XML Tree Structure Compression using RePair

Markus Lohrey[a], Sebastian Maneth[b,1], Roy Mennicke[c]

[a] *Universität Leipzig, Institut für Informatik, Germany*
[b] *University of Oxford, Department of Computer Science, UK*
[c] *Technische Universität Ilmenau, Institut für Theoretische Informatik, Germany*

Abstract

XML tree structures can conveniently be represented using ordered unranked trees. Due to the repetitiveness of XML markup these trees can be compressed effectively using dictionary-based methods, such as minimal directed acyclic graphs (DAGs) or straight-line context-free (SLCF) tree grammars. While minimal SLCF tree grammars are in general smaller than minimal DAGs, they cannot be computed in polynomial time unless $\mathsf{P} = \mathsf{NP}$. Here, we present a new linear time algorithm for computing small SLCF tree grammars, called TreeRePair, and show that it greatly outperforms the best known previous algorithm BPLEX. TreeRePair is a generalization to trees of Larsson and Moffat's RePair string compression algorithm. SLCF tree grammars can be used as efficient memory representations of trees. Using TreeRePair, we are able to produce the smallest queryable memory representation of ordered trees that we are aware of. Our investigations over a large corpus of commonly used XML documents 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. With respect to file compression we are able to show that a Huffman-based coding of TreeRePair grammars gives compression ratios comparable to the best known XML file compressors.

Key words: XML, tree structure compression, memory representation

Email addresses: lohrey@informatik.uni-leipzig.de (Markus Lohrey), sebastian.maneth@cs.ox.ac.uk (Sebastian Maneth), roy.mennicke@tu-ilmenau.de (Roy Mennicke)

[1] This work was produced while the author was affiliated to NICTA (Sydney) and to the University of New South Wales, Australia

1. Introduction

XML tree structures can conveniently be represented using node-labeled ordered unranked trees. Here, "ordered" means that the children of a node are ordered from left to right, and "unranked" means that the number of children of a node is not determined by the node label. Due to the repetitiveness of the markup in XML documents, these tree structures can be effectively compressed using dictionary based methods. A standard tool to compress trees are directed acyclic graphs (DAGs). In contrast to a tree, a node of a DAG may have several incoming edges, i.e., there is no unique parent node. In this way, subtrees that occur several times in the original tree have to be represented only once. It was shown in [6] that the minimal DAG (minimal w.r.t. the number of edges) of the structure tree of a typical XML document consists of only about 10% of the original number of edges of the tree. Minimal DAGs have many appealing features, e.g., they are unique, can be computed in linear time [13], and allow for constant time subtree equality check. On the other hand, their sharing ability is limited to complete subtrees of the given tree.

A generalization of DAGs from the sharing of repeated subtrees to the sharing of repeated tree patterns (that is, connected subgraphs of the tree) are straight-line context-free (SLCF) tree grammars, initiated in [27, 35]. An SLCF tree grammar is a context-free tree grammar, see e.g. [11], which is acyclic (from a nonterminal A, one cannot derive a tree containing A) and has exactly one defining production per nonterminal. SLCF tree grammars generalize Plandowski's singleton context-free grammars [32] (also known as straight-line programs) from strings to trees. SLCF tree grammars can be (at most) exponentially smaller than DAGs. On the negative side, a minimal SLCF tree grammar for a given tree is in general not unique and cannot be computed in polynomial time unless $\mathsf{P} = \mathsf{NP}$ (this is known already for the string case [9]). The first algorithm for computing a small linear SLCF tree grammar for a given input tree was BPLEX [7]. BPLEX repeatedly searches in a fixed window for the optimal pattern to share. It was shown in [7] that BPLEX compresses equally or better on binary tree representations (i.e., first-child/next-sibling encodings, see e.g. [38]) than on unranked trees. Note that this is not true for minimal DAGs of XML document trees: they are typically smaller in the unranked case [7, 25].

Here, we introduce an efficient algorithm called "TreeRePair" for computing small linear SLCF tree grammars. We show that our linear time

implementation of TreeRePair is superior to BPLEX in many aspects: it runs faster (by a factor of around 50), needs less memory, produces smaller grammars, and is easier to understand. As far as we know, the grammars produced by TreeRePair give rise to the smallest existing memory representation of ordered trees; see also [29] where an XML self-index using TreeRePair grammars is presented. TreeRePair is a generalization of Larson and Moffat's RePair for strings [19]. Their idea is to replace in a given string all (non-overlapping) occurrences of a most frequent *digram* (= two consecutive letters) by a new nonterminal. This process is repeated until no digram occurs more than once. It naturally gives rise to a linear SLCF tree grammar (even with all non-start productions in Chomsky normal form). In TreeRePair a digram consists of a triple (f, i, g) where f and g are node labels and i is a natural number. An occurrence of the digram (f, i, g) in a given tree consists of a node labeled f together with its i-th child node labeled g. Similar as with BPLEX, TreeRePair compresses better on the binary encoded XML document trees than on unranked trees.

Let us consider in more detail the replacement of all occurrences of the digram $(f, 2, g)$ by the new nonterminal X in a given binary tree t. If both f and g are binary, then every occurrence of the digram has 3 outgoing edges (one for the f-labeled node and two for the g-labeled node). Hence, removing an occurrence of the digram $(f, 2, g)$ from the tree leaves three dangling subtrees. Therefore the nonterminal X must be of rank 3, which means that every X-labeled node has three children. In an SLCF tree grammar, formal parameters $y_1, y_2, \ldots$ are used to indicate where dangling subtrees are to be inserted. Thus, the production of X has the form

$$X(y_1, y_2, y_3) \to f(y_1, g(y_2, y_3)).$$

In the next round of replacement, if the digram $(X, 1, X)$ is replaced, then a new nonterminal Y of rank 5 is introduced. The complexity of several algorithms on SLCF tree grammars (such as evaluating tree automata) depend on the number of parameters in the grammar [22]. For this reason, BPLEX is controlled by a user-defined `max_rank` constant and produces grammars with at most `max_rank` many parameters per nonterminal. In TreeRePair, we realize such a constant by only counting digrams which give rise to nonterminals of rank at most `max_rank`. Interestingly, the choice of `max_rank` drastically influences the compression ratio of TreeRePair. We show:

1. There is a family of trees on which TreeRePair with `max_rank`=1 produces grammars that are exponentially smaller than those produced by

TreeRePair with `max_rank`=unbounded.

2. For every k there is a family of trees on which the TreeRePair algorithm with `max_rank` $\geq k$ produces grammars that are exponentially smaller than those produced by TreeRePair with `max_rank` $< k$.

Fortunately, typical XML (binary) document trees do not depend much on `max_rank`, and setting this constant to 4 gives the smallest grammars (in terms of number of edges) for our XML corpus.

We also experimented with using the grammars produced by TreeRePair as a means for XML file compression. For this, we use a layered Huffman-based coding as in DEFLATE [12]. We show that the achieved compression is competitive with the best known XML file compressors, and that only a single compressor (XMLPPM [10]) compresses slightly better than TreeRePair (to about 0.41% of the original file sizes, as compared to 0.44% with TreeRePair).

Technically speaking, the linear running time of our implementation of TreeRePair is achieved by following the ideas of RePair for strings [19] by using a priority queue of length $\sqrt{n}$ (where n is the number of digram occurrences) and updating this queue in constant time for each round of replacement.

An extended abstract of this paper appeared as [23] and a preliminary conference version was published as [24].

Related work

Independently to our work the tree compressor CluX (for *Clustering XML-subtrees*) was developed at the University of Paderborn [18, 5]. CluX is also based on the RePair string compression algorithm but exhibits some fundamental differences to TreeRePair. One of the main differences is that CluX first generates a DAG of the input tree and then processes each part (called "repair packets") of it individually, i.e., it generates multiple grammars which are combined in the end. The maximal size of each repair packet can be specified by an input parameter. The author of [18] points out that this packet-based behavior may have a negative impact on the compression performance of CluX. Our own investigations concerning a TreeRePair version running on the DAG (in one part) of the input tree support this point of view.

Several other types of tree grammars were proposed in the literature for the purpose of tree compression: elementary ordered tree grammars [2] (were

a tree version of the bisection algorithm for strings was developed), variable replacement grammars [39], and higher-order tree grammars [17]. As mentioned at the beginning of the Introduction, DAGs (which are a special case of the grammars considered in this paper) have been applied to XML document trees in [6]. Recently, DAG variations which can give rise to stronger compression than usual DAGs have been considered: top-trees [4] and hybrid DAGs [25]. In [25] a compression combining (binary) DAGs and the string repair of [19] is considered and compared directly to TreeRePair.

Algorithmic problems for trees that are represented by SLCF tree grammars have been thoroughly studied. In [22] various membership problems for SLCF-compressed trees and tree automata were considered. It was shown that in many cases the same complexity bounds hold as for DAGs. In particular, it was pinpointed that for a given nondeterministic tree automaton $\mathcal{A}$ and a linear, k-bounded (meaning that every nonterminal has rank at most k) SLCF tree grammar $\mathcal{G}$ it can be checked in polynomial time if the tree represented by $\mathcal{G}$ is accepted by $\mathcal{A}$ – provided that k is a constant. This is important because in the context of XML, for instance, tree automata are used to type check XML documents against an XML schema (cf. [30, 31]). The result was further improved in [26], where it was shown that every linear SLCF tree grammar can be transformed in polynomial time into a linear monadic (= 1-bounded) one. Together with the above mentioned result from [22], a polynomial time algorithm for testing if a given nondeterministic tree automaton accepts a tree given by a linear SLCF tree grammar is obtained. In [22] it was proved that the evaluation problem of core XPath (the navigational part of XPath) over linear SLCF tree grammars is PSPACE-complete, which is the same complexity as for DAGs [14]. The evaluation problem for XPath asks whether a given node in a given tree is selected by a given XPath expression. This result is remarkable since SLCF tree grammars achieve higher compression ratios than DAGs. In [29] an evaluator for simple XPath expressions is investigated which runs over compact memory representations of XML, based on SLCF tree grammars; the demonstrated space requirement for the tree structures is indeed tiny: for an XMark document [34] of half a billion nodes, a single bit of storage accommodates more than 8 XML tree nodes.

Several other algorithmic problems on SLCF tree grammars that are related to term rewriting (e.g., unification) were studied in [8, 15, 20, 36, 37].

2. Preliminaries

In the following, $\mathbb{N}_+ = \mathbb{N} \setminus \{0\}$ denotes the set of non-zero natural numbers. For a set X, we denote by X^* the set of all finite words over X. For $w = x_1 x_2 \dots x_n \in X^*$, we define $|w| = n$. The empty word is denoted by ε. For readability, we sometimes surround an element of $\mathbb{N}$ by square brackets, e.g., for the sequence 222221, we write $[2]^5[1]$ instead of $2^5 1$ to clarify that we are not dealing with the fifth power of 2.

In the following two subsections, we introduce several formal definitions concerning trees and tree grammars. These definitions are needed to formally specify the TreeRePair algorithm. Most of our notations are standard, see e.g. [11].

2.1. Labeled ordered trees

A *ranked alphabet* is a pair $(\mathcal{F}, \mathsf{rank})$, where $\mathcal{F}$ is a finite set of *function symbols* and the function $\mathsf{rank} : \mathcal{F} \to \mathbb{N}$ assigns to each $a \in \mathcal{F}$ a natural number $\mathsf{rank}(a)$ that we call the *rank of* a. Furthermore, we define $\mathcal{F}_i = \{a \in \mathcal{F} \mid \mathsf{rank}(a) = i\}$. We fix a ranked alphabet $(\mathcal{F}, \mathsf{rank})$ in the following. An *$\mathcal{F}$-labeled ordered tree* is a pair $t = (\mathsf{dom}_t, \lambda_t)$ where

(1) $\mathsf{dom}_t \subseteq \mathbb{N}_+^*$ is a non-empty finite set of *nodes*

(2) $\lambda_t : \mathsf{dom}_t \to \mathcal{F}$ is the (total) node labeling function

(3) if $w = vv' \in \mathsf{dom}_t$ for $v, v' \in \mathbb{N}_+^*$, then also $v \in \mathsf{dom}_t$

(4) if $v \in \mathsf{dom}_t$ and $\lambda_t(v) \in \mathcal{F}_n$, then $vi \in \mathsf{dom}_t$ if and only if $1 \leq i \leq n$.

The node $\varepsilon \in \mathsf{dom}_t$ is called the *root* of t. If $vi \in \mathsf{dom}_t$ and $i \in \mathbb{N}_+$, then we say that vi is the i-th *child* of v. The *size* of t is defined as its number of edges, i.e., $|t| = |\mathsf{dom}_t| - 1$. The *depth* of t is $\max\{|u| \mid u \in \mathsf{dom}_t\}$. The set of all $\mathcal{F}$-labeled trees is denoted by $T(\mathcal{F})$. We identify an $\mathcal{F}$-labeled tree t with a term in the usual way: if $\lambda_t(\varepsilon) = a \in \mathcal{F}_i$, then this term is $a(t_1, \dots, t_i)$, where t_j is the term associated with the subtree of t rooted at node j for all $j \in \{1, \dots, i\}$.

Example 1. Figure 1 shows the $\mathcal{F}$-labeled ordered tree $t \in T(\mathcal{F})$ where $\mathsf{dom}_t = \{\varepsilon, 1, 2, 3, 11, 12, 21, 22, 31, 111, 112, 121, 122, 211, 212, \dots\}$, $\lambda_t(\varepsilon) = f$, $\lambda_t(1) = g$, $\lambda_t(2) = g$, $\lambda_t(3) = h$, $\lambda_t(11) = i$, $\lambda_t(12) = i$, $\lambda_t(111) = a$, etc.

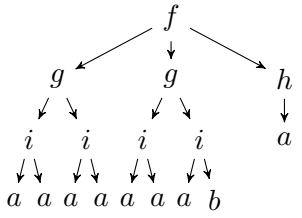

Figure 1: The $\mathcal{F}$-labeled ordered tree t from Example 1.

We fix a countable set $\mathcal{Y} = \{y_1, y_2, \ldots\}$ of *(formal context-) parameters* such that $\mathcal{Y} \cap \mathcal{F} = \emptyset$. Below, we also use a distinguished parameter $z \notin (\mathcal{Y} \cup \mathcal{F})$. The set of all $\mathcal{F}$-labeled trees with parameters from $Y \subseteq \mathcal{Y}$ is denoted by $T(\mathcal{F}, Y)$. Formally, we consider parameters as function symbols of rank 0 and define $T(\mathcal{F}, Y) = T(\mathcal{F} \cup Y)$.

The tree $t \in T(\mathcal{F}, Y)$ is said to be *linear* if every parameter $y \in Y$ occurs at most once in t. For $t \in T(\mathcal{F}, \{y_1, \ldots, y_n\})$ and $t_1, \ldots, t_n \in T(\mathcal{F}, Y)$, we denote by $t[y_1/t_1, \ldots, y_n/t_n]$ the tree that is obtained from t by replacing every y_i-labeled leaf by the tree t_i for $i \in \{1, 2, \ldots, n\}$.

A *context* is a tree $C \in T(\mathcal{F}, \mathcal{Y} \cup \{z\})$ in which the distinguished parameter z appears exactly once. Instead of $C[z/t]$ we write $C[t]$.

Let $t \in T(\mathcal{F}, \{y_1, \ldots, y_n\})$ such that for every y_i there exists a node $v \in \mathsf{dom}_t$ with $\lambda_t(v) = y_i$. We say that t *is a tree pattern occurring in* $t' \in T(\mathcal{F}, \mathcal{Y})$ if there exist a context $C \in T(\mathcal{F}, \mathcal{Y} \cup \{z\})$ and trees $t_1, \ldots, t_n \in T(\mathcal{F}, \mathcal{Y})$ such that $C\big[t[y_1/t_1, y_2/t_2, \ldots, y_n/t_n]\big] = t'$.

2.2. SLCF tree grammars

For further consideration, let us fix a countable infinite set $\mathcal{N}_i$ of symbols of rank $i \in \mathbb{N}$ with $\mathcal{F} \cap \mathcal{N}_i = \mathcal{Y} \cap \mathcal{N}_i = \emptyset$. Hence, every finite subset $N \subseteq \bigcup_{i \geq 0} \mathcal{N}_i$ is a ranked alphabet. A *context-free tree grammar (over the ranked alphabet $\mathcal{F}$)* or short *CF tree grammar* is a triple $\mathcal{G} = (N, P, S)$, where

(1) $N \subseteq \bigcup_{i \geq 0} \mathcal{N}_i$ is a finite set of *nonterminals*,

(2) P (the set of *productions*) is a finite set of pairs $(A \to t)$, where $A \in N$, $t \in T(\mathcal{F} \cup N, \{y_1, \ldots, y_{\mathsf{rank}(A)}\})$, $t \notin (\mathcal{Y} \cup N)$, each of the parameters $y_1, \ldots, y_{\mathsf{rank}(A)}$ appears in t[2], and

[2]In contrast to [26], our definition of a context-free tree grammar inherits productivity,

(3) $S \in N$ is the *start nonterminal* of rank 0.

If $(A \to t) \in P$, then A is called the *left-hand side* and t is said to be the *right-hand side* of $(A \to t)$. We assume that every nonterminal $B \in N \setminus \{S\}$ as well as every terminal symbol from $\mathcal{F}$ occurs in the right-hand side t of some production $(A \to t) \in P$.

If $\mathcal{G} = (N, P, S)$ is a CF tree grammar, then we define the derivation relation $\Rightarrow_{\mathcal{G}}$ on $T(\mathcal{F} \cup N, \mathcal{Y})$ as follows: We have $s \Rightarrow_{\mathcal{G}} s'$ if and only if there exists a production $(A \to t) \in P$ with $\mathsf{rank}(A) = n$, a context $C \in T(\mathcal{F} \cup N, \mathcal{Y} \cup \{z\})$, and trees $t_1, \ldots, t_n \in T(\mathcal{F} \cup N, \mathcal{Y})$ such that $s = C[A(t_1, \ldots, t_n)]$ and $s' = C[t[y_1/t_1 \cdots y_n/t_n]]$. The language of $\mathcal{G}$ is $L(\mathcal{G}) = \{t \in T(\mathcal{F}) \mid S \Rightarrow_{\mathcal{G}}^* t\}$ and its *size* $|\mathcal{G}|$ is defined by

$$|\mathcal{G}| = \sum_{(A \to t) \in P} |t| \,.$$

That means that $|\mathcal{G}|$ equals the sum of the numbers of edges of the right-hand sides of $\mathcal{G}$'s productions.

We consider the following restrictions on context-free tree grammars $\mathcal{G} = (N, P, S)$:

- $\mathcal{G}$ is *k-bounded* (for $k \in \mathbb{N}$) if $\mathsf{rank}(A) \leq k$ for every $A \in N$.
- $\mathcal{G}$ is *monadic* if it is 1-bounded.
- $\mathcal{G}$ is *linear* if for every $(A \to t) \in P$ the tree t is linear.

Let $\mathcal{G} = (N, P, S)$ be a CF tree grammar. Consider the binary relation

$$\rightsquigarrow_{\mathcal{G}} = \{(A, B) \in N \times N \mid (B \to t) \in P \text{ and } A \text{ occurs in } t\}.$$

The *hierarchical order* of $\mathcal{G}$ is the reflexive transitive closure $\rightsquigarrow_{\mathcal{G}}^*$ of $\rightsquigarrow_{\mathcal{G}}$.

A *straight-line context-free tree grammar (SLCF tree grammar)* is a CF tree grammar $\mathcal{G} = (N, P, S)$, where

(1) for every $A \in N$, there is *exactly one* production $(A \to t) \in P$ with left-hand side A, and

---

i.e., $t \notin \mathcal{Y}$ and each parameter $y_1, \ldots, y_{\mathsf{rank}(A)}$ appears in t for every $(A \to t) \in P$. This is justified by the fact that the grammars generated by TreeRePair are always productive.

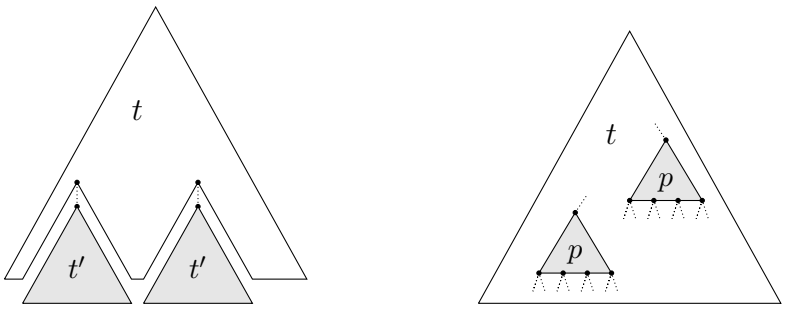

Figure 2: On the left side: A tree t containing two occurrences of the very same subtree t'. On the right side: A tree t containing two occurrences of the tree pattern p.

(2) the relation $\rightsquigarrow_{\mathcal{G}}$ is acyclic.

The conditions (1) and (2) ensure that $L(\mathcal{G})$ contains exactly one tree which we denote by $\mathsf{val}(\mathcal{G})$.

Example 2. Consider the (linear and monadic) SLCF tree grammar $\mathcal{G} = (N, P, S)$ where P is given by the following productions:

$$S \to f\big(A(a), A(b), B\big), \qquad A(y_1) \to g\big(i(a,a), i(a,y_1)\big), \qquad B \to h(a)$$

We have $\mathsf{val}(\mathcal{G}) = t$ where $t \in T(\mathcal{F})$ is the tree from Example 1.

SLCF tree grammars can be considered as a generalization of the well-known DAGs (see, for instance, [22] for a common definition). Whereas the latter is a structure preserving compression of a tree by sharing common subtrees, SLCF tree grammars broaden this concept to the sharing of repeated tree patterns (connected subgraphs of a tree); see Figure 2 for an illustration. Actually, a DAG can be considered as a 0-bounded SLCF tree grammar, where the nonterminals of the grammar correspond to the nodes of the DAG.

Let $\mathcal{G} = (N, P, S)$ be a linear SLCF tree grammar and let $(A \to t) \in P$ be the production for the nonterminal $A \in N \setminus \{S\}$. In order to define a measure of the contribution of $A \to t$ to the size of $\mathcal{G}$, we first need to make the following definition. By $\mathsf{ref}_{\mathcal{G}}(A)$, we denote the set of all pairs (t', v) where t' is a right-hand side of $\mathcal{G}$'s productions and $v \in \mathsf{dom}_{t'}$ is labeled by A. More formally, we define

$$\mathsf{ref}_{\mathcal{G}}(A) = \{(t', v) \mid \text{ there exists } (B \to t') \in P \text{ with } v \in \mathsf{dom}_{t'} \text{ and } \lambda_{t'}(v) = A\}\,.$$

Now, we define the *save value* of A to be

$$\mathsf{sav}_{\mathcal{G}}(A) = |\mathsf{ref}_{\mathcal{G}}(A)| \cdot (|t| - \mathsf{rank}(A)) - |t| \,.$$

Intuitively, $\mathsf{sav}_{\mathcal{G}}(A)$ is A's contribution (in terms of number of edges) to a small representation of the tree $\mathsf{val}(\mathcal{G})$ by the linear SLCF tree grammar $\mathcal{G}$. Note that $\mathsf{sav}_{\mathcal{G}}$ can be negative; for instance, for $(A \to t) \in P$ with $|\mathsf{ref}_{\mathcal{G}}(A)| = 1$ we have $\mathsf{sav}_{\mathcal{G}}(A) = -\mathsf{rank}(A)$. Thus, a production which is only referenced once can be removed without increasing the size of $\mathcal{G}$.

2.3. Digrams

In this section, we introduce the notion of digrams which is central to our TreeRePair algorithm. Recall that we have fixed a ranked alphabet $\mathcal{F}$ of function symbols, a set $\mathcal{N}$ of nonterminals, and a set $\mathcal{Y} = \{y_1, y_2, \ldots\}$ of parameters. A *digram* is a triple $\alpha = (a, i, b)$ where $a, b \in \mathcal{F} \cup \mathcal{N}$ and $1 \leq i \leq \mathsf{rank}(a)$. The symbol a is called the *parent symbol of the digram* α and b is called the *child symbol of the digram* α, respectively. We denote the set of all digrams by Π.

For a digram $\alpha \in \Pi$ we define its rank and pattern as

$$\mathsf{par}(\alpha) = \mathsf{rank}(a) + \mathsf{rank}(b) - 1 \in \mathbb{N}\,,$$
$$\mathsf{pat}(\alpha) = a\big(y_1, \ldots, y_{i-1}, b(y_i, \ldots, y_{j-1}), y_j, \ldots, y_{\mathsf{par}(\alpha)}\big) \in T(\mathcal{F} \cup \mathcal{N}, \mathcal{Y})\,,$$

and $j = i + \mathsf{rank}(b)$. Intuitively, $\mathsf{pat}(\alpha)$ is the tree pattern which is represented by the digram α. In this pattern, the parameters $y_1, \ldots, y_{\mathsf{par}(\alpha)}$ occur. For $m \in \mathbb{N} \cup \{\infty\}$, let

$$\Pi_m = \{\alpha \in \Pi \mid \mathsf{par}(\alpha) \leq m\}$$

be the set of digrams of rank at most m. Clearly, $\Pi_\infty = \Pi$.

Let $\alpha \in \Pi$ and $t \in T(\mathcal{F} \cup \mathcal{N}, \mathcal{Y})$. An *occurrence* of α within t is a node $v \in \mathsf{dom}_t$ at which a subtree

$$\mathsf{pat}(\alpha)[y_1/t_1, y_2/t_2, \ldots, y_{\mathsf{par}(\alpha)}/t_{\mathsf{par}(\alpha)}]$$

with $t_1, t_2, \ldots, t_{\mathsf{par}(\alpha)} \in T(\mathcal{F} \cup \mathcal{N}, \mathcal{Y})$ is rooted. The *set of all occurrences of the digram* α *in* t is denoted by $\mathsf{OCC}_t(\alpha) \subseteq \mathsf{dom}_t$.

If $\alpha = (a, i, b) \in \Pi$ and $t \in T(\mathcal{F} \cup \mathcal{N}, \mathcal{Y})$, then the occurrences $v, w \in \mathsf{OCC}_t(\alpha)$ are *overlapping* if one of the following equations holds: $v = w$, $vi = w$ or $wi = v$. A subset $\sigma \subseteq \mathsf{OCC}_t(\alpha)$ is said to be overlapping if there exist overlapping $v, w \in \sigma$.

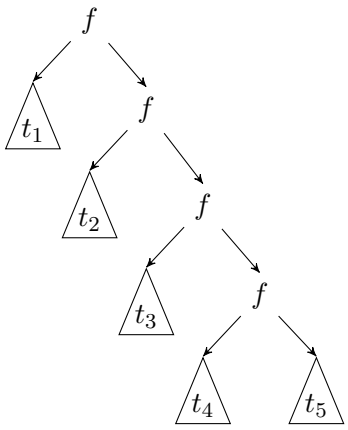

Figure 3: The Tree t from Example 3.

It is easy to see that the set $\mathsf{OCC}_t(\alpha)$ is non-overlapping if $a \neq b$. In contrast, if we have $a = b$, the set $\mathsf{OCC}_t(\alpha)$ potentially contains overlapping occurrences. Consider the following example:

Example 3. Let $t \in T(\mathcal{F})$ be the tree depicted in Figure 3 and $\alpha = (f, 2, f)$. Clearly, we have $\{\varepsilon, 2, 22\} \subseteq \mathsf{OCC}_t(\alpha)$ where on the one hand ε and 2 and on the other hand 2 and 22 are overlapping occurrences of α.

Let $\alpha \in \Pi$ and $t \in T(\mathcal{F} \cup \mathcal{N}, \mathcal{Y})$. Let $\sigma \subseteq \mathsf{OCC}_t(\alpha)$ be a non-overlapping set. Furthermore, let us assume that $\sigma \cup \{v\}$ is overlapping for all $v \in \mathsf{OCC}_t(\alpha) \setminus \sigma$, i.e., σ is maximal with respect to inclusion among non-overlapping subsets. Then the following example shows that σ is not necessarily maximal with respect to cardinality.

Example 4. Consider the tree $t \in T(\mathcal{F})$ which is depicted in Figure 3. Let $\alpha = (f, 2, f) \in \Pi$. We have $\mathsf{OCC}_t(\alpha) = \{\varepsilon, 2, 22\}$. If $\sigma = \{2\} \subseteq \mathsf{OCC}_t(\alpha)$, then σ is non-overlapping and $\sigma \cup \{v\}$ is overlapping for all $v \in \mathsf{OCC}_t(\alpha) \setminus \sigma$. However, σ is not maximal with respect to cardinality. Consider the non-overlapping subset $\sigma' = \{\varepsilon, 22\} \subseteq \mathsf{OCC}_t(\alpha)$ with $|\sigma| < |\sigma'|$.

Let us also point out that, in general, the set $\mathsf{OCC}_t(\alpha)$ may give rise to more than one maximal (with respect to cardinality) non-overlapping subset.

Example 5. Consider the tree $t = f(f(f(a)))$ over the ranked alphabet $\mathcal{F}$ and $\alpha = (f, 2, f)$. The subsets $\{\varepsilon\}$ and $\{1\}$ of $\mathsf{OCC}_t(\alpha)$ are both maximal with respect to cardinality.

```
FUNCTION retrieve-occurrences(t, α)    // α = (a, i, b)
  M := ∅; v := ε;
  while (true) do
    v := next-in-postorder(t, v);
    if (v ∈ OCC_t(α) and vi ∉ M) then
      M := M ∪ {v};
    endif
    if v = ε then
      return M;
    endif
  endwhile
ENDFUNC
```

Figure 4: The algorithm `retrieve-occurrences`.

The algorithm `retrieve-occurrences` from Figure 4 computes a non-overlapping set $M \subseteq \mathsf{OCC}_t(\alpha)$. Using the function `next-in-postorder` (not listed), it traverses the tree t in postorder. It begins by passing the parameters t and ε to `next-in-postorder` to obtain the first node $u \in \mathsf{dom}_t$ of t in postorder. The second node in postorder is then obtained by passing the parameters t and u. This step is repeated to traverse the whole tree t in postorder. For every node v which is encountered during the postorder traversal, it is checked if v is an occurrence of α and if it is non-overlapping with all occurrences already contained in the current set M. If both conditions are fulfilled, the node v is added to M.

Let $t \in T(\mathcal{F} \cup \mathcal{N}, \mathcal{Y})$ and $\alpha \in \Pi$. We define $\mathsf{occ}_t(\alpha)$ to be the set M computed by the algorithm `retrieve-occurrences`(t, α) from Figure 4. Clearly, if $a \neq b$ in $\alpha = (a, i, b)$, then we have $\mathsf{occ}_t(\alpha) = \mathsf{OCC}_t(\alpha)$. In the following, we show that the subset $\mathsf{occ}_t(\alpha) \subseteq \mathsf{OCC}_t(\alpha)$ is maximal with respect to cardinality.

Lemma 6. *Let $\alpha = (a, i, b) \in \Pi$, $t \in T(\mathcal{F} \cup \mathcal{N}, \mathcal{Y})$, and $\sigma \subseteq \mathsf{OCC}_t(\alpha)$ be non-overlapping and maximal with respect to cardinality. Then $|\mathsf{occ}_t(\alpha)| = |\sigma|$.*

Proof. If $a \neq b$, then $\sigma = \mathsf{OCC}_t(\alpha)$. Moreover, by inspecting the algorithm from Figure 4, it can be easily checked that $\mathsf{occ}_t(\alpha) = \mathsf{OCC}_t(\alpha)$.

Now, let us assume that $a = b$. We define a graph (V, E) with $V \subseteq \mathsf{dom}_t$ as follows: $V = \{v \in \mathsf{dom}_t \mid \lambda_t(v) = a\}$ and $(u, v) \in E$ if and only $v = ui$. This graph is a disjoint union of paths. Note that $|\sigma|$ is simply

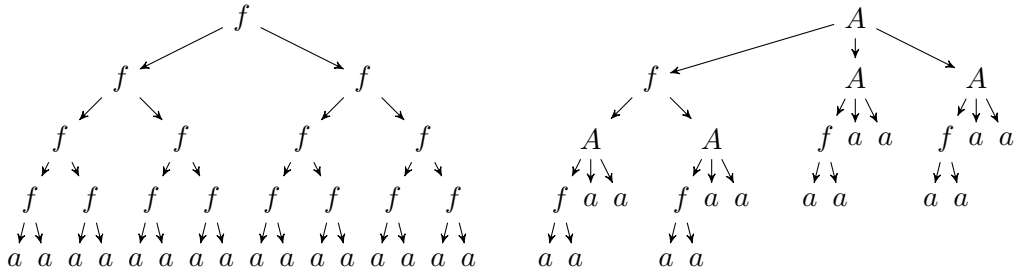

Figure 5: The trees t (left side) and $t[\alpha/A]$ (right side) from Example 7.

the number of edges in a largest (with respect to cardinality) matching of (V, E). To see that $|\mathsf{occ}_t(\alpha)| = |\sigma|$ we can assume that (V, E) consists of a single path (otherwise, consider maximal path in (V, E) separately). So, let $V = \{v, vi, \ldots, vi^m\}$. The size of a largest matching is $\lfloor (m+1)/2 \rfloor$. Moreover, the algorithm from Figure 4 selects from the path (V, E) all nodes $vi^{m+1-2\ell}$ for $1 \leq \ell \leq \lfloor (m+1)/2 \rfloor$, and these are $\lfloor (m+1)/2 \rfloor$ many nodes. □

Let $t \in T(\mathcal{F} \cup \mathcal{N}, \mathcal{Y})$, $\alpha \in \Pi$ and $A \in \mathcal{N}_{\mathsf{par}(\alpha)}$. By $t[\alpha/A]$, we denote the tree which is obtained by replacing all occurrences from $\mathsf{occ}_t(\alpha)$ in the tree t by the nonterminal A (in parallel). More precisely, we replace every subtree $\mathsf{pat}(\alpha)[y_1/t_1, y_2/t_2, \ldots, y_{\mathsf{par}(\alpha)}/t_{\mathsf{par}(\alpha)}]$ with $t_1, \ldots, t_{\mathsf{par}(\alpha)} \in T(\mathcal{F} \cup \mathcal{N}, \mathcal{Y})$ which is rooted at an occurrence $v \in \mathsf{occ}_t(\alpha)$ by the new subtree $A(t_1, t_2, \ldots, t_{\mathsf{par}(\alpha)})$. Since $\mathsf{occ}_t(\alpha)$ is non-overlapping, this parallel substitution is well-defined.

Example 7. Consider the tree $t \in T(\mathcal{F})$ which is depicted on the left side of Figure 5. For $\alpha = (f, 2, f)$, we have $\mathsf{occ}_t(\alpha) = \{\varepsilon, 11, 12, 21, 22\}$. By replacing the digram α in t by the nonterminal $A \in \mathcal{N}_3$, we obtain the tree $t[\alpha/A]$ which is depicted on the right side of Figure 5.

For $t \in T(\mathcal{F} \cup \mathcal{N})$ and $m \in \mathbb{N} \cup \{\infty\}$, we denote with

$$\mathsf{mfreq}_m(t) = \{\alpha \in \Pi_m \mid \mathsf{occ}_t(\alpha) \neq \emptyset \text{ and } \forall \beta \in \Pi_m : |\mathsf{occ}_t(\beta)| \leq |\mathsf{occ}_t(\alpha)|\}$$

the set of digrams which occur in t most frequently (with respect to all digrams from Π_m). If $m = \infty$, then $\mathsf{mfreq}_m(t) = \emptyset$ if and only if the tree t consists of exactly one node. Otherwise (i.e., $m \neq \infty$), we have $\mathsf{mfreq}_m(t) = \emptyset$ if and only if t consists of exactly one node or if for all digrams α occurring in t it holds that $\alpha \notin \Pi_m$.

3. TreeRePair algorithm

In this section, we introduce our TreeRePair algorithm which consists of two phases, namely the *replacement phase* and the *pruning phase*.

3.1. Replacement phase

During the replacement phase of our algorithm, we subsequently replace a most frequent digram by a new nonterminal. The replacement process stops if there is no digram occurring at least twice.

Let $t_0 \in T(\mathcal{F})$ be our input tree and $m \in \mathbb{N} \cup \{\infty\}$ be the maximal rank allowed for nonterminals. Formally, a *replacement phase* on the tree t during which $n \in \mathbb{N}$ digrams are replaced is described by a sequence $(\mathcal{G}_1, \alpha_1), (\mathcal{G}_2, \alpha_2), \ldots, (\mathcal{G}_n, \alpha_n)$ where, for all $1 \leq i \leq n$,

- $\mathcal{G}_i = (N_i, P_i, S_i)$ is an SLCF tree grammar with $(S_i \to t_i) \in P_i$,
- α_i is a digram from $\mathsf{mfreq}_m(t_{i-1})$,
- $t_i = t_{i-1}[\alpha_i/A_i]$ where $A_i \in \mathcal{N}_{\mathsf{par}(\alpha_i)} \setminus N_{i-1}$,
- $N_i = \{S_i, A_1, \ldots, A_i\}$,
- $P_i = \{S_i \to t_i\} \cup \{A_j \to \mathsf{pat}(\alpha_j) \mid 1 \leq j \leq i\}$, and
- $\mathsf{mfreq}_m(t_n) = \emptyset$ or $|\mathsf{occ}_t(\beta)| \leq 1$ for all $\beta \in \mathsf{mfreq}_m(t_n)$.

The linear SLCF tree grammar $\mathcal{G}_n$ is not the final output of TreeRePair. It is further processed in the pruning phase, described next.

3.2. Pruning phase

The grammar produced in the replacement phase of our algorithm potentially contains productions which do not contribute to a compact representation of the input tree. In our TreeRePair algorithm, we consider production $(A \to t)$ as unprofitable if $\mathsf{sav}_{\mathcal{G}}(A) \leq 0$. During the pruning phase, we get rid of such unprofitable productions.

Let $\mathcal{G} = (N, P, S)$ be a linear SLCF tree grammar. We *eliminate* a production $(A \to t) \in P$ with $A \in N \setminus \{S\}$ from $\mathcal{G}$ as follows:

(1) For every $(t', v) \in \mathsf{ref}_{\mathcal{G}}(A)$, we replace the subtree $A(t_1, t_2, \ldots, t_n)$ rooted at $v \in \mathsf{dom}_{t'}$ by the tree $t[y_1/t_1, y_2/t_2, \ldots, y_n/t_n]$ where $n = \mathsf{rank}(A)$ and $t_1, \ldots, t_n \in T(\mathcal{F} \cup \mathcal{N}, \mathcal{Y})$.

(2) We remove the production $A \to t$ from P.

Eliminating a production has an impact on the save value of the remaining nonterminals. Let $B_1, B_2, \ldots, B_{n-1}, B_n$ be a sequence of all nonterminals from N in hierarchical order, i.e., we have $B_n = S$ and $B_j \not\rightsquigarrow_{\mathcal{G}}^* B_i$ for all $1 \leq i < j \leq |N|$. Let $(B_i \to t_i), (B_j \to t_j) \in P$, where $1 \leq i, j < n$ and $i \neq j$. If we eliminate B_i, this may have an impact on the value of $\mathsf{sav}_{\mathcal{G}}(B_j)$. We need to distinguish two cases.

(a) If B_i occurs in t_j, i.e., $B_i \rightsquigarrow_{\mathcal{G}} B_j$, then $|t_j|$ is increased because of the elimination of B_i. At the same time, $\mathsf{sav}_{\mathcal{G}}(B_j)$ goes up if we have $|\mathsf{ref}_{\mathcal{G}}(B_j)| > 1$. The increase of $|t_j|$ is due to the fact that we can assume that the inequality $|\{v \in \mathsf{dom}_{t_i} \mid \lambda_{t_i}(v) \notin \mathcal{Y}\}| \geq 2$ holds. Every production which was introduced in the replacement phase represents a digram and therefore consists of at least two nodes labeled by the parent and child symbol, respectively, of this digram.

(b) If B_j occurs in t_i, i.e., $B_j \rightsquigarrow_{\mathcal{G}} B_i$, then $|\mathsf{ref}_{\mathcal{G}}(B_j)|$ and therefore $\mathsf{sav}_{\mathcal{G}}(B_j)$ are possibly increased by eliminating B_i. In fact, both values go up if $|\mathsf{ref}_{\mathcal{G}}(B_i)| > 1$.

The following example shows that the size of the final grammar depends on the order in which possible inefficient productions are eliminated.

Example 8. Consider the linear SLCF tree grammar $\mathcal{G} = (N, P, S)$ where $N = \{S, A, B\}$ and P is the following set of productions:

$$\begin{aligned} S &\to f(A(a, a), B(A(a, a))) \\ A(y_1, y_2) &\to f(B(y_1), y_2) \\ B(y_1) &\to f(y_1, a) \end{aligned}$$

Let us assume that the grammar $\mathcal{G}$ was generated in the replacement phase of our algorithm and that we now want to remove all inefficient productions. We have $\mathsf{sav}_{\mathcal{G}}(A) = -1$ and $\mathsf{sav}_{\mathcal{G}}(B) = 0$, i.e., the productions with left-hand sides A and B do not contribute to a small representation of the input tree $\mathsf{val}(\mathcal{G})$. Let us consider the following two cases:

(a) If we eliminate the production with left-hand side A, then we obtain the grammar $\mathcal{G}_1 = (\{S_1, B\}, P_1, S_1)$ where P_1 consists of the productions $S_1 \to f(f(B(a), a), B(f(B(a), a)))$ and $B(y_1) \to f(y_1, a)$. We have

$|\mathcal{G}_1| = 11$ and $\mathsf{sav}_{\mathcal{G}_1}(B_1) = 1$, i.e., the production with left-hand side B is not considered inefficient.

(b) In contrast, if we first eliminate the production with left-hand side B, then we obtain the grammar $\mathcal{G}_2 = (\{S_2, A\}, P_2, S_2)$ with productions $S_2 \to f(A(a,a), f(A(a,a),a))$ and $A(y_1,y_2) \to f(f(y_1,a),y_2)$. Since $\mathsf{sav}_{\mathcal{G}_2}(A) = 0$, we also eliminate the production with left-hand side A and obtain the grammar $G'_2 = (\{S'_2\}, P'_2, S'_2)$ consisting solely of the production $S'_2 \to f(f(f(a,a),a), f(f(f(a,a),a),a))$. We have $|\mathcal{G}'_2| = 12$.

As shown above, it is a complex optimization problem to find the optimal order in which inefficient productions should be removed. This forces us to use a heuristic to decide which production to remove next from the grammar.

Let $\mathcal{G} = (N, P, S)$ be the grammar produced in the replacement phase of our algorithm. The pruning phase of our TreeRePair algorithm is divided into two phases:

(1) In the first part of the pruning phase, we eliminate every production $(A \to t) \in P$ with $|\mathsf{ref}_{\mathcal{G}}(A)| = 1$. That way we achieve not only a possible reduction of the size of $\mathcal{G}$ (because we have $\mathsf{sav}_{\mathcal{G}}(A) = -\mathsf{rank}(A)$ for every $A \in N$ referenced only once) but we also decrement the number of nonterminals $|N|$ each time we eliminate such a production.

(2) In the second part of the pruning phase, we eliminate all remaining inefficient productions. We cycle through the remaining productions in their reverse hierarchical order (recall that the start nonterminal S is the last nonterminal in the hierarchical order, i.e., it is the first one in the reverse hierarchical order). For every $(A \to t) \in P$ with $A \neq S$, we check if $\mathsf{sav}_{\mathcal{G}}(A) \leq 0$. If this proves to be true, then we eliminate $(A \to t)$. That way $|\mathcal{G}|$ and $|N|$ are possibly further reduced.

Note that, in Example 8, our heuristic would indeed generate the smaller grammar $\mathcal{G}_1$ (since A is the last non-start nonterminal in the hierarchical order). Another reasonable heuristic might be to eliminate inefficient productions in the order of the $\mathsf{sav}_{\mathcal{G}}$-values of their left-hand sides, i.e., if we would proceed as follows: As long as there is a production whose left-hand side has a $\mathsf{sav}_{\mathcal{G}}$-value smaller or equal to 0 we remove a production whose left-hand side has the smallest occurring $\mathsf{sav}_{\mathcal{G}}$-value. However, our investigations show that this approach leads to unappealing final grammars. The grammars generated by this approach exhibit nearly the same number of edges

but much more nonterminals (about 50% more) compared to the grammars obtained using the above heuristic.

Note that one cannot easily detect digrams leading to inefficient productions during the replacement phase. For example, skipping digrams occurring only a small number of times and leading to productions with many parameters might be disadvantageous. Imagine an input tree exhibiting a large tree pattern occurring only twice. In order to generate a grammar which comes with a production representing this large tree pattern, one has to replace many digrams possibly occurring only twice and introducing productions exhibiting a large number of parameters.

4. Influence of the maximal rank allowed

In this section, we investigate the influence of the maximal rank allowed for nonterminals on the compression performance of TreeRePair.

4.1. Large maximal rank

In the following, we construct, for every $k \geq 2$, a family of trees on which TreeRePair with maximal rank restricted to any $\ell \geq k$ achieves an exponentially better compression ratio than TreeRePair with maximal rank restricted to any $\ell' < k$. For $n \geq 1$ and $k \geq 2$, we consider the tree $s_{n,k}$ which consists of a right-comb of 2^n-many f_k-labeled nodes of rank k. The first $k-1$ many children of each f_k-labeled node (as well as the last child of the last f_k-labeled node of the comb) are leaves labeled with a. Thus, the size of $s_{n,k}$ is precisely $k \cdot s^k$. Consider Figure 6 which shows the tree $s_{3,3}$ (of size 24). It should be intuitively clear that on this tree, an execution of TreeRePair with maximal rank set to 1 does not carry out any digram replacement. This is because in a ternary tree, any replacement of a digram introduces a nonterminal of rank at least 2. On the other hand, setting the maximal rank to 2 allows to recursively remove all repeating digrams, thus ending up with a grammar of size only $2n + k$.

Theorem 9. *Let $k, \ell, \ell', n \in \mathbb{N}$ with $k \geq 2$ and $\ell \geq k - 1 > \ell'$. For every grammar $\mathcal{G}^\ell$ (resp. $\mathcal{G}^{\ell'}$) that can be produced by a run of TreeRePair on the tree $s_{n,k}$, where the maximal rank of nonterminals is restricted to ℓ (resp., ℓ'), we have $|\mathcal{G}^\ell| \leq 2n + k$ and $|\mathcal{G}^{\ell'}| = |s_{n,k}| = k \cdot 2^n$.*

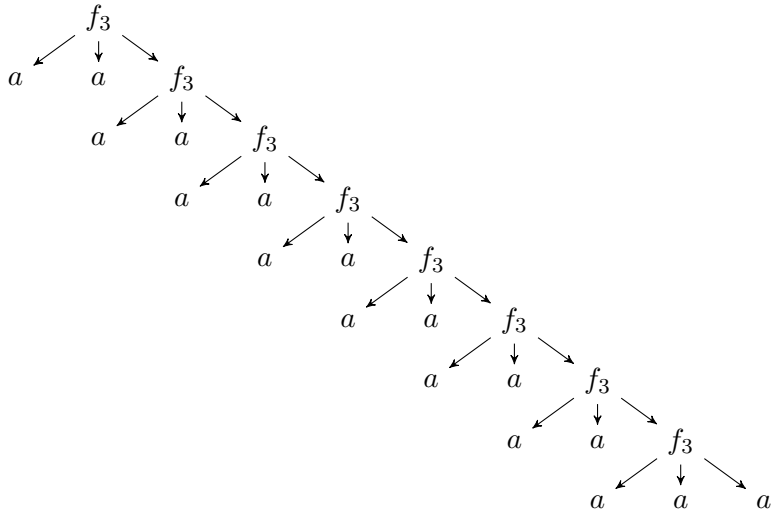

Figure 6: The tree $s_{3,3}$.

PROOF. Since the tree $s_{n,k}$ does not contain digrams of rank at most $k-2$, $\mathcal{G}^{\ell'}$ must be a trivial grammar for $s_{n,k}$, i.e., $|\mathcal{G}^{\ell'}| = |s_n|$. In contrast, if we run TreeRePair with a maximal rank for nonterminals of at least $k-1$, then the algorithm detects repeating digrams. Note that the most frequent digrams in $s_{n,k}$ are (f_k, j, a) for $1 \leq j \leq k-1$. Each of these digrams occurs 2^n many times in $s_{n,k}$. In the first $k-1$ iterations of the replacement phase, TreeRePair introduces $k-1$ productions that collapse the tree pattern $f(a, \ldots, a, y)$ (with $k-1$ many a's) into a single nonterminal $A_0(y)$, and the start production is $S \to A_0^{2^n}(a)$. During the pruning phase these $k-1$ productions are merged to a single production $A_0(y) \to f(a, \ldots, a, y)$. Moreover, the replacement phase collapses the right-hand side $A_0^{2^n}(a)$ of the start production by introducing $n-1$ additional productions $A_i(y) \to A_{i-1}(A_{i-1}(y))$ $(1 \leq i \leq n-1)$ into the tree $A_{n-1}(A_{n-1}(a))$. This gives the grammar

$$\begin{aligned} S &\to A_{n-1}(A_{n-1}(a)), \\ A_i(y) &\to A_{i-1}(A_{i-1}(y)) \quad (1 \leq i \leq n-1), \\ A_0(y) &\to f(\underbrace{a, \ldots, a}_{k-1 \text{ many}}, y) \end{aligned}$$

of size $2n + k$. □

In [23] another example was presented. Let t_n be a full binary tree of height n whose leaves are labeled with pairwise different symbols. It is shown that, for every k, there is a bound m such that, for every tree t_n with $n \geq m$, TreeRePair with maximal rank k produces a grammar that is strictly larger than TreeRePair with unlimited maximal rank.

4.2. Small maximal rank

It seems natural to assume that, in general, trees can be compressed best by TreeRePair if there are no restrictions on the maximal rank of nonterminals. However, it turns out that there are (not so uncommon) types of trees for which the opposite is true. In this section, we construct a family of trees s_n which can be compressed best if we limit the maximal rank of nonterminals to 1.

The tree s_n is a right-comb of 2^n many binary f-labeled nodes. The left child of each f-node (as well as the right-child of the last f-node of the comb) is a leaf with a label from $\{a_i \mid 0 \leq i \leq 4\}$. If we read the leaf labels of the comb from left to right, then we obtain a word that is a prefix of a word in $(a_1a_2a_3a_4a_0)^*$. The tree s_4 is shown in Figure 7.

Theorem 10. *Let $n \geq 4$. For every grammar $\mathcal{G}^\infty$ (resp. $\mathcal{G}^1$) that can be produced by a run of TreeRePair on the tree s_n, where the maximal rank of nonterminals is not restricted (resp., is restricted to 1), we have $|\mathcal{G}^\infty| > \frac{1}{2}|s_n|$ and $|\mathcal{G}^1| \in O(\log_2 |s_n|) = O(n)$.*

Proof. Let us consider a run $(\mathcal{G}_1, \alpha_1), \ldots, (\mathcal{G}_{n-1}, \alpha_{n-1})$ of TreeRePair on the tree s_n with no restriction on the maximal rank of nonterminals. For $1 \leq i \leq n-1$, let $\mathcal{G}_i = (N_i, P_i, S_i)$ and $(S_i \to t_i) \in P_i$ be the start production of $\mathcal{G}_i$. In the first iteration of the replacement phase, the digram $(f, 2, f)$ is the sole most frequent one, i.e., $\alpha_1 = (f, 2, f)$. This is because of $|\mathsf{occ}_{s_n}\big((f, 2, f)\big)| = 2^{n-1}$ whereas for every $0 \leq \ell \leq 4$ the inequality

$$|\mathsf{occ}_{s_n}\big((f, 1, a_\ell)\big)| \leq \lceil 2^n/5 \rceil < 2^{n-1}$$

holds. Therefore, we replace the digram $(f, 2, f)$ by a new nonterminal A_1 of rank 3 and obtain $\mathcal{G}_1$. In every subsequent iteration $2 \leq i \leq n-1$, we replace the most frequent digram $\alpha_i = (A_{i-1}, 2^{i-1}+1, A_{i-1})$ in t_{i-1} by a new

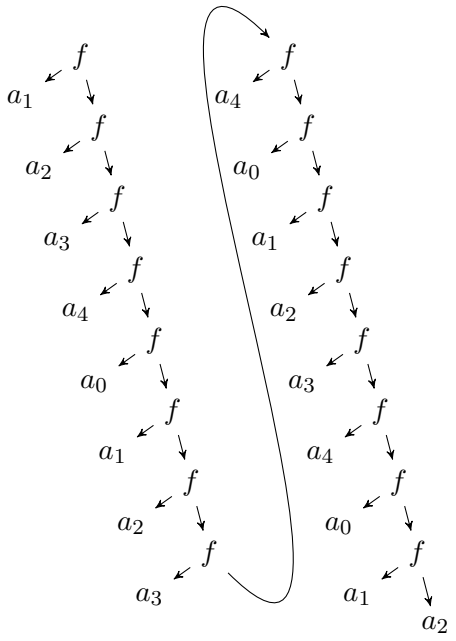

Figure 7: The tree s_4.

nonterminal A_i of rank 2^i+1. For every $1 \leq i \leq n-1$, the tree t_i is given by

$$\mathsf{dom}_{t_i} = \{\varepsilon\} \cup \{[2^i+1]^j[k] \mid 0 \leq j < 2^{n-i}, 1 \leq k \leq 2^i+1\} \quad \text{and}$$

$$\lambda_{t_i}(v) = \begin{cases} A_i & \text{if } v = [2^i+1]^j \text{ with } 0 \leq j \leq 2^{n-i}-1\,. \\ a_{j2^i+k \bmod 5} & \text{if } v = [2^i+1]^j[k] \text{ with } 0 \leq j \leq 2^{n-i}-1, 1 \leq k \leq 2^i\,. \\ a_{2^n+1 \bmod 5} & \text{if } v = [2^i+1]^{2^{n-i}}\,. \end{cases}$$

In order to argue that we have $\alpha_{i+1} = (A_i, 2^i+1, A_i)$ for every $1 \leq i \leq n-2$, we compute the number of occurrences of all digrams occurring in t_i. It is easy to verify that $|\mathsf{occ}_{t_i}(\alpha_{i+1})| = 2^{n-i-1}$. In contrast, for every $1 \leq k \leq 2^i$ and $0 \leq \ell \leq 4$, the inequality $|\mathsf{occ}_{t_i}\big((A_i, k, a_\ell)\big)| \leq \lceil 2^{n-i}/5 \rceil < 2^{n-i-1}$ holds. To see this note that $\ell \cdot 2^i \not\equiv 0 \bmod 5$ for every $0 \leq \ell \leq 4$. Hence, we have

$$j2^i + k \;\not\equiv\; (j+\ell)2^i + k \bmod 5$$

for all $j, k \in \mathbb{Z}$ and $0 \leq \ell \leq 4$. But this implies that

$$\lambda_{t_i}([2^i+1]^j[k]) \;\neq\; \lambda_{t_i}([2^i+1]^{j+\ell}[k])$$

for all $1 \leq k \leq 2^i$ and all j, ℓ with $0 \leq \ell \leq 4$ and $j + \ell \leq 2^{n-i} - 1$.

Due to the fact that we do not replace digrams with a child symbol from $\{a_0, \ldots, a_4\}$, the tree t_{n-1} has to contain at least 2^n+1 nodes labeled by these symbols. This property is also preserved during the pruning phase. Hence, we must have $|\mathcal{G}_{n-1}| > 2^n$ and, therefore, $|\mathcal{G}_{n-1}| > \frac{1}{2}|s_n|$. To sum up, if we do not restrict the maximal rank of new nonterminals and always replace a most frequent digram (which is unique in every step if we start with a tree s_n), then the compression ratio achieved on the trees s_n is larger than $1/2$.

Now, we consider the same trees s_n, but we only allow to introduce new nonterminals of rank at most 1. Hence, consider a run $(\mathcal{G}_1, \alpha_1), \ldots, (\mathcal{G}_k, \alpha_k)$ of TreeRePair on the tree s_n, where the maximal rank of nonterminals is restricted to 1. Note that we have $(f, 2, f) \notin \mathsf{mfreq}_1(s_n)$, since a replacement of $(f, 2, f)$ would result in a nonterminal of rank $3 > 1$. Therefore only the digrams $(f, 1, a_l)$ $(0 \leq l \leq 4)$ and subsequent digrams can be replaced. Note that since $n \geq 4$, every digram $(f, 1, a_l)$ occurs at least twice. It turns out that after the first nine digram replacements the tree pattern $f(a_1, f(a_2, f(a_3, f(a_4, f(a_0, y)))))$ is represented by a new nonterminal $A(y)$ of rank 1. The actual order of the replacements within the first nine iterations depends on the method used to choose a most frequent digram when there are multiple most frequent digrams.

The right-hand side of $\mathcal{G}_9$'s start production is a monadic tree mainly consisting of consecutive nonterminals A. The corresponding nodes — there are roughly $2^n/5$ of them — are then boiled down using approximately $\log_2(2^n/5)$ digram replacements. Therefore the total number of edges in the final grammar $\mathcal{G}_k$ is in $\mathcal{O}(n)$, i.e., it is of logarithmic size (the size of the input tree s_n is $2^{n+1} + 1$). □

5. Implementation details

We implemented a prototype of the TreeRePair algorithm. The source code and its documentation can be accessed by visiting the following web page

http://code.google.com/p/treerepair

Regarding our implementation we decided to focus on the compression of the tree structure of XML documents only. This allows us to better compare

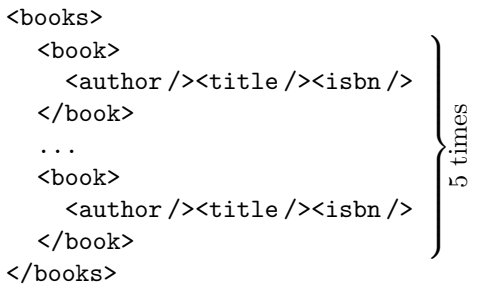

Figure 8: A simplified XML document.

the compression performance with existing compressors. More precisely, our implementation reads the document tree of an XML file by considering only element nodes and skipping text content and other node types. Internally, a first-child/next-sibling encoding of the document tree (see Section 5.1) is constructed as a pointer structure. In this pointer structure each node has pointers to its children and to its parent node. Optionally, and in fact by default, the input tree is represented by its minimal unranked DAG in memory. Since this DAG can be constructed on the fly during parsing, it allows to save a lot of memory, and hence larger document trees can be processed by our implementation. A slight drawback is that the administration and replacement of digram occurrences is more complex over the DAG representation and results in slightly worse compression rates (this does not happen if the non-DAG memory representation is used).

The mode of operation of our implementation slightly differs from the TreeRePair algorithm. In fact, it is not guaranteed that, in every iteration, one of the most frequent digrams is replaced. More precisely, the set of non-overlapping occurrences of a digram α in t that is stored by the algorithm is only guaranteed to be a subset of $\mathsf{occ}_t(\alpha)$, see Section 5.3.1. This phenomenon results from our efforts to obtain an implementation that runs in linear time. Our experiments show that this deviation from pure TreeRePair downgrades the compression ratio in an negligible way while improving running time drastically.

5.1. From XML document trees to ranked trees

An XML document tree can be considered as an unranked tree, i.e., nodes with the same label possibly have a varying number of children. Figure 9 shows the unranked XML document tree of the XML document of Figure 8.

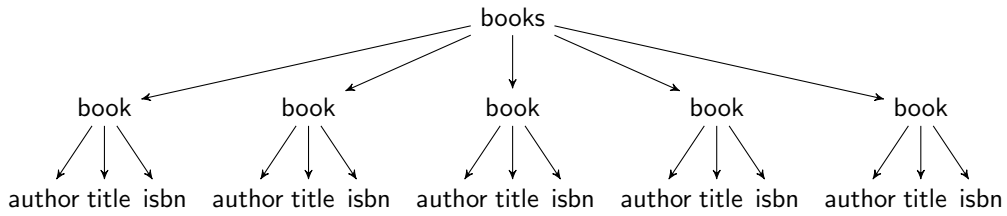

Figure 9: XML document tree of the XML document listed in Figure 8

However, our TreeRePair algorithm works on ranked trees. A common way of transforming an unranked tree into a binary tree is the *first-child/next-sibling encoding*, see, e.g., Section 2.3.2 in Knuth's first book [16] and see [38] for a discussion in the context of XML. In this encoding, the leftmost child of a node v in the unranked tree (if existing) becomes the left child of v in the binary tree, and the right sibling of v in the unranked tree (if existing) becomes the right child of v in the binary tree. Hence, a leaf node of the unranked tree has no left child but a possible right child in the corresponding binary tree. In contrast, a last sibling of the unranked tree has no right child in the binary tree.

To obtain a tree over a ranked alphabet, one introduces, for every node label a that occurs in the unranked tree, four different variants: a^{00} is used for a-labeled nodes with no children in the binary tree, a^{10} (resp. a^{01}) is used for a-labeled nodes which exhibit only a left (resp. right) child in the binary tree, and a^{11} is used for nodes coming with a left and a right child in the binary tree. Hence, a^{00} has rank 0, a^{10} and a^{01} have rank 1, and a^{11} has rank 2. Figure 10 shows the first-child/next-sibling encoding of the unranked tree from Figure 9.

Another way of transforming an unranked tree into a ranked one which circumvents the introduction of special labels is the use of placeholder nodes. These nodes are inserted to indicate missing left or right children. However, our experiments showed that our implementation of TreeRePair achieves slightly less competitive compression results under this encoding.

5.2. Data structures used

Our implementation uses similar data structures as RePair for strings [19]. In order to focus on the essentials, we first do not pay attention to the fact that, internally, the input tree is represented by a DAG. In Section 5.4, we discuss the necessary modifications for the DAG representation.

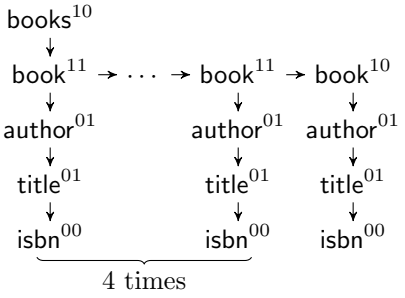

Figure 10: Binary tree representation of the XML document tree from Figure 9.

We suggest to consider Figure 11 while reading the following description. In main memory, every node v of the input tree is represented by an object exhibiting several pointers. These allow constant time access to the parent node, to all child nodes of v, and to the possible next and previous occurrences of the digrams $\alpha = \big(\lambda_t(v), i, \lambda_t(vi)\big)$ where $1 \leq i \leq \mathsf{rank}(\lambda_t(v))$. The pointers to the next and previous occurrences of the digram α form a doubly linked list of occurrences of the digram α. We call these structures *occurrences lists* in the following.[3] The specific order of the occurrences in an occurrences list is not relevant. Doubly linked lists allow to remove any specific digram occurrence in constant time.

Every digram is represented by a special object. The latter exhibits two pointers which reference the first and the last element of the corresponding occurrences list. Let us consider a digram α occurring i many times, where $i < \lfloor\sqrt{n}\rfloor$ and n is the size of the input tree. Then the corresponding object exhibits two more pointers which point to the next and previous, respectively, digram β occurring i many times. These pointers form a doubly linked list of all digrams occurring i times. We call this list the *i-th digram list*. In contrast, all digrams γ occurring at least $\lfloor\sqrt{n}\rfloor$ many times are organized in one doubly linked, unordered list which is called the *top digram list*. As for the occurrences lists, doubly linked lists allow to remove a certain digram

[3]During our investigations we also implemented a TreeRePair version avoiding these doubly linked lists of occurrences. Instead, for every digram, we used a hashed set storing pointers to all occurrences. However, this version had no benefits compared to the doubly linked list approach but lead to slightly longer running times. Considering the memory usage, in some cases it achieved better results while in others a substantial increase was noticed.

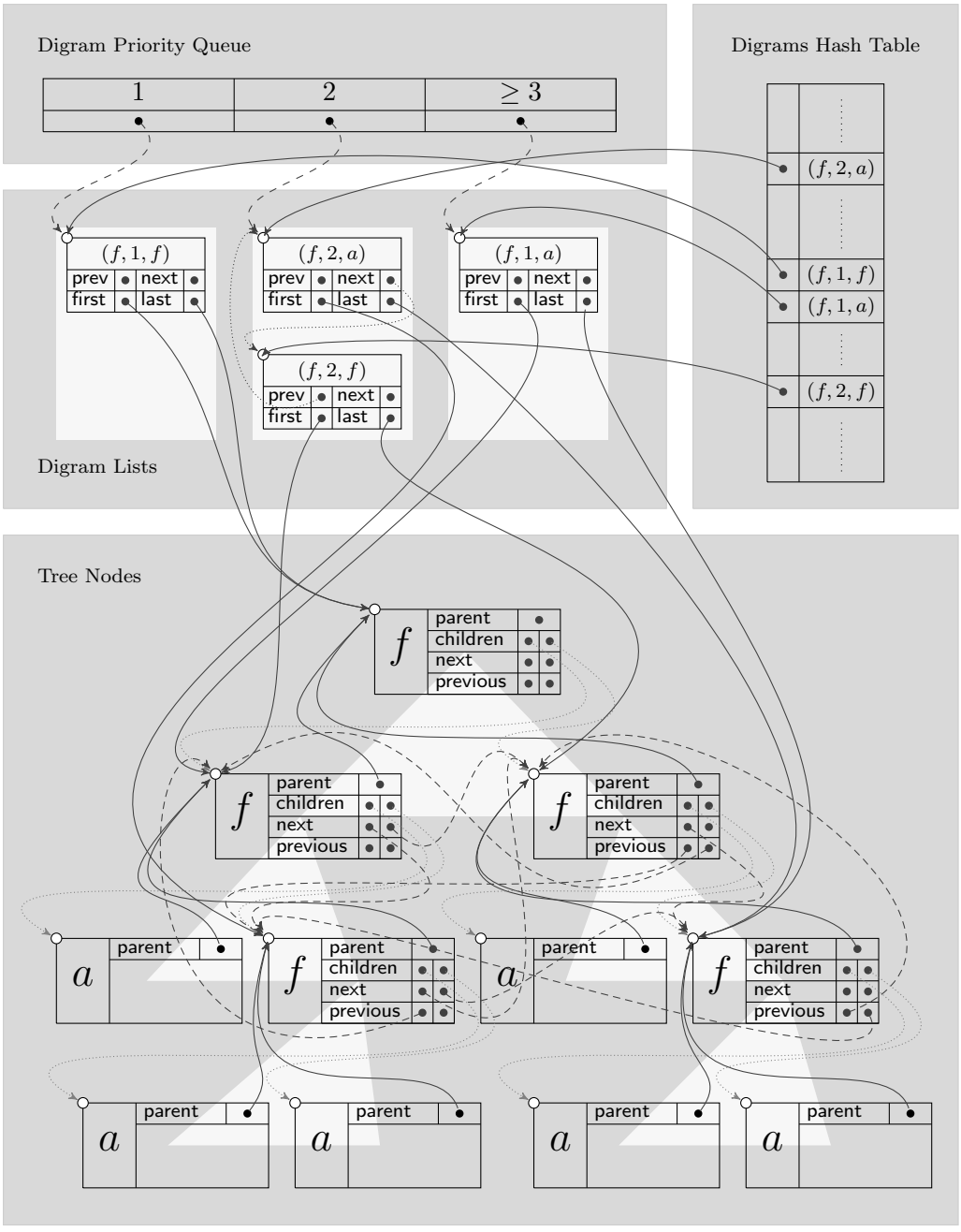

Figure 11: A simplified depiction of a part of the data structures used by our implementation. The tree represented in memory is $f(f(a, f(a, a)), f(a, f(a, a)))$.

from a digram list in constant time.

These doubly linked lists of digrams are again referenced by a *digram priority queue*. This queue consists of $\leq \lfloor\sqrt{n}\rfloor$ entries. The i-th entry stores a pointer to the head of the i-th digram list, where $1 \leq i < \lfloor\sqrt{n}\rfloor$. The $\lfloor\sqrt{n}\rfloor$-th entry references the head of the top digram list. Lastly, there is a *digram hash table* storing pointers to the objects of all occurring digrams. It allows constant time access to all digrams and therefore constant time access to the first occurrence of each digram.

5.3. Complexity of our implementation

Our implementation of the TreeRePair algorithm runs in linear time if we restrict the maximal rank of nonterminals to a constant $m \in \mathbb{N}$. In fact, our implementation allows m to be specified using a command line switch.

Theorem 11. *Fix $1 \leq m < \infty$ and a ranked alphabet $\mathcal{F}$. For any given input tree $t \in T(\mathcal{F})$, our implementation of TreeRePair run with maximal rank m for nonterminals, produces in time $\mathcal{O}(|t|)$ an m-bounded linear SLCF tree grammar $\mathcal{G}$ such that $\mathsf{val}(\mathcal{G}) = t$.*

The multiplicative constant in the time bound $\mathcal{O}(|t|)$ depends linearly on m. In our experiments $m = 4$ turned out to yield the best compression ratio on large XML structure trees, see Section 7. The choice $m = 4$ leads to a moderate constant in the time bound $\mathcal{O}(|t|)$.

Let us first argue that the pruning phase of TreeRePair, in which inefficient productions are eliminated, can be implemented in linear time. Let $\mathcal{G}$ be the grammar obtained after the replacement phase. Recall that during the pruning phase, we go over all nonterminals except for the start nonterminal S in reverse hierarchical order. Hence, we need an enumeration $A_1, A_2, \ldots, A_n$ of all nonterminals such that $A_i \rightsquigarrow_G A_j$ (i.e., A_i occurs in the right-hand side of A_j) implies $i > j$. Such an enumeration is implicitly obtained in the replacement phase: If A_i occurs in the right-hand side of A_j then A_j is introduced in the replacement phase after A_i.

Recall that the pruning phase consists of two phases:

- In a first phase, we eliminate all nonterminals A with $|\mathsf{ref}_{\mathcal{G}}(A)| = 1$.
- In a second phase, we walk over all remaining nonterminals A (the start nonterminal is excluded) in reverse hierarchical order and eliminate every nonterminal A with $\mathsf{sav}_{\mathcal{G}}(A) \leq 0$.

Let us argue that the second phase can be implemented in linear time, the arguments for the first phase are analogous. In a single traversal of the right-hand sides of $\mathcal{G}$, we can compute, for each nonterminal A, the value $|\mathsf{ref}_{\mathcal{G}}(A)|$ and a list of all occurrences of A in the right-hand sides of $\mathcal{G}$. Now, we walk over all nonterminals in reverse hierarchical order. Let us assume that we process the nonterminals in the following order: $A_1, A_2, \ldots, A_n$ where $A_1 = S$ is the start nonterminal. Let t_i be the right-hand side of A_i. At a nonterminal A_i, we traverse t_i in order to compute $|t_i|$. All in all, this can be accomplished in linear time since we have $\mathcal{O}(|\mathcal{G}|) \leq \mathcal{O}(|t|)$. Having $\mathsf{ref}_{\mathcal{G}}(A_i)$ and $|t_i|$ at hand, we are able to calculate $\mathsf{sav}_{\mathcal{G}}(A_i)$. If we decide to eliminate A_i (i.e., $\mathsf{sav}_{\mathcal{G}}(A_i) \leq 0$), then all occurrences of A_i in a right-hand side are replaced by a copy of t_i. Note that A_i can only occur in the right-hand sides of $A_1, \ldots, A_{i-1}$. Hence, when we eliminate A_i, the right-hand sides of $A_{i+1}, \ldots, A_n$ (the nonterminals that still have to be considered) are not modified.

We have to argue that the total number of node insertions during all nonterminal eliminations is at most $|t|$. For this, note that the worst case occurs if all nonterminals $A_2, \ldots, A_n$ are eliminated in this order. In this case, at most $|t|$ many new nodes are inserted, since the right-hand side t_1 of the start nonterminal $A_1 = S$ is transformed into the original tree t.

Finally, we have to argue that all additional data structures (the occurrence lists for the nonterminals and the $\mathsf{ref}_{\mathcal{G}}$-values) can be updated. But this is easy: Each time we introduce during the elimination process a new A_j-labeled node in a right-hand side, we set $\mathsf{ref}_{\mathcal{G}}(A_j) := \mathsf{ref}_{\mathcal{G}}(A_j) + 1$. In the same way, we can update the occurrences lists for nonterminals.

Now, let us investigate the complexity of the replacement phase which was described in Section 3.1. With every replacement of a digram occurrence, one edge of the input tree is absorbed. Therefore, a run of our implementation can consist of at most $n - 1$ iterations, where n is the size of the input tree. Each replacement of an occurrence can be accomplished in $\mathcal{O}(1)$ time since at most m children need to be reassigned — in our implementation, the reassignment of a child node is just a matter of updating two pointers. For every production which is introduced during a run of our algorithm it holds that its right-hand side is at most of size $m + 1$, i.e., it can be constructed in constant time.

However, in order to show that the replacement phase can be performed in linear time two more aspects need to be considered:

(1) *Updating the occurrences lists:* When replacing a digram by a new nonterminal, some digram occurrences have to be removed from the occurrences lists and new digrams are introduced for which we have to set up corresponding occurrences lists. The overall time (over the whole replacement phase) necessary for these updates has to be linear in the size of the input tree.

(2) *Retrieving a most frequent digram:* Let us assume that up-to-date occurrences lists are at hand for every digram occurring. In order to proceed, we have to determine a most frequent digram α (according to the information provided by the occurrences lists). Again, the overall time (over the whole replacement phase) needed for finding maximal digrams has to be linear in the size of the input tree.

In the following, we consider each of the above aspects in detail.

5.3.1. Updating the occurrences lists

Let t be our input tree. At the beginning of the replacement phase, the occurrences list for every digram $\alpha \in \Pi_m$ occurring in t is initially constructed. This is done by parsing the tree t in a similar way as it is done in the function `retrieve-occurrences` which is listed in Figure 4. In fact, during the traversal not only one digram is considered but for every encountered digram $\alpha \in \Pi_m$ the occurrence list is constructed in parallel. After completion, for every digram α occurring in t, the occurrence list exactly contains the nodes from $\mathsf{occ}_t(\alpha)$. However, we cannot afford to redo this traversal in every subsequent iteration. In this case we would not be able to achieve a linear running time of our algorithm.

Fortunately, there is another way of keeping track of the sets of non-overlapping occurrences. It relies on the fact that every replacement of a digram occurrence v only involves those occurrences of other digrams which overlap v. There are at most $m + 1$ such overlapping digram occurrences.

Example 12. Let us consider the tree t which is depicted in Figure 12. The occurrences which would be absorbed by the replacement of the occurrence $21 \in \mathsf{dom}_t$ of the digram $(g, 3, h)$ are highlighted. Figure 13 shows the tree t' which results from the replacement of $(g, 3, h)$ in t by the nonterminal A. All new occurrences arising from this replacement are highlighted.

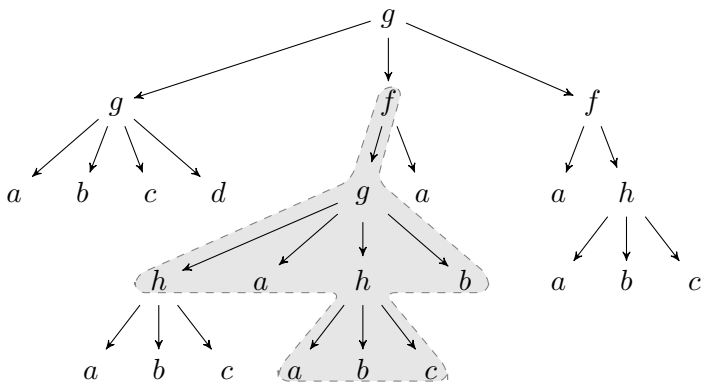

Figure 12: The tree t from Example 12.

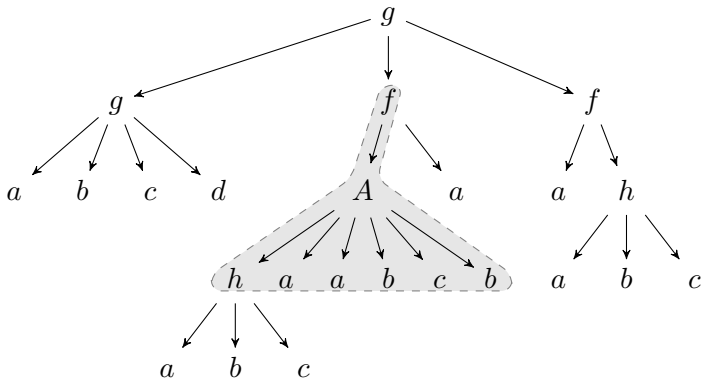

Figure 13: The tree t' from Example 12.

In our implementation, before every replacement of a digram occurrence v, we visit each digram occurrence v' which will be absorbed by the upcoming replacement. This can be done in constant time using the parent and child pointers since we limited the maximal rank of every nonterminal to the constant m. Removing the occurrence v' from the corresponding occurrence list is just a matter of setting the right pointers to null. After we replaced v, we visit every digram occurrence v' which arose from the replacement and insert it into the corresponding occurrence list. This can also be done in constant time by retrieving the corresponding digram object from the hash table and updating pointers accordingly.

However, the above approach has the following drawback: Although at

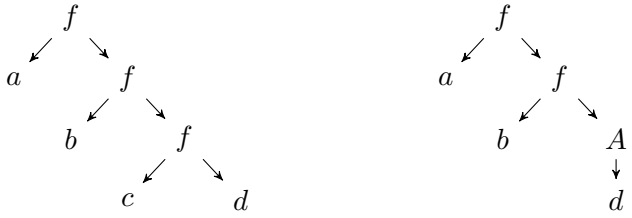

Figure 14: The trees t and t' (from left to right) from Example 13.

the beginning the occurrences list of a diagram α contains all occurrences from $\mathsf{occ}_t(\alpha)$, this property is not preserved by our implementation, i.e., in general, the set of digram occurrences contained in the occurrences list of a digram α is different from the set $\mathsf{occ}_t(\alpha)$. This is the reason, why our implementation is not guaranteed to replace a most frequent digram in each round of the replacement phase. The example below illustrates this phenomenon.

Example 13. Consider the tree t depicted in on the left side of Figure 14. At the beginning of the replacement phase, the occurrences list for the digram $\alpha = (f, 2, f)$ would exactly contain the occurrences from $\mathsf{occ}_t(\alpha) = \{2\}$. This is because the occurrences lists are constructed in the spirit of the algorithm `retrieve-occurrences` from Figure 4. Now, let us assume that we replace the digram $(f, 1, c)$ (we could easily enlarge t such that $(f, 1, c)$ is the most frequent digram and still show the same) resulting in the tree t' depicted on the right side of Figure 14. After performing this replacement and especially after updating all involved occurrences lists in the way described above, the occurrence list of α would be empty. However, we have $\mathsf{occ}_{t'}(\alpha) = \{\varepsilon\}$.

5.3.2. Retrieving a most frequent digram

We now investigate the time needed to obtain a most frequent digram in an iteration of the replacement phase of our algorithm. When referring to "most frequent diagram" we mean a digram whose current occurrences list has maximal length. As explained in the previous section, it is not guaranteed that such a digram is really a most frequent digram in the current tree.

It is easy to see that if the top digram list (which contains all digrams occurring at least $\lfloor \sqrt{n} \rfloor$ many times where n is the size of the input tree) is empty, we can obtain the most frequent digram in constant time. We just need to choose the first element of the first non-empty digram list among

the remaining $\lfloor\sqrt{n}\rfloor - 1$ digram lists. In every iteration, after we have determined the most frequent digram, we remember the first non-empty digram list in order to save our self the needless and time-consuming rechecking of the empty digram lists. This is justified due to the fact that the number of occurrences of the most frequent digrams replaced during the replacement phase is monotonically decreasing. The latter follows from the following two observations: After the replacement of all occurrences of a digram, the number of occurrences of the digrams which were already existing is unchanged or smaller than before. The number of occurrences of the digrams which arose from the replacement step (because of the new nonterminal) is less than or equal to the number of occurrences of the digram replaced.

Now, let us assume that the top digram list, i.e., the doubly linked list of all digrams occurring at least $\lfloor\sqrt{n}\rfloor$ times, is not empty. We need to scan all elements in it since the digrams contained are not ordered by their frequency. There can be roughly at most $\sqrt{n}$ digrams in the top digram list. Therefore, we need roughly $\mathcal{O}(\sqrt{n})$ time to retrieve the most frequent digram. However, by the replacement of this digram at least $\lfloor\sqrt{n}\rfloor$ edges are absorbed. It is easy to see that, all in all, obtaining the most frequent digram needs constant time on average.

In a run of our implementation, we can replace at most $n - 1$ digram occurrences and, as shown before, the replacement of each occurrence, the update of the occurrences lists and the determination of a most frequent digram can be accomplished in constant time per occurrence replacement. Thus, the whole replacement phase is completed in linear time.

5.4. Impact of the DAG representation

In the preceding section, dealing with the complexity of our implementation of TreeRePair, we did not pay attention to the underlying DAG representation of the input tree. This enabled us to concentrate on the essentials. Nevertheless, we have to clarify the impact of this representation, particularly concerning the compression performance and the running time of our implementation, since it is used by default. Only by starting our implementation with the `-no_dag` switch it avoids the DAG representation and loads the whole input tree into main memory.

Consider the sequence $\mathcal{G}_0, \mathcal{G}_1, \ldots, \mathcal{G}_n$ of grammars, where $\mathcal{G}_0$ is the grammar consisting of the single production $S_0 \to t_0$ with t_0 being the original tree to be compressed, and $\mathcal{G}_i$, $i \geq 1$ are the grammars produced during the replacement phase, see Section 3.1. Let $\mathcal{G}_i = (N_i, P_i, S_i)$ and let $(S_i \to t_i) \in P_i$,

i.e., t_i is the right-hand side of the start nonterminal of $\mathcal{G}_i$. In the DAG version of our implementation, we store t_i as an (unranked) DAG. This means in particular, that the original tree t_0 is represented as a DAG. Recall from [6] that for typical XML files, the minimal unranked DAG of the XML document tree has only 10% of the number of edges of the original unranked tree. This means that using the DAG in-memory representation, we are able to compress XML trees that can be roughly 10 times larger than without DAGs. Note also that all productions except the start production have small right-hand sides (only containing single digrams). Thus, DAG compression pays off only for the trees t_i. A DAG it is represented as a 0-bounded (linear) SLCF tree grammar $\mathcal{G}'_i$ with $\mathsf{val}(\mathcal{G}'_i) = t_i$ which we call DAG grammar in the following.

In general, the number of "physical occurrences" of a digram in the DAG grammar is smaller than the number of "virtual occurrences" of the same digram in the unfolding of the DAG. However, the computation of a most frequent digram to be replaced is still based on the virtual occurrences.

5.4.1. Initial construction of the occurrences lists

In the first iteration of the replacement phase, we need to construct the occurrences list for every digram $\alpha \in \Pi$ occurring in the input tree t. This can be accomplished by a postorder traversal of all the right-hand sides of the productions of the DAG grammar representing the input tree. If the DAG grammar contains a production $A \to s$ and the nonterminal B occurs in s (necessarily at a leaf position since all nonterminals of a DAG grammar have rank 0), then the right-hand side of B is traversed before the right-hand side s of A is traversed.

Our implementation also finds digram occurrences spanning two productions of the DAG. Assume that we visit during the postorder traversal of a right-hand side s of the DAG grammar a node v labeled with a nonterminal A of the DAG grammar. Hence v must be a leaf of s. Assume moreover that v is the i-th child of node u and that u is labeled with the symbol f. We then determine the root symbol g of the right-hand side s' of A. In case $f \neq g$, we found a new occurrence of the digram (f, i, g) which is inserted into the corresponding occurrence list (the occurrence can be specified by the pair consisting of a reference to the right-hand side s together with node u). In case $f = g$, we first have to check whether the root of s' is not an occurrence of (f, i, f), in which case this occurrence has been already inserted into the occurrence list for (f, i, f).

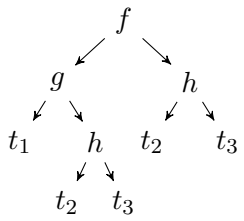

Figure 15: The tree t represented by the DAG grammar given by $S \to f\big(g(t_1, A), A\big)$ and $A \to h(t_2, t_3)$.

5.4.2. Updating the occurrences lists

Regarding the graph representation of a DAG, a tree node can exhibit multiple parent nodes. In fact, a node has multiple parent nodes if it is the root of the right-hand side of a production of the corresponding DAG grammar and if this production is referenced multiple times.

To capture all digram occurrences which are absorbed by the replacement of a digram we need to take care of the above fact. Instead of removing one occurrence formed by the node being replaced and its parent, we need to iterate over possibly multiple parents and remove all corresponding occurrences. The same holds when registering the new digram occurrences which arise from the replacement of a digram occurrence.

It is easy to see that our linear running time is not negatively affected by this loop over all parents. Far from it — as mentioned earlier, the DAG representation saves us time by avoiding repetitive re-calculations.

5.4.3. Replacing the digrams

The third and last scenario in which we have to take special care of the DAG representation is when replacing an occurrence of a digram α spanning two productions of the DAG grammar. Due to our restriction on digrams with equal parent and child symbols, the digram α has to have different parent and child symbols. In the following we want to use an example to describe what needs to be done when replacing the digram α.

Consider the DAG grammar given by the productions $S \to f\big(g(t_1, A), A\big)$ and $A \to h(t_2, t_3)$ which represents the tree t depicted in Figure 15. Imagine that we want to replace the sole occurrence of the digram $(f, 2, h)$, i.e., an occurrence spanning two productions. In order to do that we mainly have to complete the following three steps.

(1) We first have to introduce a new production for every child of the node

$2 \in \mathsf{dom}_t$. Thus, we obtain two new productions $B \to t_2$ and $C \to t_3$. Note that we can skip this step for every child node which is already labeled by a nonterminal of the DAG grammar.

(2) We need to update the production with left-hand side A to $A \to h(B, C)$.

(3) Finally, we introduce a new nonterminal D representing the digram $(f, 2, h)$ and update the production for S to $S \to D(g(t_1, A), B, C)$.

The above steps are necessary because the production with left-hand side A is referenced more than once. If this would not be the case, then we could have directly connected the children of $2 \in \mathsf{dom}_t$ to the newly introduced node labeled by D and removed the production with left-hand side A from the grammar. Since the maximal rank of a non terminal is restricted to a constant, the replacement of a digram occurrence spanning two productions of the DAG grammar can still be accomplished in constant time.

All in all, it has become clear that even when representing the input tree of our algorithm as a DAG our implementation runs in linear time. Even more, the DAG representation saves us time by avoiding repetitive re-calculations.

6. Succinct grammar coding

In order to obtain a compact representation of an XML document tree in terms of file size, we further need to compress the generated SLCF tree grammar by a binary succinct coding. Our succinct coding, of which we give an overview in this section, is independent from the TreeRePair algorithm and can actually be applied to any SLCF grammar. Hence, it could, for example, also be used to further compress grammars generated by BPLEX or CLUX.

The technique we use is loosely based on the DEFLATE algorithm described in [12]. 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 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).

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 [12], 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. We refer the reader to [23] for a detailed description of the succinct coding.

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

7. Experiments

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

7.1. 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 enabled. This tool is used to apply a succinct coding to a grammar generated by the BPLEX algorithm, as described in [28]. However, this has

XML document	File size (kb)	# Edges	Depth	# Element types	Source
1998statistics	349	28 305	5	46	1
catalog-01	4 219	225 193	7	50	9
catalog-02	44 656	2 390 230	7	53	9
dictionary-01	1 737	277 071	7	24	9
dictionary-02	17 128	2 731 763	7	24	9
dblp	117 822	10 802 123	5	35	2
EnWikiNew	4 843	404 651	4	20	3
EnWikiQuote	3 134	262 954	4	20	3
EnWikiSource	13 457	1 133 534	4	20	3
EnWikiVersity	5 887	495 838	4	20	3
EnWikTionary	99 201	8 385 133	4	20	3
EXI-Array	5 347	226 522	9	47	5
EXI-factbook	1 214	55 452	4	199	5
EXI-Invoice	266	15 074	6	52	5
EXI-Telecomp	3 700	177 633	6	39	5
EXI-weblog	1 104	93 434	2	12	5
JST_gene.chr1	4 202	216 400	6	26	8
JST_snp.chr1	13 795	655 945	7	42	8
medline02n0328	51 751	2 866 079	6	78	6
NCBI_gene.chr1	6 862	360 349	6	50	8
NCBI_snp.chr1	63 941	3 642 224	3	15	8
sprot39.dat	111 175	10 903 567	5	48	7
treebank	19 551	2 447 726	36	251	4

Table 1: Characteristics of the XML documents used in our tests. The values in the "Source"-column match the source IDs in Table 2. The depth of an XML document tree specifies the length (number of edges) of the longest path from the root of the tree to a leaf.

no great influence on the running time 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 byte-code 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. The source of the original XML documents is given in Table 2.

7.2. Comparing tree grammar compressors

We compared BPLEX [7] (sf.net/projects/bplex), CluX [5] (supplied to us by the authors), DAG and bDAG [6], each of which produces SLCF tree grammars. The latter two generate minimal DAGs, either for the unranked XML tree (DAG), or for the binary first-child/next-sibling encoded XML

ID	Source
1	http://www.cafeconleche.org/examples
2	http://dblp.uni-trier.de/xml
3	http://download.wikipedia.org/backup-index.html
4	http://www.cs.washington.edu/research/xmldatasets
5	http://www.w3.org/XML/EXI
6	http://www.ncbi.nlm.nih.gov/pubmed
7	http://expasy.org/sprot
8	http://snp.ims.u-tokyo.ac.jp
9	http://cs.uwaterloo.ca/ tozsu/ddbms/projects/xbench

Table 2: Sources of the XML documents from Table 1.

tree (bDAG), cf. [7] where it was observed already that DAG gives better compression ratios than bDAG. Note that DAGs and bDAGs can be seen as SLCF tree 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 the number of edges, number of nonterminals, size of the start production, and depth of the grammar. The latter is the maximal nesting depth of nonterminals in a chain of productions, which, as we see later, influences the traversal speed. TreeRePair was run with `-optimize edges` (which prunes all nonterminals with a save value ≤ 0) and its default setting $m = 4$ for the maximal rank of nonterminals (see the remarks below), CluX was run with configuration `ConfEdges.xml`, and BPLEX was run with its standard parameters. In the case of BPLEX, we used the `gprint` tool to eliminate nonterminals that are referenced only once.

The results of our experiments are shown in Table 3 (the column "%" gives the compression ratio with respect to the number of edges, i.e, 100 $\times$ number of edges of the output grammar / number of edges of the input tree) and Table 4 (all values are averages over our XML corpus). Note that TreeRePair yields the best results with respect to compression ratio in terms of grammar size. Recall that the size of a grammar is the total number of edges in all right-hand sides. Moreover, with respect to running time and memory consumption TreeRePair is comparable to DAG and bDAG, which are the best in these categories (memory consumption is even better than for DAG). The depth and the size of the start production are missing for CluX. The reason for this is that, as already explained in the introduction, CluX splits the input tree into several packages and generates a grammar for each package. Hence, it is not clear how to measure the whole depth and the size

XML document	TreeRePair		BPLEX		CluX		DAG		bDAG	
	%	#NT	%	#NT	%	#NT	%	#NT	%	#NT
1998statistics	1.7	54	1.8	168	1.69	37	4.9	15	8.5	31
catalog-01	1.7	400	2.2	1251	1.63	291	3.8	506	3.1	520
catalog-02	1.1	965	1.4	3045	1.52	1499	1.4	792	2.2	805
dictionary-01	7.7	1676	8.4	3994	8.71	1248	21.1	448	28.0	2058
dictionary-02	5.9	9757	6.6	23209	8.52	11672	20.0	2414	24.9	16281
EnWikiNew	2.3	667	2.4	1369	2.42	476	8.7	29	17.3	23
EnWikiQuote	2.4	452	2.6	985	2.58	323	9.1	25	18.1	19
EnWikiSource	1.1	861	1.3	1895	1.82	1106	8.8	24	17.5	19
EnWikiVersity	1.4	525	1.5	1043	1.61	423	8.8	24	17.6	19
EnWikTionary	1.0	4535	1.1	6402	1.48	6315	8.7	30	17.3	26
EXI-Array	0.4	123	0.6	383	0.53	142	42.2	13	56.5	8
EXI-factbook	2.4	145	4.1	1423	2.58	146	8.1	293	9.2	236
EXI-Invoice	0.7	14	0.6	40	0.93	20	7.1	15	13.7	6
EXI-Telecomp	0.1	21	0.1	47	0.08	21	5.6	15	11.2	10
EXI-weblog	0.1	13	0.0	24	0.05	11	9.1	2	18.2	2
JST_gene.chr1	1.8	354	2.2	1113	2.99	126	4.2	76	6.8	114
JST_snp.chr1	1.5	856	2.1	4193	1.54	634	3.6	242	6.2	282
medline02n0328	4.1	9064	5.2	33976	6.73	13010	22.8	3960	25.8	20013
NCBI_gene.chr1	1.4	504	2.4	3631	1.68	328	4.5	436	4.0	605
NCBI_snp.chr1	0.0	17	0.0	23	0.03	291	11.1	2	22.2	2
treebank	20.7	32857	23.3	76109	34.85	48358	53.8	24746	59.4	43586
sprot39.dat	2.3	20224	3.2	111167	4.27	33102	16.1	10243	13.2	31116
dblp	3.9	25250	4.3	38712	5.65	30430	11.1	3378	19.4	6592

Table 3: Performances of the tree grammar compressors on the individual XML documents.

	TreeRePair	BPLEX	CluX	DAG	bDAG
Edges (%)	2.8	3.5	4.4	12.7	18.2
Number of nonterminals	6 715	32 159	12 133	4 635	8 560
Size of start production (%)	72	61	—	89	93
Depth	24	78	—	4	24
Time (seconds)	19	934	101	15	9
Memory (MB)	72	550	395	123	59

Table 4: Summary of the performance of the tree grammar compressors

of the start production of the CluX output.

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; see Table 5 for a comparison. Our two examples from Section 4.1 and 4.2 offer a possible explanation for this fact: In the binary first-child/next-sibling encodings of XML trees, we quite often find comb-like subtrees (or long lists), where all symbols have rank at most 2. On such trees, a small m-value seems to be a better choice. Our XML documents are quite flat (depth ≈ 5, see Table 1) and have high branching degree. This 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

Max. rank	0	1	2	3	4	5	6
Edges (%)	55.02	3.29	2.92	2.89	2.86	2.89	2.89
# NTs	1 265	5 539	4 712	4 916	4 753	4 956	4 958
File size (%)	2.12	0.51	0.47	0.47	0.46	0.47	0.46
Time (sec)	7.0	8.4	9.3	9.5	9.6	9.8	9.8
Mem (MB)	44	44	45	47	47	47	47

Table 5: Impact of the maximal rank.

	TRePair	bplex	CluX	XMill	PPM	SPPM	SHuff	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 6: Performance for XML file compression

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 of all documents in our XML corpus).

7.3. 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 [21], XMLPPM in version 0.98.3 [10], see xmlppm.sf.net, SCMPPM [1], see www.infor.uva.es/∼jadiego, 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 6. CluX was run with configuration `ConfSize.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 [28]. Table 6 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 stripped XML document. 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

	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 7: Speeds for iterative pre-order traversals

structure only needs additional space O(depth of the grammar) on SLCF tree grammar compressed trees using the stack configurations from [7], see also the next paragraph.

7.4. 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 [33] (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 side 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 [7]. These stack configurations are proportional to the depth of the grammar, which in our examples has an average value of 24, see Table 4. We calculated the size of the grammar memory representation (called *index size* in Table 7) and measured traversal times (excluding the time needed to generate the grammar's memory representation) of our implementation. As a yardstick, we also give these values for the succinct trees of [33], 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 [3], which in turn are faster than our grammar compressed trees. Table 7 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.

8. Conclusions

We presented TreeRePair, a generalization to trees of Larsson and Moffat's RePair algorithm. TreeRePair generates straight-line linear context-free tree grammars. On our test corpus the produced grammars are smaller than those produced by previous grammar-based compressors. Moreover, we developed a Huffman-based succinct coding of the resulting grammars. The compression ratio of the resulting file compressor is comparable to the best known XML file compressors. Finally, we demonstrated that the grammars produced by TreeRePair can be used for traversing the original XML tree. Our experiments showed a reasonable trade-off between compression performance and tree traversal speed.

Acknowledgment. The first author was supported by the DFG research project ALKODA. We are thankful to the anonymous referees for careful reading of the submitted manuscript.

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] T. Akutsu. A bisection algorithm for grammar-based compression of ordered trees. *Inf. Process. Lett.*, 110(18-19):815–820, 2010.

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

[4] P. Bille, I. Li Gørtz, G. M. Landau, and O. Weimann. Tree compression with top trees. In *ICALP*, 2013. To appear.

[5] S. Böttcher, R. Hartel, and C. Krislin. CluX - clustering XML sub-trees. In *ICEIS (1)*, pages 142–150, 2010.

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

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

[8] A. Gascón C. Creus and G. Godoy. One-context unification with STG-compressed terms is in NP. In *RTA*, pages 149–164, 2012.

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

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

[11] H. Comon, M. Dauchet, R. Gilleron, F. Jacquemard, C. Löding, D. Lugiez, S. Tison, and M. Tommasi. Tree automata techniques and applications. Available at: http://www.grappa.univ-lille3.fr/tata, 2007.

[12] P. Deutsch. DEFLATE compressed data format specification version 1.3. 1996.

[13] P. J. Downey, R. Sethi, and R. E. Tarjan. Variations on the common subexpression problem. *J. ACM*, 27(4):758–771, 1980.

[14] M. Frick, M. Grohe, and C. Koch. Query evaluation on compressed trees (extended abstract). In *LICS*, pages 188–197, 2003.

[15] A. Gascón, G. Godoy, and M. Schmidt-Schauß. Unification and matching on compressed terms. *ACM Trans. Comput. Log.*, 12(4):26, 2011.

[16] D. E. Knuth. *The Art of Computer Programming, Volume I: Fundamental Algorithms*. Addison-Wesley, 1968.

[17] N. Kobayashi, K. Matsuda, and A. Shinohara. Functional programs as compressed data. In *PEPM*, pages 121–130, 2012.

[18] C. Krislin. Optimierung grammatik-basierter XML-Kompression. Diplomarbeit, Faculty for Electrical Engineering, Computer Science and Mathematics, University of Paderborn (Germany), 2008.

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

[20] J. Levy, M. Schmidt-Schauß, and M. Villaret. The complexity of monadic second-order unification. *SIAM J. Comput.*, 38(3):1113–1140, 2008.

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

[22] M. Lohrey and S. Maneth. The complexity of tree automata and XPath on grammar-compressed trees. *Theor. Comput. Sci.*, 363(2):196–210, 2006.

[23] M. Lohrey, S. Maneth, and R. Mennicke. Tree structure compression with RePair. *CoRR*, abs/1007.5406, 2010.

[24] M. Lohrey, S. Maneth, and R. Mennicke. Tree structure compression with RePair. In *DCC*, pages 353–362, 2011.

[25] M. Lohrey, S. Maneth, and E. Nöth. XML compression via DAGs. In *ICDT*, pages 69–80, 2013.

[26] M. Lohrey, S. Maneth, and M. Schmidt-Schauß. Parameter reduction and automata evaluation for grammar-compressed trees. *J. Comput. Syst. Sci.*, 78(5):1651–1669, 2012.

[27] S. Maneth and G. Busatto. Tree transducers and tree compressions. In *FoSSaCS*, pages 363–377, 2004.

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

[29] S. Maneth and T. Sebastian. Fast and tiny structural self-indexes for XML. *CoRR*, abs/1012.5696, 2010.

[30] M. Murata, D. Lee, M. Mani, and K. Kawaguchi. Taxonomy of XML schema languages using formal language theory. *ACM Trans. Internet Techn.*, 5(4):660–704, 2005.

[31] F. Neven. Automata theory for XML researchers. *SIGMOD Record*, 31(3):39–46, 2002.

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

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

[34] A. Schmidt, F. Waas, M. L. Kersten, M. J. Carey, I. Manolescu, and R. Busse. XMark: A benchmark for XML data management. In *VLDB*, pages 974–985, 2002.

[35] M. Schmidt-Schauß. Polynomial equality testing for terms with shared substructures. Frank report 21, Fachbereich Informatik und Mathematik. J. W. Goethe-Universität Frankfurt am Main (German), 2005.

[36] M. Schmidt-Schauß. Matching of compressed patterns with character-variables. In *RTA*, pages 272–287, 2012.

[37] M. Schmidt-Schauß, D. Sabel, and A. Anis. Congruence closure of compressed terms in polynomial time. In *FroCos*, pages 227–242, 2011.

[38] T. Schwentick. Automata for XML - a survey. *J. Comput. Syst. Sci.*, 73(3):289–315, 2007.

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