Efficient Memory Representation of XML Document Trees

Giorgio Busatto [a], Markus Lohrey [b], Sebastian Maneth [c,1]

[a] *Department für Informatik, Universität Oldenburg, Germany*
giorgio.busatto@informatik.uni-oldenburg.de

[b] *FMI, Universität Stuttgart, Germany*
lohrey@informatik.uni-stuttgart.de

[c] *National ICT Australia Ltd.*[1] *and University of New South Wales, Sydney, Australia*
sebastian.maneth@nicta.com.au

Abstract

Implementations that load XML documents and give access to them via, e.g., the DOM, suffer from huge memory demands: the space needed to load an XML document is usually many times larger than the size of the document. A considerable amount of memory is needed to store the tree structure of the XML document. In this paper, a technique is presented that allows to represent the tree structure of an XML document in an efficient way. The representation exploits the high regularity in XML documents by compressing their tree structure; the latter means to detect and remove repetitions of tree patterns. Formally, context-free tree grammars that generate only a single tree are used for tree compression. The functionality of basic tree operations, like traversal along edges, is preserved under this compressed representation. This allows to directly execute queries (and in particular, bulk operations) without prior decompression. The complexity of certain computational problems like validation against XML types or testing equality is investigated for compressed input trees.

Key words: Tree grammar, compression, in-memory XML representation

1 Introduction

There are many scenarios in which trees are processed by computer programs. Often it is useful to keep a representation of the tree in main memory in order to retain

[1] National ICT Australia is funded through the Australian Government's *Backing Australia's Ability* initiative, in part through the Australian Research Council.

fast access. If the trees to be stored are very large, then it is important to use a memory efficient representation. A recent, most prominent example of large trees are XML documents which are sequential representations of ordered (unranked) trees, and an example application which requires to materialize (part of) the document in main memory is the evaluation of XML queries. The latter is typically done using one of the existing XML data models, e.g., the DOM. Benchmarks show that a DOM representation in main memory is 4–5 times larger than the original XML file. This can be understood as follows: a node of the form `<a/>` needs 4 bytes in XML; but as a tree node it needs at least 16 bytes: a name pointer, plus three node pointers to the parent, the first child, and the next sibling (see, e.g., Chapter 8 of [24]). There are some improvements leading to more compact representations, e.g., Galax [10] uses only 3–4 times more main memory than the size of the file. Another, more memory efficient data model for XML is that of a binary tree. As shown in [25], the known XML query languages can be readily evaluated on the binary tree model.

In this paper, we concentrate on the problem of representing binary trees in a space efficient way, so that the functionality of the basic tree operations (such as the traversal along edges) is preserved. Instead of compression, this is often called *data optimization* [19]. There are two fundamentally different approaches to small tree representation: pointer-based and succinct pointer-less [19]. The latter means to pack the tree into a small bit-array in such a way that basic navigations through the tree can be realized in constant time; recently there has been new progress in succinct tree representations [11,16,17]. With respect to memory requirements, succinct representation are more competitive than pointer-based representations; with respect to traversal speed, however, pointer structures are much more competitive than succinct representations. Here we deal with pointer-based tree representations. As common, we use as size measure the number of pointers needed. The actual cost of a pointer is implementation dependent and is not considered here. Our technique is a generalization of the well-known sharing of common subtrees. The latter means to determine during a bottom-up phase, using a hash table, whether the current subtree has occurred already, and if so to represent it by a pointer to its previous occurrence. In this way the minimal unique DAG (directed acyclic graph) of the tree is obtained in linear time. For common XML documents, the size (measured in number of pointers) of the minimal DAG is about 1/10 of the size of the original tree [4]. Our representation is based on sharing of common subgraphs of a tree. The resulting sizes are 1/2–1/4 of the size of the minimal DAG (even if multiplicity counters are used in the DAG to represent consecutive edges to the same subtree). To our knowledge, this is the most efficient pointer-based tree representation currently available. At the same time, the complexity of querying, e.g. using XQuery, stays the same as for DAGs [22]. We therefore believe that our representation is better suited for in-memory storage of XML documents, than DAG-based representations. Note that our representation can also be incrementally updated; as experiments show [13], even after thousands of updates, the additional overhead on the structure stays below 40% with respect to the size of compressing from scratch.

In a succinct pointer-less representation, any tree of n nodes has the same memory requirement, no matter if the tree is highly regular (i.e. contains many occurrences of identical subtrees) or non-regular. Thus, for very large regular trees, which are typical in applications, the size of our compressed representation will be smaller than that of any succinct pointer-less representation. It should be noted that succinct pointer-less and pointer-based representations are *not* competing approaches, but can be combined: our compression algorithm generates certain tree grammars which themselves consist of many small trees. These trees can of course be stored succinctly instead of using pointers [13]. The result is guaranteed to be smaller than those obtained by any of the two approaches in separation.

Of course, an XML document contains more components than just tree nodes: a node may have attributes, and a leaf may have character data. Both types of values we keep in string buffers. When traversing the XML tree, we keep information on how many nodes before (in document order) the current node (i) have attributes and (ii) how many have character data. Thus, it suffices to store two additional bits per node indicating whether the node has attributes or character data. These numbers determine for a node the correct indices into the attribute and data value buffers, respectively. With this in mind, it is straightforward to implement a DOM proxy for our representation. Note that attribute and character values can be stored more space efficiently using standard techniques [1]. The XML file compression tool XMill [21] separates data values into containers and compresses them individually using standard methods. The result is stored together with the tree structure. For typical XML files, about 50% of the total file size is made up by data values, whereas the remaining 50% is made up by the tree structure. It is likely that compressing the tree structure by the technique presented here will further improve XMill's compression ratio.

We now describe our representation in more detail. Consider the tree $c(c(a,a), c(a,a))$, or, in XML `<c><c><a/><a/></c><c><a/><a/></c></c>`. It consists of seven nodes and six edges. The minimal DAG for this tree has three nodes u, v, w and four edges ('first-child' and 'second-child' edges from u to v and from v to w). The minimal DAG can also be seen as the minimal *regular tree grammar* that generates the tree [23]: the shared nodes correspond to nonterminals of the grammar. For example, the above DAG is generated by the regular tree grammar with productions $S \to c(V,V)$, $V \to c(W,W)$, and $W \to a$. A generalization of sharing of subtrees is the sharing of arbitrary patterns, i.e., connected subgraphs of a tree. In a graph model it leads to the well-known notion of sharing graphs which are graphs with special "begin-sharing" and "end-sharing" edges, called fan-ins and fan-outs [20]. Since fan-in/out pairs can be nested, this structure makes a doubly-exponential compression ration possible. In contrast, a DAG is at most exponentially smaller than the tree it represents. A sharing graph can be seen as a *context-free (cf) tree grammar* [23]. In a cf tree grammar nonterminals can appear inside of an intermediate tree (as opposed to at the leaves in the regular case); formal parameters $y_1, y_2, \dots$ are used in productions in order to indicate where to glue the subtrees of the nonterminal which is being replaced. Finding the smallest sharing graph for a given tree is equivalent to finding the smallest cf tree grammar that generates the tree. Unfortunately, the latter problem is NP-hard: already finding the smallest cf (string) grammar for a given string is NP-complete [5]. The first main contribution of this paper is a linear time algorithm that finds a cf tree grammar for a given tree. On common XML documents the algorithm performs well, obtaining grammars that are 1.5–2 times smaller than the minimal DAGs. As an example, consider the tree $t = c(c(a,a), d(c(a,a), c(c(a,a), d(c(a,a), c(a,a)))))$ from Fig. 1 which has 18 edges. The minimal DAG, written as a tree grammar,

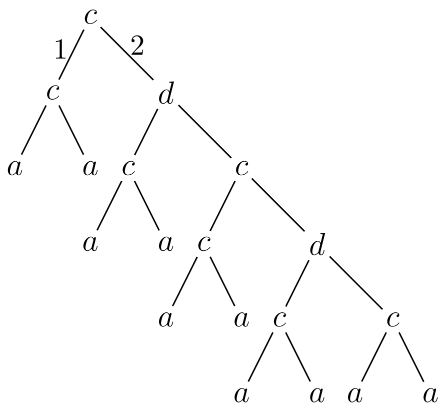
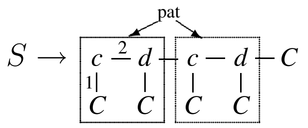


Fig. 1. Tree $t = c(c(a,a), d(c(a,a), c(c(a,a), d(c(a,a), c(a,a)))))$.



$$C \to c(A,A)$$
$$A \to a$$

$$\begin{array}{lcl} S & \to & B(B(C)) \\ B(y_1) & \to & c(C, d(C, y_1)) \\ C & \to & c(A,A) \\ A & \to & a \end{array}$$

Fig. 2. Regular and cf tree grammars generating $\{t\}$ from Fig 1.

can be seen on the left of Fig. 2. It is the initial input to our algorithm BPLEX which tries to transform the grammar into a smaller cf tree grammar. It does so by going bottom-up through the right-hand sides of productions, looking for multiple (non-overlapping) occurrences of patterns, i.e., connected subgraphs. In our example, the tree pattern pat (consisting of two nodes labeled c and d and their left children labeled C) appears twice in the right-hand side of the first production. A pattern p in a tree can conveniently be represented by a tree t_p with formal parameters $y_1, \ldots, y_r$ at leaves: simply add to p for each edge, leading from a node of p to a node outside of p, a new leaf and label the jth such leaf (in preorder) by the parameter y_j. Thus, $t_{\text{pat}} = c(C, d(C, y_1))$ in our example. This tree becomes the right-hand side of a new nonterminal B and the right-hand side of the production for the start nonterminal S becomes $B(B(C))$. The resulting cf tree grammar is shown on the right of Fig. 2. Clearly, this grammar generates exactly one tree. Such a cf tree grammar is called *straight-line* (for short, SL). The straight-line notion is well-known from string grammars (see, e.g., [29,30]).

The BPLEX algorithm is presented in Section 3. In Section 4 we discuss the application of BPLEX to XML documents. Experimental results are presented in Section 5. In Section 6 we study two problems for SL cf tree grammars that are important for XML documents: (1) to validate against an XML type and (2) to test equality of the trees generated by two SL cf tree grammars. Since BPLEX generates SL context-free tree grammars of a more restricted form (additionally: linear in the parameters) we also consider problems (1) and (2) for this restricted case. Concerning (1), it is shown that for an XML type T, represented by a (deterministic) bottom-up tree automaton B with m states, we can test whether or not the tree represented by G has type T in time $O(m^k \times |B| \times \mathrm{size}(G))$. Here, k is the maximal number of parameters of the nonterminals of G, $\mathrm{size}(G)$ is the sum of the sizes of all right-hand sides of the grammar G, and $|B|$ is the size of the transition function of the automaton B. Note that a Core XPath query Q can be transformed into an equivalent deterministic bottom-up tree automaton with $O(2^{|Q|})$ many states. This leads to an algorithm for Core XPath evaluation on XML documents represented by SL context-free tree grammars, whose running time is (i) exponential in $k \cdot |Q|$ (where k is the maximal number of parameters) and (ii) polynomial in the size of the grammar. This result nicely fits to a result from [4], where it was shown that a Core XPath query Q can be evaluated on an XML document represented by a DAG D in time $O(2^{|Q|} \times |D|)$, where $|D|$ is the number of nodes of D. Concerning problem (2), it is proved that the equivalence of two SL cf tree grammars can be tested in (i) polynomial space with respect to the sum of sizes of the two grammars and (ii) in polynomial time with respect to the sum of sizes if the grammars are assumed to be linear (i.e., no parameter appears more than once in the right-hand side of any production).

2 Preliminaries

The empty string over some alphabet will be denoted by ε. A finite set Σ together with a mapping $\mathrm{rank} : \Sigma \to \mathbb{N}$ is called a *ranked alphabet*. The set of all Σ-labeled, ordered, rooted, and ranked *trees* is denoted by T_Σ. Here, "ordered" and "ranked" means that the children of an f-labeled node ($f \in \Sigma$) has exactly $\mathrm{rank}(f)$ many children, which are linearly ordered. Such a tree t will be also represented as a term: If the root of t is labeled with f and t_i is the subtree of t, which is rooted at the i-th child of the root of t ($1 \le i \le n = \mathrm{rank}(f)$), then t can be represented by the term $f(t_1, \ldots, t_n)$. For a set A, $T_\Sigma(A)$ is the set of all trees over $\Sigma \cup A$, where all elements of A have rank 0. We fix a set of *parameters* $Y = \{y_1, y_2, \ldots\}$ and, for $k \ge 0$, $Y_k = \{y_1, \ldots, y_k\}$. For a ranked tree t, $V(t)$ denotes its set of nodes and $E(t)$ its set of edges. Each node in $V(t)$ can be represented by a sequence u of integers describing the path from the root of t to the desired node (Dewey notation). Formally, for a sequence $u \in \mathbb{N}^*$ and a tree t we define the node $u[t] \in V(t)$ inductively as follows: Let $\varepsilon[t]$ be the root of t. Now assume that $u = i.v$ with $i \in \mathbb{N}$ and $v \in \mathbb{N}^*$. If t is not of the form $t = f(t_1, \ldots, t_n)$ with $f \in \Sigma$ and $n \geq i$, then t_u is undefined. Otherwise, we set $u[t] = v[t_i]$. In the rest of the paper, we will often identify the node $u[t]$ with the sequence u. The label of the node $u[t]$ is denoted by $t[u]$ and the subtree of t rooted at u is denoted by t/u. For example, for the tree t from Fig. 1, $1.1[t]$ is the left-most leaf of t and we have $t[1.1] = a$. For symbols $a_1, \ldots, a_n$ of rank zero and trees $t_1, \ldots, t_n$, $[a_1 \leftarrow t_1, \ldots, a_n \leftarrow t_n]$ denotes the substitution of replacing each leaf labeled a_i by the tree t_i, $1 \leq i \leq n$.

Tree Grammars

Context-free (cf) tree grammars are a natural generalization of cf grammars to trees (see, e.g., Section 15 in [18]). A cf tree grammar G consists of ranked alphabets N and Σ of nonterminal and terminal symbols, respectively, of a start nonterminal S (of rank zero), and of a finite set of productions of the form $A(y_1, \ldots, y_k) \to t$, where A is a nonterminal in N of rank $k \geq 0$ and t is a tree over nonterminal symbols, terminal symbols, and parameters in Y_k which may appear at leaves, i.e., $t \in T_{N\cup\Sigma}(Y_k)$. For trees $s, s' \in T_{N\cup\Sigma}$ we write $s \Rightarrow_G s'$ if s' is obtained from s by replacing a subtree $A(s_1, \ldots, s_k)$ by the tree $t[y_1 \leftarrow s_1, \ldots, y_k \leftarrow s_k]$ where $A(y_1, \ldots, y_k) \to t$ is a production of G. Thus, the parameters are used to indicate where to glue the subtrees of a nonterminal occurrence, when applying a production to it. The language generated by G is

$$L(G) = \{s \in T_\Sigma \mid S \Rightarrow_G^* s\}.$$

Observe that a parameter can cause copying (if it appears more than once in a right-hand side) or deletion (if it does not appear in a right-hand side). For example, the cf tree grammar with productions $S \to A(a)$, $A(y_1) \to A(c(y_1, y_1))$, $A(y_1) \to y_1$ generates the language of all full binary trees over the binary symbol c and the constant symbol a. The grammars which are generated by our BPLEX algorithm ("SLT grammars", see below) will have neither copying nor deletion, i.e., every parameter will appear exactly once in a right-hand side.

A cf tree grammar is *regular* if all nonterminals have rank 0. It is *straight-line* (for short, SL) if each nonterminal A has exactly one production (with right-hand side denoted $\mathrm{rhs}(A)$) and the nonterminals can be ordered as $A_1, \ldots, A_n$ in such a way that A_1 is the start nonterminal and $\mathrm{rhs}(A_i)$ has no occurrences of A_j for $j \leq i$ (such an order is called "SL order"). Thus, an SL cf tree grammar G can be defined by a tuple $(N, \Sigma, \mathrm{rhs})$ where the set of nonterminals N is ordered (let $A_1, \ldots, A_n$ be the order on N) and rhs is a mapping from N to $T_{N\cup\Sigma}(Y)$ such that for all $1 \leq i \leq n$: $\mathrm{rhs}(A_i) \in T_{\{A_{i+1},\ldots,A_n\}\cup\Sigma}(Y_k)$, where k is the rank of A_i. A cf tree grammar is *linear* if, for every production $A(y_1, \ldots, y_k) \to t$, each parameter y_i

occurs at most once in t, and it is *nondeleting* if each y_i occurs at least once in t. In the sequel we use *SLT grammar* to stand for “SL linear nondeleting cf tree grammar”.

3 The BPLEX Algorithm

The purpose of grammar-based tree compression is to find a small grammar that generates a given tree. The size of such a grammar can be considerably smaller than the size of the tree, depending on the grammar formalism chosen. For example, finding the smallest regular tree grammar that generates a given tree t can be done in linear time, and the resulting grammar is equivalent to the minimal DAG of the tree. The minimal regular tree grammar G_t is also straight-line (any grammar that generates exactly one element can be turned into an SL grammar). The initial input to our compression algorithm BPLEX is the grammar G_t: BPLEX takes an arbitrary SL regular tree grammar as input and outputs a (smaller) SLT grammar. As mentioned in the Introduction, moving from regular to cf tree grammars corresponds to generalizing the sharing of common subtrees to the sharing of arbitrary tree patterns (connected subgraphs of a tree).

The basic idea of the algorithm is to find tree patterns that appear more than once in the input grammar (in a non-overlapping way), and to replace them by new nonterminals that generate the corresponding patterns. We call this technique *multiplexing* because multiple occurrences of the replaced patterns are represented only once in the output. The order in which the algorithm scans the nodes in the right-hand sides of the input grammar corresponds to scanning the generated tree bottom up; for this reason, the algorithm is called BPLEX (for *bottom-up multiplexing*).

Before we explain the BPLEX-algorithm in more detail, we first need some definitions. Assume that G is an SLT grammar with nonterminals $A_1, \ldots, A_h$ (in SL order). Let $l \leq h$. With $<^l_G$ we denote the ordering on the union

$$V^l_G = \bigcup_{i=1}^{l} V(\mathrm{rhs}_G(A_i))$$

(here we assume w.l.o.g. that the node sets $V(\mathrm{rhs}_G(A_1)), \ldots, V(\mathrm{rhs}_G(A_l))$ are pairwise disjoint) obtained by scanning $\mathrm{rhs}_G(A_l)$ through $\mathrm{rhs}_G(A_1)$, each in left-to-right postorder. A node $\rho \in V^l_G$ will be identified with its *address* $z = (j, u) \in \{1, \ldots, l\} \times \mathbb{N}^*$ in G, where j is such that $\rho \in V(\mathrm{rhs}_G(A_j))$ and $u[\mathrm{rhs}_G(A_j)] = \rho$ (i.e., u is the path in the tree $\mathrm{rhs}_G(A_j)$ to the node ρ). If z is a node in V^l_G that is not the root of $\mathrm{rhs}_G(A_1)$, then $\mathrm{next}(<^l_G, z)$ is the node following z in the order $<^l_G$.

BPLEX (see Fig. 3) takes as input an SL regular tree grammar G (with say l nonterminals $A_1, \ldots, A_l$) and three parameters specifying

```
procedure BPLEX(G: grammar, K_N: int, K_S: int, K_R: int): grammar
begin
    A_l := last symbol in the SL ordering of G
    z := leftmost leaf of rhs_G(A_l)
    while true do
        RepM := RepM(G, z, K_N)
        NewM := NewM(G, z, K_N, K_S, K_R)
        if NewM ≠ ∅ or RepM ≠ ∅ then
            m := max-match(NewM, RepM)
            if m ∈ RepM then
                G := G[m ← A], with rhs_G(A) = p_m
            else
                k := rank(p_m)
                A := fresh(G, k)
                G := add(G, A(y_1, ..., y_k) → p_m)
                G := G[m, c_m ← A]
            fi
        elseif ∃w ∈ V_G^l : z <_G^l w then z := next(<_G^l, z)
        else break
        fi
    od
    return G
end BPLEX
```

Fig. 3. The BPLEX algorithm.

(1) the maximum number K_N of nodes and productions that are examined when computing patterns matching at a given node,
(2) the maximum size K_S of a new pattern, and
(3) the maximum rank K_R of a new pattern.

In our analysis of BPLEX, we will consider K_N, K_S, and K_R as fixed constants. This will be crucial in order to obtain a linear running time for BPLEX. ¿From G, BPLEX computes a sequence of SLT grammars, each having at least l nonterminals. Moreover, we store a current address z in the current SLT grammar. At each step, BPLEX computes a set of *repeated matches* by comparing the tree patterns occurring at z with the right-hand sides of the last K_N productions of G with index greater than l, and a set of *new matches* by finding pairs of non-overlapping occurrences of tree patterns at z and at the K_N most recently visited nodes (thus exploiting the well-known idea of a sliding window that appears e.g. in many implementations of the LZ77 compression scheme, cf. the discussion in Section 7). If at least one match is found, BPLEX performs the sharing that provides the highest size reduction for the grammar, it moves to the next node otherwise. If there is no next node, then it returns the current SLT grammar.

We now examine the algorithm in detail. We describe the progress of the computation through a sequence of configurations $(G_1, z_1), \ldots, (G_h, z_h)$ where, for each $1 \leq i \leq h$, G_i is an SLT grammar generating the uncompressed tree, and z_i is an address in G_i denoting the node that is examined during the i-th iteration (the *current* address). $G_1 = G$ is the input to the algorithm; G_h is the output. The starting address z_1 is the left-most leaf of $\mathrm{rhs}_{G_1}(A_l)$ and the final address $z_h = (1, \varepsilon)$ is the root of $\mathrm{rhs}_{G_h}(A_1)$. For $1 \leq i \leq h$, grammar G_i has nonterminals $A_1, \ldots, A_{l_i}$, with $l_1 = l$ and, for $i > 1$, either $l_i = l_{i-1}$ or $l_i = l_{i-1} + 1$ and $A_{l_i} = \mathrm{fresh}(G_{i-1}, k)$ for some $k > 0$. By $\mathrm{fresh}(G, k)$ we denote a nonterminal of rank k that does not occur in G.

A tree pattern can be described by a tree with parameters at leaves (parameters denote subtrees that are not part of the pattern). Formally, a (*tree*) *pattern* p (*of rank* k) is a tree in which each $y \in Y_k$ occurs exactly once. Given a tree t and a node $u \in \mathbb{N}^*$ of t, the pattern p *matches* t at u if there are trees $t_1, \ldots, t_k$ such that $t/u = p\Theta$, where Θ is the substitution $[y_1 \leftarrow t_1, \ldots, y_k \leftarrow t_k]$. The triple (p, u, Θ) is called a *match* of p (in t) at u. Given a match $m = (p, u, \Theta)$, let $p_m = p$. Two matches (p, u, Θ) and (p', u', Θ') in the same tree t are *overlapping* if either there is a node $v \in \mathbb{N}^*$ in p such that $p[v] \notin Y$ and $uv = u'$ or there is a node $v' \in \mathbb{N}^*$ in p' such that $p'[v'] \notin Y$ and $u'v' = u$. Two matches $m = (p, u, [y_1 \leftarrow t_1, \ldots, y_k \leftarrow t_k])$ and $m' = (p, u', [y_1 \leftarrow t'_1, \ldots, y_k \leftarrow t'_k])$ of the same pattern p (but in possibly different trees) are *maximal* if $t_i[\varepsilon] \neq t'_i[\varepsilon]$ for all $1 \leq i \leq k$ (intuitively: there is no possibility to extend m and m' at the leafs to matches of some larger common pattern). Given a grammar G with nonterminals $A_1, \ldots, A_h$, a pattern p *matches* G at the address $z = (j, u)$ ($1 \leq j \leq h$, $u \in \mathbb{N}^*$) if p matches $\mathrm{rhs}_G(A_j)$ at u; in this case we call the triple $m = (p, z, \Theta)$ a match of p (in G) at $z = (j, u)$ and say that z is the address of m in G.

The replacement of patterns is defined as follows. Let G be an SLT grammar with nonterminals $A_1, \ldots, A_h$, p a pattern of rank k with a corresponding production $A_i(y_1, \ldots, y_k) \to p$ in G, and $m = (p, (j, u), [y_1 \leftarrow t_1, \ldots, y_k \leftarrow t_k])$ a match of p in G where $i \neq j$. The match m is replaced by A_i by replacing the subtree $\mathrm{rhs}_G(A_j)/u$ of $\mathrm{rhs}_G(A_j)$ with the tree $A_i(t_1, \ldots, t_k)$. The resulting grammar is denoted by $G[m \leftarrow A_i]$. Similarly, for two non-overlapping matches m_1 and m_2 of p in G (which means that either m_1 and m_2 are matches in two different right-hand sides of G or m_1 and m_2 are matches in the same right-hand side of G but are not overlapping according to the above definition), $G[m_1, m_2 \leftarrow A_i]$ is the grammar obtained from G by replacing each match m_1 and m_2 by A_i.

We now discuss how the size of an SLT grammar changes when occurrences of a tree pattern are replaced by a nonterminal that generates the pattern. The *size of a tree* (without parameters) is its *number of edges*. Since the SLT grammars that are generated by BPLEX have the property that all k parameters of a nonterminal appear exactly once in the right-hand side of its production, and in the order $y_1, y_2, \ldots, y_k$, we do not need to explicitly represent the parameters as nodes of the tree. Instead, we attach to a node label the information which of its subtrees is a parameter (note that the number of parameters in grammars generated by BPLEX is typically *very* small, 10 or less; experiments show that compression hardly improves when allowing more than 10 parameters, i.e., when setting $K_R > 10$). Hence, we do not count the edges to parameters and define the *size* of a tree t with k occurrences of parameters as $\mathrm{size}(t) = |E(t)| - k$. For a tree grammar G, $\mathrm{size}(G)$ is the sum of the sizes of the right-hand sides of the productions of G. Clearly, $\mathrm{size}(G) - \mathrm{size}(G[m \leftarrow A]) = \mathrm{size}(p)$ and $\mathrm{size}(G) - \mathrm{size}(G[m_1, m_2 \leftarrow A]) = 2 \times \mathrm{size}(p)$. If the production prod is not in G already then the size of the grammar $\mathrm{add}(G, \text{prod})$ obtained by adding prod to G is $\mathrm{size}(G) + \mathrm{size}(\mathrm{rhs}(\text{prod}))$.

Let us turn our attention to the computation of pattern sets. During the i-th iteration, when computing grammar G_{i+1} from G_i, BPLEX computes first the set

$$\begin{aligned}&\mathrm{RepM} = \mathrm{RepM}(G_i, z_i, K_N) = \\ &\{m \mid \exists j \in \{l, \ldots, \min\{l_i, l + K_N\}\} : m \text{ is a match of } \mathrm{rhs}_{G_i}(A_j) \text{ in } G_i \text{ at } z_i\}\end{aligned}$$

of all matches at the current address z_i of patterns from $\{\mathrm{rhs}_{G_i}(A_j) \mid l < j \leq \min\{l_i, l + K_N\}\}$. This computation considers at most K_N productions of index greater than l. Note that one can check whether $\mathrm{rhs}_{G_i}(A_j)$ matches G_i at z_i in at most $\mathrm{size}(\mathrm{rhs}_{G_i}(A_j))$ steps by reading the label of at most $\mathrm{size}(\mathrm{rhs}_{G_i}(A_j))$ many descendant-nodes of z_i and thereby binding parameters of $\mathrm{rhs}_{G_i}(A_j)$ to descendants of z_i. Since, every right-hand side $\mathrm{rhs}_{G_i}(A_j)$ with $j > l$ will have size at most K_S (see the discussion below) we can bound the total cost of computing RepM by $K_N \times K_S$. This is a constant, since K_N and K_S are assumed to be constants.

BPLEX also computes the set

$$\mathrm{NewM} = \mathrm{NewM}(G_i, z_i, K_N, K_S, K_R)$$

of all matches m at z_i such that:

- $0 < \mathrm{size}(p_m) \leq K_S$;
- there exists a *companion match* c_m of the same pattern p_m in G_i at some node w among the last K_N nodes preceding z_i in the order $<^l_{G_i}$ such that m and c_m are non-overlapping;
- if $\mathrm{size}(p_m) < K_S$, then either (1) m and c_m are maximal, or (2) m and c_m can only be extended to larger matches that overlap;
- the rank of p_m is at most K_R.

The set NewM can be computed by comparing top-down the tree rooted at z_i with the trees rooted at the last K_N nodes preceding z_i in the order $<^l_{G_i}$. Since the computation stops whenever it encounters a pattern that is larger than K_S, the cost of computing NewM is bounded by the constant $K_N \times K_S$.

BPLEX chooses a match $m \in \mathrm{RepM} \cup \mathrm{NewM}$ with maximal size, denoted by

$$\begin{array}{llll} S & \to E(E(C)) & C \to c(A, A) \\ E(y_1) & \to c(C, D(y_1)) & A \to a \\ D(y_1) & \to d(C, y_1) & \end{array}$$

Fig. 4. SLT grammar generating $\{c(c(a,a), d(c(a,a), c(c(a,a), d(c(a,a), c(a,a)))))\}$.

max-match$(\mathrm{RepM}, \mathrm{NewM})$. If $m \in \mathrm{RepM}$, then the match is replaced by the right-hand side of the corresponding production. If $m \in \mathrm{NewM}$, then BPLEX adds a new production $A \to p_m$ to the grammar, with $A = \mathrm{fresh}(G_i, \mathrm{rank}(p_m))$, and replaces the matches m and c_m by A. In both cases, the size of the grammar is reduced by $\mathrm{size}(p_m)$. If no matches are found, BPLEX tries to move the address z_i to the next node with respect to the order $<^l_{G_i}$.

Theorem 1 *For a given SL regular tree grammar G, BPLEX produces in time $O(\mathrm{size}(G))$ an equivalent SL cf tree grammar, where each nonterminal has rank at most K_R.*

PROOF. Each of the sets RepM and NewM can be computed in time $K_N \times K_S$, which is a constant. Hence, each execution of the while loop in Fig. 3 takes constant time. The linear running time of BPLEX follows, since for an input grammar G, the while loop cannot be executed more than $2 \times \mathrm{size}(G)$ times (each execution of the loop either moves the address forward or reduces the size of the grammar). $\Box$

We now illustrate the computation of BPLEX on the regular tree grammar on the left of Fig. 2. BPLEX does not perform any sharing in the third and second production; it then scans the first production. When the highest d is encountered (address $(1, 2)$) a match m of pattern $d(C, y_1)$ is found, together with a companion c_m matching in $(1, 2.2.2)$. This has size 1 and is chosen for replacement. The new nonterminal D of rank 1 is added to the grammar together with the production $D(y_1) \to d(C, y_1)$, and the two matches are replaced so that the first production becomes $S \to c(C, D(c(C, D(C))))$. The new pattern $\mathrm{rhs}(D)$ does not match the new grammar in $z = (1, 2)$ and no pairs of new matches are found either. Therefore z is changed to the root of the S production ($z = (1, \varepsilon)$). Here, the right-hand side of D does not match, while the pattern $c(C, D(y_1))$ matches in $(1, \varepsilon)$ and in $(1, 2.1)$, and these two matches are maximal. Therefore a new nonterminal E of rank 1 is added together with the production $E(y_1) \to c(C, D(y_1))$, and the matches are replaced by E, producing the output grammar shown in Fig. 4. Both this grammar and the cf tree grammar on the right of Fig. 2 have size 7. Note that BPLEX has not detected pattern $p = c(C, d(C, y_1))$ appearing in Fig. 2, because the smaller pattern $d(C, y_1)$ is replaced before p has been scanned completely.

```
<agenda>
   <person><name/><street/></person>  ⎫
   ...                                ⎬ 5 times
   <person><name/><street/></person>  ⎭
</agenda>
```

Fig. 5. An XML skeleton.

4 Memory-Efficient XML Tree Representation using BPLEX

An XML document is a sequential representation of a nested list structure. As an example, consider the XML document skeleton (i.e., without data values) in Fig. 5.

As mentioned in the Introduction, there are different data models for XML, which vary in their sizes. For example, DOM trees contain bidirectional pointers between a node and its children, its parent node, and its direct left and right sibling; the resulting size is approximately 4-5 times larger than the size of the original XML document. In this section we explain how BPLEX can be used to generate a small pointer-based representation of the tree structure of an XML document. BPLEX operates on ranked trees, that is, trees in which the rank of each symbol d is a fixed number $\mathrm{rank}(d)$. We now discuss two different ranked tree representations for XML document trees.

Binary Tree Model

One convenient and well-known way of modeling the tree structure of an XML document in a ranked way is to view it as *binary tree*: the first-child of an XML element node is represented by a left-child pointer and the next-sibling of an element node is represented by a right-child pointer. In this way, the pointers allow constant time top-down and next-sibling access through the tree structure. Other accessors can be supported by storing additional information dynamically when traversing the tree. For instance, if the sequence of parents of traversed first-child nodes is maintained, then it is possible to access in constant time the parent node and in $O(\textit{depth})$ time the next-in-preorder node, where *depth* refers to the depth of the XML document, i.e., to the maximal length of such ancestor sequences; more precisely, the next-in-preorder node of a given node is its next-sibling, i.e., right child, if it exists, and otherwise it is the next-sibling of the lowest parent of traversed first-child nodes that has a next-sibling node. Similarly, keeping a sequence of parents of traversed right-child nodes provides constant time previous-sibling and $O(depth)$ time previous-in-preorder access.

Note that a leaf (resp. the last sibling) of the document tree has no left (resp. no right) child edge in the binary tree representation; this is denoted by the superscript

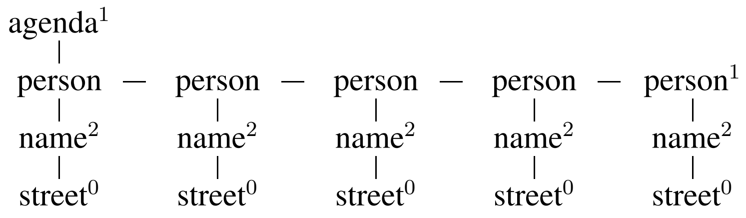


Fig. 6. Binary tree representation of an unranked tree.

2 (resp. 1), and by 0 for a last sibling leaf. In Fig. 6 the binary tree representation of the tree structure for the XML document from Fig. 5 is shown (with second child edges of person-nodes drawn horizontally). This binary tree has 15 edges. Let us now consider how BPLEX works on this tree. As before, we first turn a (ranked) tree into its minimal DAG, represented as a regular tree grammar, and then apply BPLEX to the grammar. In our example, the corresponding regular tree grammar has the three productions

$$
\begin{aligned}
S &\to \text{agenda}^1(\text{person}(A, \text{person}(A, \text{person}(A, \text{person}(A, \text{person}^1(A))))))\\
A &\to \text{name}^2(B)\\
B &\to \text{street}^0
\end{aligned}
$$

and its size is 11. Consider the S-production of this grammar. Its right-hand side contains four occurrences of the pattern $p = \text{person}(A, y_1)$. Thus, given a production $C(y_1) \to \text{person}(A, y_1)$, each of the occurrences can be replaced by the nonterminal C. However, there is one further occurrence of a similar pattern $p' = \text{person}^1(A)$, which can be obtained by removing the parameter y_1 from the pattern p. Note that, since A is a first child in p, removing y_1 changes person into person^1. In general, we allow a nonterminal K of rank m to appear with any rank $0 \leq r \leq m$ in the right-hand sides of productions, provided it is indicated which parameters are to be deleted; in the implementation, missing parameters are marked by a special "empty tree marker". With this "overloading" semantics of productions in mind, BPLEX turns the above regular tree grammar into the following grammar of size 6:

$$
\begin{aligned}
S &\to \text{agenda}^1(C(D(D))) & A &\to \text{name}^2(B)\\
D(y_1) &\to C(C(y_1)) & B &\to \text{street}^0\\
C(y_1) &\to \text{person}(A, y_1) & &
\end{aligned}
\tag{1}
$$

In this grammar, the D-production generates copies along a path of the binary tree. Repeated applications of such copying productions cause exponential size increase. In this way, the size of the input grammar can, in certain cases, be reduced exponentially. Consider our example, but now with 10000 person entries (thus, a binary tree with 30000 edges). The corresponding minimal regular tree grammar G_{10000}

has size 20001 while BPLEX outputs the following grammar of size 20:

$$
\begin{array}{lcl}
S & \to & \text{agenda}^1(A_8(A_5(A_4(A_3(A_1(A_1)))))) \\
A_1(y_1) & \to & A_2(A_2(y_1)) \\
A_2(y_1) & \to & A_3(A_3(y_1)) \\
 & \vdots & \\
A_{12}(y_1) & \to & A_{13}(A_{13}(y_1)) \\
A_{13}(y_1) & \to & \text{person}(A_{14}, y_1) \\
A_{14} & \to & \text{name}^2(A_{15}) \\
A_{15} & \to & \text{street}^0
\end{array}
$$

In this grammar, the symbol A_{13} generates the tree person$(\text{name}(\text{street}, y_1))$. More generally, for $j = 1, \ldots, 13$, A_j generates a chain with 2^{13-j} occurrences of this pattern and one parameter y_1 at the end of the chain. It is easy to see that S generates the correct tree with 10000 person entries.

Multiary Tree Model

Another way of modeling the tree structure of an XML document in a ranked way is to explicitly store the number k of children of an element node u with its label, and to provide, for each $1 \leq i \leq k$, a child$_i$-pointer from u to its i-th child (that is, to the $(i-1)$-sibling of u's first-child). In this way, the pointers provide constant time top-down access. If, dynamically, the sequence of parent nodes together with their child numbers are stored, then parent, previous-sibling, and next-sibling nodes can be accessed in constant time, and previous-in-preorder and next-in-preorder nodes can be accessed in $O(depth)$ time. We refer to this model as the *multiary tree model*.

It should be clear that, given an XML document, the number of pointers in the binary tree model is precisely equal to the number of pointers in the multiary tree model. The multiary tree model is slightly more flexible than the binary tree model with respect to traversal operations; however, with respect to update operations the multiary tree model is less flexible than the binary tree one, because in the latter, children pointers are stored in fixed-size arrays. Before we present our experimental results in the next section, we now want to show that both tree representations, DAGs and SLT grammars, are sensitive to the choice of the representation (binary or multiary tree).

A multiary tree representation of the XML document of before consists of a root node labeled `agenda` which has associated with it an array of five pointers, each to a node labeled `person` which in turn has an array of two pointers to nodes labeled `name` and `street`, respectively. For each pointer to a child node we can additionally also keep the inverse pointer from the child to its parent node. This doubles the number of pointers in the representation. Our investigations are independent of this choice: we always count in number of edges (these numbers have to be multiplied by the implementation cost of an edge, which possibly involves the cost of two pointers). The size of the multiary tree representation of the XML document in Fig. 5 is thus 15 edges.

DAGs: Binary Trees versus Multiary Trees

Before presenting experimental results with BPLEX, we discuss its relation to another tree compression method that has been applied to XML. Recall that we applied BPLEX to the minimal regular tree grammar of a binary tree representation of an unranked tree. An unranked tree has itself a unique minimal DAG (minimal regular tree grammar) which can be obtained in the same way as for ranked trees. However, the size of the minimal DAG of an unranked tree can be different from the one of the minimal DAG of its binary representation! In most cases the minimal unranked DAG is smaller than the binary one. The reason is that chains of second child edges in the binary tree become sibling subtrees in the unranked tree. To see this, consider the binary tree in Fig. 6. Clearly, its minimal DAG has only one copy of the subtree $\text{name}^2(\text{street})$ and hence has only 11 edges. On the other hand, the minimal DAG of the corresponding unranked tree has only one copy of the subtree $\text{person}(\text{name}, \text{street})$ and therefore has only 7 edges. As an example of a binary tree with a minimal DAG that is smaller than the one of the corresponding unranked tree, consider the unranked tree

$$t = u(p(x, b, c, b, c), p(y, b, c, b, c), p(z, b, c, b, c)) \tag{2}$$

Its minimal unranked DAG has 18 edges, but the minimal binary DAG has only 12, because only one copy of the subtree $b^2(c^2(b^2(c^0)))$ appears.

Multiplicities. In fact, the size of the minimal DAG representation can even be further reduced by using multiplicity counters for consecutive equal subtrees [4]. Then the DAG for the unranked tree of the agenda-example from Fig. 5 can be represented using only 3 edges, or equivalently, by an (unranked) regular tree grammar with multiplicity counters and productions

$$A \to \text{agenda}([5]P), \quad P \to \text{person}(\text{name}, \text{street}).$$

Of course, multiplicity counters take up space, but following Koch et al. this space is neglected. Thus, BPLEX produces the grammar in (1), which is smaller (size 6) than the minimal DAG of the unranked tree (size 7), but such a minimal DAG has a smaller representation (size 3) when multiplicity counters are added. From now on, we call the minimal DAG with multiplicity counters for an unranked tree its mDAG. This representation can easily be turned into a regular tree grammar with the *same size* that generates the binary representation of the original unranked tree.

This grammar also contains multiplicity counters at nodes, which are expanded to chains of nodes. We implemented a version of BPLEX which works on such grammars (and does not change the multiplicity counters). As it turns out, only in a few cases we obtained small improvements over BPLEX on the binary regular tree grammar corresponding to the minimal DAG. Thus, the advantage of counters is compensated for, by the ability of BPLEX to exponentially compress chains of nodes. On a few files, the minimal binary DAG was even smaller than the mDAG, due to similar chains as in the tree t from (2); cf. in Tab. 2 the two catalog files and the file NCBI_gene.chr1.

SLT Grammars: Binary Trees versus Multiary Trees

As we have seen before, copies of subtrees in the binary tree might not be copies of subtrees in the multiary tree, and vice versa. This means that the minimal DAGs for the binary and multiary tree may differ (in both ways). We now want to show that this property carries over to SLT grammars too: there are copies of connected subgraphs that appear in the binary tree but not in the multiary tree, and vice versa. First, consider the tree

$$A(a(p), b(q), c(r), d(a(s), b(t), c(u), d(v), e(w)), e(x)).$$

In the multiary tree representation, there are no patterns of size at least one, which appear at least twice in the tree. Hence, the minimal multiary tree SLT grammar for this tree has the same size as this tree. In the binary representation, however, we obtain two copies of the right-child chain of nodes a, b, c, d, e; hence, the minimal binary SLT grammar is by four edges smaller than the tree.

Now, consider the tree

$$a(b(k, c(l, d(a(b(m, c(n, d(o, e(q))))), e)))).$$

Clearly, the multiary tree representation of this tree has two occurrences of the chain of nodes a, b, c, d, e; hence, the minimal multiary tree SLT grammar is by four edges smaller than the tree. In the binary representation, however, there are no patterns of size at least one, which appear at least twice. Hence, the minimal binary SLT grammar for this tree has the same size as the tree itself.

These examples show that, in principle, it is unclear whether the multiary tree or binary tree will give rise to better compression by DAGs or SLT grammars. Interestingly, our experimental results in the next section show that, in the DAG case almost always the unranked DAG is smaller than the binary one, while in the SLT grammar case, BPLEX performs equally well on the multiary and binary representation.

Input File	Size (MB)	Element Count	Max Depth	Average Depth
SwissProt	457,4	10,903,568	6	4.45
DBLP	103.6	2,611,931	5	3.00
Treebank	55.8	2,447,727	37	9.42
1998statistics	0.64	28,306	6	5.99
catalog-01 / 02	11 / 104	225,194 / 2,240,231	8	5.65
dictionary-01 / 02	11 / 104	277,072 / 2,731,764	8	6.91
JST_snp.chr1	36	655,946	8	4.82
JST_gene.chr1	11	216,401	7	5.77
NCBI_snp.chr1 / gene	190 / 24	3,642,225 / 360,350	4	4
medline_0378	123	2,790,421	7	4.95

Table 1
Characteristics of XML documents used in experiments.

input file	size of tree in #edges	min. binary DAG size		min. unranked mDAG size		BPLEX(30000,20,10) output size		#NTs
SwissProt	10,903,568	1,437,445	13.2%	1,100,648	10.1%	311,328	2.9%	112,822
DBLP	2,611,931	533,183	20.4%	222,754	8.5%	115,902	4.4%	21,724
Treebank	2,447,727	1,454,494	59.4%	1,301,688	53.2%	519,542	21.2%	81,900
1998statistics	28,306	2,403	8.5%	726	2.6%	410	1.4%	169
catalog-01	225,194	6,990	3.1%	8,503	3.8%	3,817	1.7%	1,252
catalog-02	2,240,231	52,392	2.3%	32,267	1.4%	26,774	1.2%	2,385
dictionary-01	277,072	77,554	28.0%	46,993	17.0%	20,150	7.3%	4,446
dictionary-02	2,731,764	681,130	24.9%	441,322	16.2%	160,329	5.9%	25,288
JST_snp.chr1	655,946	40,663	6.2%	25,047	2.3%	12,858	1.8%	4,231
JST_gene.chr1	216,401	14,606	6.7%	5,658	2.6%	4,000	1.8%	1,114
NCBI_snp.chr1	3,642,225	809,394	22.2%	15	<0.1%	59	<0.1%	26
NCBI_gene.chr1	360,350	14,356	4.0%	11,767	3.3%	7,160	2.0%	3,634
medline_0378	2,790,421	629,853	22.6%	695,505	24.9%	132,733	4.8%	34,873

Table 2
BPLEX in highest compression mode. All sizes are in number of edges. Window size = 30000, max. pattern size = 20, max. rank = 10.

5 Experimental Results

We implemented BPLEX in C using gcc and the Expat XML parsing library (see http://expat.sourceforge.net/). See http://bplex.sourceforge.net/ for a preliminary version of BPLEX. Our experiments were done on a Pentium 3Ghz, 1GB RAM, running Linux. We tested BPLEX on three different sets of XML documents. The first one contains documents used in [4]: SwissProt (protein data), DBLP (a bibliographic database), Treebank (a linguistic database), and 1998statistics (baseball statistics). The second set contains XML documents generated by XBench [34], and the third contains documents from the Japanese Single Nucleotide Polymorphism database (see http://snp.ims.u-tokyo.ac.jp).

input file	multiary tree BPLEX(500,10,10)			binary tree BPLEX(500,10,10)		
	output size		#NTs	output size		#NTs
SwissProt	686,399	6.3%	15,153 (10,291)	330,595	3.0%	108,724 (103,551)
DBLP	218,638	8.4%	2,452 (2,048)	128,231	4.9%	16,270 (9,922)
Treebank	594,208	24.0%	34,119 (27,443)	593,770	24.0%	69,672 (62,484)
1998statistics	724	2.6%	45 (44)	418	1.5%	160 (86)
catalog-01	3,259	1.4%	590 (534)	3,894	1.7%	1,201 (1,040)
catalog-02	25,832	1.2%	876 (817)	27,725	1.2%	2,049 (1,606)
dictionary-01	37,506	14.0%	1,485 (498)	23,498	8.5%	3,154 (2,597)
dictionary-02	337,456	12.0%	6,962 (2,784)	188,088	6.9%	20,857 (19,211)
JST_snp.chr1	11,225	1.7%	353 (309)	12,876	1.8%	3,607 (2,732)
JST_gene.chr1	4,962	2.3%	135 (130)	4,374	1.8%	933 (559)
NCBI_snp.chr1	14	<0.1%	11 (11)	1,606	<0.1%	19 (10)
NCBI_gene.chr1	360,349	1.4%	606 (553)	7,390	2.1%	3,517 (3,284)
medline_0378	223,861	7.8%	8,411 (4,019)	143,792	5.0%	32,234 (29,461)

Table 3
Binary versus ranked BPLEX. Window size = 500, max. pattern size = 10, max. rank = 10. Number of nonterminals in brackets are for minimal DAG grammars.

Table 2 shows for each document the size of its tree structure (in number of edges) together with the sizes in three different representations. The minimal unranked DAG (with multiplicities) is in most of the cases smaller than the minimal binary DAG. The smallest sizes are generated by BPLEX, ranging between 0.1% and 21% of the size of the original tree structure; as input for BPLEX we used the minimal binary DAG, represented as an SLT grammar. As input parameters for BPLEX we used: window size 30000, maximal pattern size 20, maximal rank 10. The last column shows the number of nonterminals (#NTs) of the BPLEX-output.

The only examples where the binary DAG is smaller than the mDAG are catalog-01 and a file of the medical bibliographies medline. As can be seen, BPLEX performs surprisingly well on medline. Note that for the file NCBI_snp.chr1, the small size of the minimal mDAG (15 edges) is due to a multiplicity counter: a long list of siblings all labeled by the same nonterminal is represented by just one edge (plus a counter). In the binary minimal DAG we do not have multiplicities, and hence its size is much larger (809,394 edges); interestingly, BPLEX is able to reduce this size to only 59 edges (and, if the window size is increased to $40,000$ then we obtain only 48 edges). This is because a long list is broken down exponentially by BPLEX – viz copy productions or the form $A_i(y_1) \to A_{i+1}(A_{i+1}(y_1))$, as outlined in the example of the person list in the previous section. With a small window size of 500, BPLEX introduces fewer copy productions and hence compressed only to $1,606$ edges.

We also implemented a version of BPLEX that runs on the multiary tree model of an XML document, instead of the binary tree model. The results are shown in Table 3; note that the numbers were obtained with lower parameters than those of Table 2: window size $= 500$, maximal rank $= 10$, and maximal pattern size $= 10$.

The table also shows in the last column the number of nonterminals in the grammars generated by BPLEX, and, in brackets, the number of nonterminals of the corresponding minimal DAG grammars. As can be seen, the achieved compression ratios are similar to those of BPLEX running on binary trees. In most cases, running on the binary tree gives slightly better compression; note however that in the multiary tree model we obtain grammars with far fewer nonterminals than in the binary tree model. One of the reasons for this is the use of multiplicities in the minimal mDAG; we run multiary tree BPLEX on the mDAG, i.e., we take advantage of the multiplicity counters. This can be seen on NCBI_snp.chr1: it has an mDAG with 15 edges which is transformed by BPLEX to an SLT grammar with 14 edges. These experiments suggest that, in practice, tree compression by BPLEX is *not* sensitive to un-/rankedness of the input, i.e., whether we work in the binary or multiary tree model.

Performance and Parameter Tuning

Recall from Fig. 3 the three parameters of BPLEX: the window size K_N, the maximal rank K_R of a pattern, and the maximal size K_S of a pattern. Our experiments show that the algorithm performs well with small values of K_R and K_S and that values above 5 and 10 respectively do not increase compression much. For instance, with $K_S = 3$ (and $K_N = 30000$, $K_R = 10$) we obtain for Treebank a compression ratio of 22.0%, as compared to the 21.2% obtained with $K_S = 20$; similarly, for dictionary-02 we obtain 6.0% compression ratio for $K_S = 3$, compared with 5.9% for $K_S = 20$. The same happens for K_R: taking it equal to 3 gives 22% and 6% for Treebank and dictionary-02, respectively.

The main factor for good compression is the window size. BPLEX achieves best compression with a window size of > 100; values above $20,000$ do not change compression much in our examples. For instance, fix $K_R = 10$ and $K_S = 20$; then on medline we get for $K_N = 10, 100, 1000$ the compression ratios 5.4%, 5.2%, and 4.9%, respectively; Similarly, for Treebank we obtain 30%, 26%, and 24%, respectively, and for dictionary-02 we obtain 7.9%, 7.4%, and 6.7%, respectively. Our current implementation runs slowly on large window sizes, requiring several hours to obtain all the results shown in Tab. 2. For instance, running on medline with $K_N = 10, 100, 1000$ takes 10, 41, and 116 seconds, respectively; similarly, running on Treebank takes 545, 2094, and 3165 seconds, respectively. This is mainly due to the non-optimized way in which matches of patterns are found and recorded, which results in a large constant hidden in the $O(n)$-expression for the running time of BPLEX. Interestingly, even with a small window size, BPLEX already compresses considerably better than binary DAGs and unranked mDAGs. If we use $K_N = K_R = K_S = 3$ then each of our examples compresses in less than one minute; compression rates are SwissProt 4.1%, Treebank 34%, and dictionary-01 12%. It remains to test the impact of our compression with respect to the total memory consumption for an XML document in main memory.

6 Algorithms on SLT Grammars

SLT grammars are well suited to efficiently represent XML documents. Consider now a grammar in memory which represents a large XML document. How can we process the XML tree, without decompressing the grammar? Any read access like, e.g., reading the label of the root node, or moving along an edge from one node to another, can be realized on the grammar representation with an additional per-step overhead of at most the size h of the grammar [23]. Additionally, a stack of height at most h must be maintained at all times. Thus, the price to be paid for having a small representation that can be accessed without decompression, is a slow down for each read operation. For some special applications, however, it is possible to eliminate the slow-down, or to even achieve speed ups. In this section we investigate such applications.

XML Type Validation

XML type validation means to check whether a given XML document is valid with respect to an XML type. Popular formalism for XML types (varying in their expressiveness) are DTD, XML Schema, or RELAX NG. Here we want to check whether the tree structure of an XML document, represented by an SL cf tree grammar G, is valid with respect to an XML type. Essentially this can be done in time linear in the size of the grammar G, if both the maximal number of parameters k of G and the XML type definition are fixed. In particular, the number k appears as an exponent in the constant of the algorithm (see Proposition 2). In BPLEX, k is controlled by the input parameter K_R. Practical experiments show that small values of k (smaller than 5) already achieve very competitive compression ratios; in fact, we observed that for all the files shown in Tab. 2 taking K_R bigger than 10 does not improve compression anymore. It can therefore be assumed that k is very small with respect to the size of G.

All XML type formalisms mentioned above can conveniently be modeled by regular tree languages [27], a classical concept from formal language theory [18]. We therefore consider the problem of checking whether, for an SL cf tree grammar G, $L(G)$ is included in a regular tree language R. Assume that R is given by a deterministic bottom-up tree automaton B (formally defined below). Our inclusion check is similar to constructing the (context-free) intersection grammar $H_\cap$ of a context-free tree grammar H with R: the productions of $H_\cap$ are obtained by running the automaton B on the right-hand sides of H's productions. For simplicity, consider first the string case, i.e., H is a context-free grammar and B is a DFA: $H_\cap$'s nonterminals are triples of the form $[q, A, p]$ denoting that B moves from state q to state p on some string generated by H's nonterminal A. If $A \to XY$ is a production of H then for every state r the grammar $H_\cap$ has the production $[q, A, p] \to [q, X, r][r, Y, p]$. This well-known triple-construction [2] can be generalized to context-free tree grammars by considering nonterminals of the form $[(q_1, q_2, \ldots, q_k), A, p]$ where A is a nonterminal of rank k and $q_1, \ldots, q_k, p$ are states of the tree automaton B (see Theorem 3.2.8 of [12] where a similar triple-construction is presented for inside-out (IO) macro grammars; that construction can easily be generalized to trees, thus showing that IO context-free tree languages are closed under intersection with regular tree languages).

We now come back to the case of validating an SL cf tree grammar G. Since G generates only one tree we do not construct an intersection grammar, but a single run of (an appropriate extension of) the automaton B in order to determine whether $L(G) = \{t\} \subseteq L$.

Formally, a deterministic bottom-up finite-state tree automaton is a tuple $B = (Q, \Sigma, \{\delta_\sigma\}_{\sigma\in\Sigma}, F)$ where Q is a finite set of states, Σ is a ranked alphabet, $\delta_\sigma : Q^k \to Q$ for $\sigma \in \Sigma$ of rank k, and $F \subseteq Q$ is a set of final states. For a tree $t \in T_\Sigma(Y_k)$ and a mapping $\Theta : Y_k \to Q$ which assigns to each parameter a state, we define the state $\delta(t, \Theta) \in Q$ inductively as follows: If $t = y_i$ for some $1 \leq i \leq k$, then $\delta(t, \Theta) = \Theta(y_i)$. Now assume that $t = \sigma(t_1, \ldots, t_n)$ for some $\sigma \in \Sigma$ of rank n and trees $t_1, \ldots, t_n \in T_\Sigma(Y_k)$. Then, $\delta(t, \Theta) = \delta_\sigma(\delta(t_1, \Theta), \ldots, \delta(t_1, \Theta))$. In case $t \in T_\Sigma$ (i.e., t does not contain parameters), let $\delta(t) = \delta(t, \Theta)$, where Θ is the empty mapping. The language accepted by B is $L(B) = \{s \in T_\Sigma \mid \delta(s) \in F\}$. The size $|B|$ of B is the size of the transition function δ. In [22] it was shown that our validation problem is PSPACE-complete, and that the following proposition holds. For completeness, we present a proof.

Proposition 2 *(cf. Theorem 1 of [22]) Given an SL cf tree grammar G and a deterministic bottom-up tree automaton B it can be checked whether $L(G) \cap L(B) = \varnothing$ in worst case time $O(m^k \times |B| \times \mathrm{size}(G))$, where m is the number of states of B and k is the maximal number of parameters of nonterminals of G.*

PROOF. The proof of this proposition is straightforward: let $G = (N, \Sigma, \mathrm{rhs})$ with $N = \{A_1, \ldots, A_n\}$ and $L(G) = \{t\}$. Let $B = (Q, \Sigma, \delta, F)$ be a deterministic bottom-up tree automaton. We now run the tree automaton B on the right-hand sides of the productions of G. We do this bottom-up, starting with the right-hand side $\mathrm{rhs}(A_n)$. More formally, we compute for every mapping $\Theta : Y_k \to Q$ (where k is the rank of A_n) the state $\delta(\mathrm{rhs}(A_n), \Theta)$. In this way, we obtain a mapping $\delta_{A_n} : Q^k \to Q$ with $\delta_{A_n}(q_1, \ldots, q_n) = \delta(\mathrm{rhs}(A_n), \Theta)$, where Θ is the mapping with $\Theta(y_i) = q_i$ for all $1 \leq i \leq k$. We add the mapping δ_{A_n} to the transition mappings δ_σ ($\sigma \in \Sigma$) of the automaton B. Extended in this way, we can now run B on the right-hand side $\mathrm{rhs}(A_{n-1})$ and compute the mapping $\delta_{A_{n-1}}$. Note that the nonterminal A_n may occur in $\mathrm{rhs}(A_{n-1})$, hence we need the transition mapping δ_{A_n} when running B on $\mathrm{rhs}(A_{n-1})$. We continue in this way and compute all mappings δ_{A_i} for $1 \leq i \leq n$. Note that for each nonterminal A_i of rank k, $|Q|^k = m^k$ many values of δ_{A_i} are computed and that the computation of each value takes time

$O(\mathrm{size}(\mathrm{rhs}(A_i)) \times |B|)$. Hence, in total at most $O(m^k \times |B| \times \mathrm{size}(G))$ computation steps are needed. This number can be greatly decreased by going *top-down* in a "lazy" manner through G, starting with $\mathrm{rhs}(A_1)$. Note though, that the price for the improvement is the necessity to maintain recursive calls. Consider the run of B on $\mathrm{rhs}(A_1)$. If B arrives at a nonterminal A_i ($i > 1$) of rank k, in states $q_1, \ldots, q_k$, then we issue a recursive call to compute $\delta_{A_i}(q_1, \ldots, q_k)$. Such a call means to substitute q_j for y_j, $1 \leq j \leq k$, in $\mathrm{rhs}(A_i)$ and then to run B on the resulting tree. During the run further recursive calls may be generated. If a value $\delta_{A_i}(q_1, \ldots, q_k)$ is determined, then it is stored in a table in order to avoid its recomputation. The total number of δ_{A_i}-values that are actually computed in this way may be much smaller than the worst case bound of m^k. $\square$

Note that in order to use Proposition 2 in the context of XML types, the corresponding type definition has to first be transformed into a (deterministic bottom-up finite) tree automaton. If the type is given as DTD or as XML Schema, then the transformation into a deterministic tree automaton can be done in time linear in the size of the representation. However, the details are more convoluted: in the case of DTD, a *last*-child, *previous*-sibling binary tree encoding should be used to guaranteed that the resulting automaton is of linear size. And in the case of XML Schemas, a deterministic *top-down* tree automaton should be used (note that a result similar to Proposition 2 holds for top-down tree automata; see [22]). For RELAX NG (which employs full regular tree languages and nondeterminism) it cannot be avoided that the size of the corresponding deterministic tree automaton (no matter if top-down or bottom-up) is sometimes exponential in the size of the RELAX NG type definition. This is not a serious issue though, when using BPLEX (which outputs linear grammars): for *linear* SL cf tree grammars (SLT grammars) Proposition 2 can be extended to the case that the automaton B is *nondeterministic*: the δ_{A_i} are now functions from Q^k to 2^Q, where k is the rank of A_i; they are computed by checking for all states $p, p_1, \ldots, p_k$ of B whether there is a run on $\mathrm{rhs}(A_n)[y_1 \leftarrow p_1, \ldots, y_k \leftarrow p_k]$ arriving in p. It follows that the problem can be solved in time $O(m^{k+1} \times |B| \times \mathrm{size}(G))$, see [22] for a detailed proof and a discussion explaining the importance of the linearity of the input SL cf tree grammar.

Equality Test

Consider two SL cf tree grammars G_1 and G_2. Is it possible to test whether both G_1 and G_2 generate the same tree t, without fully uncompressing the grammars, i.e., without deriving the tree t? More precisely, we are interested in the time complexity of testing equivalence of G_1 and G_2.

In the string case, i.e., if G_1 and G_2 are SL cf string grammars, then the problem can be solved in polynomial time with respect to the sum of the sizes of G_1 and G_2 [29]. The proof relies on the fact that, for an SL cf string grammar G (in Chomsky normal form) of size n, the length of the string derivable from a nonterminal of G is $\leq 2^n$, and therefore can be stored in n bits. Since basic operations (comparing, addition, subtraction, multiplication, etc.) on such numbers work in polynomial time with respect to n, we can compute in polynomial time the length of the word generated by any nonterminal of G. Since in the tree case this property does *not* hold anymore (because the size of the tree generated by an SL cf tree grammar of size n can be 2^{2^n}) it looks unlikely that the equivalence problem can also be solved by an algorithm running in polynomial time. In fact, we do not know whether such an algorithm exists. The following theorem shows that the problem can be solved using polynomial space, and hence in exponential time. On the other hand, if the grammars G_1 and G_2 are linear, then they can be transformed into SL cf string grammars generating a depth-first left-to-right traversal of the corresponding tree; then, the result of [29] can be used to show that in this case testing equivalence can be done in polynomial time.

Theorem 3 *Testing equivalence of two SL cf tree grammars G_1 and G_2 can be done in polynomial space, and in polynomial time if G_1 and G_2 are linear.*

Before we prove Theorem 3, we first have to introduce some definitions concerning derivations of SL tree grammars. Let $G = (\{A_1, \dots, A_n\}, \Sigma, \mathrm{rhs})$ be an SL tree grammar. The grammar G generates precisely one tree t, i.e., $L(G) = \{t\}$. Given a node $u \in \mathbb{N}^*$ of t, how can we obtain from G the label $t[u]$?

As shown in [23], $t[u]$ can be obtained using only space $O(\mathrm{size}(G))$ as follows. A *pointed production* is a pair (i, ρ) where $1 \leq i \leq n$ and ρ is a node in $\mathrm{rhs}(A_i)$. A *stack configuration* is a non-empty sequence $w = (i_1, \rho_1) \cdots (i_m, \rho_m)$, $m \geq 1$, of pointed productions such that $i_1 = 1$ and for every $1 \leq \nu \leq n-1$, $\mathrm{rhs}(A_{i_\nu})[\rho_\nu] = A_{i_{\nu+1}}$. The stack configuration w *points* to the node ρ_n of the tree $\mathrm{rhs}(A_{i_n})$, and the *label* of w, denoted lab(w), is defined as the label $\mathrm{rhs}(A_{i_n})[\rho_n] \in \Sigma$ (and undefined if it is not in Σ). The idea of determining the label $t[u]$ is to build up a stack configuration by scanning the sequence u from left to right. This sequence will finally point to a Σ-labeled node and the label of this node is $t[u]$.

As an example for the above definitions, consider the SL tree grammar H with productions

$$\begin{aligned} A_1 \quad &\to A_2(A_2(e)) \\ A_2(y_1) &\to A_3(A_3(y_1)) \\ A_3(y_1) &\to c(y_1, y_1). \end{aligned}$$

Then, $w_1 = (1, \varepsilon)$ points to the A_2-labeled root node of A_1's right-hand side, while $w_2 = (1, \varepsilon)(2, \varepsilon)(3, 1)$ points to the left occurrence of the parameter y_1 in A_3's right-hand side. The stack configuration $w_3 = (1, \varepsilon)(2, \varepsilon)(3, \varepsilon)$ points to the c-labeled node of A_3's right-hand side; thus, the label of w_3 is c: lab$(w_3) = c$.

We now define two operations on stack configurations: down$_\nu$ ($\nu \in \mathbb{N}$) and Expand.

For a stack configuration $w = w'(i, \rho)$, down$_\nu(w)$ is defined as $w'(i, \rho\nu)$ if $\rho\nu$ is a node in $\mathrm{rhs}(A_i)$, and is undefined otherwise. Thus, down$_\nu(w)$ is a stack configuration, which points to the ν-th child of the node to which the stack configuration w points. The idea of the Expand-operation is to expand a stack configuration, until it points to a Σ-labeled node. Formally, the stack configuration Expand(w), where $w = w'(i, \rho)$, is defined as

(i) w if $\mathrm{rhs}(A_i)[\rho] \in \Sigma$,
(ii) Expand(down$_\nu(w')$) if rhs$(A_i)[\rho] = y_\nu$, and
(iii) Expand$(w'(i, \rho)(j, \varepsilon))$ if rhs$(A_i)[\rho] = A_j$.

Thus, if w already points to a Σ-labeled node then Expand just returns w, see (i). In case (ii), where w points to a parameter y_ν, w' cannot be the empty sequence, because then we would have $i = 1$. But since the rank of the initial non-terminal A_1 is zero, rhs(A_1) does not contain the parameter y_ν. Hence, w' is a stack configuration, which points to an A_i-labeled node, where the rank of A_i is at least ν (so that the ν-th parameter y_ν can appear in rhs(A_i)). Hence, down$_\nu(w')$ is defined. The Expand-operation now removes the pointed production (i, ρ) from w (which points to a leaf of rhs(A_i) labeled with the parameter y_ν) and directly moves to the ν-th child of the A_i-labeled node to which w' points (by applying down$_\nu$ to w'). Finally, if w points to an A_j-labeled node (case (iii)) then the Expand-operation first adds the pointed production (j, ε) to w (so that the resulting stack configuration points to the root of rhs(A_j)) and continues expansion.

In our example,

$$\text{Expand}(w_1) = \text{Expand}((1, \varepsilon)) = (1, \varepsilon)(2, \varepsilon)(3, \varepsilon) = w_3$$
$$\text{down}_1(w_3) = \text{down}_1((1, \varepsilon)(2, \varepsilon)(3, \varepsilon)) = (1, \varepsilon)(2, \varepsilon)(3, 1) = w_2$$

and

$$\begin{aligned} \text{Expand}(w_2) &= \text{Expand}(\text{down}_1((1, \varepsilon)(2, \varepsilon))) \\ &= \text{Expand}((1, \varepsilon)(2, 1)) \\ &= \text{Expand}((1, \varepsilon)(2, 1)(3, \varepsilon)) \\ &= (1, \varepsilon)(2, 1)(3, \varepsilon). \end{aligned}$$

Using these definitions we can now determine the label of $t[u]$ as lab(Find(G, u)) where Find is recursively defined, for an SLT grammar G, a sequence of integers v, and an integer i as follows:

$$\begin{aligned} \text{Find}(G, \varepsilon) &= \text{Expand}((1, \varepsilon)) \\ \text{Find}(G, vi) &= \text{Expand}(\text{down}_i(\text{Find}(G, v))). \end{aligned}$$

Note that in the definition of Find, the mapping down$_i$ is only applied to a stack configuration which points to a terminal-labeled node. It should be clear that the stack configuration Find(G, u) points to a node with label $t[u]$. Additionally, no stack configuration in the computation ever consists of more than n pairs, because the first components of pointed productions in a stack configuration are pairwise different.

In the example

$$\begin{aligned}
\text{Find}(H,1) &= \text{Expand}(\text{down}_1(\text{Find}(H,\varepsilon))) \\
&= \text{Expand}(\text{down}_1(\text{Expand}((1,\varepsilon)))) \\
&= \text{Expand}(\text{down}_1(w_3)) \\
&= \text{Expand}(w_2) = (1,\varepsilon)(2,1)(3,\varepsilon).
\end{aligned}$$

This means that $t[1] = c$, because lab(Find$(H, 1)) = \text{rhs}(A_3)[\varepsilon] = c$.

PROOF OF THEOREM 3. Let $G_1 = (\{A_1, \ldots, A_{n_1}\}, \Sigma, \text{rhs}_1)$ and $G_2 = (\{B_1, \ldots, B_{n_2}\}, \Sigma, \text{rhs}_2)$. By Savitch's Theorem (see, e.g., [28]) and the complement closure of PSPACE, it suffices to give a nondeterministic polynomial space algorithm that tests *in*equivalence. Roughly speaking, the algorithm guesses corresponding paths in the trees t_1 and t_2, generated by G_1 and G_2, respectively. The key issue is that any node $u \in \mathbb{N}^*$ of t_i can be (non-uniquely) represented in polynomial space with respect to the size of G_i. In fact, the node u in t_i can be represented by the stack configuration Find(G_i, u). Recall that the length of Find(G_i, u) is at most n_i. Of course, the node u is in general *not* uniquely represented by Find(G_i, u); in particular, corresponding nodes generated by parameter copying have the same Find-representation (as an example: consider the grammar G with productions $A_1 \to A_2(e)$ and $A_2(y_1) \to c(y_1, y_1)$; then Find$(G, 1) = (1, 1)$ equals Find$(G, 2)$, and points to the e-node). The algorithm starts with the two sequences Find(G_1, ε) and Find(G_2, ε), representing the root nodes of t_1 and t_2, respectively. If their labels are different we accept. Otherwise, we guess a child number i and move down to the i-th child (by applying down$_i$ and Expand), resulting in Find(G_1, i) and Find(G_2, i). If the corresponding labels are different we accept, etc. If there is no child number (we are at a leaf) we reject. Since G_1 and G_2 move synchronized through t_i, we do not need to store $u \in \mathbb{N}^*$; in fact, u's length might be exponential in the sizes of G_i. A pseudo code of this procedure is shown in Fig. 7. It should be clear that there is a run returning **true** if and only if there is a node u with $t_1[u] \neq t_2[u]$.

Now let G_1 and G_2 be linear. This means that for any nonterminal A of G_1 or G_2, of rank k, the tree $A(y_1, \ldots, y_k)$ derives to a tree t over $\Sigma \cup Y_k$ in which y_j occurs at most once, $1 \leq j \leq k$. It is straightforward to change the grammars in such a way that (1) every y_j occurs exactly once in t and (2) the order of the parameters in t (going depth-first left-to-right) is $y_1, \ldots, y_k$. The idea is now to construct cf string grammars H_1, H_2 which generate depth-first left-to-right traver-

```
procedure INEQUIVALENT(G1, G2: grammar): bool
begin
    s1 := Find(G1, ε)
    s2 := Find(G2, ε)
    while true do
        if lab(s1) ≠ lab(s2) then return true
        else
            f := lab(s1)
            if rank(f) = 0 then return false
            guess an integer i ∈ {1, ..., rank(f)}
            s1 := Expand(down_i(s1))
            s2 := Expand(down_i(s2))
        fi
    od
end INEQUIVALENT
```

Fig. 7. Checking inequivalence of two SL cf tree grammars G_1 and G_2.

sals of t_1 and t_2, respectively. Let $i \in \{1, 2\}$. For every nonterminal A of G_i of rank $k > 0$ let $A_{0,1}, A_{1,2}, \ldots, A_{k-1,k}, A_{k,0}$ be new nonterminals of H_i, and for every $\sigma \in \Sigma$ of rank $k > 0$ let $\sigma_{0,1}, \sigma_{1,2}, \ldots, \sigma_{k-1,k}, \sigma_{k,0}$ be new terminals of H_i. Nonterminals and terminals of rank zero are taken over to H_i. The right-hand side of the nonterminal $A_{0,1}$ is the traversal starting at the root of the right-hand side of A (indicated by the index 0) up to the first parameter y_1 in the right-hand side of A (indicated by the parameter 1). The right-hand side of $A_{\nu,\nu+1}$ is the traversal starting at the parameter y_ν in the right-hand side of A up to the parameter $y_{\nu+1}$. Similarly, a terminal symbol $g_{\nu,\nu+1}$ means that g was entered coming from its ν-th child and was exited by moving to its $(\nu + 1)$-th child. It should be clear how to construct the productions of H_i. As an example, consider the tree grammar production $A(y_1, y_2, y_3) \to B(g(y_1, a, b), h(B(y_2, y_3)))$ and the nonterminal $A_{1,2}$ of the constructed string grammar; its production is $A_{1,2} \to g_{1,2}\, a\, g_{2,3}\, b\, g_{3,0}\, B_{1,2}\, h_{0,1}\, B_{0,1}$. Clearly, $t_1 = t_2$ if and only if the string w_1 generated by H_1 equals the string w_2 generated by H_2. Moreover, H_1 and H_2 are SL cf string grammars of polynomial size with respect to G_1 and G_2, respectively. By the result of [29], testing $w_1 = w_2$ can be done in polynomial time with respect to the sizes of H_1 and H_2. $\square$

7 Related Work

Grammar-based tree compression was independently presented in [33]. However, their algorithm seems less effective than BPLEX, and in particular it generates grammars with a very large number of parameters (typically several thousands). This means that our algorithms of Section 6 are not likely to be applicable to the grammars they produce, because they sensibly depend on the number of parameters in a grammar.

There are succinct pointer-less representations of trees, see, [19,16,11]. In this way, an n-node tree can be represented by $2n + o(n)$ bits, while allowing $O(1)$ time for most read operations on a tree [16]. In the context of XML, pointer-less tree representations can, e.g., be found in XPRESS [26]: label paths in an XML document are encoded by real number intervals following an arithmetic encoding; this allows to run path queries directly on the compressed instance. This method is typically applied directly to XML documents on the file system. While XPRESS has smaller query evaluation times than other systems working on compressed XML files (like, e.g., XGrind [32]), it is unclear how well it compares to other approaches (like ours) when documents are loaded into memory. In [13] a succinct representation for SLT grammars is introduced; using this representation, it is, for instance, possible to represent our SLT grammars for DBLP and medline, using only 288KB and 358KB, respectively.

It is also possible to use strings to represent XML trees in memory [35]; their experiments show that this offers good compression, while still being able to query efficiently the representation. XQueC uses a queriable XML representation that is based on compression of data values [1]. An advanced implementation which basically uses DAG sharing together with compression of data values is presented in [9]; their results are convincing, which strengthens the belief in our approach, because replacing DAG sharing by SLT grammars should immediately improve their system.

Consider now the problem of finding the smallest cf string grammar for a given string. This problem is NP-complete and various approximation algorithms have been studied [5]. In particular, the size of the smallest cf grammar is lower bounded by the size of the smallest LZ77 representation of the string (when the size of the sliding window is unbounded) [5,31]. The question arises whether a similar result holds in the tree case. But for trees it is unclear how an efficient LZ77 representation would look like. The problem is how to specify tree prefixes that have appeared before [8]. In the string case, the LZ77 representation is obtained by performing a left-to-right scan of the input string; at each moment, the string starting at the current position is matched against all prefix strings, and the longest match is selected. For example, the string $abbbaabbabbb$ is compressed by LZ77 into $abbba[1,3][1,4]$, where a pair $[i,j]$ represents the substring starting at position i of length j. In order to bound the time needed for matching, many implementations of LZ77 use a sliding window of fixed size instead of the complete prefix. In the tree case there is no accepted version of LZ77. The problem is that i should be replaced by a path p, and j should be replaced by an unlabeled tree t with parameters at leaves (or, alternatively, by a list of paths to parameters) [8], but such pairs $[p,t]$ require too much space in order to obtain good compression.

In [31] a technique to decrease the size of an SL cf grammar is presented; the idea is to change the grammar in such a way that its derivation trees become balanced trees, in the sense of AVL trees. This technique gives good compression ratios, when applied to an SL cf grammar obtained from the minimal LZ77 representation of the string. Even though there is no obvious way to extend LZ77 to trees, it might be possible to apply the technique of [31] to SL cf tree grammars. Another variation of Lempel-Ziv compression, known as LZ78, can more readily be extended to trees. For LZ78 on strings, new patterns are composed by adding a letter to already existing patterns. A pattern is specified as a pair (i, a) where i is the index of a previous pattern and a is a letter; the case $i = 0$ represents the one-letter pattern a. In this scheme the string $abbbaabbabbb$ is compressed to $(0, a)(0, b)(2, b)(1, a)(3, a)(3, b)$. Thus, the pair $(2, b)$ is the concatenation bb of b (the second pattern) and b, and similarly $(3, a)$ represents bba. The LZ78 encoding has a natural interpretation as an SL cf string grammar (see e.g. [5]). LZ78 can be extended to trees by using a dictionary of tree patterns where, during a top-down scan of the input tree, new patterns are obtained from existing ones by appending subpatterns at parameter positions; in the simplest case, only a one-node subpattern is appended. Such a technique is presented in [6]; other variations, each using a different method for extending the patterns, are presented in [7]. In [6] no experimental results are provided. In [7] the proposed algorithms are applied to term compression, and the best performance is a size reduction to about 50% of the original. It remains to be investigated how these techniques perform on XML documents.

In [14] it was shown that evaluation of Core XPath queries on DAGs is PSPACE complete. Recently we have shown that this result can be extended to linear SL cf tree grammars [22]; this means that, while achieving better compression than DAGs by using BPLEX, the complexity of evaluating a Core XPath still remains the same for outputs of BPLEX as it is for DAGs.

8 Conclusions and Future Work

A linear time algorithm was presented that transforms a given tree into a small SLT grammar. The algorithm can be used to "compress" the tree structure of an XML document into a highly efficient, pointer-based memory representation. The representation preserves the basic tree operations and can be accessed via DOM (using an appropriate proxy). On average, the size of a compressed instance is one half of the size of the minimal unique DAG of the tree, which in turn is about 1/10 of the size of the original tree [4]. For some computational problems on trees, we presented efficient algorithms that directly work on SLT grammars; in particular we considered (1) validation against XML types given by deterministic bottom-up tree automata and (2) testing equality of documents. In [22] we considered Core XPath evaluation. It remains to implement these ideas and test how well they behave on practical queries. To further increase memory efficiency, our representation could be combined with a compression of data values (e.g., similar to the one of [1]). It is also possible to directly keep results of queries in compressed format; this idea has been considered for DAG compression and a fragment of XQuery [3]. It also has been considered for compression by SLT grammars and macro tree transducers as query formalism [23]. It is not difficult to change BPLEX to take arbitrary SL cf tree grammars as input; in this way it might be possible to achieve further compression by running BPLEX on its own output.

Several recent programming languages allow to process XML documents via pattern matching constructs. Such constructs are compiled into automata which carry out the matching in the document. It seems straightforward to extend this compilation to automata which directly work on SLT grammars. In this way an efficient XML query evaluator is obtained because XQueries and XSLTs can be translated to pattern matching statements. In this context, other optimizations might become important (e.g. lazy sequences [15]).

We would like to test how our technique can be used for XML file compression. We hope that the performance of existing compressors, like XMill, can be further improved by using BPLEX for the compression of tree structures.

Acknowledgments. We would like to thank the anonymous referees for many useful comments.

References

[1] A. Arion, A. Bonifati, G. Costa, S. D'Aguanno, I. Manolescu, and A. Pugliese. XQueC: Pushing queries to compressed XML data. In *Proc. VLDB 2003*, pages 1065–1068. Morgan Kaufmann, 2003.

[2] Y. Bar-Hillel, M. Perles, and E. Shamir. On formal properties of simple phrase structure grammars. *Z. Phonetik. Shrachwiss. Kommunikationsforsch.*, 14:143–172, 1961.

[3] P. Buneman, B. Choi, W. Fan, R. Hutchison, R. Mann, and S. Viglas. Vectorizing and querying large XML repositories. In *Proc. ICDE 2005*, pages 261–272. IEEE Press, 2005.

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

[5] 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.

[6] S. Chen and J. H. Reif. Efficient lossless compression of trees and graphs. In *Proc. DCC'96*, page 428. IEEE Press, 1996.

[7] J. R. Cheney. First-order term compression: techniques and applications. Master's thesis, Carnegie Mellon University, August 1998.

[8] J. R. Cheney. Personal communication. 2004.

[9] J. Cheng and W. Ng. XQzip: Querying compressed XML using structural indexing. In *Proc. EDBT 2004*, volume 2992 of *LNCS*, pages 219–236. Springer, 2004.

[10] M. F. Fernandez, J. Siméon, B. Choi, A. Marian, and G. Sur. Implementing XQuery 1.0: The galax experience. In *Proc. VLDB 2003*, pages 1077–1080. Morgan Kaufmann, 2003.

[11] P. Ferragina, F. Luccio, G. Manzini, and S. Muthukrishnan. Structuring labeled trees for optimal succinctness, and beyond. In *Proc. FOCS 2005*, pages 184–196. IEEE Press, 2005.

[12] M. J. Fischer. *Grammars with macro-like productions*. PhD thesis, Harvard University, Massachusetts, May 1968.

[13] D. K. Fisher and S. Maneth. Structural selectivity estimation for XML documents. To appear in *Proc. ICDE 2007*, IEEE Press, 2007

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

[15] V. Gapeyev, M. Y. Levin, B. C. Pierce, and A. Schmitt. XML goes native: Run-time representations for Xtatic. In *Proc. CC 2005*, volume 3443 of *LNCS*, pages 43–58. Springer, 2005.

[16] R. F. Geary, R. Raman, and V. Raman. Succinct ordinal trees with level-ancestor queries. In *Proc. SODA 2004*, pages 1–10. SIAM Press, 2004.

[17] R. F. Geary, N. Rahman, R. Raman, and V. Raman. A simple optimal representation for balanced parentheses. *Theor. Comput. Sci.*, 368(3): 231–246, 2006.

[18] F. Gécseg and M. Steinby. Tree languages. In *Handbook of Formal Languages, Volume 3*, Chapter 1. Springer, 1997.

[19] J. Katajainen and E. Mäkinen. Tree compression and optimization with applications. *Intern. J. of Foundations of Comput. Sci.*, 1:425–447, 1990.

[20] J. Lamping. An algorithm for optimal lambda calculus reductions. In *Proc. POPL 1990*, pages 16–30. ACM Press, 1990.

[21] H. Liefke and D. Suciu. XMill: An efficient compressor for XML data. In *Proc. SIGMOD 2000*, pages 153–164. ACM Press, 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] S. Maneth and G. Busatto. Tree transducers and tree compressions. In *Proc. FOSSACS 2004*, volume 2987 of *LNCS*, pages 363–377. Springer, 2004.

[24] D. Megginson. *Imperfect XML: Rants, Raves, Tips, and Tricks ... from an Insider*. Addison-Wesley, 2004.

[25] T. Milo, D. Suciu, and V. Vianu. Typechecking for XML transformers. *J. Comp. Syst. Sci.*, 66:66–97, 2003.

[26] J. Min, M. Park, and C. Chung. XPRESS: A queriable compression for XML data. In *Proc. SIGMOD 2003*, pages 122–133. ACM Press, 2003.

[27] M. Murata, D. Lee, and M. Mani. Taxonomy of XML schema languages using formal language theory. In *Proc. Extreme Markup Languages 2001*.

[28] C. H. Papadimitriou. *Computational Complexity*. Addison-Wesley, New York, 1994.

[29] W. Plandowski. Testing equivalence of morphisms on context-free languages. In *Proc. ESA 1994*, volume 855 of *LNCS*, pages 460–470. Springer, 1994.

[30] W. Rytter. Algorithms on compressed strings and arrays. In *Proc. SOFSEM 1999*, volume 1725 of *LNCS*, pages 48–65. Springer, 1999.

[31] W. Rytter. Application of Lempel-Ziv factorization to the approximation of grammar-based compression. *Theoret. Comput. Sci.*, 302:211–222, 2002.

[32] P. M. Tolani and J. R. Hartisa. XGRIND: A query-friendly XML compressor. In *Proc. ICDE 2002*, pages 225–234. IEEE Press, 2002.

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

[34] B. B. Yao, M. T. Özsu, and N. Khandelwal. XBench benchmark and performance testing of XML DBMSs. In *Proc. ECDE 2004*, pages 621–633. IEEE Press, 2004.

[35] N. Zhang, V. Kacholia, and M. T. Özsu. A succinct physical storage scheme for efficient evaluation of path queries in XML. In *Proc. ICDE 2004*, pages 54–65. IEEE Press, 2004.