2011 Data Compression Conference

Tree Structure Compression with RePair

Markus Lohrey
Universität Leipzig
Germany

Sebastian Maneth
NICTA and UNSW
Australia

Roy Mennicke
Universität Leipzig
Germany

Abstract

Larsson and Moffat's RePair algorithm is generalized from strings to trees. The new algorithm (TreeRePair) produces straight-line linear context-free tree (SLT) grammars which are smaller than those produced by previous grammar-based compressors such as BPLEX. Experiments show that a Huffman-based coding of the resulting grammars gives compression ratios comparable to the best known XML file compressors. Moreover, SLT grammars can be used as efficient memory representation of trees. Our investigations show that tree traversals over TreeRePair grammars are 14 times slower than over pointer structures and 5 times slower than over succinct trees, while memory consumption is only $1/43$ and $1/6$, respectively.

1 Introduction

Grammar-based compression [6] offers an interesting alternative to other compression methods such as entropy-based compression (e.g., arithmetic coding). It exploits repetitions of substrings rather than frequencies, and works particularly well for highly repetitive strings. The idea is to find a small *straight-line context-free* (SL) grammar that generates the given string. Intuitively, each nonterminal of the grammar represents a repeated substring. The output of dictionary-based compressors (e.g., LZ77, LZ78) can be seen as a small grammar for the input string. Besides their use for file compression, grammars have the advantage that for particular tasks, they can be processed without decompression. For instance, it is possible to run a finite-state automaton over a grammar in time proportional to the size of the grammar (which can be exponentially smaller than the original string), see, e.g., [19]; other efficient tasks for SL grammars include equivalence checking [18] and pattern matching [12]. Recently a self-index for SL grammars was proposed [8] which achieves linear extracting and linear substring search time.

XML documents are highly repetitive w.r.t. their markup. The latter describes an ordered (unranked) tree. For instance, the unique minimal directed acyclic graph (DAG), which can be constructed in linear time, only exhibits about 10% of the edges of common XML document trees [4]. While DAGs can be seen as regular tree grammars, there is also a generalization of SL grammars to trees: straight-line linear context-free tree (SLT) grammars [5, 15]. SLT grammars have similarly nice properties as SL grammars; for instance, executing tree automata (or even certain XPath queries) can be done in linear time in the size of the grammar [13, 15] and equivalence can be decided in cubic time [5]. On the other hand, finding a minimal SLT grammar is NP-complete; this already holds for SL grammars [6]. The first approximation algorithm for SLT grammars is the sliding-window based BPLEX [5]. For common XML document trees, BPLEX produces grammars with 50% less edges than the minimal DAG. In this paper we present a new approximation algorithm for SLT grammars. Our algorithm is a generalization of RePair [10] from strings to trees. For strings, several approximation algorithms exist, e.g., LZ78 or Sequitur [17].

1068-0314/11 $26.00 © 2011 IEEE
DOI 10.1109/DCC.2011.42

RePair is one of the stronger such algorithms, and most of all, its idea is intriguingly simple: (1) find the most frequent pair of adjacent symbols in the input string, (2) replace it by a new nonterminal and add the corresponding production, and (3) repeat this process until no repeated pair exists. The input to our algorithm is a node-labeled, ordered tree. Instead of adjacent pairs of symbols we now replace adjacent nodes in the tree, that is, a node together with one of its child nodes (this is called a *digram*). Consider the tree $f(a(e,e), f(a(e,e),e))$. The most frequent digram is either f with left-child a, or a with left-child e, or a with right-child e. Replacing the first one gives $A_1(e,e,A_1(e,e,e))$ and the new production $A_1(y_1,y_2,y_3) \to f(a(y_1,y_2),y_3)$. After two more iterations we obtain the following grammar (S is the start nonterminal): $\{S \to A_3(A_3(e)), A_3(y_1) \to A_2(e,y_1), A_2(y_1,y_2) \to A_1(e,y_1,y_2), A_1(y_1,y_2,y_3) \to f(a(y_1,y_2),y_3)\}$. The symbols $y_1, y_2, \ldots$ are *formal parameters* and indicate how to embed the right-hand side of the production when applying it. Each parameter y_i occurs at most once in a right-hand side (such tree grammars are called *linear*). Of course, also other grammars could have been derived; for instance, if we first replace a-nodes with first child e, then we obtain the grammar $\{S \to B_3(B_3(e)), B_3(y_1) \to f(B_2,y_1), B_2 \to B_1(e), B_1(y_1) \to a(e,y_1)\}$. The size of this grammar (in number of edges) is only 7, as opposed to 11 for the previous grammar (and 9 for the original tree). Moreover, the maximal number of parameters y_j, called the *rank of the grammar*, is one for the second grammar, while it is three for the first grammar. In our implementation of TreeRePair, the user can specify the maximal allowed rank of the output grammar as a parameter m. It turns out that the compression ratio of TreeRePair very much depends on m: for certain families of trees, setting $m = 1$ is optimal, while for other families leaving $m = \infty$ is optimal, see Section 2.3. Interestingly, for our XML corpus, $m = 4$ gives the smallest grammars. Another phenomenon due to parameters is that new productions can potentially *increase* the size of the grammar. For instance, removing the productions for B_1 and B_2 from the second grammar above results in the grammar $\{S \to B_3(B_3(e)), B_3(y_1) \to f(a(e,e),y_1)\}$, which consists of only 6 edges. The removal of inefficient productions is called *pruning*, and is carried out in our TreeRePair implementation at the end.

Our implementation uses similar data structures as the one for strings [10] which guarantee that the whole algorithm runs in linear time. To save memory, we do not store the original tree but instead work over its minimal DAG. Our experiments show that the implementation outperforms previous tree grammar compressors: on average we are 32 times faster and use $1/10$ of the memory of BPLEX [5], and are 6 times faster with $1/8$ of the memory of CluX [3]. The latter compressor is another tree adaptation of RePair, which was recently and independently developed. CluX differs from TreeRePair in several important details, see Section 2.4. The size of our grammars (in number of edges) is 20% less than BPLEX and 40% less than CluX.

We implemented a Huffman-based coding of our grammars, which allows to use TreeRePair as an XML file compressor. Our experiments show that the resulting files are smaller than most previous XML compressors such as XMill [11], SCMPPM [1], SCMHuff [1], BPLEX [16], or running gzip or bzip2 over the XML markup file. A notable exception is XMLPPM [7] which compresses on average 12% better than our TreeRePair, all averaged over our corpus of 24 XML documents. Note that we work on *XML tree structures* only, i.e., we remove all text and attribute values from the XML docu-

ment prior to running experiments. Most of the compressors mentioned above are mainly concerned with compressing those text values which we ignore in this work. Moreover, SCMPPM and SCMHuff use different PPM models and Huffman trees, resp., for the text within different XML tags. But this only causes space overhead, when text values are removed from the documents. This partly explains the poor compression ratio of SCMPPM and SCMHuff in our experiments.

We implemented a memory mapping of our grammars which allows to traverse the original tree, (essentially) without decompression of the grammar. This mapping offers a good trade-off between space and time: the (iterative) tree traversal speed over TreeRePair grammars is 14 times slower than over pointer structures and 5 times slower than over succinct trees [20], while memory consumption is $1/43$ and $1/6$, respectively. Compared to DAGs and BPLEX, traversals over TreeRePair grammars are approximately 4-times as fast and about 30% slower, while using $1/7$ and $1/2$ of the memory, respectively.

Let us remark that TreeRePair is not restricted to XML tree structure compression. It is a general purpose compressor for ordered trees, which works well if the input tree has a regular structure. Due to space restrictions, most concepts in this paper are only introduced informally, see the full version [14] for more details. The source code of our implementation is available (under GPL 3.0 license) at http://code.google.com/p/treerepair.

2 TreeRePair

We work over ordered, ranked trees (or equivalently terms) in which each node is labeled by a symbol a of fixed rank $\mathsf{rank}(a)$ (equal to the number of children of the node). In fact, we use binary trees to represent XML structure trees: the left child represents the first child and the right child the next sibling; this is called the *first-child/next-sibling encoding*. The TreeRePair algorithm consists of two steps: *digram replacement* and *pruning*. A *digram* is a triple $\alpha = (a, i, b)$, where a and b are node labels and i is a number. It denotes an edge between an a labeled node and its b-labeled i-th child. For instance, the tree $t_1 = f(a, f(c, f(a, f(a, b))))$ contains the digrams $(f, 1, a)$, $(f, 2, f)$, $(f, 1, c)$, and $(f, 2, b)$. The node v of the tree t is an *occurrence of the digram* $\alpha = (a, i, b)$ if v is labeled a and v's i-th child is labeled b. For instance, for t_1, the occurrences of $(f, 2, f)$ are ε, 2, and 2.2, where tree nodes are denoted by their Dewey numbers. Two occurrences of α are *overlapping*, if they share a common tree node; this is only possible if $a = b$. A set of occurrences of α in t is *overlapping*, if it contains two overlapping occurrences. For instance, in t_1, $\{\varepsilon, 2\}$ is overlapping for $(f, 2, f)$. A non-overlapping set of occurrences of α in t can be computed in time $O(|t|)$: during a post-order traversal through t we keep a set S of non-overlapping occurrence of α and add a new occurrence to S if it does not overlap with any previous occurrence in S. We denote this set of occurrences by $\mathsf{occ}_t(\alpha)$; it is easily seen to be maximal (w.r.t. cardinality) among all non-overlapping sets of occurrences of α.

2.1 Replacement of digrams

In an SLT grammar, the digram $\alpha = (a, i, b)$ is represented by a nonterminal A with production $A(y_1, \ldots, y_k) \to a(y_1, \ldots, y_{i-1}, b(y_i, \ldots, y_{j-1}), y_j, \ldots, y_k)$, where $j = i + \mathsf{rank}(b)$. We denote the right-hand side of this production by t_α. The number k equals $\mathsf{rank}(a) + \mathsf{rank}(b) - 1$ and is called the *rank* of α. It denotes the number of "dangling edges" of α. In A's production, the j-th dangling edge, from left-to-right, is denoted by the parameter y_j. Since the rank of a grammar can influence algorithms that execute over the grammar, TreeRePair takes as input a user-defined number, the maximal rank m, and produces a grammar of rank $\leq m$. We often omit the parameters in the left-hand side of productions and simply write $A \to t_\alpha$.

We describe a run of TreeRePair on input tree t by a sequence of SLT tree grammars $\mathcal{G}_0, \mathcal{G}_1, \ldots, \mathcal{G}_h$. The first grammar $\mathcal{G}_0$ has exactly one production: $S_0 \to t$. For a grammar $\mathcal{G}_j$ with start production $S \to s$ we choose a digram $\alpha = (a, i, b)$ of rank $\leq m$ for which $|\mathsf{occ}_s(\alpha)|$ is maximal (and at least two) among all digrams of rank $\leq m$. If multiple such α exist, then the algorithm has to choose one of them for replacement. Our implementation maintains a list of digrams with maximal $|\mathsf{occ}_s(\alpha)|$-value, and takes the first one in this list for replacement. In the examples that follow we choose (for simplicity) the first one in pre-order of s. If $|\mathsf{occ}_s(\alpha)| \leq 1$ for all digrams α, then the replacement phase terminates and $h = i$. Otherwise, we construct $\mathcal{G}_{j+1}$ from $\mathcal{G}_j$ by changing the start production to $S \to s'$ and adding the production $A \to t_\alpha$, where A is a new nonterminal not present in $\mathcal{G}_j$. The tree s' is obtained from s by replacing each occurrence $v \in \mathsf{occ}_s(\alpha)$ of the digram α by the nonterminal A. This means that in s', v is labeled A and has the following subtrees from left to right: the first $(i-1)$ subtrees of v from s, followed by the subtrees of the i-th child of v from s, followed by the $(i+1)$-th to last subtree of v from s.

The SLT grammar $\mathcal{G}_h$ is only an intermediate result, because it potentially consists of productions which do not contribute to a compact representation of the input tree t. Such productions are removed through *pruning*.

2.2 Pruning

The size $|t|$ of a tree t is its number of edges, and the size $|\mathcal{G}|$ of an SLT grammar $\mathcal{G}$ is the sum of sizes of its productions' right-hand sides. We denote by $\mathsf{ref}_\mathcal{G}(A)$ the number of occurrences of the nonterminal A in the right-hand sides of the productions of $\mathcal{G}$. We define for a production $A \to s$ its *save-value*, denoted by $\mathsf{sav}_\mathcal{G}(A)$, as $|\mathsf{ref}_\mathcal{G}(A)| \cdot (|s| - \mathsf{rank}(A)) - |s|$, which can be negative. This is the number of edges that is "saved" by the production $A \to s$. It is simple to eliminate A from the SLT grammar $\mathcal{G}$: remove A's production $A \to s$ and replace every occurrence of A in the remaining productions by s.

In a first pruning phase, we eliminate every nonterminal A with $|\mathsf{ref}_\mathcal{G}(A)| = 1$. This possibly reduces the size of $\mathcal{G}$ (because we have $\mathsf{sav}_\mathcal{G}(A) = -\mathsf{rank}(A)$) and also decrements the number of nonterminals. See the example in the introduction where B_2 and B_1 are removed which reduces the size of the grammar by one.

A nonterminal A is called *inefficient*, if $\mathsf{sav}_\mathcal{G}(A) \leq 0$. The goal of the second pruning phase is to eliminate inefficient nonterminals. This turns out to be a rather complex optimization problem, because the save-values of nonterminals may increase when elim-

inating other nonterminals. For instance, consider the SLT grammar $\mathcal{G}_1$ with productions $\{S \to f(A(a,a), B(A(a,a))), A \to f(B(y_1), y_2), B \to f(y_1, a)\}$. Hence, $\mathsf{sav}_{\mathcal{G}_1}(A) = -1$ and $\mathsf{sav}_{\mathcal{G}_1}(B) = 0$. Case (1): We eliminate A and obtain the grammar $\mathcal{G}_2 = \{S \to f(f(B(a), a), B(f(B(a), a))), B \to f(y_1, a)\}$ of size 11. Now, we have $\mathsf{sav}_{\mathcal{G}_2}(B) = 1$. Case (2): We eliminate B and obtain the grammar $\{S \to f(A(a,a), f(A(a,a), a)), A \to f(f(y_1, a), y_2)\}$. We further eliminate A because its save-value is now zero. The resulting grammar consists of the single production $S \to f(f(f(a,a),a), f(f(f(a,a),a),a))$, and its size is 12. This case distinction shows that the order in which inefficient productions are eliminated influences the size of the final grammar.

An SLT grammar defines a *hierarchical order* on its nonterminals: if a nonterminal Y appears in X's production, then Y follows X in this order. Our heuristic for eliminating inefficient nonterminals is to follow their *reverse hierarchical order*. Note that in the example above, this heuristic leads to the larger grammar. In fact our example might suggest that a better compression ratio is obtained by eliminating inefficient nonterminals in order of increasing save-values. However, our experiments showed that this strategy leads to unappealing final grammars on our test corpus: the final grammars exhibit nearly the same number of edges but many more nonterminals (about 50% more) when compared to our "reverse hierarchical order heuristic". Note that it is not possible to already detect digrams leading to inefficient productions during the replacement step, since the value $\mathsf{ref}_{\mathcal{G}}(A)$ may grow during the replacement step.

2.3 Influence of the maximal rank m

As mentioned before, the user of TreeRePair may specify the maximal number m of parameters in the output grammar as a parameter. In this section, we show that the size of the generated grammar subtly depends on the choice of m. We present two examples for tree families. In the first example, setting $m = \infty$ leads to to the best compression ratio, whereas in the second example setting $m = 1$ leads to the best compression ratio.

Example 1: Let t_i be a full binary tree of height 2^i with each inner node labeled f, and each leaf labeled by a distinct symbol. As an example, consider t_3. It has 2^8-many leaves. We first replace the digram $(f, 1, f)$ and introduce the production $A_1 \to f(f(y_1, y_2), y_3)$. Now the most frequent digram is $(A_1, 2, f)$, which appears as many times as $(f, 1, f)$ did before. We replace it by A_2 and add the production $A_2 \to A_1(y_1, y_2, f(y_3, y_4))$. This production is changed in the pruning step to $A_2 \to f(f(y_1, y_2), f(y_3, y_4))$, because A_1 is only referenced once. The right-hand side of the start production is now a full 4-ary tree of height 4. Finally, we obtain a 16-ary tree with two levels of A_6-labeled inner nodes. In this tree, no digram appears more than once. The size of this final grammar is 298. It can be shown that limiting the maximal number of parameters in TreeRePair over t_i with $i > 3$ always produces larger grammars than with unlimited rank. For instance, on t_3, limiting the rank to $m = 4$ results in a grammar of size 346.

Example 2: Let s_n be a right comb of f's of the form $f(a, f(b, f(c, f(d, f(e, f(a, f(b, \dots)$ of height 2^n. Thus, the sequence of leaf labels is a repetition of $abcde$ of length $2^n + 1$. For $m = \infty$, TreeRePair first replaces the digram $(f, 2, f)$ by A_1. Next, it replaces $(A_1, 2, A_1)$ by A_2 of rank 5, etc. For instance, for s_4, TreeRePair produces after three iterations a

grammar with start production $S \to A_3(a, b, c, d, e, a, b, c, A_3(d, e, a, b, c, d, e, a, b))$. In fact, for any s_n, the right-hand side of the start production of the final grammar contains all initial leaf labels. Thus, it is of size at least 2^n. In contrast, if we limit the maximal rank to $m = 1$, then in the first two iterations TreeRePair replaces the digrams $(f, 1, a)$ and $(f, 1, b)$, resulting in the tree $A_1(A_2(f(c, f(d, f(e, A_1(A_2(f(c, \ldots)$. Now $(A_1, 1, A_2)$ is replaced, resulting in $A_3(f(c, f(d, f(e, A_3(f(c, \ldots)$. In the next two iterations, $(f, 1, c)$, and $(f, 1, d)$ are replaced. This process continues (when starting with the tree s_n) for $O(n)$ iterations and results in a tree of size $O(1)$. Hence, the size of the final grammar is $O(n)$, i.e., is logarithmic in $|s_n|$.

2.4 Implementation details

In order to save memory, we transform, on the fly during parsing, the given input tree into the minimal DAG for the binary first-child/next-sibling encoding of the input tree. In [4] it has been demonstrated that for common XML tree structures, the minimal (unranked) DAG has $\approx 10\%$ of the original tree's edges, (in [5] the average over their 13 documents is 11.2%, as compared to 17% for binary DAGs). Moreover, transforming the initial tree into the minimal DAG for the binary first-child/next-sibling encoding saves CPU time as well, because repetitive computations of digrams are avoided.

The data structures we use in TreeRePair for the replacement of digrams are similar to those used in [10] — mainly hash tables, doubly linked lists, and priority queues — and ensure a linear running time of the replacement step. The crucial point is that if we replace a diagram then only the occurrence numbers of diagrams overlapping the replaced diagrams change. Again, details can be found in the long version [14]. CluX, which is also based on the RePair string compressor, differs from TreeRePair in several important implementation details. To the knowledge of the authors, it does not use pruning. Moreover, CluX splits the input tree into several packages, which arise from the DAG structure of the input tree. For each of these packages a separate grammar is generated. This excludes the possibility of exploiting dependencies between different packages for compression. We conjecture that these differences are responsible for the poorer compression ratio of CluX, see Section 3.

2.5 Succinct grammar coding

In order to get a compact representation of XML structure trees, we further compress the generated SLT grammar by a binary succinct coding. The technique we use is loosely based on the DEFLATE algorithm described in [9]. In fact, we use a combination of a fixed-length coding, multiple Huffman codings, and a run-length coding to encode different aspects of the grammar. We encode all symbols (terminal symbols and nonterminal symbols) of the generated grammar by integers. Since the parameters always occur in the order $y_1, y_2, \ldots$ in right-hand sides, it suffices to use one fixed place holder for parameters. Element names of the input XML tree structure become terminal symbols of our tree grammar. Since under the first-child/next-sibling encoding of unranked trees, a symbol can have (i) no children, (ii) only a left child, (iii) only a right child, or (iv) both a left and a right child, each element type corresponds to four terminal symbols; one for each of the

four possibilities (i)-(iv). These four terminal symbols are distinguished by two additional bits. We obtained the best compression ratio by using four different Huffman encodings for different parts of the grammar. Three of them encode (a) the right-hand side of the start production, (b) the remaining productions, the children characteristics of the terminal symbols (i.e., which of the above 4 possibilities (i)-(iv) holds), and the number of terminal and nonterminals, and finally (c) the names (element types) of the terminals. Moreover, the three Huffman trees for these encodings (the base Huffman encodings) are encoded by a fourth Huffman encoding (the super Huffman encoding). As explained in [9], it is sufficient to only write out the lengths of the generated Huffman codes to be able to reconstruct the actual Huffman trees. The code lengths for the three base Huffman encodings are further encoded using a run-length encoding, see [14] for more details.

An interesting aspect of the succinct encoding is that smaller file sizes are obtained if we eliminate in the pruning step nonterminals with a save-value ≤ 2 (instead of ≤ 0, which yields minimal edge numbers). In the implementation, this modification of the pruning step is triggered by the switch "-optimize filesize".

3 Experiments

We conduct three types of experiments: we compare our implementation of TreeRePair to other SLT grammar based compressors, to other XML file compressors, and we compare traversal speeds over memory representations of the generated grammars.

Testing environment: Our experiments were performed on a machine with an Intel Core2 Duo CPU T9400 processor, four gigabytes of RAM, and the Ubuntu Linux operating system, kernel 2.6.32. TreeRePair and BPLEX were compiled with version 4.4.3 of `gcc` using the "-O3" (compile time optimizations) and "-m32" (i.e., we generated them as 32bit-applications) switches. We were not able to compile the `succ`-tool of the BPLEX distribution with compile time optimizations (i.e., using the "-O3" switch). This tool is used to apply a succinct coding to a grammar generated by the BPLEX algorithm, as described in [16]. However, this has no great influence on the runtime for BPLEX since the `succ`-tool executes quite fast compared to BPLEX. In contrast, CluX is an application written in Java for which we only had the bytecode at hand. We executed CluX using the Java SE Runtime Environment™ , version 1.6.0_20. We measure memory usage by constantly polling the `VmRSS`-value under Linux. We tested over 24 different XML documents, most of which are known from previous articles about XML compression. For all tests, we first remove all text contents from each document and only keep the start and end element tags (and empty element tags), thus obtaining "stripped" documents. Table 1 shows the characteristics of the 24 stripped XML documents.

Comparing tree grammar compressors: We compared BPLEX [5] (obtained from http://sourceforge.net/projects/bplex), CluX [3] (supplied to us by the authors), DAG and bDAG, see e.g., [4] (included e.g. at http://sourceforge.net/projects/bplex), each of which produces SLT grammars. The latter two generate minimal DAGs, either on the unranked XML tree (DAG), or on the binary first-child/next-sibling encoded XML tree (bDAG), cf. [5] where it was observed already that DAG gives better compression ratios

XML document	File size (KB)	# Edges	Depth	# Element types	Source (http://. . .)
1998statistics	341	28 305	5	46	www.cafeconleche.org/examples
catalog-01	4 120	225 193	7	50	softbase.uwaterloo.ca/~ddbms/projects/xbench
catalog-02	43 609	2 390 230	7	53	softbase.uwaterloo.ca/~ddbms/projects/xbench
dictionary-01	1 696	277 071	7	24	softbase.uwaterloo.ca/~ddbms/projects/xbench
dictionary-02	16 726	2 731 763	7	24	softbase.uwaterloo.ca/~ddbms/projects/xbench
dblph20090930	254 866	24 211 585	5	36	dblp.uni-trier.de/xml
EnWikiNew	4 729	404 651	4	20	download.wikipedia.org/backup-index.html
EnWikiQuote	3 060	262 954	4	20	download.wikipedia.org/backup-index.html
EnWikiSource	13 141	1 133 534	4	20	download.wikipedia.org/backup-index.html
EnWikiVersity	5 748	495 838	4	20	download.wikipedia.org/backup-index.html
EnWikTionary	96 876	8 385 133	4	20	download.wikipedia.org/backup-index.html
EXI-Array	5 221	226 522	9	47	www.w3.org/XML/EXI
EXI-factbook	1 185	55 452	4	199	www.w3.org/XML/EXI
EXI-Invoice	260	15 074	6	52	www.w3.org/XML/EXI
EXI-Telecomp	3 613	177 633	6	39	www.w3.org/XML/EXI
EXI-weblog	1 078	93 434	2	12	www.w3.org/XML/EXI
JST_gene.chr1	4 103	216 400	6	26	snp.ims.u-tokyo.ac.jp
JST_snp.chr1	13 471	655 945	7	42	snp.ims.u-tokyo.ac.jp
medline02n0328	50 537	2 866 079	6	78	www.ncbi.nlm.nih.gov/pubmed
NCBI_gene.chr1	6 701	360 349	6	50	snp.ims.u-tokyo.ac.jp
NCBI_snp.chr1	62 442	3 642 224	3	15	snp.ims.u-tokyo.ac.jp
sprot39.dat	108 569	10 903 567	5	48	expasy.org/sprot
treebank	19 092	2 447 726	36	251	www.cs.washington.edu/research/xmldatasets
xmlgen_f10	205 953	16 703 209	11	74	www.xml-benchmark.org/generator.html

Table 1: Characteristics of our XML corpus

than bDAG. Note that DAGs and bDAGs can bee seen as SLT grammars of rank zero — nodes of the (b)DAG correspond to nonterminals of the grammar (for DAGs, different copies of a symbol have to be introduced, in case that symbol occurs with different ranks). We compare the grammars produced by the different compressors in terms of numbers of edges and numbers of nonterminals. TreeRePair was run with "-optimize edges", which sets the maximal number of parameters m to 4 (see the remarks below), CluX was run with configuration "ConfEdges.xml", and BPLEX was run with its standard parameters (and the gprint tool was used to eliminate nonterminals that are referenced only once). The results of our experiments are shown in Table 2 (all values are averages over our XML corpus). Note that TreeRePair yields the best results w.r.t. compression ratio (in terms of number of edges), running time, and memory consumption. Recall that the number of edges of a grammar is the number of edges in all right-hand sides. It is interesting to note that on our XML corpus, TreeRePair achieves the best compression ratio with value $m = 4$ for the maximal number of parameters. Our two examples from Section 2.3 offer a possible explanation for this fact: for deep binary trees, $m = \infty$ is the best choice, whereas for comb-like trees (or long lists), a small m-value is a better choice. The binary first-child/next-sibling encodings of real XML document trees tend to be closer to the latter shape. As can be seen from Table 1, our XML documents are quite flat (depth ≈ 5), which results in long combs under the first-child/next-sibling encoding. On the other hand, this is not true for treebank, which has a depth of 36. Indeed it turns out that for treebank $m = 42$ yields the best compression ratio (20.445 % in contrast to 20.719 % for $m = 4$ — this is also the worst compression ratio on our XML corpus).

Comparing XML file compressors: Besides the above mentioned tree grammar compressors, we consider the following XML file compressors: XMill in version 0.8 with bzip2 as backend compressor [11], XMLPPM in version 0.98.3 [7], see http://xmlppm.

	TreeRePair	BPLEX	CluX	DAG	bDAG
Edges (%)	2.8	3.5	4.4	12.7	18.2
# NTs	6 715	32 159	12 133	4 635	8 560
Time (seconds)	19	934	101	15	9
Memory (MB)	72	550	395	123	59

Table 2: Performance of different tree grammar compressors

sourceforge.net/, SCMPPM [1], see http://www.infor.uva.es/~jadiego/download.php, and SCHuff [1] (an implementation was kindly provided to us by the authors). As a yardstick we also include numbers for the general purpose compressors gzip and bzip2 in the comparison. TreeRePair was run with "-optimize filesize", which generates a succinct grammar encoding, as described in Section 2.5. CluX was run with configuration "Conf-Size.xml" and the "-s 4" switch. For BPLEX we used gprint with "--threshold 14" and the succ-tool with "--type 68", which generates a Huffman-based coding that was reported to give the smallest output files [16]. Table 3 shows the outcomes of our experiments. As "Memory" we show the ratio of the program's peak memory usage over the size of the original file. Thus, on average, TreeRePair's memory consumption is 2.4-times the size of the markup file. As can be seen, only XMLPPM achieves a slightly better compression ratio than TreeRePair. A disadvantage of XMLPPM is that due to the adaptive nature of the PPM algorithm, traversing the XML tree structure is not possible on the compressed document. The latter has to be fully decompressed, see also [1]. The same holds for SCMPPM, gzip, and bzip2. In contrast, navigating in the XML tree structure only needs additional space $O(\text{depth of the grammar})$ on SLT grammar compressed trees using the stack configurations from [5], see also the next paragraph.

	TreeRePair	BPLEX	CluX	XMill	XMLPPM	SCMPPM	SCMHuff	gzip	bzip2
File size (%)	0.44	0.59	0.63	0.49	0.41	0.74	4.39	1.41	0.60
Time (seconds)	19	946	296	128	4	6	16	1	25
Memory (peak/orig)	2.4	60.3	115	1.0	0.4	0.4	0.3	0.1	2.1

Table 3: Performance for XML file compression

Comparing grammar traversal speeds: In order to achieve fast tree traversals, we map the output grammar of TreeRePair into memory as follows: the initial right-hand side of the grammar is represented using an implementation of Sadakane and Navarro's succinct trees [20] (for moderate-size trees), which was generously supplied to us by Sadakane. The rest of the grammar is transformed into Chomsky normal form (so that every right-hand size has precisely two non-parameter symbols) and each such production is represented by a single 64-bit machine word. We then perform an iterative pre-order traversal through the tree represented by the grammar, using down_1 (go to first child), down_2 (go to second child), and up (go to parent) over nodes represented by the stack configurations mentioned after Theorem 3 in [5]. We calculated the size of the grammar memory representation (called *index size* in Table 4) and measured traversal times of our implementation. As a yardstick, we also give these values for the succinct trees of [20], and for a simple pointer-based representation, where each tree node has three machine pointers to its parent, first, and second child. Note that for arbitrary root-node path traversals, machine pointers are *much* faster (about 100 times) than succinct trees [2], which in turn are

	TreeRePair	BPLEX	bDAG	Succinct	Pointer
Traversal speed (ms)	771	597	3 220	164	56
Index size (KB)	463	794	3 070	2 724	19 995

Table 4: Speeds for iterative pre-order traversals

faster than our grammar compressed trees. Table 4 shows the results of our experiments. Our comparison does not include CluX, since its output (consisting of a separate grammar for each package) cannot be processed by our tool for measuring the traversal speed.

References

[1] J. Adiego, G. Navarro, and P. de la Fuente. Using structural contexts to compress semistructured text collections. *Inf. Process. Manage.*, 43(3):769–790, 2007.

[2] D. Arroyuelo, R. Cánovas, G. Navarro, and K. Sadakane. Succinct trees in practice. In *ALENEX 2010*, 84–97.

[3] S. Böttcher, R. Hartel, and C. Krislin. CluX: Clustering XML sub-trees. In *ICEIS 2010*.

[4] P. Buneman, M. Grohe, and C. Koch. Path queries on compressed XML. In *VLDB 2003*, 141–152.

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

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

[7] J. Cheney. Compressing XML with multiplexed hierarchical PPM models. In *DCC 2001*, 163–172.

[8] F. Claude and G. Navarro. Self-indexed grammar-based compression. To appear in *Fund. Inform.*.

[9] P. Deutsch. DEFLATE compressed data format specification version 1.3. http://tools.ietf.org/html/rfc1951, 1996.

[10] N. J. Larsson and A. Moffat. Offline dictionary-based compression. In *DCC 1999*, 296–305.

[11] H. Liefke and D. Suciu. XMill: An efficient compressor for XML data. In *SIGMOD Conference 2000*, 153–164.

[12] Y. Lifshits. Processing compressed texts: A tractability border. In *CPM 2007*, 228–240.

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

[14] M. Lohrey, S. Maneth, and R. Mennicke. Tree structure compression with RePair. Technical report, arXiv.org, 2010.

[15] M. Lohrey, S. Maneth, and M. Schmidt-Schauß. Parameter reduction in grammar-compressed trees. In *FOSSACS 2009*, 212–226.

[16] S. Maneth, N. Mihaylov, and S. Sakr. XML tree structure compression. In *DEXA Workshops 2008*, 243–247.

[17] C. G. Nevill-Manning, I. H. Witten, and D. Maulsby. Compression by induction of hierarchical grammars. In *DCC 1994*, 244–253.

[18] W. Plandowski. Testing equivalence of morphisms on context-free languages. In *ESA 1994*, 460–470.

[19] W. Rytter. Grammar compression, LZ-encodings, and string algorithms with implicit input. In *ICALP 2004*, 15–27.

[20] K. Sadakane and G. Navarro. Fully-functional succinct trees. In *SODA 2010*, 134–149.