Grammar-Based Tree Compression

EPFL Technical Report IC/2004/80*

Giorgio Busatto[1], Markus Lohrey[2], and Sebastian Maneth[3]

[1] Department für Informatik, Universität Oldenburg, Germany
[2] Fakultät Informatik, Universität Stuttgart, Germany
[3] School of Computer and Communication Sciences, EPFL, Switzerland

Abstract. Grammar-based compression means to find a small grammar that generates a given object. Such a grammar reveals the structure of the object (according to the grammar formalism used); the main advantage of this compression method is that the resulting grammar can often be used in further computations without prior decompression. A linear time bottom-up algorithm is presented which transforms a tree into a particular context-free tree grammar. For common XML documents the algorithm performs well, compressing the tree structure to about 5% of the original size. The validation of an XML document against an XML type can be done without decompression, in linear time w.r.t. the size of the grammar (for a fixed type). While the involved grammars can be double exponentially smaller than the represented trees, testing them for equivalence can be done in polynomial space w.r.t. the sum of their sizes.

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 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. There are some improvements leading to more compact representations, e.g., Galax [FSC+03] 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 [MSV03], 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 movement along the edges of the tree) are preserved. Instead of compression, such a representation is also called "data optimization". A well-known method of tree compression is sharing of common subtrees: during a bottom-up phase we determine, using a hash

* See `http://icwww.epfl.ch/publications/`

table, whether we have seen a particular subtree already, and if so we delete it and replace it by the corresponding pointer. In this way we obtain in linear time the minimal (unique) DAG (directed acyclic graph) that represents the tree. For usual XML documents the minimal DAG is about 1/10 of the size of the original tree [BGK03]. As an example, consider the tree $c(c(a,a),c(a,a))$ consisting 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 [MB04]: the shared nodes correspond to nonterminals of the grammar. For example, the above DAG is generated by the regular tree grammar with productions $U \to c(V,V)$, $V \to c(W,W)$, and $W \to a$.

A generalization of sharing of subtrees is the sharing of arbitrary patterns, i.e., connected subgraphs of the tree. In a graph model it leads to the well-known notion of sharing graphs [Lam90] (which are graphs with special "begin-sharing" and "end-sharing" edges, called fan-ins and fan-outs in [Lam90]). As opposed to DAGs which can be at most exponentially smaller than the represented trees, sharing graphs can be at most double-exponentially smaller than the tree. A sharing graph can be seen as a context-free (cf) tree grammar [MB04]. In a cf tree grammar nonterminals can appear inside of a tree (as opposed to at the leaves in the regular case); formal parameters $y_1, y_2, \ldots$ are used in productions in order to indicate where to glue the subtrees of the nonterminal to which the production is applied. 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 because finding the smallest cf (string) grammar for a given string is a well-known NP-complete problem [LS02]. The first main result of this paper is a linear time algorithm for finding a small cf tree grammar for a given tree. On usual XML documents the algorithm performs well, obtaining a grammar that is 1.5-2 times smaller than the minimal DAG. As an example, consider the tree

$$t = c(c(a,a), d(c(a,a), c(c(a,a), d(c(a,a), c(a,a)))))$$

which has 18 edges. The minimal DAG, written as tree grammar, can be seen on the left

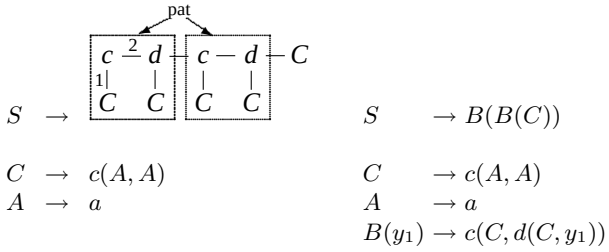

Fig. 1. Regular and cf tree grammars generating $\{t\}$.

of Fig. 1. It is the starting point of our algorithm which tries to transform the grammar into a smaller cf tree grammar. This is done by going bottom-up through the right-hand side of productions, looking for multiple (non-overlapping) occurrences of patterns. 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 t_p all children of nodes of p (and the edges), and label the jth such node (in preoder) by y_j. Thus, $t_{\text{pat}} = c(C, d(C, y_1))$. This tree becomes the right-hand side of a new nonterminal B and the right-hand side of the first production becomes $B(B(C))$. The resulting minimal cf tree grammar is shown on the right of Fig. 1.

The BPLEX algorithm is presented in Section 3. In Section 4 we discuss the application of BPLEX to XML documents and present experimental results which were obtained by running BPLEX on various benchmark XML documents. In Section 5 we study two problems for our tree grammars G that are important for XML documents: (1) to validate against an XML type and (2) to test equivalence. In fact, we consider both of these problems for so called "straight-line" (for short, SL) context-free tree grammmars, which are grammars that are guaranteed to generate at most one tree; the "straight-line" notion is well-known from string grammars (see, e.g., [Ryt04,Pla94]). Since BPLEX generates grammars of a more restricted form (namely, they additionally are linear in the paramters) we also consider problems (1) and (2) for the case that the SL cf tree grammar is linear. It is shown that for an XML type T represented by a (deterministic) bottom-up tree automaton we can test whether or not $L(G)$ has type T in time $O(a^m \times |G|)$, where m is the maximal number parameters of the nonterminals of G and a is the size of the automaton. Note that running a tree automaton is similar to evaluating a query; in [BGK03] it was shown that a 'Core XPath query' Q can be evaluated on an XML document represented by its minimal DAG D in time $O(|Q| \times |D|)$. Moreover, in the string case a similar result of $O(a \times |G|)$ is well-known (see, e.g., Theorem 28(b) of [Ryt99]). Next it is proved that testing the equivalence of two SL cf tree grammars can be done in polynomial space w.r.t. the sum of sizes of the two grammars. If the grammars are linear then testing their equivalence can even be done in polynomial time.

2 Preliminaries

For $k \in \mathbb{N}$, we denote the set $\{1, \ldots, k\}$ with $[k]$. A finite set Σ together with a mapping $\text{rank} : \Sigma \to \mathbb{N}$ is called a ***ranked alphabet***. The set of all (ordered, rooted) ranked trees over Σ is denoted by T_Σ. 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 $k \geq 0$, $Y_k = \{y_1, \ldots, y_k\}$. For a ranked tree t, $V(t)$ denotes its set of nodes and $E(t)$ denotes 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), the node itself is denoted with t_u; for example, 1.1.1 denotes the left-most leaf of the tree t from the Introduction (labeled a). The label at node u is denoted $t[u]$ and the subtree rooted at u is denoted t/u. 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, $1 \leq i \leq n$.

Context-free (cf) tree grammars are a natural generalization of cf grammars to ranked trees (see, e.g., Section 15 in [GS97]). A cf tree grammar G consists of ranked alphabets N and Σ of nonterminal and terminal symbols, respectively, of a start symbol (of rank zero), and of a finite set of productions of the form $A(y_1, \ldots, y_k) \to t$. The right-hand side t of a production of the nonterminal A is a tree over nonterminal and terminal symbols and over the parameters in Y_k which may appear at leaves, where k is the rank of A. Sentential forms are trees s, s' in $T_{N \cup \Sigma}$ and $s \Rightarrow_G s'$ 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 t is the right-hand side of an A-production. Thus, the parameters are used to indicate where to glue the subtrees of a nonterminal, when applying a production to it. The language generated by G is $\{s \in T_\Sigma \mid S \Rightarrow_G^* s\}$. Note that a parameter can cause copying (if it appears more than once in a rhs) or deletion (if it does not appear). For example, the cf tree grammar with productions

$$S \to A(a), \qquad A(y_1) \to A(c(y_1, y_1)), \qquad A(y_1) \to y_1$$

generates the language of all full binary trees over the binary symbol c and the constant symbol a.

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 $\mathrm{rhs}(A_i)$ has no occurrences of A_j for $j < i$. Thus, an SL cf tree grammar can be defined by a tuple $(N, \Sigma, \mathrm{rhs})$ where N is ordered and rhs is a mapping from N to right-hand sides. A 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.

3 BPLEX: A Bottom-Up Algorithm for Grammar-Based Tree Compression

Grammar-based tree compression means 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 can be done in linear time, and the resulting grammar is isomorphic to the minimal DAG of the tree. The minimal regular tree grammar is also straight-line (SL) and, in general, any grammar that generates exactly one element can be turned into an SL grammar. Our starting point for compressing a tree is an SL regular tree grammar, and our algorithm takes such a grammar as input and generates as output a (smaller) SL cf tree 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 (which are 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 which generate the corresponding patterns. We call this technique *multiplexing* because multiple occurrences of the replaced patterns are represented only once in the

output. The algorithm is called BPLEX (for *bottom-up multiplexing*) since the right-hand sides of productions in the input grammar are scanned bottom-up while searching for patterns. Note that a tree pattern can be conveniently described by a tree with parameters at leaves (parameters denote connected subtrees that are not part of the pattern). Formally, a (*tree*) *pattern* p (*of rank* k) is a ranked tree in which each $y \in Y_k$ occurs exactly once. Given a tree t and a node u of t, the pattern p *matches* t in u if there are trees $t_1, \ldots, t_k$ and a pattern p' isomorphic to p such that $t/u = p'\Theta$ where Θ is the substitution $[y_1 \leftarrow t_1, \ldots, y_k \leftarrow t_k]$. The pair (p', Θ) is called a *match* of p (in t) at u. Given a match m, p_m denotes the corresponding pattern. Two matches (p', Θ'), (p'', Θ''), are *overlapping* if p' and p'' have at least one common node. Two matches $m' = (p', [y_1 \leftarrow t'_1, \ldots, y_k \leftarrow t'_k])$, $m'' = (p'', [y_1 \leftarrow t''_1, \ldots, y_k \leftarrow t''_k])$ of the same pattern p are *maximal* if, for all $i \in [k]$, $t'_i[\varepsilon] \neq t''_i[\varepsilon]$ (intuitively: there is no possibility to extend m', m'' to matches of some larger common pattern).

We now discuss how the size of a cf tree 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 SL cf tree 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 rule, and in the order $y_1, y_2, \ldots, y_k$, we do not need to explicitely represent the parameters as nodes of the tree. Hence, we do not count the edges to parameters; thus in general, for a tree t, $\mathrm{size}(t)$ is defined as $|E(t)| - |E_y(t)|$ where $E_y(t)$ are the edges to parameters in t. For a tree grammar G, $\mathrm{size}(G)$ is the sum of sizes of the right-hand sides of the productions of G. Let G be an SL cf tree grammar, p a pattern of rank k with a corresponding production $A(y_1, \ldots, y_k) \rightarrow p$ in G, and $m = (p', [y_1 \leftarrow t_1, \ldots, y_k \leftarrow t_k])$ a match of p in the right-hand side of some other production of G. The match m is replaced by A by deleting the subtree rooted at the root of p' and replacing it by the tree $A(t_1, \ldots, t_k)$. The resulting grammar is denoted by $G[m \leftarrow A]$. Similarly, for two non-overlapping matches m_1, m_2 of p in G, $G[m_1, m_2 \leftarrow A]$ is the grammar obtained from G by replacing each match m_1 and m_2 by A. Clearly, $\mathrm{size}(G) - \mathrm{size}(G[m \leftarrow A]) = \mathrm{size}(p)$ and $\mathrm{size}(G) - \mathrm{size}(G[m_1, m_2 \leftarrow A]) = 2 \times \mathrm{size}(p)$. For a production prod we denote by $\mathrm{add}(G, \mathrm{prod})$ the grammar obtained from G by adding prod. Clearly, if prod is not in G already, then the size of $\mathrm{add}(G, \mathrm{prod})$ is $\mathrm{size}(G) + \mathrm{size}(\mathrm{rhs}(\mathrm{prod}))$. If G is a cf tree grammar, we denote by $\mathrm{fresh}(G, k)$ a nonterminal of rank k that does not occur in G.

The execution of our algorithm produces a sequence $G_1, \ldots, G_h$ of SL cf tree grammars that generate the same tree and share the nonterminals $A_1, \ldots, A_l$. $G_1 = G$ is the SL cf tree grammar that the algorithm takes as input; G_h is the result. New nonterminals $A_{l+1}, A_{l+2}, \ldots$ may appear in $G_2, \ldots, G_h$. Given a grammar G_i, $i \in [h]$, scanning the nodes of the trees $\mathrm{rhs}_{G_i}(A_l)$ through $\mathrm{rhs}_{G_i}(A_1)$ in postorder induces a total order on the set of nodes

$$V^l_{G_i} = \bigcup_{j \in [l]} V(\mathrm{rhs}_{G_i}(A_j))$$

denoted by $<^l_{G_i}$. The reflexive closure of $<^l_{G_i}$ is denoted by $\leq^l_{G_i}$. Scanning the input regular grammar in this order corresponds to scanning the generated tree bottom-up.

Let $i \in [h]$. For $j \in [l]$, a node in $V(\mathrm{rhs}_{G_i}(A_j))$ is denoted by the *address* $z = (j, u)$, where u is the path to that node in the tree $\mathrm{rhs}_{G_i}(A_j)$. If z is a node in $V^l_{G_i}$ that is not the root of $\mathrm{rhs}_{G_i}(A_1)$, then $\mathrm{next}(<^l_{G_i}, z)$ is the node following z in the order $<^l_{G_i}$. The execution of the algorithm on an input grammar G can be described by a sequence of configurations $(G_1, z_1), \ldots, (G_h, z_h)$, where $G = G_1, \ldots, G_h$ are SL cf tree grammars and, for each $i \in [h]$, z_i is the address of some node in $V^l_{G_i}$, called the *current node*, which is examined during the i-th step. Address z_1 is the left-most leaf of $\mathrm{rhs}_{G_1}(A_l)$ and $z_h = (1, \varepsilon)$ is the root of $\mathrm{rhs}_{G_h}(A_1)$. A pattern p *matches* grammar G in $z = (j, u)$ if p matches $\mathrm{rhs}(A_j)$ in u; if $m = (p', \Theta)$ is the match of p in $z = (j, u)$, z is the address of m in G.

At step $i \in [h]$, the algorithm computes the set $M_1(G_i, z_i, K_P)$ of all matches in z_i of patterns that are isomorphic to a right-hand side $\mathrm{rhs}_{G_i}(A_j)$ for some $j > l$. The parameter $K_P \in \mathbb{N}$ is introduced for efficiency reasons, and indicates that only the latest K_P productions with index greater than l are considered. This is similar to the idea of a sliding window as it is used in many implementations of the LZ77 compression scheme, cf. also the discussion in Section 6. Note that one can check whether $p = \mathrm{rhs}_{G_i}(A_j)$ matches G_i in z_i in at most $\mathrm{size}(p)$ steps by comparing the two trees top-down and binding parameters of p to descendants of z_i. The total cost of computing $M_1(G_i, z_i, K_P)$ is bounded by $K_S \times K_P$, where K_S (see below) is the maximum size for a production with index $j > l$. Additionally, the algorithm computes the set $M_2(G_i, z_i, K_N, K_S)$ of all matches in z_i such that, for each $m \in M_2(G_i, z_i, K_N, K_S)$, we have

- there exists a (non-overlapping) *companion match* c_m of the same pattern in some node w among the last K_N nodes preceding z_i in the order $<^l_{G_i}$, and
- $0 < \mathrm{size}(p_m) \leq K_S$ and, if $\mathrm{size}(p_m) < K_S$, then m and c_m are maximal.

The set $M_2(G_i, z_i, K_N, K_S)$ can be computed by comparing top-down the tree rooted at z_i with trees rooted at nodes preceding z_i. The cost of computing $M_2(G_i, z_i, K_N, K_S)$ is bounded by $K_S \times K_N$. After computing the two sets of matches, the algorithm chooses a match $m \in M_1(G_i, z_i, K_P) \cup M_2(G_i, z_i, K_N, K_S)$ with maximal size denoted by $\max(M_1, M_2)$. If the chosen match m belongs to $M_1(G_i, z_i, K_P)$, then the match is replaced by the right-hand side of the corresponding production. If m belongs to $M_2(G_i, z_i, K_N, K_S)$, a new production with a nonterminal A and $\mathrm{rhs}_{G_i}(A) = p_m$ is added to the grammar, and the matches m, c_m are replaced by A. In both cases, the size of the grammar is reduced by $\mathrm{size}(p_m)$. If $M = \emptyset$, we move the current node to the next node with respect to the order $<^l_{G_i}$. This procedure is repeated until the root of $\mathrm{rhs}_{G_i}(A_1)$ is reached. The linearity of the algorithm derives from the fact that, for an input grammar G, the body of the loop cannot be executed more than $2 \times |G|$ (each run through the body either moves the address forward or reduces the size of the grammar), and from the fact that the two sets $M_1(G_i, z_i, K_P)$ and $M_2(G_i, z_i, K_N, K_S)$ can be computed in constant time.

The algorithm is shown in Fig. 2. Let us consider its behavior on the regular tree grammar on the left of Fig. 1. A postorder traversal of the grammar goes through the third and second production without finding any pair of matches that can be replaced. It then continues scanning 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

$G :=$ input grammar
$z :=$ leftmost leaf of $\mathrm{rhs}_G(A_l)$
while true do
  $M_1 := M_1(G, z, K_P)$
  $M_2 := M_2(G, z, K_N, K_S)$
  if $M_1 \neq \emptyset$ **or** $M_2 \neq \emptyset$ **then**
    $m := \max(M_1, M_2)$
    if $m \in M_1$ **then**
      $G := G[m \leftarrow A]$, with $\mathrm{rhs}_G(A) = p_m$
    else
      $k := \mathrm{rank}(p_m)$
      $A := \mathrm{fresh}(G, k)$
      $G := \mathrm{add}(G, A(y_1, \ldots, y_k) \rightarrow p_m)$
      $G := G[m, c_m \leftarrow A]$
    endif
  elif $\exists w \in V_G^l : z <_G^l w$ **then**
    $z := \mathrm{next}(<_G^l, z)$
  else
    break
  endif
enddo

Fig. 2. The complete algorithm

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 production $D(y_1) \rightarrow d(C, y_1)$, and the two matches are replaced so that the first production becomes $S \rightarrow c(C, D(c(C, D(C))))$. 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 maximal pattern $c(C, D(y_1))$ matches in $(1, \varepsilon)$ and in $(1, 2.1)$. Therefore a new nonterminal E of rank 1 is added together with the production $E(y_1) \rightarrow c(C, D(y_1))$, and the matches are replaced by E, producing the output grammar shown in Fig. 3. This grammar and the cf tree grammar on the right of Fig. 1 have both size

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

Fig. 3. Cf tree grammar generating $\{t\}$.

7. Note, however, that the algorithm has split the pattern $p = c(C, d(C, y_1))$ into two sub-patterns, since pattern $d(C, y_1)$ is detected and replaced before the larger pattern p is scanned completely.

4 XML Compression using BPLEX

An XML document is a sequential representation of a nested list structure. 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 more than the size of the original XML document. Another data model are (ordered) unranked trees which are like DOM trees, but without pointers between siblings. As an example, consider the following XML document skeleton (i.e., without data values).

```
<agenda>
    <person> <name/> <street/> </person>
    <person> <name/> <street/> </person>
    <person> <name/> <street/> </person>
    <person> <name/> <street/> </person>
    <person> <name/> <street/> </person>
</agenda>
```

An (ordered) unranked tree representation of this XML document 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 unranked tree representation of the above XML document is 15 edges.

The BPLEX algorithm works on ranked trees; on the other hand every unranked tree can be turned into a binary ranked tree without changing the number of edges: delete all edges to non-first children, and add a (second child) edge from any node to its next sibling. Note that a leaf (resp. the last sibling) in the unranked tree has no left (resp. no right) child edge in the binary tree representation; this is denoted by the superscript 2 (resp. 1), and by 0 for a last sibling leaf. In Fig. 4 the binary representation of the unranked tree for the XML document of above is shown (with second child edges drawn

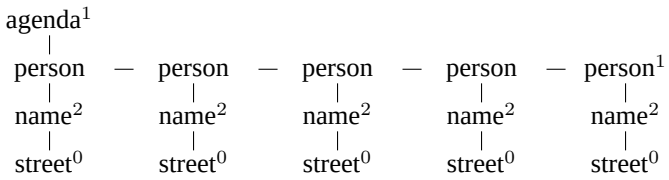

Fig. 4. Binary tree representation of an unranked tree.

horizontally). 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. For the example the corresponding grammar G_5 has the two productions

$$S \to \text{agenda}^1(\text{person}(A, \text{person}(A, \text{person}(A, \text{person}(A, \text{person}^1(A))))))$$
$$A \to \text{name}^2(\text{street}^0).$$

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

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

Fig. 5. Output of BPLEX on G_5.

grammar into the cf tree grammar shown in Fig. 5. In this grammar, the D-productions 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 grammar shown in Fig. 6, which has size 20. In this grammar, the symbol A_3 generates the tree person(name(street, y_1)). More generally, for $j = 3, \ldots, 15$, A_j generates a chain with 2^{j-3} 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.

$$\begin{array}{lcl} S & \to & \text{agenda}^1(A_7(A_{11}(A_{12}(A_{13}(A_{15}(A_{15})))))) \\ A_1 & \to & \text{street}^0 \\ A_2 & \to & \text{name}^2(A_1) \\ A_3(y_1) & \to & \text{person}(A_2, y_1) \\ A_4(y_1) & \to & A_3(A_3(y_1)) \\ A_5(y_1) & \to & A_4(A_4(y_1)) \\ A_6(y_1) & \to & A_5(A_5(y_1)) \\ A_7(y_1) & \to & A_6(A_6(y_1)) \end{array} \qquad \begin{array}{lcl} A_8(y_1) & \to & A_7(A_7(y_1)) \\ A_9(y_1) & \to & A_8(A_8(y_1)) \\ A_{10}(y_1) & \to & A_9(A_9(y_1)) \\ A_{11}(y_1) & \to & A_{10}(A_{10}(y_1)) \\ A_{12}(y_1) & \to & A_{11}(A_{11}(y_1)) \\ A_{13}(y_1) & \to & A_{12}(A_{12}(y_1)) \\ A_{14}(y_1) & \to & A_{13}(A_{13}(y_1)) \\ A_{15}(y_1) & \to & A_{14}(A_{14}(y_1)) \end{array}$$

Fig. 6. Output of BPLEX on G_{10000}.

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 4. 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 $t_u = u(p(x, b, c, b, c), p(y, b, c, b, c), p(z, b, c, b, c))$. 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.

In fact, the size of the minimal DAG representation can even be further reduced by using "multiplicity" counters for consecutive equal subtrees [BGK03]. Then the DAG for the unranked tree of the agenda-example can be represented using only 3 edges, or equivalently, by an (unranked) regular tree grammar with multiplicity counters and productions $A \to \text{agenda}([5]P), P \to \text{person}(\text{name}, \text{street})$. Of course, multiplicity counters take up space, but following Koch et al. this space is neglected (similar to the fact that we do not count edges to parameters in cf tree grammars, see Section 3). Thus, BPLEX produces the grammar of Fig. 5 that 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 this DAG representation for an unranked tree its mDAG (minimal DAG with multiplicities). Such 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_u of above; cf. in Table 1 the two catalog files and the file NCBI_gene.chr1.

Experimental results

We tested BPLEX on three different sets of XML documents. The first one contains most of the documents used in [BGK03], namely SProt (protein data), DBLP (a bibliographic database), Treebank (a linguistic database), and 1998statistics (baseball statistics). The second set contains XML documents generated with the XBench benchmark [YÖK04]. The third set contains documents from the Japanese Single Nucleotide Polymorphism database [HTH$^+$02]. BPLEX was implemented in C on a Linux platform and the tests were run with $K_N = 50000$, K_P unbounded, and $K_S = 500$; the

results are shown in Fig. 1. For each XML document, the table shows the size (number of edges) of its skeleton, the size and compression ratio of the corresponding minimal binary DAG, the size and compression ratio of the mDAG, and finally the size and compression ratio of the output of BPLEX.

On the larger files from the first group, the output of BPLEX has approximately half the size of the corresponding mDAG. On the files of the second group BPLEX performs even better, producing in most cases a grammar whose size is one third of the corresponding mDAG. In many of the files from the snp repository, BPLEX produces an output with at most half the size of the corresponding mDAG (see the NCBI_gene files in the table). On other files (e.g. JST_snp.chr1) BPLEX improves the compression ratio by about 1.5 only. On other files (like NCBI_snp.chr1) the construction of the mDAG already provides an exponential size reduction; on these files BPLEX also gives exponential compression.

Input file	$\mathrm{size}(t)$	Min. binary reg. gr.		Min. unranked reg. gr. (mDAG)		BPLEX output	
SwissProt	$10,903,568$	$1,664,451$	15.0%	$1,100,648$	10.1%	$337,129$	3.1%
DBLP	$2,611,931$	$533,188$	20.0%	$222,755$	8.5%	$115,954$	4.4%
Treebank	$2,447,727$	$1,456,707$	60.0%	$1,301,690$	53.2%	$550,230$	22.0%
1998statistics	$28,306$	$2,404$	8.5%	727	2.6%	411	1.5%
catalog-02	$2,240,231$	$58,529$	2.6%	$74,165$	3.3%	$30,451$	1.4%
catalog-01	$225,194$	$9,581$	4.3%	$20,298$	9.0%	$4,916$	2.2%
dictionary-02	$2,731,764$	$681,155$	25.0%	$547,834$	20.0%	$161,611$	5.9%
dictionary-01	$277,072$	$77,555$	28.0%	$58,573$	21.0%	$20,193$	7.3%
JST_snp.chr1	$655,946$	$44,321$	6.8%	$25,047$	3.8%	$15,807$	2.4%
JST_gene.chr1	$216,401$	$15,314$	7.1%	$7,386$	3.4%	$4,839$	2.2%
NCBI_snp.chr1	$3,642,225$	$809,395$	22.0%	15	$< 0.1\%$	49	$< 0.1\%$
NCBI_gene.chr1	$360,350$	$19,715$	5.5%	22491	6.2%	$9,506$	2.6%

Table 1. Experimental results with BPLEX.

5 Algorithms on SL Context-Free Tree Grammars

As our experimental results show, SL cf tree grammars are well suited to efficiently represent XML documents; especially if the underlying tree model is that of a binary tree (unlike, e.g., DOM). Consider now a grammar in memory which represents a large XML document. How can we process the XML tree that is represented, without decompressing the grammar?

Any read access to the tree like, e.g., reading the label of the root node, or moving along an edge from one node to another node, can be realized on the grammar representation with an additional per-step overhead of at most the size h of the grammar [MB04]. Additionally, a stack of height at most h must be maintained at all times. Thus, the price to be payed for the fact that we have 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 drastic speed ups. In this section we investigate such applications.

5.1 XML Type Validation

The first application we consider is XML type validation: an XML document represented by an SL cf tree grammar should be validated against an XML type. There are several formalisms for describing XML types, with varying expressiveness, e.g., DTDs, XML Schema, or RELAX NG. All of these can conveniently be modeled by the regular tree languages [MLM00], a classical concept well known from formal language theory [GS97]. Our first result states that XML type checking can be done in time linear in the size of the grammar G, if the maximal number of parameters m is fixed. The involved constant depends on the size of the XML type definition, and on the maximal number m of parameters of the nonterminals in G; in fact, m appears as an exponent. Note that our algorithm can easily be adapted to take m as an input parameter. Practical experiments show that small values of m already achieve competitive compression rates. It can therefore be assumed that m is very small with respect to the size of G (the average value of m w.r.t. the size of G in Table 1 is around 10^{-4}). As formal model for regular tree languages we use (deterministic bottom-up finite) tree automata. Such an automaton can be defined by a tuple $A = (Q, \Sigma, \{\delta_\sigma\}_{\sigma\in\Sigma}, F)$ 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. The transition function δ of A is extended to trees over Σ in the usual way: $\delta(\sigma(t_1, \ldots, t_k)) = \delta_\sigma(\delta(t_1), \ldots, \delta(t_k))$ for $\sigma \in \Sigma$ of rank k and $t_1, \ldots, t_k \in T_\Sigma$. The language accepted by A is $\{s \in T_\Sigma \mid \delta(s) \in F\}$.

Theorem 1. *Given an SL cf tree grammar G and a tree automaton B it can be checked whether or not $L(G) \cap L(B) = \varnothing$ in time $O(s^m \times |G|)$, where s is the number of states of B and m is the maximal number of parameters of the nonterminals of G.*

Proof. Let $G = (N, \Sigma, \mathrm{rhs})$ with $N = \{A_1, \ldots, A_n\}$ and $B = (Q, \Sigma, \delta, F)$. We assume that G is reduced, i.e., each nonterminal is used in a (successful) derivation of G. We now run the tree automaton B on the right-hand sides of G, starting bottom-up with the right-hand side $\mathrm{rhs}(A_n)$ of the last nonterminal of G. For parameters $y_1, \ldots, y_k$ which (possibly) appear in $\mathrm{rhs}(A_n)$ we do not yet know the actual trees; we therefore try all possible combinations $(q_1, \ldots, q_k)$ of states of B, and store this in the function $\Psi_{A_n}(q_1, \ldots, q_k) = \delta(\mathrm{rhs}(A_n)[y_1 \leftarrow q_1, \ldots, y_k \leftarrow q_k])$. In a similar way we compute $\Psi_{A_{n-1}}$, using Ψ_{A_n} at occurrences of A_n in $\mathrm{rhs}(A_{n-1})$. Continuing in this way, we obtain after n steps the constant function $\Psi_{A_1}() \in Q$ which is the state in which B arrives for the tree t with $L(G) = \{t\}$.

For each nonterminal of rank k, $|Q|^k$ many values of Ψ are computed. Hence, in total at most $s^m \times |G|$ computations steps are needed. □

Note that it is not necessary to compute Ψ_{A_i} for *all* combinations of $(q_1, \ldots, q_k)$; rather, this computation can be deferred until a concrete tuple of states has been determined. In this 'lazy' manner, the factor s^m can be reduced significantly. Note further that in order to use Theorem 1 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 tree automaton can be done in time linear in the size of the representation; the reason is that these formalisms are deterministic: there is only one rule per nonterminal, and the regular expressions which are used in right-hand sides are also deterministic (which implies that the corresponding Glushkov automaton is deterministic, which can be constructed in time linear in the size of the expression). Hence, the algorithm of the proof of Theorem 1 is highly practical for DTDs and XML Schemas. For RELAX NG (which employs full regular tree languages) it might be less practical, because the size of the corresponding tree automaton can be exponential in the size of the representation r. However, if the SL cf tree grammar is linear (which it is, if it was produced by BPLEX), then Theorem 1 can be extended to the case that the automaton B is nondeterministic: the Ψ_A are now functions from Q^k to 2^Q, where k is the rank of A; they are computed by checking for every state p and states $p_1, \ldots, p_k$ 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. Thus the problem can be solved in time $O(s^{m+1} \times |G|)$.

Validation of an XML document against a type description can also be done in an approximative way. In fact, in [MdR95] it was shown that it can be decided in constant time whether an XML document validates, or if is "far" from it. As distance measure they use the tree edit distance with moves. Let us now show that their approximative validation can also be done on compressed XML documents. The proof of Theorem 1 can easily be adapted in order to construct a finite tree automaton which runs on a tree representation of the grammar (such a tree is simply an abstract derivation tree in which copies of nonterminals are not taken into account, i.e., the rank of a production equals the number of different nonterminals in its right-hand side). The states of the automaton are finite functions from Q^m to Q and the transitions are computed in exactly the same way as in the proof of Theorem 1. Note that this result can also be formulated in terms of MSO (monadic second-order) logic: every MSO-definable property on trees is also MSO definable on (SL cf grammar-) compressed trees. Even though the above constructed automaton allows to do approximative validation, it is not clear yet how good the approximation is with respect to one on the original tree. Therefore it should be investigated how edit distances on trees change when moving to a compressed structure.

5.2 Testing Equivalence of SL Context-Free Tree Grammars

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, 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 [Pla94]. The proof relies on the fact that, for an SL cf string grammar G (in Chomsky nf) 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 t 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, 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 [Pla94] can be used to show that in this case testing equivalence can be done in polynomial time.

Theorem 2. *Testing equivalence of two SL cf tree grammars G_1 and G_2 can be done in* PSPACE*, and in polynomial time if G_1 and G_2 are linear.*

Proof. Let $G_1 = (\{A_1, \ldots, A_m\}, \Sigma, \mathrm{rhs}_1)$, $G_2 = (\{B_1, \ldots, B_n\}, \Sigma, \mathrm{rhs}_2)$ be SL cf tree grammars. By Savitch's Theorem (see, e.g., [Pap94]) and the complement closure of PSPACE, it suffices to give a nondeterministic algorithm that tests *in*equivalence. Roughly speaking, the algorithm guesses a node u in d_1 (the DAG represented by G_1) and accepts if the label of u in d_1 is different from u's label in d_2 (the DAG represented by G_2). The key issue is that a node in d_1 (d_2) can be represented in polynomial space w.r.t. the size of G_1 (G_2). This representation is discussed in the end of [MB04]. It consists of a sequence

$$(i_1, u_1), (i_2, u_2) \ldots, (i_p, u_p)$$

where $i_1 = 1$, $i_1 < \cdots < i_p$ are indices in $\{1, \ldots, n\}$, and for $1 \leq \nu \leq p$, u_ν is a node in $\mathrm{rhs}_1(A_{i_\nu})$ with label $A_{i_{\nu+1}}$; moreover $\mathrm{rhs}_1(A_{i_p})[u_p] \in \Sigma$. The first pair $(1, u_1)$ denotes that we start a derivation of G_1 with the right-hand side of A_1 and node u_1 marked; the next pair (i_2, u_2) means u_1 is labeled A_{i_2} and that we apply its production with u_2 is marked, etc. Since u_p is terminal, the sequence represents a derivation of a node of t_1. Given such a sequence h representing a node u of t_1 it is straightforward to construct a sequence h' representing the i-th child ui of u in t_1 [MB04]. Note that any such sequence has length $< n$. The algorithm starts with two empty sequences. It then generates the sequences h_1, h_2 representing the root nodes of t_1, 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, resulting in h_1', h_2'. If the corresponding labels are different we accept, etc. If there is no child number (we are at a leaf) we reject.

Now let G_1, G_2 be linear. This means that for any nonterminal A of G_1, 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, $1 \leq j \leq k$. In fact, 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 traversals of t_1 and t_2, respectively. Let $i \in \{1, 2\}$. For every nonterminal X of G_i of rank $k > 0$ let $X_{0,1}, X_{1,2}, \ldots, X_{k-1,k}, X_{k,0}$ 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 nonterminal $A_{0,1}$ generates the traversal starting at the root node of the corresponding right-hand side (indicated by the index 0) up to the first parameter y_1 of the right-hand side (indicated by the index 1). The

nonterminal $A_{\nu,\nu+1}$ generates the traversal starting at y_ν (and going up), until the parameter $y_{\nu+1}$ is encountered. Similarly, a terminal symbol $g_{2,3}$ means that g was entered coming from its second child and was exited by moving to its third 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 w_2 (gen. by H_2). Moreover, H_1, H_2 are SL cf string grammars of polynomial size w.r.t. G_1, G_2, respectively. By the result of [Pla94], testing $w_1 = w_2$ can be done in polynomial time w.r.t. the sizes of H_1, H_2. □

6 Conclusions and Further Research Topics

We have presented a linear time algorithm which can be used to find a small SL cf tree grammar for a given (ranked) tree. The size of the resulting grammar is usually 75%-50% of the size of the unique minimal DAG of the tree. We have adapted the algorithm to compress memory representations of XML documents, obtaining for large files about half the size of the representation of Koch et. al. [BGK03].

Consider 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 [LS02]. In particular, the size of the smallest cf grammar is lower bounded by the size of the smallest LZ77 representation of the string (when no sliding window is used) [CLL$^+$02,Ryt02]. The question arises whether a similar result holds in the tree case. But, what is an LZ77 representation of a tree? For LZ77 on strings, the prefix to the current position is considered for finding the longest substring that matches at the current position; often, only a fixed length prefix – the sliding window – is considered. 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 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) [Che04], but such pairs $[p,t]$ require too much space in order to obtain good compression. The main idea of [Ryt02] is to obtain grammars with balanced derivation trees, called "AVL-grammars". This technique seems to be applicable to cf tree grammars too, and it remains to be checked whether it gives rise to better approximation algorithms than the one presented here.

Another variation of Lempel-Zip 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 $(3,a)$ is bba. A simple extension to trees is to consider complete subtrees as

patterns [CR96]. It seems however, that the size of such a representation will be lower-bounded by the size of the minimal DAG. A more powerful extension is to consider trees with parameters as patterns and to compress a tree into a "pattern substitution tree" which has edges labeled by substitutions [Che98].

There are also succinct tree representations that do not use pointers to represent edges, see, e.g., [KM90,DCW93]. Recently it has been shown that an n-node tree can be represented by $2n + o(n)$ bits, while allowing $O(1)$ time for most read operations on the tree [GRR04]. Also in the context of XML pointerless representations exists; for example, in XPRESS [MPC03] label paths in an XML document are encoded by real number intervals following an arithmetic encoding technique; this allows to run path queries directly on the compressed instance.

It should be mentioned that context-free tree grammars are inspired by macro grammars [Fis68] which are cf grammars with parameters. Such grammars can be used for grammar-based string compression and support at most double exponential compression. It remains further research to investigate whether our algorithm can be used to find small macro grammars for given strings.

References

[BGK03] P. Buneman, M. Grohe, and C. Koch. Path queries on compressed XML. In J. C. Freytag et al., editor, *Proc. VLDB'2003*, pages 141–152. Morgan Kaufmann, 2003.

[Che98] J. R. Cheney. First-order term compression: techniques and applications. Master's thesis, Carnegie Mellon University, August 1998.

[Che04] J. R. Cheney. Personal communication. 2004.

[CLL+02] M. Charikar, E. Lehman, D. Liu, R. Panigrahy, M. Prabhakaran, A. Rasala, A. Sahai, and A. Shelat. Approximating the smallest grammar: Kolmogorov complexity in natural models. In *Proc. STOC'02*, pages 792–801. ACM Press, 2002.

[CR96] S. Chen and J. H. Reif. Efficient lossless compression of trees and graphs. In J. A. Storer and M. Cohn, editors, *Proceedings of the 6th Data Compression Conference – DCC '96*, page 428. IEEE Computer Society Press, 1996.

[DCW93] J. J. Darragh, J. G. Cleary, and I. H. Witten. Bonsai: a compact representation of trees. *Softw., Pract. Exper*, 23:277–291, 1993.

[Fis68] M.J. Fischer. *Grammars with macro-like productions*. PhD thesis, Harvard University, Massachusetts, May 1968.

[FSC+03] M. F. Fernandez, J. Siméon, B. Choi, A. Marian, and G. Sur. Implementing xquery 1.0: The galax experience. In J. C. Freytag et al., editor, *Proc. VLDB'2003*, pages 1077–1080. Morgan Kaufmann, 2003.

[GRR04] R. F. Geary, R. Raman, and V. Raman. Succinct ordinal trees with level-ancestor queries. In *Proc. SODA'2004*, pages 1–10, 2004.

[GS97] F. Gécseg and M. Steinby. Tree languages. In G. Rozenberg and A. Salomaa, editors, *Handbook of Formal Languages, Volume 3*, chapter 1. Springer-Verlag, 1997.

[HTH+02] M. Hirakawa, T. Tanaka, Y. Hashimoto, M. Kuroda, T. Takagi, and Y. Nakamura. JSNP : a database of common gene variations in the japanese population. *Nucleic Acids Research*, 30:158–162, 2002.

[KM90] J. Katajainen and E. Mäkinen. Tree compression and optimization with applications. *Intern. J. of Foundations of Comput. Sci.*, 1:425–447, 1990.

[Lam90] J. Lamping. An algorithm for optimal lambda calculus reductions. In *Proc. POPL'1990*, pages 16–30. ACM Press, 1990.

[LS02] E. Lehman and A. Shelat. Approximation algorithms for grammar-based compression. In *Proceedings of the 13th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA'2002)*, pages 205–212. SIAM Press, 2002.

[MB04] S. Maneth and G. Busatto. Tree transducers and tree compressions. In I. Walukiewicz, editor, *Foundations of Software Science and Computation Structures - FOSSACS'04*, volume 2987 of *LNCS*, pages 363–377, Barcelona, Spain, 2004. Springer-Verlag.

[MdR95] F. Magniez and M. de Rougemont. Property testing of regular tree languages. In *Proc. ICALP 2004*, volume 3142 of *LNCS*, pages 932–944. Springer-Verlag, 1995.

[MLM00] M Murata, D. Lee, and M. Mani. Taxonomy of XML schema languages using formal language theory. In *Proc. Extreme Markup Languages '2000*, 2000.

[MPC03] J. Min, M. Park, and C. Chung. XPRESS: A queriable compression for XML data. In *Proc. SIGMOD 2003*, pages 122–133. ACM Press, 2003.

[MSV03] T. Milo, D. Suciu, and V. Vianu. Typechecking for XML transformers. *J. of Comp. Syst. Sci.*, 66:66–97, 2003.

[Pap94] Christos H. Papadimitriou. *Computational Complexity*. Addison-Wesley, New York, 1994.

[Pla94] W. Plandowski. Testing equivalence of morphisms on context-free languages. In Jan van Leeuwen, editor, *Proc. Second European Symposium on Algorithms – ESA'94*, volume 855 of *LNCS*, pages 460–470. Springer-Verlag, 1994.

[Ryt99] W. Rytter. Algorithms on compressed strings and arrays. In *Proc. SOFSEM 1999*, volume 1725 of *LNCS*, pages 48–65. Springer-Verlag, 1999.

[Ryt02] W. Rytter. Application of lempel-ziv factorization to the approximation of grammar-based compression. *Theoret. Comput. Sci.*, 302:211–222, 2002.

[Ryt04] W. Rytter. Grammar compression, LZ-encodings, and string algorithms with implicit input. In J. Diaz, J. Karhumäki, A. Lepistö, and D. Sannella, editors, *Proceedings of the 31st International Colloquium on Automata, Languages and Programming (ICALP 2004)*, volume 3142 of *LNCS*, pages 15–27, 2004.

[YÖK04] 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 Computer Society, 2004.