Annotated Trees and their Applications to XML Compression

Tomasz Müldner[1], Jan Krzysztof Miziołek[2] and Tyler Corbin[1]
[1] *Jodrey School of Computer Science, Acadia University, Wolfville, B4P 2A9 NS, Canada*
[2] *Faculty 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

Müldner T., Miziołek J. and Corbin T..
Annotated Trees and their Applications to XML Compression.
DOI: 10.5220/0004839900270039
In *Proceedings of the 10th International Conference on Web Information Systems and Technologies* (WEBIST-2014), pages 27-39
ISBN: 978-989-758-023-9

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 Σ be an alphabet, called the label alphabet. A **labeled tree** is an ordered tree with nodes labeled by strings from Σ^* 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 $\mathbf{Label}(\mathbf{x})$ denote the label of the node x, $\mathbf{Height}(\mathbf{D})$ denote the height of D, $\mathbf{Nodes}(\mathbf{D})$ denote the set of all nodes of D, and $\mathbf{D_i}$ be the tree consisting of all nodes and edges in D at levels $1, \ldots, i, where 1 \leq i \leq height(D)$. Example of a tree is shown in Figure 1.

Definition 2. *A **path** in a tree is defined to be of the form $/x_1/x_2, \ldots, /x_k$, 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 $\mathbf{Paths}(\mathbf{D})$ to denote the set of all paths in D. For the path $p = /x_1/x_2, \ldots, /x_k$, let $\mathbf{Length}(\mathbf{p})$ be the number of nodes in the path p,

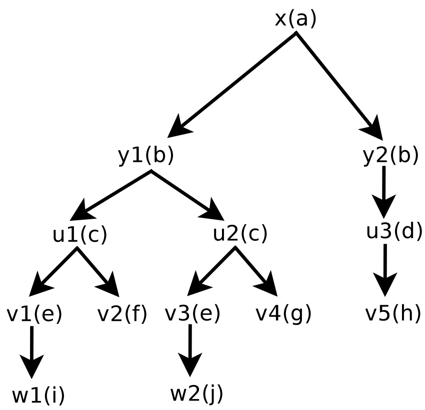

Figure 1: Example of a tree D.

$\mathbf{Last}(\mathbf{p})$ be the last element x_k, $\mathbf{Label}(\mathbf{p}) = Label(x_k)$ be the label of p, and for $Length(p) > 1$, let $\mathbf{p}\!\downarrow$ be the path p except the last element.

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

The similarity relation is an equivalence relation and we denote by $[\![\mathbf{p}]\!]$ the equivalence class of p, by $\mathbf{Similar}(\mathbf{D})$ the quotient set of this relation, i.e., the set of all different sets of similar paths. For $p \in Paths(D)$ let $\mathbf{Length}([\![\mathbf{p}]\!]) = Length(p)$ be the length of the equivalence class, $\mathbf{Label}([\![\mathbf{p}]\!]) = Label(p)$ 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 $q \in Similar(D)$ let $|\mathbf{q}|$ be the number of paths in q, and $\mathbf{Last}(\mathbf{q})$ be the sequence of nodes that are last elements of all paths in this class, *ordered* from left to right. Clearly, $|Last(q)| = |q|$.

In Figure 1, $Similar(D) = \{[\![/x]\!], [\![/x/y_1]\!], [\![/x/y_1/u_1]\!], [\![/x/y_1/u_3]\!], [\![/x/y_1/u_3/v_5]\!], [\![/x/y_1/u_1/v_1]\!], [\![/x/y_1/u_1/v_2]\!], [\![/x/y_1/u_2/v_4]\!], [\![/x/y_1/u_1/v_1/w_1]\!], [\![/x/y_1/u_1/v_3/w_2]\!]\}$ and for $p = /x/y_1, [\![p]\!] = \{/x/y_1, /x/y_2\}$; $Length([\![p]\!]) = 2$, $Label(p) = b$, $Last([\![p]\!]) = <y_1, y_2>$, and $|[\![p]\!]| = 2$.

Definition 4. *A partial relation $\prec$ in the set $Similar(D)$ is defined as follows: $[\![p_1]\!] \prec [\![p_2]\!]$ iff $Length([\![p_1]\!]) = Length[\![p_2]\!]) > 1, [\![p_1\!\downarrow]\!] = [\![p_2\!\downarrow]\!], Label([\![p_1]\!])$ is different from $Label([\![p_2]\!])$, and there exist paths $p1 \in [\![p_1]\!]$ and $p2 \in [\![p_2]\!]$ such that the node $Last(p1)$ is the left sibling of the node $Last(p2)$. The tree D has a **cycle** if there exist $q_1, q_2 \in Similar(D)$ 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 $y_1(b)$ by b_1. In this notation, an equivalence class $[\![p]\!]$ will be denoted using the label $Label([\![p]\!])$, 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 permuted to create an annotated tree with nodes represented by equivalence classes of the similarity relation. In the worst case, if for each $q \in Similar(D)$, $|q| = 1$ 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 $q \in Similar(D)$, $|q| \gg 1$ 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 such 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 g-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 ∈ Nodes(A) except the root has its nodes annotated; and (3) for every node H ∈ Nodes(A) and for every child G of H there exists exactly one node n ∈ 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., $\mathbf{X}(\mathbf{n})[\alpha_1, \ldots, \alpha_j]$ 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 $\mathbf{AnnotationSum}(\mathbf{X}) = \sum_{i=1}^{j} \alpha_i$ 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.

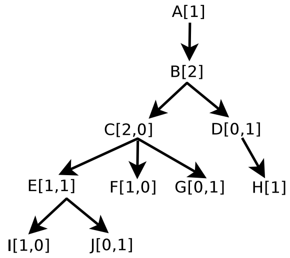

Figure 2: Annotated tree.

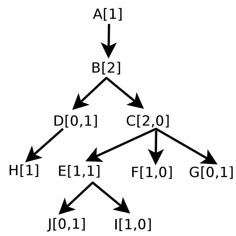

Figure 3: Another annotated tree.

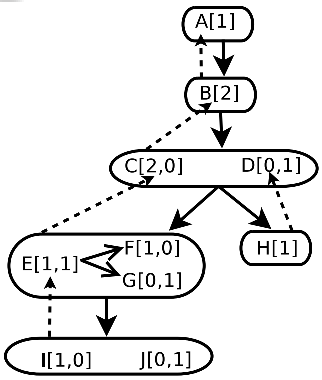

Figure 4: Example of a g-tree.

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 h 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* $<m_1,\ldots,m_k>$ *(in left-to-right order) in* D_2 *at level* $i, k = j$*, and for* $1 \leq r \leq k$*, nodes* n_r *and* m_r *have the same degree and label.*

2.4 Mapping Trees to Sets of Annotated Trees

We define the mapping $\tau : Acyclic \rightarrow Annotated{-}G$ 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 $[\![p]\!](Label(p))[\alpha_1,\ldots,\alpha_j]$ or using a simplified notation $Label(p)[\alpha_1,\ldots,\alpha_j]$. If x is a node in D, which is not a root, then by $\mathbf{p_x}$ we denote a (unique) path $p \in Paths(D)$ which ends in x.

Definition 7. *Definition of mapping* τ : $Acyclic \rightarrow Annotated{-}G$*. Let* $D \in Acyclic$*.*

- *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* $[\![/r]\!](Label(r))$*.*
  2. *For any level* $i > 1$ *of* D *and equivalence class* $q \in Similar(D)$ *of length* $i-1$ *let* $N_{q,\,i}$ *denote the set of nodes* x *in* D *at level* i *such that* $[\![p_x\!\downarrow]\!] \in q$*. Clearly, the sets* $\{N_{q,\,i} : q \in Similar(D)\}$ *form a disjoint coverage of the set of all nodes in* D *at level i. Each set* $N_{q,\,i}$ *is mapped by* τ *to the single graph* $G \in Nodes(\tau(D))$ *at the level i;* $G = \{[\![p_x]\!](Label(p_x)\}$ *where the source of* G *is the node* q*. For any graph* $G \in Nodes(\tau(D))$ *and two nodes* $q_1, q_2 \in G$*, there is an edge* $q_1 \Longrightarrow q_2$ *in* G *iff* $q_1 \prec q_2$*, (see Definition 4). Since* D *is assumed to have no cycles,* G *is acyclic. The node* $H \in Nodes(\tau(D))$ *is the parent of the node* G *if the source of* G *belongs to the dag* $Nodes(H)$*. 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* $i, 1 \leq i < Height(D)$ *and for any equivalence class* q *of length i,* $|Last(q)| = AnnotationSum(q)$*, 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 the level* $i+1$ *with the source* X*, and a node* $Y \in Nodes(G)$*. Let* $Y = [\![p]\!](r)[\alpha_1,\ldots,\alpha_j], X = [\![p\!\downarrow]\!](m)$ *and* $k = AnnotationSum(X)$*. First, we set the number* j *of annotations in* Y *to be* k*. From the inductive assumption it follows that the sequence* $Last([\![p\!\downarrow]\!])$ *has* k *elements* $s_1,\ldots,s_k$*. For* $1 \leq j \leq k$*, we define* α_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* $|Last([\![p]\!])| = AnnotationSum(Y)$*.*

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

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

For the tree D from Figure 1, 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* $\tau : Acyclic \rightarrow Trees_\tau$ *is injective.*

Proof. We will define the mapping τ^{-1} : $Trees_\tau \rightarrow Acyclic$ 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 $A \in Trees_\tau$. Then τ^{-1} is defined inductively on levels i of A:

1. for $i = 1$, let the root of A be a graph G consisting of a single node $X(a)[1]$). Then $\tau^{-1}(G) = x(a)$
2. Assume that for all levels $i, 1 < i < Height(A), D_i = \tau^{-1}(A)_i \in Acyclic$. We define $\tau^{-1} : A_{i+1} \rightarrow D_{i+1}$. Let G be a dag in A at level $i+1$, and $Nodes(G) = \{Y_1,\cdots,Y_m\}$, where for $j, 1 \leq j \leq m$, $Y_j = [\![q_j]\!](r_j)[\alpha_1^j,\ldots,\alpha_k^j]$, the node $X = [\![p]\!](m)$ be the source of of G , $k = AnnotationSum(X)$, and $Last([\![p]\!])$ 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 the 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 the level $i+1$ consisting of α_h^j nodes in D labeled by r_j. We define a partial order between these sets as follows: for j_1 and $j_2, 1 \leq j_1, j_2 \leq m$, $N_{h,\,j_1} \prec N_{h,\,j_1}$ iff in G there is

an arc between $[\![q_{j_1}]\!]$ and $[\![q_{j_2}]\!]$. Finally, children of s_h 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 $N_{h,\,j_1}$ and $N_{h,\,j_2}$ all nodes from $N_{h,\,j_1}$ appear to the left of all nodes in N_{h,j_2} iff $N_{h,j_1} \prec N_{h,j_2}$; 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 $D \in Acyclic, \tau^{-1}\circ\tau(D) = D$, for $A \in Trees_\tau$, $\tau^{-1}\circ\tau(A) = A$. $\square$

Next, we define the mapping $\gamma : Trees_\tau \rightarrow 2^{Annotated-G}$. Let $TS(G)$ be the set of all topological sortings of a dag G, and for $G_1,\cdots,G_n$ $P(G_1,\ldots,G_n)$ be a Cartesian product $\times_{i=1}^{n} TS(G_i)$. If $<g_1,\ldots,g_n> \in P(G_1,\ldots,G_n)$ then each graph g_i represents a topologically sorted graph G_i.

Definition 8. *Mapping γ is defined as follows: For $T \in Trees_\tau$, $\gamma(T) = \{\gamma_{<g_1,\ldots,g_n>}(T) : <g_1,\ldots,g_n> \in P(G_1,\ldots,G_n)\}$ where $\gamma_{<g_1,\ldots,g_n>}(T)$ is the g-tree T 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 $\mathbf{Trees}_\gamma$ denote the image $\gamma(Trees_\tau) \subset 2^{Annotated-G}$.

Theorem 2. *The mapping $\gamma : Trees_\tau \rightarrow Trees_\gamma$ is injective.*

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

It is easy to see that each g-tree $T \in Trees_\gamma$ is in one-to-one correspondence with an annotated tree. Since $\tau : Acyclic \leftrightarrow Trees_\tau$ and $\gamma : Trees_\tau \leftrightarrow Trees_\gamma$, each tree D can be mapped to the set of annotated trees, denoted by **Annotated(D)** and defined by the composition of τ and γ. It is easy to see that every annotated tree from the set $Annotated(D)$ represents

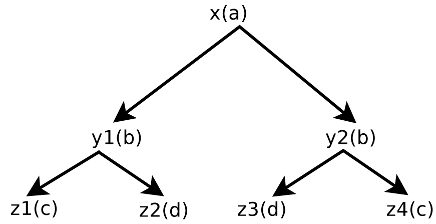

Figure 5: Cyclic Tree.

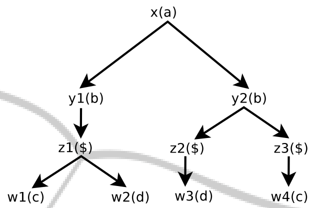

Figure 6: Acyclic tree with dummy nodes.

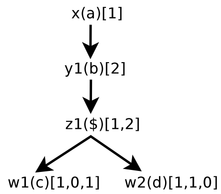

Figure 7: The annotated tree with dummy nodes.

D. For example, both annotated trees shown in Figure 2 and Figure 3 belong to the set $Annotated(D)$ that uniquely represents the tree D.

If there is a cycle in D, then we map D to an acyclic tree with the so-called dummy nodes, denoted by \$. After adding a dummy node to a cyclic document D in Figure 5, this tree will be acyclic, see Figure 6, and it can be mapped to the annotated tree shown in Figure 7. We do not formally prove that the mapping from the tree with cycles to a tree with the dummy nodes is injective but provide Algorithm 1 in Section 4 showing how cycles can be removed.

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 Δ be an alphabet, called the text alphabet, its elements are \0 terminated strings. A **text tree** is a tree with two kinds of nodes; element nodes labeled by strings from Σ^* (see Definition 1) and text nodes labeled by strings from Δ^* 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-

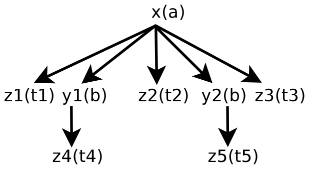

Figure 8: Text tree.

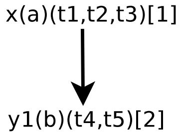

Figure 9: The annotated text tree.

dices if needed), see Figure 8. By **TextTrees** we denote the set of all text trees, and by **AcyclicTextTrees** we denote the set of all acyclic text trees. A text tree can be used to represent an XML document with text values represented by labels of text nodes. Text nodes may or may not be present and now we define complete text trees corresponding to the full-mixed content of XML documents, see (Müldner et al., 2012).

Definition 10. *An element leaf node in the text tree is the element node that has no element child. A text tree is called* ***complete*** *if every non-leaf element node has the left and the right text sibling, and every element leaf node has exactly one text child node.*

The text tree from Figure 8 is complete; it would not be complete if any of the text nodes were missing. Note that in XSAQCT when the input XML document D is parsed then for any missing text node, a text node labeled by an empty text (consisting of $\backslash 0$ only) is added. To support a unique representation of an XML document using this technique, added text nodes are removed while D is restored. In what follows, we assume that *text trees are complete*.

3.2 Compressed Representation of Text Trees

Definition 11. *For a node X in the annotated tree with m children $Y_1,\ldots,Y_m$, $m \geq 0$ let* $\mathbf{Number(X)}$ $= (\sum_{j=1}^{m} AnnotationSum(Y_j)) + AnnotationSum(X)$. *An* ***annotated text tree*** *is an annotated tree with nodes additionally labeled by concatenations of strings from Δ^*, called text labels, such that a text label T of a node $X(a)(T)$ is equal to the concatenation of $Number(X)$ text labels.*

Example of an annotated text tree is shown in Figure 9. By **AcyclicAnnTextTrees** we denote the set of all acyclic annotated text trees.

3.3 Mapping Text Elements

For every equivalence class $[\![p]\!]$, labels of text nodes that are children of element nodes (in left to right order) from the sequence $Last([\![p]\!])$ will be concatenated into a single text label of the node in the annotated tree that corresponds to this equivalence class.

For example, the text tree shown in Figure 8 will be mapped to the annotated text tree from Figure 9, where t_1,t_2 denotes a concatenation of texts. The reason for mapping text nodes this way is that for querying of the XML documents, the query of the form $/X$ returns the concatenated texts appearing in this path.

Definition 12. *Let $D \in AcyclicTextTrees$ and let A be the image of D under τ (see Definition 7) as if there was no text nodes in D. Now, for every node $X \in A$, we will define its text label T such that the number of texts concatenated in T will be equal to $Number(X)$. Let $\tau^{-1}(X)$, as defined the proof of Theorem 1, be the sequence $x_1,\ldots,x_k$ of element nodes, where $k = AnnotationSum(X)$ and let us 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, $Number(X) = AnnotationSum(X) = k$.*
2. *X is a not leaf, and it has m children, $Y_1,\ldots,Y_m$. Thus, there are $Number(X)$ 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 $X \in Nodes(A)$ and let $k = Number(X)$. The image of X under the reverse mapping is the sequence of element nodes $x_1,...,x_k$. We consider two cases:

1. $X(T)$ is a leaf. Then $k = Number(X) = AnnotationSum(X)$ and T is a concatenation of k texts $t_1,\ldots,t_k$. For every $i, 1 \leq i \leq k$, 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 $Y_1(T_1),\ldots,Y_m(T_m)$, where T is a concatenation of $r = (\sum_{j=1}^{m} AnnotationSum(Y_j)) + AnnotationSum(X)$ texts $s_1,\ldots,s_r$. Given a node x in a tree from *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,

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 $D \in Acyclic(D), \tau^{-1} \circ \tau(D) = D$. □

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 $\mathbf{p}/\mathbf{x}$ denotes the path p extended by $/x$, and $\mathbf{X0}(\mathbf{p})$ denotes the so-called *current node* (which could be empty). Recall from Section 2.1 that $p\downarrow$ denotes the path p except its last element, and from Section 2.2 that $AnnotationSum(X)$ denotes the sum of all annotations of the node X. Let $\mathbf{A}(\mathbf{n})$ denote a list of $AnnotationSum(X0(p\downarrow)) - 1$ 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 $p\downarrow$. For example, if $AnnotationSum(X0(p\downarrow)) - 1$ 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 $\mathbf{Inc}(\mathbf{X}, \mathbf{i})$ 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 $\mathcal{P}$ 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 /root of D). For every equivalence class $q \in \mathcal{P}$, let $G(q)$ be a pair consisting of (a graph with annotated nodes, a dummy node $), and let $(p) denotes the annotation of the dummy node for the class $[\![p]\!]$. Now, we define several auxiliary notations and operations: (1) `int insertP(path p)` returns 0 if $[\![p]\!] \in \mathcal{P}$; otherwise it inserts $[\![p]\!]$ into $\mathcal{P}$, sets $G([\![p]\!])$ to be an empty graph and empty dummy symbol, and returns 1; (2) `Annotatednode insertG(path p, label n, annotation A)` ; (3) `node memberG(path p, label n)` returns the unique node X in $G([\![p]\!])$ such that X is labeled with n; (4) `addArc(node X1, node X2, path p)` adds a new arc connecting $X1$ and $X2$ in $G([\![p]\!])$; (5) `annotation Ann(path p)` if the path p is of the form \root or S(X0(p↓)) is equal to 1 then return 1, otherwise return the annotation S(X0(p↓))-1 of 0's, 1; (6) `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; (7) `sortG(path p)` performs a topological sort of $G([\![p]\!])$; (8) `update(path p)` for each node X in $G([\![p]\!])$ performs $Inc(X, 0)$.

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. The algorithm maintains the current path $p0$ in D, and for each path p in D it maintains the set $\mathcal{P}$ 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 $x0$ to be the root of D and $p0$ to be the path $/x0$. 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 $X0(p0\downarrow)$ is not null; (2) If the algorithm moves down to the node x, such that the path p/x already exists, then there exists a unique annotated node X in the graph $G([\![p0]\!])$ which has the same label as the label of x. Once the algorithms complete their actions, the set $\mathcal{P}$ 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 <u>underlined</u>. If the tuple (path q, G(q)) 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 $q \in \mathcal{P}$

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 $\mathcal{P}$ and associated graphs.

```
Require: x0 = root of D, p0 = /x0, G([[p0]])=(graph and
    dummy node; 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 = insertG(p0, label of x, [A(0),1]);
10:         if X0(p0) ≠ ∅ then
11:           addArc(X0(p0), X1, p0);
12:         end if
13:       else// p1 is already defined
14:         X1 = memberG(p0,label of x);
15:         if X0(p0) ≠ ∅ AND X1 ≠ X0(p0) then
16:           // check if arc can be added
17:           if !reachableG(X0(p), X1, p0) then
18:             // can add
19:             Inc(X1,1);
20:             update(p1);
21:             addArc(X0(p0), X1, p0);
22:             if $(p0) ≠ ∅ then
23:               Inc($ ,0);
24:             end if
25:           else// needs a dummy node
26:             if $(p0) ≠ ∅ then
27:               Inc($,1);
28:             else
29:               $(p0) = $[A(1), 2];
30:             end if
31:             update(p0);
32:             Inc(X1,1);
33:           end if
34:         else
35:           X0(p0) = X1;
36:           Inc(X1,1);
37:           update(p1);
38:           if $(p1) ≠ ∅ then
39:             Inc($ ,0); Inc($ ,1);
40:           end if
41:         end if
42:       end if
43:       X0(p0) = X1;
44:       p0 = p1;
45:     else// moving up to the node x
46:       X0(p0) = ∅;
47:       p0 = p0↓;
48:     end if
49:   end while
50: 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

Table 1: Trace of the execution of Algorithm 1.

Move	p0	p1	$[\![p]\!]$: G($[\![p]\!]$), dummy
	/a		A: ∅
↓b1	/a/b1	/a/b1	A: B[1], ∅
↓c1	/a/b1/c1	/a/b1/c1	B: C[1], ∅
↑b1	/a/b	/a/b1/c1	
↓d1	/a/b1/d1	/a/b1/d1	B: C[1]→D[1], ∅
↑b1	/a/b1/		
↑a	/a		B:, C[1]→D[1], ∅
↓b2	/a/b2	/a/b2	A:Y(b)[2], ∅
			B: C[1,0]→D[1,0], ∅
↓d2	/a/b2/d2	/a/b2/d2	B: C[1,0]→D[1,1], ∅
↑b2	/a/b2		
↓c2	/a/b2/c2	/a/b2/c2	B: C[1,0]→D[1,1]
			B: C[1,0,1]→D[1,1,0]
			$[1,2]
			$[1,2]
↑b2	/a/b2		
↑a	/a		

Algorithm 2: Algorithm which maps a set $\mathcal{P}$ and associated graphs to an annotated tree.

```
Require: Initially p = \root, X = root(label of the
    root of D)[1]
  function FINALIZE(Path p, Node X)
    if $(p) ≠ ∅ then
      make the node $(p) a child of X;
      X = $(p);
    end if
    Node R = sort(p); // now G(p) is a chain
    make annotated nodes in G(p) children of X
    for every node X1 in G(p) do
      if G(p) ≠ ∅ then
        Finalize(p/(label of X1), X1);
      end if
    end for
  end function
```

(line 17) to the function *reachableG*(), 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) ann(n): first digit in the annotation of the node n; (2) chop(n): remove the first digit in the annotation of n (always 0); (3) dec(n): decrement by one the first

Algorithm 3: Algorithm which inputs an annotated tree and outputs the XML document.

```
Require: Called for the root of the annotated tree
function RESTORE(Node c)
    n = LC(c);
    while n ≠ ∅ do
        if ann(n) > 0 then
            if n is not a dummy node then
                output < + label of n + >;
            end if
            Restore(n);
        else
            chop(n);
        end if
        n = RS(n);
    end while
    dec(c);
    if c is not a dummy node then
        output </ + label of c + >;
    end if
    if ann(c) = 0 then
        chop(c);
    else
        output < + label of n + >;
        Restore(c);
    end if
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 $\mathbf{I}(\mathbf{X}|\mathbf{Y})$, the reduction in the uncertainty of X due to the knowledge of Y and consider two general cases in the transfer of information: (1) $[I(\text{Child}|\text{Parent}^N)]$ - 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) $[I(\text{Sibling}_i|\text{Sibling}_j)|\forall j < i]$ - 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.

$\mathbf{I(Child|Parent)}$: 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 bold 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.)

$\mathbf{I(Sibling_i|Sibling_j)}$: 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:

```
<Questions>
<Q>How many bits to an octet?</Q>
<A>There are 8-bits to an octet.</A>
<Q>How many bits to a byte?</Q>
<A>Generally speaking, 8.</A>
</Questions>
```

. Although functional relations can be exploited, they require the relationship to be known before compression, i.e., there is an underlying schema to the semi-structured data. Thus, these type of similarities are often not considered for general-purpose compression. With respect to non-leaf children, we expect the data to be quite statistically (and structurally) similar, because the data of a sibling set would be consistently formatted for human readability (or lack thereof). However, with leaf children, the annotative representation of a text tree will only exploit the sequence of character data local to an equivalence class and will not consider the sequence of character data local to some subtree (as shown above). Therefore, no compression algorithm will be able to use the knowledge of "How many bits to an octet" to compress the information of "There are 8-bits to an octet.". However, to compress "How many bits to a byte", we can use the knowledge of previously asking "How many bits to an octet". Therefore, semantically related text of the same vertex would have a high statistical similarity, whereas semantically related text of the same subtree will not be considered for compression.

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 $Annotated(D)$. 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 $|\mathbf{T}|$ we denote the number of nodes in T. Let the width of D at any level i be denoted by $\mathbf{width}(\mathbf{D},\mathbf{i})$ and let $\mathbf{Ann}(\mathbf{X})$ denote the annotation list of the node $X \in Nodes(A)$ and let $|\mathbf{Ann}(\mathbf{X})|$ denote the length of the annotated list. Finally let $X_1, X_2, \ldots, X_{N_i}$ be all nodes in A at level i (i.e., $width(A,i) = N_i$).

From the construction of an annotated tree, it follows that (see also Properties 2.4): (1) $width(D,i) = \sum_{k=1}^{N_i} AnnotationSum(X_k)$; and (2) In A, for any node X and its child Y, $AnnotationSum(X) = |Ann(Y)|$. 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 $AnnotationSum(Y) = |Ann(X)|$, the annotation list $Ann(X)$ 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 $Ann(X)$ 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 $\frac{(3\times C\times|A|)+(\sum_{X\in A}|Ann(X)|)\times C}{3\times C\times|D|}$. Performing a few simplifications, gives the following formula (independent of the cost C):

Definition 13. *The compressibility measure is defined as follows:*

$$\mu_c(D) = \frac{|A|+(\sum_{X\in A}|Ann(X)|)/3}{|D|}.$$

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 $V{*}10^N$, Size is the size of file in Bytes, E:A denotes the number of elements and attributes, AC denotes the number

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

XML File	Size	E:A	AC	AT	C
1gig	1.17(9)	1.6(7):3.83(6)	2.05(8)	680	0.41
BaseBall	6.72(5)	2.8(5):0	6.6(5)	47	0.78
enw.books	1.56(8)	5.3(6):4.9(5)	6.38(5)	29	0.40
enw.latest	5.96(9)	1.84(9):1.85(8)	2.59(9)	39	0.47
lineitem	3.22(7)	1.02(6):1	9.6(6)	19	0.31
UniProt	1.15(8)	9.86(5):1.44(9)	1.05(9)	217	0.36

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)), baseball.xml (Baseball.xml, 2013), enwikibooks.xml and lineitem.xml from the Wratislavia 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. These results show that the annotated tree provides a well-compressed representation of the original files, even in the presence of very large files.

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 Kolmogorov 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, GZIP[1] and XZ[2], BWT-based (Burrows and Wheeler, 1994), BZIP2[3], Predic-

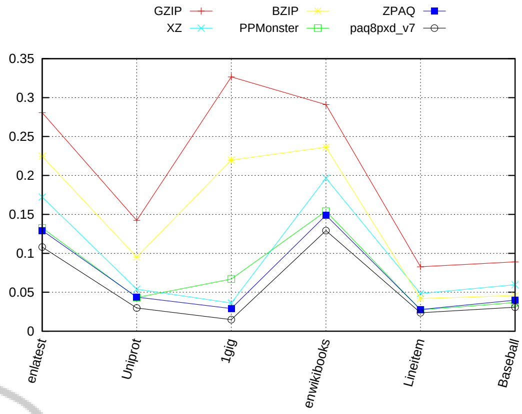

Figure 10: Compression of entire XML document with single instances of Vanilla Compressors.

Algorithm 4: Annotated Process.

```
function ENCODE(AnnotatedTree A, File file)
    write(encode(A.schema), file.schema)
    DepthFirstIterator dfs
    // For each node, its annotation list.
    for (dfs = A.iterator(); dfs.hasNext();) do
        AnnotatedNode node = dfs.next()
        write(node.annotationList.length(), file.annot).
        write(node.annotationList, file.annot).
    end for
    // For each node write its text container.
    for (dfs = A.iterator(); dfs.hasNext();) do
        AnnotatedNode node = dfs.next()
        write(node.textContainer.length(), file.text).
        write(node.textContainer, file.text).
    end for
end function
```

[1] gzip -9 FILE
[2] xz -9 -e FILE
[3] bzip2 -9 FILE

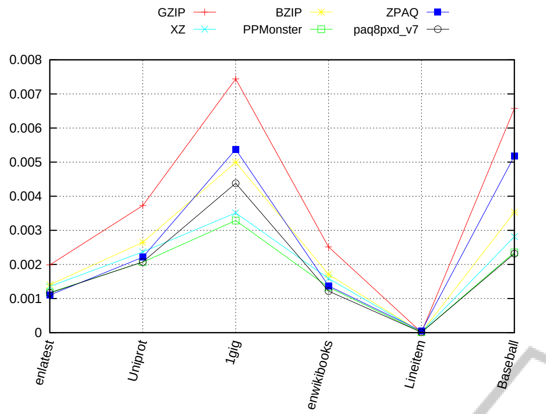

Figure 11: Compression of Annotated Tree (the number of bytes to represent the XML Syntax/Structure) over Markup Density.

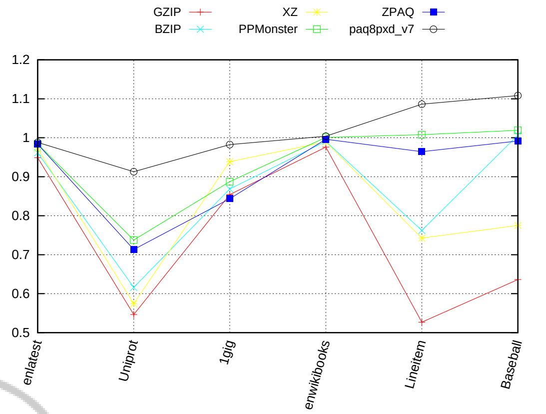

Figure 12: Compression of Annotated Transform (collection of Annotated Tree, Text Container, and Schema Tree) over Compression of XML Data.

tion by Partial Matching based PPMonster[4], and the Context-Mixing compressors ZPAQ (ZPAQ, 2013)[5] and PAQ8PXD_V7[6]; for the source code, or executables, to each compressor see (Mahoney, 2012). Results of tests showing applications of the annotated transform to each document and compressing only the structure ($\frac{\text{Annotations + Annotated Tree}}{\text{Markup Density}}$) are shown in Figure 11. It represents the syntax of the each XML document in a fraction a percent. The annotation list of each equivalence class has two components: (1) A single byte header, that signifies if any transform (e.g., a run length encoding) has been applied to the annotations; (2) A list of annotations, each encoded as a 32-bit integer (although this can be much improved by using variable sized bytes). In the worst case, the annotated representation only requires one tenth of a percent of the original markup amount (including tag names, and XML syntax data). In the best case, a very-regular (a complete tree) document, lineitem, only requires one one-thousandth of a percent of the original markup amount. In either situation, both of these situations offer a very faithful yet succinct representation of the 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.

[4]ppmonstr -m1700 -o64 FILE
[5]zpaq add FILE.zpaq FILE -method 69 -noattributes
[6]paq8pxd_v7 -8 FILE

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

Arion, A., Bonifati, A., Manolescu, I., and Pugliese, A. (2007). XQueC: a query-conscious compressed XML database. *ACM Transactions on Internet Technology*, 7(2).

Baseball.xml (2013). baseball.xml, retrieved October 2013 from http://rassyndrome.webs.com/cc/baseball.xml.

Benoit, D., Demaine, E., Munro, J., and Raman, V. (1999). Representing Trees of Higher Degree. In Dehne, F., Sack, J., Gupta, A., and Tamassia, R., editors, *Algorithms and Data Structures*, volume 1663 of *Lecture Notes in Computer Science*, pages 169–180. Springer Berlin Heidelberg.

Bille, P., Gortz, I., Weimann, O., and Landau, G. M. (2013). Tree Compression with Top Trees. In *In Proceedings of the 40th International Colloquium on Automata, Languages, and Programming*.

Burrows, M. and Wheeler, D. (1994). A block-sorting lossless data compression algorithm. *Technical Report, Digital Equipment Corporation*.

Busatto, G., Lohrey, M., and Maneth, S. (2005). Efficient Memory Representation of XML Documents. In Bierman, G. and Koch, C., editors, *Database Programming Languages*, volume 3774 of *Lecture Notes in Computer Science*, pages 199–216. Springer Berlin Heidelberg.

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

bzip2 (2013). bzip2 compression, retrieved October 2013 from http://www.bzip.org/.

Chen, S. and Reif, J. (1996). Efficient Lossless Compression of Trees and Graphs. In *In IEEE Data Compression Conference (DCC)*.

Consortium, T. U. (2013). Update on activities at the Universal Protein Resource (UniProt) in 2013. http://dx.doi.org/10.1093/nar/gks1068. Retrieved on June 20, 2013.

Corbin, T., Müldner, T., and Miziołek, J. (2013). Pre-order Compression Schemes for XML in the Real Time Environment. In *The Ninth International Conference on Web Information Systems and Technologies*, Aachen, Germany. WEBIST.

Corpus, W. (2013). Wratislavia XML corpus, retrieved October 2013 from http://www.ii.uni.wroc.pl/ inikep/research/wratislavia/.

Ferragina, P., Luccio, F., Manzini, G., and Muthukrishnan, S. (2009). Compressing and indexing labeled trees, with applications. *J. ACM*, 57(1):4:1–4:33.

Gottlob, G., Koch, C., and Pichler, R. (2005). Efficient algorithms for processing xpath queries. *ACM Trans. Database Syst.*, 30(2):444–491.

GZIP (2013). The gzip home page, retrieved October 2013 from http://www.gzip.org.

Jacobson, G. (1989). Space-efficient static trees and graphs. In *Proceedings of the 30th Annual Symposium on Foundations of Computer Science*, SFCS '89, pages 549–554, Washington, DC, USA. IEEE Computer Society.

Mahoney, M. (2012). Large Text Compression Benchmark, Retrieved October 2013 from http://mattmahoney.net/dc/zpaq.html.

Müldner, T., Corbin, T., Miziołek, J., and Fry, C. (2012). Design and Implementation of an Online XML Compressor for Large XML Files. *International Journal On Advances in Internet Technology*, 5(3):115–118.

Müldner, T., Fry, C., Miziołek, J., and Durno, S. (2009). XSAQCT: XML queryable compressor. In *Balisage: The Markup Conference 2009*, Montreal, Canada.

XML (2013). Extensible markup language (XML) 1.0 (Fifth edition), retrieved October 2013 from http://www.w3.org/tr/rec-xml/.

xmlgen (2013). The benchmark data generator, retrieved October 2013 from http://www.xml-benchmark.org/generator.html.

Ziv, J. and Lempel, A. (2006). A universal algorithm for sequential data compression. *IEEE Trans. Inf. Theor.*, 23(3):337–343.

ZPAQ (2013). Zpaq, retrieved October 2013 from http://www.w3.org/tr/rec-xml/.