\), and \(X\) and \(Y\) are dependant.

It is easy to see that each \(g\)-tree \(T \in \text{Trees}_\tau\) is in one-to-one correspondence with an annotated tree. Since \(\tau : \text{Acyclic} \rightarrow \text{Trees}_t\) and \(\gamma : \text{Trees}_t \rightarrow \text{Trees}_\tau\), each tree \(D\) can be mapped to the set of annotated trees, denoted by \(\text{Annotated}(D)\) and defined by the composition of \(\tau\) and \(\gamma\). It is easy to see that every annotated tree from the set \(\text{Annotated}(D)\) represents

### 3 TEXT TREES AND THEIR COMPRESSION

Since the procedure of compressing text trees is almost identical to the procedure of compressing labeled trees, in this section we provide only the description of how text nodes are dealt with.

**3.1 Text Trees**

**Definition 9.** Let \(\Delta\) be an alphabet, called the text alphabet, its elements are \(\setminus 0\) terminated strings. A text tree is a tree with two kinds of nodes: element nodes labeled by strings from \(\Sigma^*\) (see Definition 1) and text nodes labeled by strings from \(\Delta^*\) such that text nodes are always leaves, the root of the text tree is an element node, and any two sibling text-nodes are separated by at least one element node.

For text trees we use the same notations as for trees; text labels are denoted using letter \(t\) (with in-
An element leaf node in the text tree is $X$.

Let $X$ is a leaf. Then for every

Let $D$ is a not leaf and it has $m$ children, $Y$

For a node $X$ in the annotated tree

We consider two cases:

1. $X$ is a leaf. Then for every $1 \leq i \leq k$, $x_i$ has exactly one text child, and let $T$ be the concatenation of labels of these $k$ children. Clearly, $\text{Number}(X) = \text{AnnotationSum}(X) = k$.

2. $X$ is a not leaf, and it has $m$ children, $Y_1, \ldots, Y_m$. Thus, there are $\text{Number}(X)$ text children of nodes $x_1, \ldots, x_k$, and we define $T$ to be the concatenation of labels of these children (in left-to-right order).

**Theorem 3.** The mapping of text elements is injective.

**Proof.** Let $A$ be an annotated text tree and $D$ be the text tree; we will describe how text labels from $A$ are mapped into the text elements in $D$. Consider the node $X \in \text{Nodes}(A)$ and let $k = \text{Number}(X)$. The image of $X$ under the reverse mapping is the sequence of element nodes $x_1, \ldots, x_k$. We consider two cases:

1. $X(T)$ is a leaf. Then $k = \text{Number}(X) = \text{AnnotationSum}(X)$ and $T$ is a concatenation of $k$ texts $t_1, \ldots, t_k$. For every $1 \leq i \leq k$, we define a text child $y_i(t_i)$ of the node $x_i$.

2. $X(T)$ is a not leaf and it has $m$ children $Y_T(t_1), \ldots, Y_T(t_m)$, where $T$ is a concatenation of

$$r = (\sum_{j=1}^{m} \text{AnnotationSum}(Y_j)) + \text{AnnotationSum}(X)$$

texts $s_1, \ldots, s_k$. Given a node $x$ in a tree from $\text{AcyclicTextTrees}$ with $k$ element children $y_1, \ldots, y_k$ and $k + 1$ texts $t_0, \ldots, t_k$, text nodes are defined as follows: (1) the node labeled $t_0$ is the leftmost child of the node $x$, and (2) for $i, 1 \leq i \leq k$, the node labeled $t_i$ is the right sibling of the node $y_i$. Assume that each element node $x_i$ has $n_{r_i}$ element children, so $r = (\sum_{j=1}^{k} n_{r_j}) + k$.

Now, let us create text children of nodes $x_1, \ldots, x_k$ using the above method and consecutive groups of texts from $T$, i.e., for $x_1$ and its element children $y_1, \ldots, y_{n_{r_1}}$ we use the first $n_{r_1} + 1$ texts from $T$, and...
for \( x_2 \) and its element children \( y_2, \ldots, y_{n_{r_2}} \) we use the next \( n_{r_2} + 1 \) texts from \( T \), etc.

Clearly the tree \( \tau^{-1}(A) \) is acyclic and for \( D \in \text{Acyclic}(D), \tau^{-1}\nu(D) = D \).

4 ALGORITHMIC APPROACH

This section provides the algorithm to create an annotated tree for a fixed labeled tree \( D \), which may or may not be cyclic. There are two steps in the process of creating an annotated tree, respectively implemented by Algorithm 1 and Algorithm 2. After the first step, when parsing of \( D \) is performed, for each set of similar paths there will be an associated graph of annotated nodes, which may include (in case of a cyclic input document) a special annotated dummy symbol \( \$ \). The second step uses data created by the first step to create an annotated tree. We also show the Algorithm 3 for the inverse mapping for XML, which inputs an annotated tree for the XML document \( D \) and outputs \( D \).

4.1 Notations and Auxiliary Abstract

Data Types

Recall that \( x(n) \) denotes a node \( x \) labeled by \( n \) in \( D \), and \( X(n)[A] \) denotes an annotated node \( X \) labeled by \( n \) with the annotation \( A \). For the path \( p \in P \) and a node \( x \in D \), let \( p/x \) denotes the path \( p \) extended by /\( x \), and \( X0(p) \) denotes the so-called current node (which could be empty). Recall from Section 2.1 that \( p_x \) denotes the path \( p \) except its last element, and from Section 2.2 that \( \text{AnnotationSum}(X) \) denotes the sum of all annotations of the node \( X \). Let \( A(n) \) denote a list of \( \text{AnnotationSum}(X(0)(p_x)) - 1 \) occurrences of integer \( n \), i.e., \( k - 1 \) occurrences of \( n \), where \( k \) is the sum of all annotations of the current node for the path \( p_x \). For example, if \( \text{AnnotationSum}(X(0)(p_x)) = 1 \) is equal to four, then \( [A(5), 1] \) denotes the annotation \( [5,5,5,5,1] \). If \( X \) is a node or the dummy node then by \( \text{Inc}(X, 1) \) we denote the following modification of the annotation of \( X \): if \( i = 1 \) then the last annotation of \( X \) is incremented by one; if \( i = 0 \) then ",.0" is added after the last annotation of the dummy symbol. Let \( P \) be the set of equivalence classes of the similarity relation (see Definition 3) maintained by the Algorithm 1 (initially, it consists of the equivalence class for the path /\( x/0 \) of \( D \). For every equivalence class \( q \in P \), let \( G(q) \) be a pair consisting of (a graph with annotated nodes, a dummy node \( $ \)), and let \( S(p) \) denotes the annotation of the dummy node for the class \( [p] \).

We define several auxiliary notations and operations:

1. \( \text{int insertP}(path p) \) returns 0 if \( [p] \in P \); otherwise it inserts \( [p] \) into \( P \), sets \( G([p]) \) to be an empty graph and inserts dummy symbol, and returns 1;
2. \( \text{AnnotateNode insertG}(path p, label n, annotation A) \);
3. \( \text{node member}(path p, label n) \) returns the unique node \( X \) in \( G([p]) \) such that \( X \) is labeled with \( n \); (4) \( \text{addArc}(node X1, node X2, path p) \) adds a new arc connecting \( X1 \) and \( X2 \) in \( G([p]) \);
5. \( \text{annotation Ann}(path p) \) when the path \( p \) is of the form /\( root \) or \( S(X0(p_\uparrow)) \) is equal to 1 then return 1, otherwise return the annotation \( S(X0(p_\uparrow)) - 1 \) of 0's, 1; (6) \( \text{int reachableG}(node X1, node X2, path p) \) returns 1 if the node \( X2 \) is reachable from the node \( X2 \) in \( G([p]) \), otherwise it returns 0;
6. \( \text{sortG}(path p) \) performs a topological sort of \( G([p]) \);
7. \( \text{update}(path p) \) for each node \( X \) in \( G([p]) \) performs \( \text{Inc}(X, 0) \).

4.2 Mapping a Labeled Tree \( D \) to an Annotated Tree

This section presents Algorithms 1 and 2 which map a labeled tree \( D \), possibly cyclic, into an annotated tree, which in case of cycles adds dummy nodes labeled by \( \$ \) (thus algorithms in this section are designed for arbitrary trees, rather than for XML trees). Algorithm 1 performs a depth-first search traversal of the input tree, moving down and up. For the XML document \( D \), these actions would be triggered by entering the beginning of the element, i.e., /\( x \) and the end of the element, i.e., /\( x/0 \). The algorithm maintains the current path \( p_0 \) in \( D \), and for each path \( p \) in \( D \) it maintains the set \( P \) of paths and associated graphs \( G([p]) \). It also maintains the current annotated node \( X0(p) \) in \( G([p]) \) and the annotated dummy node.

The Algorithm 1 is initialized by setting the node \( x/0 \) to be the root of \( D \) and \( p_0 \) to be the path /\( x/0 \). Then it performs a loop moving down and up until it reaches the root node and there are no more un-visited children of the root, while maintaing the following two invariants: (1) The current node \( X0(p_\uparrow) \) is not null; (2) If the algorithm moves down to the node \( x \), such that the path /\( x/ \) already exists, then there exists a unique annotated node \( X \) in the graph \( G([p_\uparrow]) \) which has the same label as the label of \( x \). Once the algorithms complete their actions, the set \( P \) stores all equivalence classes of the similarity relation for \( D \).

Table 1 shows the trace of the execution of Algorithm 1 for the document \( D \) from Figure 5, using simplified notation. The current annotated node \( X0(p) \) (if any) is underlined. If the tuple \( \langle \text{path q, G(q)} \rangle \) has not changed, then it is not shown again. The action of going down to the node \( x \) is shown as /\( x \) and the action of going up to the node \( x \) is shown as /\( x/ \). Each \( q \in P \)
will represent a node of the annotated tree, and the annotated nodes from the graph \(G(q)\) will represent children of these nodes. Algorithm 2 inputs the output of the Algorithms 1 and produces the annotated tree.

**Algorithm 1**: Algorithm which maps a tree to a set \(P\) and associated graphs.

Require: \(x0 = \text{root of } D\), \(p0 = /x0\), \(G([p0])=(\text{graph and dummy node}; \text{both empty})\)

1: function TRAVERSE
2:   while true do
3:     if current is root and no more nodes then
4:       return;
5:   end if
6:   if moving down to \(x\) then
7:     \(p1 = p0/x\);
8:     if insertP(p1) then // \(p1\) is a new path
9:       \(X1 = \text{insertG}(p0, \text{label of } x, [A(0),1]);\)
10:      \(X0(p0)\) \(\rightarrow\) \(X1\);
11:     end if
12:   end if
13:   else// \(p1\) is already defined
14:     \(X1 = \text{memberG}(p0, \text{label of } x);\)
15:     if \(X0(p0) \neq 0 \text{ AND } X1 \neq X0(p0)\) then
16:       if check-if arc can be added
17:         if reachableG(X0(p), X1, p0) then
18:           // can add
19:           \(\text{Inc}(X1,1);\)
20:           update(p1);
21:           addArc(X0(p0), X1, p0);
22:         end if
23:       end if
24:     end else// needs a dummy node
25:     if \(S(p0) \neq 0\) then
26:       \(\text{Inc}(S(0));\)
27:     end if
28:     else
29:       \(S(p0) = S[1,2];\)
30:     end if
31:     end function
32:   end if
33:   \(X0(p0) = X1;\)
34:   \(\text{Inc}(X1,1);\)
35:   end if
36: end function
37: end function
38: end while
39: end function

Let \(n\) be the number of nodes in the input tree. The Algorithm 1 is based on DFS-traversal of a tree, which has \(O(n)\) time complexity, with the nested call (line 17) to the function \(\text{reachableG}(\cdot)\), which has \(O(|V| + |E|)\) time complexity, where \(|V|\) is the number of vertices in the graph passed as a parameter to the function, and \(|E|\) is the number of vertices in this graph. Therefore, time complexity of this algorithm is \(O(n^2)\). Similarly, Algorithm 2 has \(O(n^2)\) complexity.

### 4.3 Mapping an Annotated Tree to the Labeled Tree

Now we present the algorithm 3, which describes for an XML document a mapping reverse to the previously described mapping, i.e., it inputs an annotated tree (possibly with dummy nodes) and outputs the XML document. The following notations are used: (1) \(\text{ann}(n)\): first digit in the annotation of the node \(n\); (2) \(\text{chop}(n)\): remove the first digit in the annotation of \(n\) (always 0); (3) \(\text{dec}(n)\): decrement by one the first digit.

<table>
<thead>
<tr>
<th>Move</th>
<th>(p0)</th>
<th>(p1)</th>
<th>([p] = G([p]), \text{dummy})</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\beta b)</td>
<td>(a/b_1)</td>
<td>(a/b_1)</td>
<td>(A: B[1], 0)</td>
</tr>
<tr>
<td>(\beta c)</td>
<td>(a/b_1/c_1)</td>
<td>(a/b_1/c_1)</td>
<td>(B: C[1], 0)</td>
</tr>
<tr>
<td>(\beta d)</td>
<td>(a/b_1/d_1)</td>
<td>(a/b_1/d_1)</td>
<td>(B: C[1] \rightarrow D[1,1], 0)</td>
</tr>
<tr>
<td>(\beta a)</td>
<td>(a/b_1)</td>
<td>(\beta a)</td>
<td>(B: C[1,0] \rightarrow D[1,0], 0)</td>
</tr>
<tr>
<td>(\beta a)</td>
<td>(a/b_2)</td>
<td>(\beta a)</td>
<td>(B: C[1,0] \rightarrow D[1,1], 0)</td>
</tr>
<tr>
<td>(\beta b)</td>
<td>(a/b_2/d_2)</td>
<td>(a/b_2/d_2)</td>
<td>(B: C[1,0] \rightarrow D[1,1])</td>
</tr>
<tr>
<td>(\beta c)</td>
<td>(a/b_2/c_2)</td>
<td>(a/b_2/c_2)</td>
<td>(B: C[1,1] \rightarrow D[1,1,0])</td>
</tr>
<tr>
<td>(\beta a)</td>
<td>(a/b_2)</td>
<td>(\beta a)</td>
<td>(B: C[1,1,0] \rightarrow D[1,0])</td>
</tr>
</tbody>
</table>

Table 1: Trace of the execution of Algorithm 1.
Algorithm 3: Algorithm which inputs an annotated tree and outputs the XML document.

Require: Called for the root of the annotated tree
1: function RESTORE(Node c)
2:  | n = LC(c);
3:  | while n ≠ 0 do
4:  |  | if ann(n) > 0 then
5:  |  |  | if n is not a dummy node then
6:  |  |  |  | output < + label of n >;
7:  |  |  | end if
8:  |  | Restore(n);
9:  |  else
10:  |  | chop(n);
11:  |  | end if
12:  | n = RS(n);
13: end while
14: dec(c);
15: if c is not a dummy node then
16:  | output </ + label of c >;
17: end if
18: if ann(c) = 0 then
19:  | chop(c);
20: else
21:  | output < + label of n >;
22:  | Restore(c);
23: end if
24: end function

digit in the annotation of n (never 0); and (4) LC(n) and RS(n): respectively the leftmost child and right sibling of n. Given the annotated tree T, to restore the XML document, the following actions should be executed: output <root label>; Restore(root of T); output </root label> Algorithm 3 has $O(n^2)$ complexity.

5 EXPERIMENTAL RESULTS AND GENERAL ANALYSIS

This section describes an analysis of distribution of text elements in semi-structured data, and the definition of a compressibility measure for annotated followed by experimental results using an XML suite. Finally, it provides a general analysis of XML compression.

5.1 Quantification of Text Trees and Mutual Information

The use of annotated representation of a tree T implies the following hypothesis about the distribution of text elements in the tree: Each equivalence class in a tree T also defines a unique random variable in which the text children are sampled from. The hypothesis does not imply anything about the mutual information within the set of random variables; however, there are some implicit consequences as to how mutual information is dealt with. Specifically, mutual information is the amount of information that one random variable contains about another random variable; or the amount of reduction in uncertainty of one random variable due to the knowledge of the other. We define mutual information as $I(X|Y)$, the reduction in the uncertainty of X due to the knowledge of Y and consider two general cases in the transfer of information: (1) $I(\text{Child}|\text{Parent})$ - the transfer of information from a parent node to each of its children, or the transfer of information from a $N^th$ ancestor to each of its $N^th$ descendants; (2) $I(\text{Sibling},|\text{Sibling})_i | i < j$ - the transfer of information from a sibling node to each of its prior siblings. Although we cannot explicitly prove any general relationships of the two clauses above, we can extract information about the general use of semi-structured data and their affects on these relationships.

$I(\text{Child}|\text{Parent})$: With respect to most semi-structured data formats, e.g., XML and JSON, the text of non-leaf elements is mostly the whitespace data to make the semi-structured data human-readable, i.e., there is almost no relation between the information in any parent and the information of a leaf node. However, there is a clear relation between the information in any parent and the information of any non-leaf child. The data is only whitespace, and we can restrict the text alphabet to ASCII: SPC (0x20), TAB (0x09), LF (0x0a), VT (0x0b), FF (0x0c), and CR(0x0d). One caveat to this general statement, for example, are formatting tags in HTML, such as "display <b>bold</b> text". However, in the optimal case, and for semantic equivalence of the XML, we can functionally, and not statistically, relate the whitespace characters and the depth of the node, i.e., the depth multiplied by an indent (tab, sequence of whitespace, etc.) $I(\text{Sibling},|\text{Sibling})$: The relationship among siblings is slightly more complicated. If we consider the set of text elements for each equivalence class of leaf nodes, we can describe similarity between two sets using Statistical and Alphabetical Similarities, Functional Relations, and Temporal/Semantic and Structural Relations. Although statistical and alphabetical similarities form the basis of our hypothesis about the relationship of data among individual equivalence classes, they can also be used describe the relationship of data across equivalence classes. For example, two nodes: LastName and FirstName would be highly informationally related. However, if two tags consist of only free-formed English, the alphabets
may be similar, but the words, sentences, etc. may be different. Therefore, while there may be some statistical relationship among the character frequency, it quickly declines as we increase the degree of our statistics. Functional relations would describe the tags whose information is just some deterministic function of another tag. For example, a text tag and a sha-256 tag are functionally related. Temporal/Semantic and Structural Relations, while being a form of statistical similarity, describe the tags that have some temporal or sequential relation with its siblings. Consider this example: Despite their similarities, the tags whose information is just some deterministic statistics. Functional relations would describe the tags whose information is just some deterministic function of another tag. For example, a text tag and a sha-256 tag are functionally related. Temporal/Semantic and Structural Relations, while being a form of statistical similarity, describe the tags that have some temporal or sequential relation with its siblings. Consider this example: 

5.2 Experimental Results

Throughout this section we use the following notations: $D$ is a tree and $A$ belongs to the set of annotated trees $\text{Annotated}(D)$. Recall from Section 2.2 that this set uniquely represents $D$, therefore the description provided in this section does not depend on the choice of $A$. For any tree $T$ (annotated or not), by $|T|$ we denote the number of nodes in $T$. Let the width of $D$ at any level $i$ be denoted by $\text{width}(D, i)$ and let $\text{Ann}(X)$ denote the annotation list of the node $X \in \text{Nodes}(A)$ and let $|\text{Ann}(X)|$ denote the length of the annotation list. Finally let $X_1, X_2, \ldots, X_n$ be all nodes in $A$ at level $i$ (i.e., $\text{width}(A, i) = N_i$).

From the construction of an annotated tree, it follows that (see also Properties 2.4): (1) $\text{width}(D, i) = \sum_{k=1}^{N_i} \text{AnnotationSum}(X_k)$; and (2) In $A$, for any node $X$ and its child $Y$, $\text{AnnotationSum}(X) = |\text{Ann}(Y)|$.

Therefore, the larger the sum of all annotations of $X$, the longer the annotation list of $Y$. The implication of zeros appearing in the annotations of node $X$ is two-fold: (a) in $A$, they shorten the length of the annotation list; (b) in $D$, $Y$ does not appear as a child of $X$. If $D$ was “completely regular” and there were no missing children, then there would be no 0s on the annotation lists. From our experiments, it follows that a leaf annotation lists are usually very long, while for the leaf’s ascendants, annotations they are getting progressively shorter. To analyze this phenomenon consider a leaf node $X$ in $A$; its parent $Y$, and $Y$’s parent $Z$ (grandparent of $X$). Since $\text{AnnotationSum}(Y) = |\text{Ann}(X)|$, the annotation list $\text{Ann}(X)$ must have been increased because of the structure of $D$, specifically $Y$ has “very often” appeared as a child of $Z$, likely multiple times (resulting in annotations greater than 1). On the other hand, many 0s appearing in $\text{Ann}(X)$ indicates that $X$ has “very rarely” appeared as a child of $Y$.

Let us now consider the compressibility measure, which measures the cost of storing the annotated tree $A$ compared to the cost of storing the original document $D$. Our definition is to be implementation-independent and data are not compressed using a backend compressor. As discussed before, in general $A$ has fewer nodes than $D$, but there is an additional cost of storing annotation lists. Let $C$ be the cost of storing a single information, such as a single integer annotation or a node label. Therefore, the storage cost of a single tree node is $3 \times C$, and the cost of storing an annotation list of length $L$ is $L \times C$. The total storage cost of $D$ is equal to the cost of storing all nodes in $A$, including all annotation lists over the cost of storing all nodes in $D$, resulting in $\sum_{X \in D} |\text{Ann}(X)| \times C$.

Performing a few simplifications, gives the following formula (independent of the cost $C$):

Definition 13. The compressibility measure is defined as follows:

$$\mu_c(D) = \frac{|A| + \sum_{X \in A} |\text{Ann}(X)|}{\sum_{X \in D} |\text{Ann}(X)|}$$

In this section we provide experimental results showing values of the measure from Definition 13 applied to a suite of XML files. Characteristics of these files are shown in the first four columns of Table 2, where $V(N)$ denotes the value $N \times 10^N$. Size is the size of file in Bytes, $E:A$ denotes the number of elements and attributes, $AC$ denotes the number
Algorithm 4: Annotated Process.

1: function ENCODE(AnnotatedTree A, File file)
2:  write(encode(A.schema), file.schema)
3:  DepthFirstIterator dfs
4:  // For each node, its annotation list.
5:  for (dfs = A.iterator(); dfs.hasNext();)
6:    AnnotatedNode node = dfs.next()
7:    write(node.annotationList.length(), file.annot).
8:    write(node.textContainer.length(), file.text).
9:  end for
10: // For each node write its text container.
11: for (dfs = A.iterator(); dfs.hasNext();)
12:  AnnotatedNode node = dfs.next()
13:  write(node.textContainer, file.text).
14: end for
15: end function

Figure 10: Compression of entire XML document with single instances of Vanilla Compressors.

Table 2: Overview of XML Test Suite and Results of Testing.

<table>
<thead>
<tr>
<th>XML File</th>
<th>Size</th>
<th>E.A</th>
<th>AC</th>
<th>AT</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>1gig</td>
<td>1.17(9)</td>
<td>1.6(7)</td>
<td>3.8(6)</td>
<td>2.0(8)</td>
<td>680</td>
</tr>
<tr>
<td>BaseBall</td>
<td>6.72(5)</td>
<td>2.8(5)</td>
<td>0.0</td>
<td>6.6(5)</td>
<td>47</td>
</tr>
<tr>
<td>enwikibooks</td>
<td>x</td>
<td>5.36(5)</td>
<td>4.9(5)</td>
<td>6.3(6)</td>
<td>29</td>
</tr>
<tr>
<td>enwikibooks</td>
<td>x</td>
<td>1.84(9)</td>
<td>1.85(8)</td>
<td>2.59(9)</td>
<td>39</td>
</tr>
<tr>
<td>lineitem</td>
<td>3.22(7)</td>
<td>1.02(6)</td>
<td>3.3</td>
<td>9.6(6)</td>
<td>19</td>
</tr>
<tr>
<td>UniProt</td>
<td>1.15(8)</td>
<td>9.86(5)</td>
<td>1.44(9)</td>
<td>1.05(8)</td>
<td>217</td>
</tr>
</tbody>
</table>

of annotations, AT denotes the number of nodes in the annotation tree, and C denotes the compressibility measure. The files are 1gig.xml (a randomly generated XML file, using xmlgen (xmlgen, 2013)), base-
ball.xml (Baseball.xml, 2013), enwikibooks.xml and lineitem.xml from the Wratilavslavia corpus (Corpus, 2013), and uniprot_sprot (Consortium, 2013). This suite has been chosen because XML files included there have an ability to represent specific extremes of semi-structured data. For example, enwiki-latest.xml, the current revision of English Wikipedia, while being a very large document, encompasses two extremes: the distribution of character data is very non-uniform (i.e., the majority of the data falls within one node) and that path is predominantly free-formed English. Conversely, uniprot_sprot.xml is a highly uniform XML file (i.e., the data is evenly distributed), and the file is predominantly markup. The file 1gig.xml has the property that the subtree entropy is extremely low (subtrees are quite similar); however, each subtree differs by a parent node (for example, /a/b/d/e/f vs. /a/z/d/e/f ). The file lineitem.xml, has the property that it is an incredibly regular tree (few missing nodes), and in addition, has a nice mixture of text and numeric data. The file enwikibooks.xml is quite structurally similar to enwiki-latest.xml but is a fraction of its size. Finally, baseball.xml is an extremely irregular XML file. The last column of the Table 2 shows the test results, specifically values of the compressibility measures (see Definition 13) for the six XML file.

Table 2: Overview of XML Test Suite and Results of Testing.

5.2.1 Analysis of XML Compression

In measuring the compressibility of the annotated transform, each XML file was transferred into three separate files: (1) the annotated tree (a list of strings encoded in depth-first ordering); (2) the annotations (written in depth-first ordering); and (3) the text containers (written in depth-first ordering), see Algorithm 4. Consequently, this transform does nothing but to act as a pre-processor for other text compressors.

Ignoring Kolmogorov complexity (and Kol-
mogorov compressors), we assume the XML data to be distributed according to some random variable (or set of random variables). Therefore, the ideal way to analyze the compression benefits induced by the annotated transform would be to analyze how much benefit, with respect to the absolute lower bound, was obtained. However, since it is nearly impossible to calculate the exact entropy of the XML sources, the next ideal step would be to approximate the entropy using a lossless compression algorithm, which would be infeasible for the analysis of the proposed transform. Analyzing the percentage increase in compression does not take into consideration how substantial a percent decrease in size is, but it will be used as the base metric of comparison. For this analysis, a series of different compressors were used, see Figure10. LZ77-based (Ziv and Lempel, 2006) compressors, GZIP1 and XZ2, BWT-based (Burrows and Wheeler, 1994), BZIP23, Predic-

1 gzip -9 FILE
2 xz -9 -e FILE
3 bzip2 -9 FILE
Figure 11: Compression of Annotated Tree (the number of bytes to represent the XML Syntax/Structure) over Markup Density.

Figure 12: Compression of Annotated Transform (collection of Annotated Tree, Text Container, and Schema Tree) over Compression of XML Data.

Finally, Figure 12 plots the compression ratio of the size of the annotated tree over the compression ratio of the XML document shown in Figure 10. The first noticeable feature of Figure 12 is the fact that paq8pxd and PPMonster compress the data much better as vanilla compressors than with the annotated transform for the smaller XML files. Since the files are so small, these compressors can often build a model of the entire document, allowing those compressors to compress each tag-name, and the XML markup, quite compactly. In addition, by merging all of the text-containers into one compressible document, the data among container boundaries will often harm the initial compression of the subsequent container (the internal models have to adapt). The next general trend shown in Figure 12 is that the more markup-dense XML documents receive the more optimal compression ratios, whereas the more content-dense XML documents, only receive slight improvements for the larger XML documents. With respect to enlatest and enwikibooks, the majority of text is free-formed English, e.g., each \text{tag} contains a substantial amount of text data (all of the content you would see on a Wikipedia page). If some nodes text data were of significant size, only the compressors that incorporate a very large scope of the data would be able to exploit the tag-to-tag redundancy, otherwise, it would only be able to exploit the redundancy local to that tag, and the data at the boundary of two tags. From this, we can infer that the scope of compression is a necessary and sufficient factor in the performance of compression shown in Figure 10 (i.e., because a larger scope allows better compression of the XML syntax and the semantic/temporal relations among subtrees). Another factor may be attributed to the fact that the lower bound of lossless compression for these documents is “close” to the obtained compression ratios.

6 CONCLUSIONS AND FUTURE WORK

This paper showed that annotated trees form a faithful representation of the trees, and so the XSAQCT compression process is lossless. The formal approach and
specific algorithms have both been provided. Besides the formal and algorithmic approaches, experiments showed that the annotated tree compressibility, without using any backend compressors is high, on average approximately 0.4. Finally, a general analysis and results of testing of compression of entire XML document with single instances of vanilla compressors, compression of annotated tree over markup density, and compression of annotated transform over compression of XML data were provided, showing the usefulness of the annotated tree approach.

Simple queries, such as finding all children of a given node can be efficiently evaluated using the annotated trees. Our future work will extend queries to the subset of XPath expressions known as the core XPath as defined in (Gottlob et al., 2005), as well as more sophisticated navigational queries, e.g. asking for the $j$-th level-ancestor of $u$.

ACKNOWLEDGEMENTS

The work of the first and third authors are partially supported by the NSERC RGPIN grant and NSERC CSG-M (Canada Graduate Scholarship-Masters) grant respectively.

REFERENCES


