Dictionary-Based Compression Algorithms for Tree Structured Data

Yuko Itokawa* Koichiro Katoh† Tomoyuki Uchida‡ Takayoshi Shoudai§

Abstract— **Electronic data like XML/HTML documents, called tree structured data, have been rapidly increasing and have become larger day by day. In this paper, we propose an efficient compression and decompression algorithms based on the Lempel-Ziv compression scheme by improving XMill and XDemill (Liefke and Suciu, SIGMOD 2000) which is a compressor and a decompressor for tree structured data, respectively. Moreovre, in order to show the effectiveness and efficiency of our algorithms, we report experimental results of applying our algorithms to randomly created artificial large trees and real-world data.**

Keywords: Tree Structured Data, Lempel-Ziv Compression Scheme, Dictionary Based Lossless Compression

1 Introduction

Due to the rapid growth of information technologies, electronic data such as XML/HTML documents have been rapidly increasing. Since such data have no rigid structure but have tree structures, they are called tree structured data and can be represented by a rooted tree which has no vertex label but which has edge labels. A rooted tree whose internal vertices have ordered children is called an *ordered tree*. In Figure 1, we give a part of HTML data *Sample_html* as an example of tree structured data and an ordered tree T which represents *Sample_html*. The number in the left side of a vertex in T denotes the ordering on its siblings.

The purpose of this paper is to present an efficient compression algorithm for an ordered tree and a decompression algorithm for a compressed tree. Firstly, we give a Lempel-Ziv compression scheme for ordered trees. In a Lempel-Ziv compression scheme for strings such as LZSS [5], a previously seen text is used as a dictionary, and phrases in the input text are replaced with pointers into the dictionary to achieve compression. In our Lempel-Ziv compression scheme for an ordered tree, a firstly occurred tree f in postorder traversal of an ordered tree T is used as a dictionary and subgraphs in T which are isomorphic to f are replaced by variables with pointers into the dictionary to achieve compression. Matsumoto et al. [4] gave a *term tree* as a tree pattern having internal structural variables. In this paper, a compression of an ordered tree T is represented by a pair of a term tree t and a substitution θ such that T is obtained by applying θ to t. We give a term tree t in Figure 1 as an example of a term tree and a substitution $\{x := [g, (u6, u1, u3)]\}$ as an example of a dictionary, where g is given in Figure 1. The variables in a term tree are represented by squares with lines to its elements. Letter in the square represents the label of the variable.

Secondly, based on our Lempel-Ziv compression scheme, we present efficient compression and decompression algorithms for an ordered tree. Next, using our compression and decompression algorithms for ordered trees, we give a compression algorithm for tree structured data by improving XMill which is a compressor for tree structured data given by Liefke and Suciu[3]. Moreover, we also give a decompression algorithm for tree structured data by improving XDemill[3] which is a decompresser for tree structured data.

Several compressors for tree structured data are already proposed, i.e. XMill[3], LZCS[1], XGrind[6] and XMLPPM[2]. Liefke and Suciu[3] presented a compressor (XMill) and a decompressor(XDemill) whose architecture leverages existing compressing algorithms and tools to XML data. XMill separates the tree structure which obtains by parsing XML data with respect to XML tags and attributes and the sequence of data items (strings) which represent contents and attribute values from given XML data. Then, XMill compresses a sequence describing its tree structure by a compressor over strings such as gzip. Our compressor given in this paper can directly treat a tree structure of given XML data and gives a compression over ordered trees. Our compressor can be used as a compressor employed in structure container of XMill. As other related works, Adiego et al. [1] proposed a compressor LZCS which obtained by replacing frequently repeated subtrees by backward references to their first occurrence, where a subtree having a vertex v as a root consists of v and all of its descendants. Our compression algorithm replaces frequently repeated connected subgraphs by references to their first occurrence. In [7], Yamagata et al. presented gave a grammar-based

*Faculty of Psychological Science, Hiroshima International University, 555-36 Kurose-Gakuendai, Higashi-Hiroshima, Hiroshima Japan Tel/Fax: 81-823-70-4880/4852 Email: y-itoka@he.hirokoku-u.ac.jp

†Dep. of Computer and Media Technologies, Hiroshima City University, Japan

‡Dep. of Intelligent Systems, Hiroshima City University, Japan

§Dep. of Informatics, Kyushu University, Japan

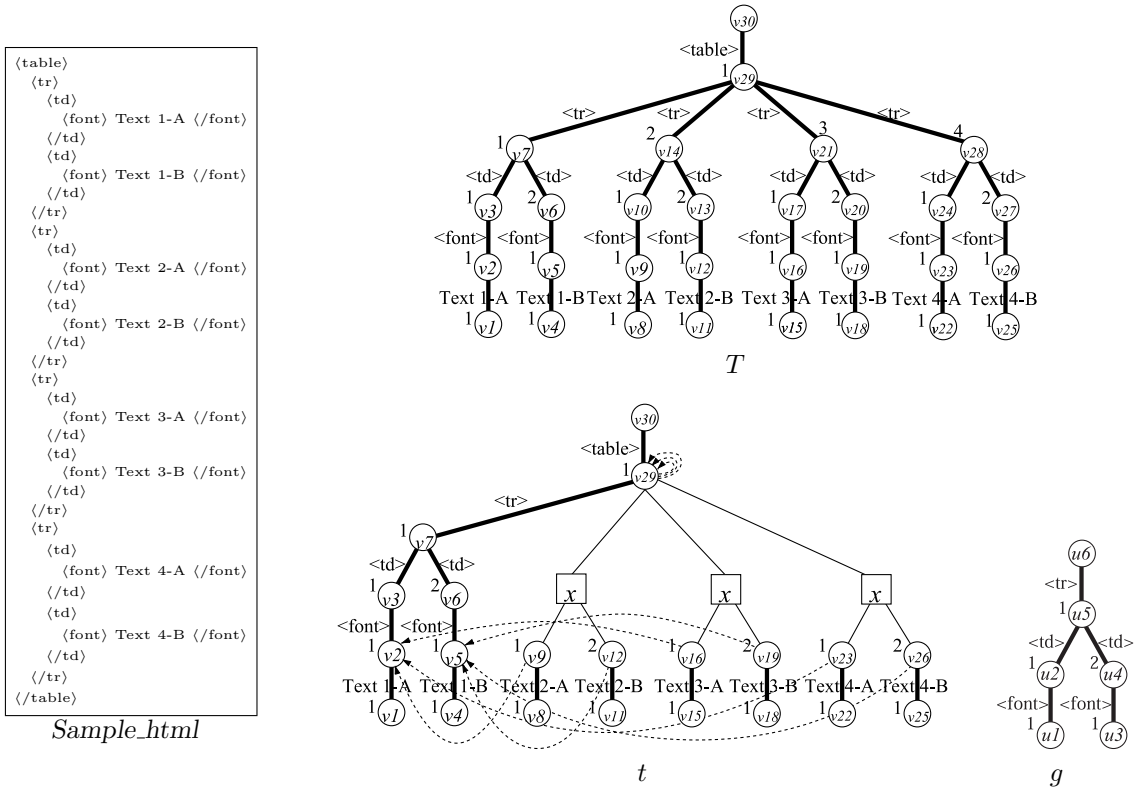

Figure 1: An HTML data *Sample_html*, the ordered rooted tree T which represents *Sample_html*, a term tree t and an ordered rooted tree g.

lossless compression algorithm for a given ordered tree T.

This paper is organized as follows. In Section 2, we formally define a term tree and its substitution which leads us to dictionary-based compression for an ordered tree without loss of information. In Section 3, we introduce a Lempel-Ziv compression scheme for ordered trees and propose compression and decompression algorithms based on Lempel-Ziv compression schemes by improving XMill and XDemill [3]. In Section 4, we evaluate the performance of our algorithms by reporting experimental results of applying our algorithms to both artificial large trees and HTML data which are real-world data.

2 Ordered Term Trees and Substitutions

Let $T = (V_T, E_T)$ be an ordered tree with a vertex set V_T and an edge set E_T. Let $\ell \geq 1$ be an integer. A list $h = (u_0, u_1, \ldots, u_\ell)$ of vertices in V_T is called a *variable* if $u_1, \ldots, u_\ell$ is a sequence of consecutive children of u_0, i.e., u_0 is the parent of $u_1, \ldots, u_\ell$ and u_{j+1} is the next sibling of u_j for j $(1 \leq j < \ell)$. Two variables $h = (u_0, u_1, \ldots, u_\ell)$ and $h' = (u'_0, u'_1, \ldots, u'_{\ell'})$ are said to be *disjoint* if $\{u_1, \ldots, u_\ell\} \cap \{u'_1, \ldots, u'_{\ell'}\} = \emptyset$. Let H_T be a set of pairwise disjoint variables of $T = (V_T, E_T)$. An *ordered term tree* on T and H_T is a triplet $t = (V_t, E_t, H_t)$ where $V_t = V_T$, $E_t = E_T - \bigcup_{h=(u_0,u_1,\ldots,u_\ell)\in H_T}\{(u_0, u_i) \in E_T \mid 1 \leq i \leq \ell\}$ and $H_t = H_T$. Because T and H_T are easily found from a triplet $t = (V_t, E_t, H_t)$, we do not write T and H_T explicitly. Below we call an ordered term tree a *term tree*, simply. A term tree $t = (V_t, E_t, H_t)$ is called a *ground term tree* if $H_t = \emptyset$. Let Λ and X be finite alphabets such that $\Lambda \cap X = \emptyset$. An element of Λ (resp. X) is called an *edge label* (resp. a *variable label*). Every variable label $x \in X$ has a nonnegative integer $rank(x)$. Every variable h has a variable label x such that $rank(x) = |h|$. A *term tree over* $\langle \Lambda, X \rangle$ is a term tree t such that all edges and variables in t are labeled with elements in Λ and X, respectively. If Λ and X are clear from the context, we often omit them. We use the same terminologies of ordered trees for term trees, for example, *parent*, *child*, *leaf*, and so on. For a set or a list D, the number of elements in D is denoted by $|D|$.

For three vertices u, u', u'' of a term tree t, we write $u' <^t_u u''$ if u' and u'' are two children of u and u' is smaller than u'' in the order of the children of u. A term tree $t = (V_t, E_t, H_t)$ *is isomorphic to* a term tree $g = (V_g, E_g, H_g)$, denoted by $t \equiv g$, if there is a bijection $\pi : V_t \to V_g$, called an *isomorphism between f and g*, such that for $v_0, v_1, \ldots \in V_t$, (1) $(v_1, v_2) \in E_t$ if and only if $(\pi(v_1), \pi(v_2)) \in E_g$, (2) (v_1, v_2) in E_t and $(\pi(v_1), \pi(v_2))$ in E_g have the same edge label, (3) $v_1 <^f_{v_0} v_2$ if and only if $\pi(v_1) <^g_{\pi(v_0)} \pi(v_2)$, (4) $(v_0, \ldots, v_\ell) \in H_t$ if and only if $(\pi(v_0), \ldots, \pi(v_\ell)) \in H_g$, and (5) two variables $(v_0, \ldots, v_\ell) \in H_t$ and $(v'_0, \ldots, v'_\ell) \in H_t$ have the same variable label if and only if $(\pi(v_0), \ldots, \pi(v_\ell)) \in H_g$ and

$(\pi(v'_0), \ldots, \pi(v'_\ell)) \in H_g$ have the same variable label.

Let $t = (V_t, E_t, H_t)$ be a term tree. A triplet $f = (V_f, E_f, H_f)$ is called a *term subtree* of t if f is a term tree such that $V_f \subseteq V_t$, $E_f \subseteq E_t$ and $H_f \subseteq H_t$. If f is a ground term tree, we call f a *subtree* of t simply. Let f and g be term trees over $\langle \Lambda, X \rangle$ having at least two vertices. Let $h = (v_0, v_1, \ldots, v_\ell)$ $(\ell \geq 1)$ be a variable in f and $\sigma = (u_0, u_1, \ldots, u_\ell)$ a list of $\ell + 1$ distinct vertices in g such that u_0 is the root of g and $u_1, \ldots, u_\ell$ are leaves of g. The pair $[g, \sigma]$ is called an $(\ell+1)$-*hypertree over* $\langle \Lambda, X \rangle$. Below we often omit $\langle \Lambda, X \rangle$. For two $(\ell+1)$-hypertrees $[g, (u_0, \ldots, u_\ell)]$ and $[f, (w_0, \ldots, w_\ell)]$, $[g, (u_0, \ldots, u_\ell)]$ is said to be *equivalent to* $[f, (w_0, \ldots, w_\ell)]$ if there is an isomorphism π between g and f such that for each i $(0 \leq i \leq \ell)$, $w_i = \pi(u_i)$. For a variable h, a term tree g and a list σ of distinct vertices of g, the form $h \leftarrow [g, \sigma]$ is called a *variable replacement* for h. A new term tree $f' = f\{h \leftarrow [g, \sigma]\}$ is obtained by applying $h \leftarrow [g, \sigma]$ to f in the following way. For the variable $h = (v_0, \ldots, v_\ell)$ in f, we attach g to f by removing h from f and identifying $v_0, \ldots, v_\ell$ of f with $u_0, \ldots, u_\ell$ of g in this order. We define a new ordering $<_v^{f'}$ on every vertex v in f' so that for a vertex v in f' with at least two children v' and v'' of v, (1) if $v, v', v'' \in V_g$ and $v' <_v^g v''$ then $v' <_v^{f'} v''$, (2) if $v, v', v'' \in V_f$ and $v' <_v^f v''$ then $v' <_v^{f'} v''$, (3) if $v = v_0 (= u_0)$, $v' \in V_f - \{v_1, \ldots, v_\ell\}$, $v'' \in V_g$, and $v' <_v^f v_1$ then $v' <_v^{f'} v''$, and (4) if $v = v_0 (= u_0)$, $v' \in V_f - \{v_1, \ldots, v_\ell\}$, $v'' \in V_g$, and $v_\ell <_v^f v'$ then $v'' <_v^{f'} v'$. Let x be a variable label with $rank(x) = \ell + 1$ and $[g, \sigma]$ an $\ell+1$-hypertree. Then, the form $x := [g, \sigma]$ is called a *binding* for x. A finite collection of bindings $\theta = \{x_1 := [g_1, \sigma_1], \ldots, x_n := [g_n, \sigma_n]\}$ is called a *substitution* if x_i's are mutually distinct variable labels in X and no variable in g_i has a variable label in $\{x_1, \ldots, x_n\}$. Let t be a term tree and $H_t(x_i)$ the set of all variables in H_t whose labels are x_i for a variable label x_i in X. Then, for a subtitution $\theta = \{x_1 := [g_1, \sigma_1], \ldots, x_n := [g_n, \sigma_n]\}$, we can obtain a new term tree, denoted by $t\theta$, by applying all variable replacements in $\bigcup_{i=1}^n \{e \leftarrow [g_i, \sigma_i] \mid e \in H_t(x_i)\}$ to t.

3 Lempel-Ziv Compression Algorithm for Tree Structured Data

In a Lempel-Ziv compression scheme over strings such as LZSS [5], a previously seen text is used as a dictionary, and phrases in the input text are replaced with pointers into the dictionary to achieve a compression. Using the framework given in the previous section, we present a Lempel-Ziv compression scheme for ordered trees by regarding a substitution as a dictionary.

Let T be a tree and t a term tree. If there is a substitution θ for t such that $T \equiv t\theta$ and the sum of the representation sizes (defined later) of t and θ is less than that of T, the pair (t, θ) gives us a compression of T by regarding θ as a dictionary. First we define the representation of a binding in θ in order to store it into a dictionary effectively. We assume that for each binding $x := [g, \sigma]$ in θ, t has a term subtree g' such that $g \equiv g'$. Let π be an isomorphism from g to g'. Let u_0 be the root of g and $u_1, \ldots, u_n$ all leaves of g which are listed in left-to-right order. Then the *corresponding list of* g *in* t, denoted by CL_t^g, is the list $(\pi(u_1), \ldots, \pi(u_n), \pi(u_0))$. The *corresponding dictionary of* θ *in* t, denoted by CD_t^θ, is the list obtained by sorting all elements of $\{CL_t^g \mid x := [g, \sigma] \in \theta\}$ lexicographically. For a binding $x := [g, \sigma]$ in θ, the *port index* of σ in t, denoted by PI_t^σ, is a list $(k_2, k_3, \ldots, k_{|\sigma|})$ of $|\sigma| - 1$ distinct integers from 1 to n such that $CL_t^g[k_i] = \pi(\sigma[i])$ for any i $(2 \leq i \leq |\sigma|)$. We suppose that the corresponding list CL_t^g is the ℓ_g-th member in CD_t^θ $(1 \leq \ell_g \leq |CD_t^\theta|)$. Then a pair (ℓ_g, PI_t^σ) is called the *corresponding variable label* of a binging $x := [g, \sigma]$. Below we identify a term tree g in a binding $x := [g, \sigma]$ with a term subtree g' of t such that $g \equiv g'$. For example, let T be a tree and t a term tree described in in Figure 2. Let $\theta = \{x := [g, (4, 2)]\}$ be a substitution where g is a term subtree of t shown in Figure 2. Obviously $T \equiv t\theta$ holds. Then, the corresponding list of g in t is $CL_t^g = (1, 2, 4)$. The corresponding dictionary of θ is the list $((1, 2, 4))$. The port index of σ in t is $PI_t^{(4,2)} = (2)$. And the corresponding variable label of the binding $x := [g, (4, 2)]$ is $(1, (2))$. Finally, we define a Lempel-Ziv compression for tree structure data. Let T be a tree. A pair (t, θ) of a term tree t and a substitution θ is called a *Lempel-Ziv compression of* T if the following conditions (1)–(3) hold.

(1) $T \equiv t\theta$.
(2) For each binding $x := [g, \sigma]$, t has a term subtree isomorphic to g.
(3) The sum of sizes of t and θ is less than that of T.

In Figuer 2, we show an example of Lempel-Ziv compressions.

In Figure 3, we describe an algorithm *Compressing_Ordered_Tree* for a given tree T and two integers Min and Max with $0 < Min < Max$, which outputs a Lempel-Ziv compression (t, θ) of T. In this section, we present compression algorithms for tree structured data by improving XMill [3]

For a vertex v in a given tree, a function *Make_Term_Subtree* at the line 3 of *Compressing_Ordered_Tree* outputs the set of all hypertrees $[f, \sigma]$ satisfying the following conditions (1)–(6). Let $f = (V_f, E_f, H_f)$ and $\sigma = (u_1, \ldots, u_n)$.

(1) f is a subtree of t and v is the rightmost child of the root of f.
(2) $Min \leq |V_f| \leq Max$.
(3) For a vertex $u \in V_f$, if there exists a vertex w in $V_T - V_f$ such that w is adjacent to u in T then u is the root of f or a leaf of f.
(4) The vertex u_1 in σ is the root of f.
(5) If $n > 2$, all $u_2, \ldots, u_n$ are adjacent to vertices in $V_T - V_f$.

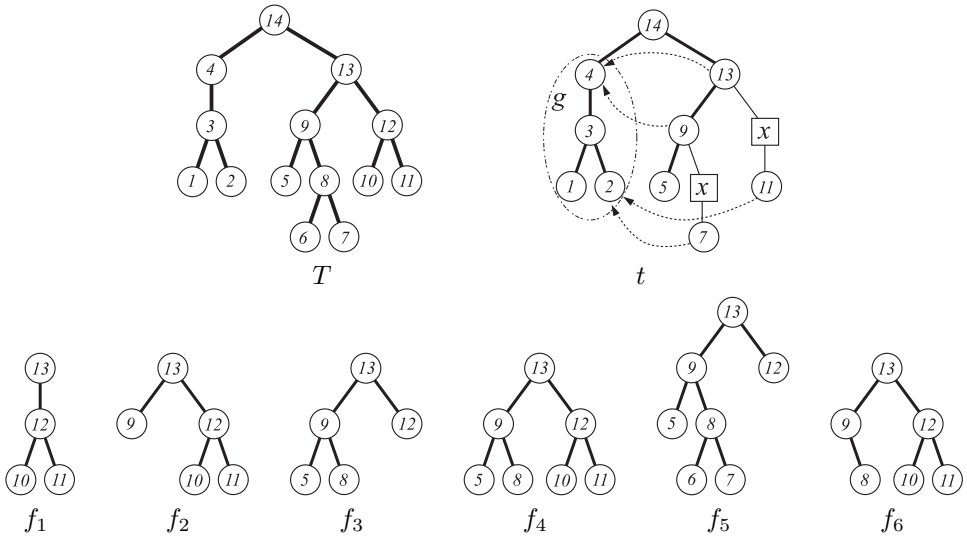

Figure 2: A tree T is an input tree-structured data and f_1, f_2, f_3, f_4, f_5, and f_6 are subtrees of T. Then $(t, \{x := [g, (4, 2)]\})$ is a Lempel-Ziv compression of T. The tree f_1 and the list $(4, 2)$ in the binding are indicated by the subtree of t in the circle and the four pointers from 9 to 4 and from 7 to 2, and from 13 to 4 and from 11 to 2.

(6) If $n = 2$, either u_2 is adjacent to a vertex in $V_T - V_f$ or u_2 is the rightmost leaf of f.

By appropriately setting integers Min and Max, we can bound the maximum number of hypertrees which *Make_Term_Subtree* outputs. Since the probability that a large subtree of t is occurred in t in several times is low, in general, and the number of hypertrees in S affects the time complexity of *Compressing_Ordered_Tree*, Min and Max must be appropriately given for tree structured data by users. For example, let T be a tree in Figure 2 and f_1, f_2, f_3, f_4, f_5 and f_6 subtrees of T in Figure 2. All subtrees f_1, f_2, f_3, f_4, f_5, and f_6 satisfy the conditions (1) and (2) for a vertex 12, $Min = 4$ and $Max = 7$. However f_6 does not satisfy the condition (3) for the same parameters while f_1, f_2, f_3, f_4, and f_5 satisfy it. Let us consider two hypertrees $[f_1, (13, 10)]$ and $[f_1, (13, 11)]$. Since 11 is the rightmost leaf of f_1 but 10 is not, $[f_1, (13, 11)]$ is constructed by *Make_Term_Subtree*$(T, 12, 4, 7)$ but $[f_1, (13, 10)]$ is not. Hence, it outputs the set $S = \{[f_1, (13, 10, 11)], [f_2, (13, 9, 10, 11)], [f_3, (13, 5, 8, 12)], [f_4, (13, 5, 8, 10, 11)], [f_5, (13, 5, 6, 7, 12)]\}$.

Compressing_Ordered_Tree repeats the following processes. Let g be one of term subtrees generated by *Make_Term_Subtree* for a vertex $v \in V_t$. If there is a vertex v' which is visited before v in postorder such that *Make_Term_Subtree* for v' outputs a term subtree g' isomorphic to g, we revise the term tree t by replacing g with a new variable having a new variable label x and add a new binding $x := [g', \sigma]$ to θ. We introduce a new pointer from g into a firstly occurred subtree $g' = (V'_g, E'_g)$ in $t = (V_t, E_t, H_t)$ such that g' is isomorphic to g and for any vertex $v \in V'_g$, v is not adjacent to any vertex in $V_t - V'_g$ in t if v is neither the root of g' nor a leaf of g'. Such a pointer consists of the list of pointers into the root of g' and into some of the leaves of g'.

Next we present an algorithm for decompressing a Lempel-Ziv compression (t, θ) by applying repeatedly the following variable replacement process to $t = (V_t, E_t, H_t)$ until t has no variable:

(1) choose a variable h in t which firstly appears in depth-first search,

(2) find a binding $x := [g, \sigma]$ such that x is a variable label of h,

(3) apply the variable replacement $h \leftarrow [g, \sigma]$ to t.

The structure of an HTML file is encoded as follows. Start-tags are assigned positive integers while all end-tags are replaced by the token "/". Tags, such as `<table>`, `<tr>`, ... in *Sample_html* in Fig. 1 are encoded to positive integers `T1`, `T2`, ... In XMill[3], to obtain a higher compression ratio by `gzip`, PCDATAs whose parents have the same tag label are stored in same container. PCDATAs whose parent tags are assigned a positive integer `a` are assigned a negative integer value $-$`a`. For example, for *Sample_html* in Fig. 1, we assigned `<table>` = 1, `<tr>` = 2, `<td>` = 3, `<font>` = 4, respectively. Then we obtain the following sequence which represents T:

```
1 2 3 4 -4//3 4 -4///2 3 4 -4//3 4 -4///2 3
          4 -4//3 4 -4///2 3 4 -4//3 4 -4////
```

A term tree $t = (V_t, E_t, H_t)$ and a substitution $\theta = \{x_i := [g_i, \sigma_i] \mid 1 \leq i \leq |\theta|\}$ are encoded as follows. Let $h \in H_t$ be a variable which has a corresponding variable label $(d, (p_1, \cdots, p_N))$ and a a decimal integer. We encode a port index $(p_1, \cdots, p_N)$ into a decimal integer a such that the q-th bit $(1 \leq q \leq p_N)$ of the binary representation of a is 1 if and only if q is in $(p_1, \cdots, p_N)$. Then

Algorithm Compressing_Ordered_Tree

Input: A tree $T = (V_T, E_T)$, integers Min and Max $(0 < Min < Max)$.
Output: A Lempel-Ziv compression (t, θ) of T.

```
1.  t := T, θ := ∅, δ := ∅;
2.  for each vertex v in T, in postorder do
3.      S := Make_Term_Subtree(t, v, Min, Max);
4.      if θ = ∅ and δ = ∅ then δ := S;
5.      else
6.          for each hypertree [f, σ] ∈ S in decreasing order of the size of f do
7.              if there exists a binding x := [f', σ'] in θ such that [f', σ'] is equivalent to [f, σ] then
8.                  replace a term subtree f in t with a new variable labeled with x;
9.              else if there exists a hypertree [f', σ'] in δ such that [f', σ'] is equivalent to [f, σ] then
10.                 replace a term subtree f in t with a new variable having a new variable label y;
11.                 θ := θ ∪ {y := [f', σ']}; δ := δ − {[f', σ']};
12.             else δ := δ ∪ {[f, σ]};
13.     end do
14. end do
15. return the pair (t, θ).
```

Figure 3: Sequential algorithm *Compressing_Ordered_Tree*

we encode a variable h into "$d(a)$". The child ports of h are surrounded by a delimiter "`$`". The code of binding $x_i := [g_i, \sigma_i] \in \theta$ is represented by a corresponding list of g_i. So the substitution θ is encoded by a sequence of codes of binding delimited by "`#`". We describe a Lempel-Ziv compression (t, θ) as the sequence of the description of θ, a delimiter "`%`", and the description of t. For example, we obtain the following sequence which represents a Lempel-Ziv compression (t, θ) in Fig. 1:

```
6 1 3%1 2 3 4 -4//3 4 -4///1(40)$-4/
                $-4/1(40)$-4/$-4/1(40)$-4/$-4//
```

Let $|T|$ be the description length of a tree T and $|(t, \theta)|$ the description length of a Lempel-Ziv compression of a tree T, then $(|(t, \theta)|/|T|) \times 100$ is defined as the *compression ratio* of T w.r.t. (t, θ). The smaller this ratio, the better the compression scheme. We count integer object tokens like start-tags as 4 bytes and delimiter tokens like `/` or white space as 1 byte. For the tree T in Fig. 1, $|T| = 158$ and for the Lempel-Ziv compression (t, θ), $|(t, \theta)| = 125$. Then, the compression ratio of T w.r.t. (t, θ) is $(125/158) \times 100 \approx 79$.

4 Implementation and Experimental Results

In order to evaluate our sequential lossless compression and decompression algorithm presented in the previous sections, we implemented them on a PC with 3.4GHz CPU (XEON) and 1.00GB main memory.

At first we implemented a data generator to randomly produce an artificial large tree satisfying that the degree of each vertex is less than or equal to 4 and the number of edge labels is less than or equal to 3. For each $N \in \{20,000,\ 40,000,\ 60,000,\ 80,000,\ 100,000\}$, we generated a set $D(N)$ by using the data generator. Under the following settings, we applied our algorithms to the set $D(20,000)$, $D(40,000)$, $D(60,000)$, $D(80,000)$, and $D(100,000)$. (a) $Min = 4$ and $Max = 7$. (b) A temporary dictionary can store 10,000 candidate subtrees. We report experimental results in Figure 4.

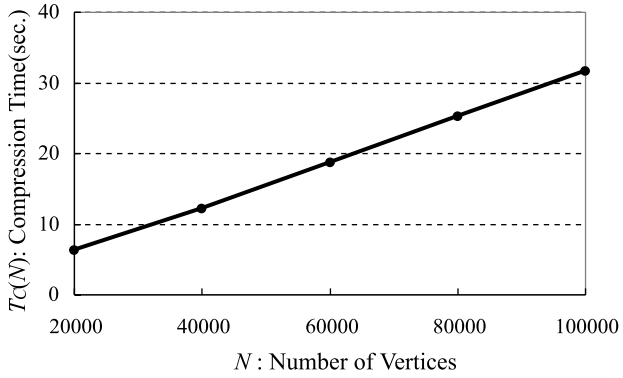

Figure 4: Compression Time vs Number of Vertices

Let $T_C(N)$ and $T_D(N)$ be the average running times of compressions and decompressions of one tree in $D(N)$, respectively. Figure 4 shows the relationship between $T_C(N)$ and N. The running time depends on the capacity of a temporal dictionary and settings of two integers Min and Max. From Figure 4, $T_C(N)$ increases almost linearly w.r.t. N. Our algorithm could process in about 30 seconds even a tree which has 100,000 vertices. Each tree is compressed to a Lempel-Ziv compression whose size is about 80% of the original size. For all randomly generated trees, over 50 elements of dictionary were generated. It is no wonder that artificial tree data were not

Table 1: Experimental results

HTML file set	#Vertices	Compression ratio(%)		Running time for Compression(sec)		Running time for Decompression(sec)	
		Our Algorithm	XMill	Our Algorithm	XMill	Our Algorithm	XMill
nissan*1	5142	2.84	16.08	5.69	1.86	1.87	1.76
honda*2	7369	2.56	15.61	7.32	1.84	2.10	1.90
toyota*3	5295	3.08	17.08	5.97	1.88	1.90	1.78
various files*4	10980	11.94	15.63	10.67	2.56	2.52	2.33

∗1 http://www.nissan.co.jp/CARLINEUP/ ∗2 http://www.honda.co.jp/auto-lineup/

∗3 http://toyota.jp/Showroom/carlineup/index.html ∗4 Mixed data from official sites of European football teams.

compressed and a number of dictionary abound because they contain little same structure.

We applied our decompression algorithm to the Lempel-Ziv compressions obtained by the above compression experiments on $D(20,000)$, $D(40,000)$, $D(60,000)$, $D(80,000)$, and $D(100,000)$. $T_D(N)$ is quite faster than $T_C(N)$. For example, the average of the running time of decompressions on the Lempel-Ziv compressions of D(100,000) is about 1 second. All the other Lempel-Ziv compressions are decompressed in less than a second. This result shows that our Lempel-Ziv compression scheme has the same characteristics as a Lempel-Ziv compression scheme such as LZSS over strings. Finally, these results also show that our algorithms have high durabilities against huge data.

We also experimented applying our compression algorithm to HTML files as real-world data. We used 3 sets of HTML files that contain a lot of same tree structures and a set of HTML files that contains little same tree structure. In this experiment, we applied a huge tree structured data by connecting some tree structured data which represent HTML files to our algorithm. Table 1 shows the compression ratios for HTML file sizes and running times for compressions and decompressions of each set. From Table 1, the running time for our compression and decompression algorithms was lower that XMill

These experimental results show higher compression ratios than those of the experiments on XMill. Similarly to the experiments on random data sets, the decompression time is quite fast. In fact, the Lempel-Ziv compression of the set "toyota" of HTML files is decompressed in about 1 second. The other three sets of HTML files are decompressed in less than a second.

5 Concluding Remarks

We have presented efficient compression and decompression algorithms for ordered trees based on a Lempel-Ziv compression scheme, by improving XMill and XDemill presented by Liefke and Suciu [3]. In order to evaluate the performance of our algorithms, we have reported some experimental results of applying our algorithms to artificial large ordered trees and real-world tree structured data which are HTML documents.

As future works, when a Lempel-Ziv compression (t, θ) and a term tree s are given, we will design a pattern matching algorithm without decompression that decides whether there exists a substitution δ such that $t\theta \equiv s\delta$ or not

References

[1] J. Adiego and G. Navarro and P. de la Fuente. Lempel-Ziv compression of structured text. *Data Compression Conference (DCC 2004)*, pages 112–121, 2004.

[2] J. Cheney. Compression XML with multiplexed hierarchical PPM models. Proc. Data Compression Conference (DCC 2001), pages 163, 2001.

[3] H. Liefke and D. Suciu. Xmill: an efficient compressor for xml data. *Proc. ACM SIGMOD Conf.*, 29(1):57–62, 2000.

[4] S. Matsumoto, Y. Hayashi, and T. Shoudai. Polynomial time inductive inference of regular term tree languages from positive data. *Proc. ALT-97, Springer-Verlag, LNAI 1316*, pages 212–227, 1997.

[5] J.A. Storer and T.G. Szymanski. Data compression via textual substitution. *Journal of the ACM*, 29(4):928–951, 1982.

[6] P. Tolani and J. Haritsa. XGRIND: A query-friendly XML compressor. *Proc. of 18th International Conference of Data Engineering (ICDE'02)*, pages 225–234, 2002.

[7] K. Yamagata, T. Uchida, T. Shoudai, and Y. Nakamura. An effective grammar-based compression algorithm for tree structured data. *Proc. ILP-2003, Springer, LNAI 2835*, pages 383–400, 2003.