XML Compression via DAGs

Markus Lohrey
Universität Leipzig

Sebastian Maneth*
University of Oxford

Eric Noeth
Universität Leipzig

ABSTRACT

Unranked trees can be represented using their minimal dag (directed acyclic graph). For XML this achieves high compression ratios due to their repetitive mark up. Unranked trees are often represented through first child/next sibling (fcns) encoded binary trees. We study the difference in size (= number of edges) of minimal dag versus minimal dag of the fcns encoded binary tree. One main finding is that the size of the dag of the binary tree can never be smaller than the square root of the size of the minimal dag, and that there are examples that match this bound. We introduce a new combined structure, the *hybrid dag*, which is guaranteed to be smaller than (or equal in size to) both dags. Interestingly, we find through experiments that last child/previous sibling encodings are much better for XML compression via dags, than fcns encodings. This is because optional elements are more likely to appear towards the end of child sequences.

Categories and Subject Descriptors

E.4 [**Coding and Information Theory**]: Data compaction and compression

Keywords

XML, Tree Compression, Directed Acyclic Graph

1. INTRODUCTION

The tree structure of an XML document can be conveniently represented by an ordered unranked tree [24, 20]. For tree structures of common XML documents it was shown in [4] that *dags* (*directed acyclic graphs*) offer high compression ratios: the number of edges of the minimal dag is only about 10% of the number of edges of the original unranked tree. In a minimal dag, each distinct subtree is represented only once. A dag can be exponentially smaller than the represented tree. It was shown in [8] that queries can be evaluated directly on the dag (without prior decompression). Dags and their (average) time construction via hashing are folklore in computer science (see e.g. [7]); they are a popular data structure used for sharing of common subexpressions (e.g., in programming languages) and in binary decision diagrams, see [19]. Through a clever pointer data structure, worst-case linear time construction is shown in [6].

Unranked trees of XML tree structures are often represented using binary trees, see [23] for a discussion. A common encoding for XML is the *first child/next sibling encoding* [10] (in fact, this encoding is well-known in computer science, see Paragraph 2.3.2 in Knuth's first book [9]). The binary tree $\mathrm{fcns}(t)$ is obtained from an unranked tree t as follows. Each node of t is a node of $\mathrm{fcns}(t)$. A node u is a left child of node v in $\mathrm{fcns}(t)$ if and only if u is the first child of v in t. A node u is the right child of a node v in $\mathrm{fcns}(t)$ if and only if u is the next sibling of v in t. From now on, when we speak of the size of a graph we mean its number of edges. Consider the minimal dag of $\mathrm{fcns}(t)$ (called *bdag* for *binary dag* in the following) in comparison to the minimal dag of t. It was observed in [5] that the sizes of these dags may differ, in both directions. For some trees the difference is dramatic, which motivates the work of this paper: to study the precise relationship between the two dags, and to devise a new data structure that is guaranteed to be of equal or smaller size than the minimum size of the two dags.

Intuitively, the dag of t shares *repeated subtrees*, while the dag of $\mathrm{fcns}(t)$ shares *repeated sibling end sequences*. Consider

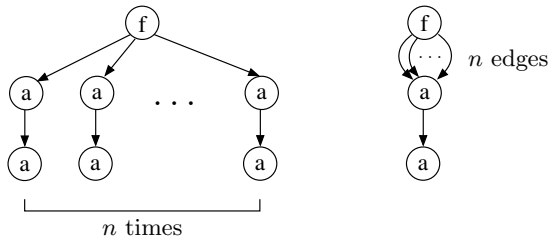

Figure 1: The unranked tree t_n and dag(t_n).

the tree t_n in the left of Figure 1. Its minimal dag is shown on the right. As can be seen, each repeated subtree is removed in the dag. The dag consists of $n+1$ edges while t_n consists of $2n$ edges. Moreover, $\mathrm{fcns}(t_n)$ does not have any repeated subtrees (except for leaves), i.e., the bdag of t_n has $2n$ edges as well. Next, consider the tree s_n in the left of

*This research was carried out while the author was visting University of Leipzig on a Mercator (DFG) grant.

EDBT/ICDT'13, March 18-22, 2013, Genoa, Italy

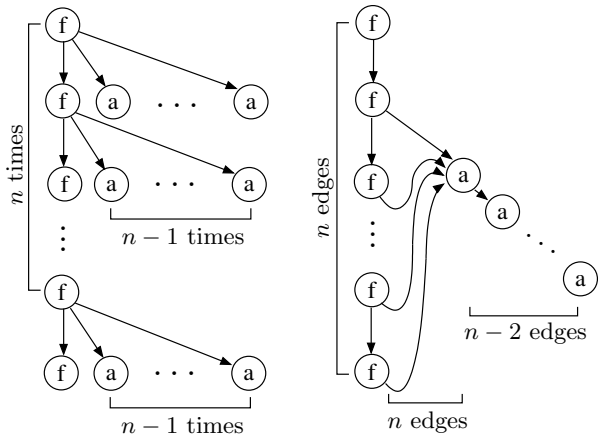

Figure 2: The unranked tree s_n and bdag(s_n).

Figure 2. Its bdag is shown on the right, it has $3n-2$ edges. On the other hand, s_n has n^2 edges and the same is true for the dag of s_n since this tree has no repeated subtrees (except for leaves). These two examples show that (i) the size of the dag of an unranked tree can be half the size of the dag of the fcns-encoded tree and (ii) the size of the dag of the fcns-encoded tree can be quadratically smaller than the size of the dag of the unranked tree. We prove in this paper that these ratios are maximal: The size of the dag of the unranked tree is (i) lower bounded by half of the size of the bdag and (ii) upper bounded by the square of the size of the bdag. Actually, we derive these bounds from stronger statements concerning a combination of the unranked dag and the binary dag, called the *hybrid dag*, which combines both ways of sharing. The idea is as follows. Given an unranked tree, we compute its minimal dag. The dag can be naturally viewed as a regular tree grammar: Introduce for each node v of the dag a nonterminal A_v for the grammar. If a node v is labeled with the symbol f and its children in the dag are $v_1, \ldots, v_n$ in this order, then we introduce the production $A_v \to f(A_{v_1}, \ldots, A_{v_n})$. We now apply the fcns encoding to all right-hand sides of this grammar. Finally, we compute the minimal dag of the forest consisting of all these fcns encoded right-hand sides. See Figure 3 which shows a tree t of size 9. Its unranked and binary dags are each of size 6. The hybrid dag consists of a start tree plus one rule, and is of size 5. For the XML document trees of our corpus, the average size of the hybrid dag is only 76% of the average size of the unranked dag.

We show that the size of the hybrid dag is always bounded by the minimum of the sizes of the unranked dag and the binary dag. Moreover, we show that (i) the size of the hdag is at least half of the size of the binary dag and (ii) the size of the unranked dag is at most the square of the size of the hdag. The above mentioned bounds for the unranked dag and binary dag are direct corollaries of these bounds.

The tree grammar of a hybrid dag is not a regular tree grammar anymore (because identifier nodes may have a right child). It can be unfolded in three passes: first undoing the sharing of tree sequences, then the binary decoding, and then undoing sharing of subtrees. We show that these grammars can be translated into a well known type of grammars: straight-line linear context-free tree grammars, for short *SLT grammars* (produced by BPLEX [5] or TreeRePair [15]). This embedding increases the size only slightly. One advantage is that SLT grammars can be unfolded into the original tree in one pass. Moreover, it is known that finite tree automata (even with sibling equality constraints) and tree walking automata can be executed in polynomial time over trees represented by SLT grammars [14, 16].

While in the theoretical limit the binary dag can be smaller in comparison than the dag, it was observed in [5] that for common XML document trees t, almost always the dag of t is smaller than the binary dag of t. One explanation is that t contains many small repeated subtrees, which seldomly are part of a repeating sibling end sequence. For each repetition we (possibly) pay a "penalty" of one extra edge in the dag of fcns(t); see the tree t_n which has penalty n. On the other hand, there are very few repeating sibling end sequences in common XML; this is because optional elements typically appear towards the end of a child sequence. Hence, the additional feature of sharing sibling sequences is not useful for XML. On real XML documents, we show in experiments that the "reverse binary dag" that arises from the *last child/previous sibling encoding* is typically smaller than the binary dag, and almost as small as the dag. Moreover, for our test corpus, the average size of the *reverse hybrid dag* built from the last child/previous sibling encoding of the dag is only 62% of the average size of the minimal dag (in contrast to 76% for the hdag described above).

Observe that in the second sharing phase of the construction of the hybrid dag, only sequences of identifiers (nonterminals of the regular tree grammar corresponding to the dag) are shared. Thus, we are sharing repeated string suffixes in a sequence of strings. We experimented with applying a grammar-based string compressor to this sequence of strings. It is not difficult to incorporate the output into an SLT grammar. As our experiments show, the obtained grammars are smaller than those of the hybrid dag and almost as small as TreeRePair's grammars. Moreover, they have the advantage that checking equivalence of subtrees is simple (each distinct subtree is represented by a unique identifier), a property not present for arbitrary SLT grammars. For hybrid dags, even equality of sibling end sequences can be checked efficiently.

2. TREES AND DAGS

Let Σ be a finite set of node labels. An *ordered Σ-labeled multigraph* is a tuple $M = (V, \gamma, \lambda)$, where V is a finite set of nodes, $\gamma : V \to V^*$, and $\lambda : V \to \Sigma$. The idea is that for a node $v \in V$, $\gamma(v)$ is the ordered list of v's successor nodes and $\lambda(v)$ is the label of v. The *underlying graph* is the directed graph $G_M = (V, E)$ where $(u, v) \in E$ if and only if v occurs in $\gamma(u)$. The *node size* of M is $\#_N(G) = |V|$ and the *edge size* or simply *size* of M is $|M| = \sum_{v \in V} |\gamma(v)|$. Note that the labeling function λ does not influence the size of M. The motivation for this is that the size of M can be seen as the number of pointers that are necessary in order to store M and that these pointers mainly determine the space consumption for M.

Two ordered Σ-labeled multigraphs $M_1 = (V_1, \gamma_1, \lambda_1)$ and $M_2 = (V_2, \gamma_2, \lambda_2)$ are isomorphic if there exists a bijection $f : V_1 \to V_2$ such that for all $v \in V_1$, $\gamma_2(f(v)) = f(\gamma_1(v))$ and $\lambda_2(f(v)) = \lambda_1(v)$ (here we implicitly extend f to a morphism $f : V_1^* \to V_2^*$). We do not distinguish between isomorphic multigraphs.

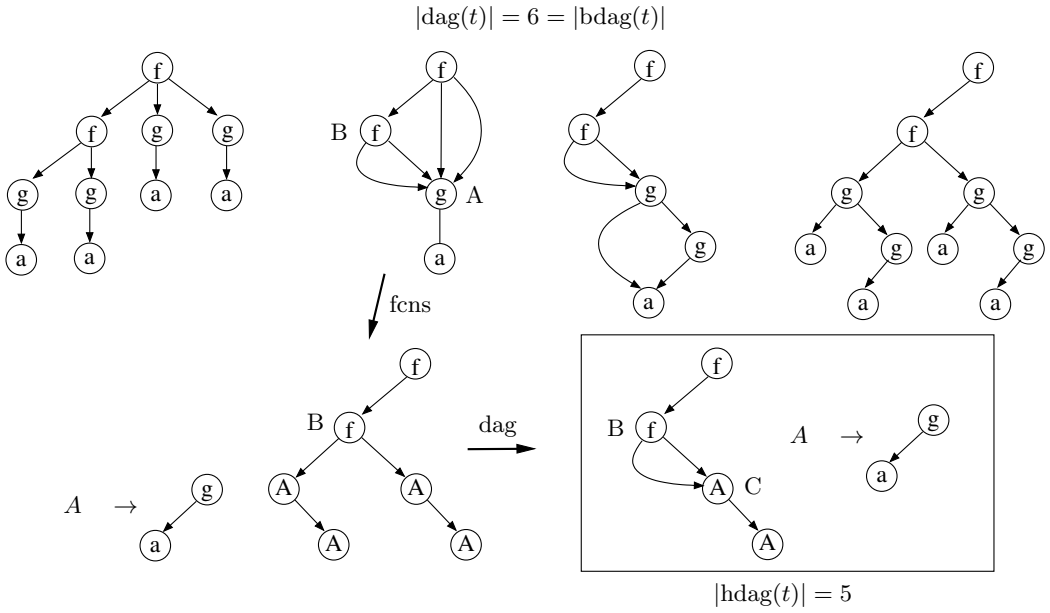

Figure 3: The tree t, its dag, its bdag, and its fcns encoding; its hybrid dag is shown in the box.

A Σ-*labeled ordered dag* is a Σ-labeled ordered multigraph $d = (V, \gamma, \lambda)$ such that the underlying graph G_d is acyclic. The nodes $r \in V$ for which there is no $v \in V$ such that (v, r) is an edge of G_d (r has no incoming edges) are called the *roots* of d. A Σ-labeled ordered *rooted* dag is a Σ-labeled order dag with a unique root. In this case every node of d must be reachable in G_d from the root node. A Σ-*labeled ordered tree* is a Σ-labeled order rooted dag $t = (V, \gamma, \lambda)$ such that every non-root node $v \in V \setminus \{r\}$ has exactly one occurrence in the concatenation of all strings $\gamma(u)$ for $u \in V$. In other words, the underlying graph G_t is a rooted tree in the usual sense and in every string $\gamma(u)$, every $v \in V$ occurs at most once. We define $\mathcal{T}(\Sigma)$ as the set of all Σ-labeled ordered trees. We denote Σ-labeled ordered trees by their usual term notation, i.e., for every $a \in \Sigma$, $n \geq 0$, and all trees $t_1, \ldots, t_n \in \mathcal{T}(\Sigma)$, we also have $a(t_1, \ldots, t_n) \in \mathcal{T}(\Sigma)$. Note that trees from $\mathcal{T}(\Sigma)$ are *unranked* in the sense that the number of children of a node does not depend on the label of the node. We therefore frequently speak of unranked trees for elements of $\mathcal{T}(\Sigma)$.

Let $d = (V, \gamma, \lambda)$ be a Σ-labeled ordered dag. With every node $v \in V$ we associate a tree $\mathrm{eval}_d(v) \in \mathcal{T}(\Sigma)$ inductively as follows: $\mathrm{eval}_d(v) = f(\mathrm{eval}_d(v_1), \ldots, \mathrm{eval}_d(v_n))$, if $\lambda(v) = f$ and $\gamma(v) = v_1 \cdots v_n$ (where $f(\varepsilon) = f$). Intuitively, $\mathrm{eval}_d(v)$ is the tree obtained by unfolding d starting in the node v. If d is a Σ-labeled ordered rooted dag, then we define $\mathrm{eval}(d) = \mathrm{eval}_d(r)$, where r is the root node of d. Note that if t is a Σ-labeled ordered tree and v is a node of t, then $\mathrm{eval}_t(v)$ is simply the subtree of t rooted at v and we write $t/v = \mathrm{eval}_t(v)$ in this case. If for nodes $u \neq v$ of t we have $t/u = t/v$ then the tree $t/u = t/v$ is a *repeated subtree* of t.

Let $t = (V, \gamma, \lambda) \in \mathcal{T}(\Sigma)$ and let $G_t = (V, E)$ be the underlying graph (which is a tree). For an edge $(u, v) \in E$, v is a *child* of u, and u is the *parent* of v. If two nodes v and v' have the same parent node u, then v and v' are *siblings*. If moreover $\gamma(u)$ is of the form $u_1 v v' u_2$ for $u_1, u_2 \in V^*$ then v' is the *next sibling* of v, and v is the *previous sibling* of v'. If a node v does not have a previous sibling, it is a *first child*, and if it does not have a next sibling, it is a *last child*.

For many tree-processing formalisms (e.g. standard tree automata), it is useful to deal with ranked trees, where the number of children of a node is bounded. There is a standard binary encoding of unranked trees, which we introduce next. A *binary* Σ-*labeled dag* d, or short *binary dag*, can be defined as a $(\Sigma \cup \{\Box\})$-labeled ordered dag $d = (V, \gamma, \lambda)$, where $\Box \notin \Sigma$ is a special dummy symbol such that the following holds: (i) For every $v \in V$ with $\lambda(v) \in \Sigma$ we have $|\gamma(v)| = 2$ and (ii) for every $v \in V$ with $\lambda(v) = \Box$ we have $|\gamma(v)| = 0$. A *binary* Σ-*labeled tree* t, or short *binary tree*, is a binary dag which is moreover a $(\Sigma \cup \{\Box\})$-labeled ordered tree. The symbol $\Box$ basically denotes the absence of a left or right child of a node. For instance, $f(\Box, g)$ denotes the binary tree that has an f-labeled root with a g-labeled right child but no left child. Note that $f(\Box, g)$ and $f(g, \Box)$ are different binary trees. Instead of introducing the dummy symbol $\Box$ one may introduce four copies $a_{i,j}$ $(i, j \in \{0, 1\})$ for every label $a \in \Sigma$, where $i = 0$ (resp., $j = 0$) means that the node does not have a left (resp., right) child. For a binary dag $d = (V, \gamma, \lambda)$, we define its node size $\#_N(d) = |\{v \in V \mid \lambda(v) \neq \Box\}|$ and size $|d| = \sum_{v \in V} |\gamma(v)|_\Sigma$, where $|v_1 v_2 \cdots v_m|_\Sigma = |\{i \mid 1 \leq i \leq m, \lambda(v_i) \neq \Box\}|$. In other words: In the definitions of $\#_N(d)$ and $|d|$ we disregard all dummy nodes. The reason for this is that we do not need dummy nodes if we introduce 4 copies of each label as explained above. Let $\mathcal{B}(\Sigma)$ denote the set of binary Σ-labeled trees.

We define a mapping $\mathrm{fcns} : \mathcal{T}(\Sigma)^* \to \mathcal{B}(\Sigma)$, where as usual $\mathcal{T}(\Sigma)^*$ denotes the set of all finite words (or sequences) over the set $\mathcal{T}(\Sigma)$, as follows ("fcns" refers to "first-child-next-sibling"): For the empty word ε let $\mathrm{fcns}(\varepsilon) = \Box$ (the empty binary tree). If $n \geq 1$, $t_1, \ldots, t_n \in \mathcal{T}(\Sigma)$ and $t_1 = f(u_1, \ldots, u_m)$ with $m \geq 0$, then $\mathrm{fcns}(t_1 t_2 \cdots t_n) = f(\mathrm{fcns}(u_1 \cdots u_m), \mathrm{fcns}(t_2 \cdots t_n))$. Note that $\mathrm{fcns}(a) = a(\Box, \Box)$ for $a \in \Sigma$. In the following we always simply write a for $a(\Box, \Box)$. The encoding fcns is bijective, hence the bijection fcns^{-1} :

$\mathcal{B}(\Sigma) \to \mathcal{T}(\Sigma)^*$ is defined. Moreover, for every $t \in \mathcal{T}(\Sigma)$, $\#_N(\text{fcns}(t)) = \#_N(t)$ and $|\text{fcns}(t)| = |t|$, see e.g. [9]. The fcns-encoding is also known as Rabin encoding.

EXAMPLE 1. *Let $t_1 = f(a_1, a_2, a_3)$ and let $t_2 = g(b_1, b_2)$. Then fcns$(t_1 t_2) = f(a_1(\Box, a_2(\Box, a_3)), g(b_1(\Box, b_2), \Box))$.*

One can construct fcns(t) also by keeping all nodes of t and redirecting edges as follows: For each node u of t, the left child of u is the first child of u in t (if it exists) and the right child of u is the next sibling of u (if it exists).

An ordered tree can be compacted by representing occurrences of repeated subtrees only once. Several edges then point to the same subtree (which we call a *repeated* subtree), thus making the tree an ordered dag. It is known that the minimal dag of a tree is unique and that it can be constructed in linear time (see e.g., [6]). For later purpose it is actually useful to define the minimal dag dag(d) for every Σ-labeled ordered dag $d = (V, \gamma, \lambda)$. It can be formally defined as $\text{dag}(d) = (\{\text{eval}_d(u) \mid u \in V\}, \gamma', \lambda')$ with $\lambda'(f(t_1, \ldots, t_n)) = f$ and $\gamma'(f(t_1, \ldots, t_n)) = t_1 \cdots t_n$. Thus, the nodes of dag$(d)$ are the different trees represented by the nodes of d. This is just a convenient way to formally define dag(d). The internal structure of the nodes of dag(d) (trees in our definition) has no influence on the size of dag(d), which is still the number of its edges. Note that in general we cannot recover d from dag(d): If for instance d is the disjoint union of two copies of the same tree t, then dag$(d) =$ dag(t), but this will not be a problem. Dags are only used for the compression of forests consisting of different trees. Such a forest can be reconstructed from its minimal dag. Note also that if d is a rooted dag, then dag(d) is also rooted and we have eval$(d) =$ eval(dag(d)).

EXAMPLE 2. *Consider the full binary tree t_n of depth n with all nodes labeled a. While $|t_n| = 2^{n+1} - 1$, $|dag(t_n)| = 2n$. Hence dag(t) can be exponentially smaller than t.*

The *binary dag* of $t \in \mathcal{T}(\Sigma)$, denoted bdag$(t)$, is defined as bdag$(t) =$ dag(fcns(t)); it is a binary dag as defined above. See Figure 2 in the introduction for an example.

Clearly, the number of nodes of dag(t) equals the number of different subtrees t/v for a node v of t. In order to count the number of nodes of bdag(t) the following definition is useful: For a node v of an unranked tree $t = (V, \gamma, \lambda)$ define sibseq$(v) \in \mathcal{T}(\Sigma)^*$ (the *sibling sequence* of u) as follows: If v is the root of t then sibseq$(v) = t$. Otherwise, let u be the parent node of v and let $\gamma(u) = wv_0v_1 \cdots v_m$, where $w \in V^*$ and $v_0 = v$. Then

$$\text{sibseq}(v) = (t/v_0)(t/v_1) \cdots (t/v_m).$$

EXAMPLE 3. *The different sibling sequences of the tree $t = f(a, f(b, a), b, a)$ are: t, $af(b, a)ba$, $f(b, a)ba$, ba, and a.*

The following lemma follows directly from the definition:

LEMMA 1. *The number of nodes of bdag(t) is equal to the number of different sibling sequences sibseq(v), for all $v \in V$.*

3. THE HYBRID DAG

While the dag shares repeated subtrees of a tree, the binary dag shares repeated sibling sequences (see Lemma 1). Consider an unranked tree t. As we have seen in the introduction, the size of dag(t) can be smaller than the size of bdag(t). On the other hand, it can also be that the size of bdag(t) is smaller than the size of dag(t). We now wish to define an object that combines both types of sharing (trees and tree sequences) and which is guaranteed to be smaller than or equal to the minimum of the sizes of dag(t) and bdag(t). Our starting point is dag(t). Basically, the reason for this is that if the same sibling sequence occurs at two nodes u and v of t, then also $t/u = t/v$ (but not the other way round). In this dag we now want to share all repeated sibling sequences. As an example, consider the tree $t = f(f(g(a), g(a)), g(a), g(a))$ shown on the top left of Figure 3. Its size is 9. The dag of this tree consists of a unique occurrence of the subtree $g(a)$ plus two f-labeled nodes, shown to the right of t in the figure. Thus $|\text{dag}(t)| = 6$. We represent this dag by the following regular straight-line tree grammar $\mathcal{G}$ with three productions (straight-line grammars are *acyclic* and for each nonterminal X there is exactly one production with left-hand side X):

$$\begin{array}{lcl} S & \to & f(B, A, A) \\ B & \to & f(A, A) \\ A & \to & g(a) \end{array} \tag{1}$$

Each non-leaf node of the dag becomes a nonterminal of the tree grammar. Thereby, the right-hand side of each grammar rule becomes a tree of height 1 (we exclude trees consisting of a single node in the following). Note that it is well-known that minimal dags are isomorphic to minimal acyclic tree automata and to minimal straight-line regular tree grammars (see, e.g., [17]). By eval$(\mathcal{G})$ we denote the unique tree $t \in \mathcal{T}(\Sigma)$ obtained from the start nonterminal S by applying $\mathcal{G}$'s productions. In order to share repeated sibling sequences in dag(t) we apply the fcns-encoding to the right-hand sides of the grammar's productions. For the above example we obtain the following new binary tree grammar:

$$\begin{array}{lcl} S & \to & f(B(\Box, A(\Box, A)), \Box) \\ B & \to & f(A(\Box, A), \Box) \\ A & \to & g(a, \Box) \end{array}$$

We now build the minimal dag of the dag obtained by taking the disjoint union of all right-hand sides of this grammar. In the example, the subtree $A(\Box, A)$ appears twice and is shared. We write the resulting dag again as a grammar, using the new nonterminal C for the new repeated tree $A(\Box, A)$ (corresponding to the repeated sibling sequence AA in (1)).

$$\begin{array}{lcl} S & \to & f(B(\Box, C), \Box) \\ B & \to & f(C, \Box) \\ C & \to & A(\Box, A) \\ A & \to & g(a, \Box) \end{array} \tag{2}$$

This grammar is the *hybrid dag* (*hdag* for short) of the initial tree. Its size is the total number of edges in all right-hand side trees; it is 5 in our example (here, as usual, we do not count edges to $\Box$-labeled nodes). Compare this to dag(t) and bdag(t), which are both of size 6.

In our example, the production $B \to f(A, A)$ in the grammar (1) does not save any edges, since the nonterminal B occurs only once in a right-hand side (namely $f(B, A, A)$). Eliminating this production yields the grammar

$$\begin{array}{lcl} S & \to & f(f(A, A), A, A) \\ A & \to & g(a) \end{array}$$

with the fcns encoding

$$\begin{array}{lcl} S & \to & f(f(A(\Box, A), A(\Box, A)), \Box) \\ A & \to & g(a, \Box). \end{array}$$

Sharing repeated subtrees in this grammar gives

$$\begin{array}{lcl} S & \to & f(f(C, C), \Box) \\ C & \to & A(\Box, A) \\ A & \to & g(a, \Box). \end{array} \quad (3)$$

The size of this grammar (number of edges to non-$\Box$ nodes in all right-hand sides) is still 5, but it has only 3 nonterminals in contrast to 4 for the above hdag. In practice, having a grammar with fewer nonterminals is preferable. In fact, our implementation avoids redundant nonterminals like B in our example. On the other hand, having only trees of height 1 as right-hand sides of the dag (seen as a grammar) does not influence the number of edges in the final grammar. Moreover, it slightly simplifies the proofs in the next section, where we show that the size of the hdag of a tree t is smaller than or equal to the minimum of the sizes of $\mathrm{dag}(t)$ and $\mathrm{bdag}(t)$.

In general, the hybrid dag is produced by first building a dag, then constructing its fcns-encoding (using grammars), and then building a dag again. More formally, consider $\mathrm{dag}(t)$ as a straight-line regular tree grammar with productions $A_1 \to t_1, \ldots, A_n \to t_n$, where $A_1, \ldots, A_n$ are nonterminals that may appear in the right-hand sides as leaf labels and every tree t_i has height 1. Then

$$\mathrm{hdag}(t) = \mathrm{dag}(\mathrm{fcns}(t_1), \ldots, \mathrm{fcns}(t_n)),$$

where the tuple $(\mathrm{fcns}(t_1), \ldots, \mathrm{fcns}(t_n))$ is viewed as the dag obtained by taking the disjoint union of the binary trees $\mathrm{fcns}(t_i)$. Clearly $\mathrm{hdag}(t)$ is unique up to isomorphism. In the second step when $\mathrm{dag}(\mathrm{fcns}(t_1), \ldots, \mathrm{fcns}(t_n))$ is constructed from the tuple $(\mathrm{fcns}(t_1), \ldots, \mathrm{fcns}(t_n))$, only sequences of leaf nodes from $t_1, \ldots, t_n$ can be shared, since these trees are pairwise different and of height 1.

We defined the hdag as a particular dag. Alternatively (and as in our example), we can define the hdag as a particular tree grammar which has a set of nonterminals that is partitioned into two sets. Recall that a straight-line grammar is acyclic and contains exactly one production per nonterminal. A *hybrid dag grammar (or hdag grammar)* is a straight-line tree grammar $h = (N_1, N_2, \Sigma, S, P)$ such that N_1 and N_2 are disjoint sets of nonterminals (of rank zero), $S \in N_1$, and for every $A \in N_1 \cup N_2$ there is exactly one production $(A \to t) \in P$, where t is a binary tree over $\Sigma \cup N_1 \cup N_2$ such that

- nonterminals of N_2 only appear at leaf positions in t,
- if $A \in N_1$ then $\mathrm{fcns}^{-1}(t) \in \mathcal{T}(\Sigma \cup N_1 \cup N_2)$ and nonterminals from N_1 only appear at leaf positions in $\mathrm{fcns}^{-1}(t)$, and
- if $A \in N_2$ then $\mathrm{fcns}^{-1}(t) \in (\Sigma \cup N_1)^*(\Sigma \cup N_1 \cup N_2)$.

Both grammars (2) and (3) are hdag grammars. For (2) we set $N_1 = \{S, A, B\}$ and $N_2 = \{C\}$ and for (3) we set $N_1 = \{S, A\}$ and $N_2 = \{C\}$. In general, the hdag of a tree can be seen as an hdag grammar. The set N_1 consists of those nonterminals that were introduced in the construction of $\mathrm{dag}(t)$ from t, while N_2 comprises those nonterminals that were introduced in the construction of $\mathrm{dag}(\mathrm{fcns}(t_1), \ldots, \mathrm{fcns}(t_n))$ from $\mathrm{dag}(t)$. The *size* of h is defined as $|h| = \sum_{A \to t \in P} |t|$.

Note that an hdag grammar is *not* a regular tree grammar because nodes labeled with nonterminals from N_1 may have right children. Also note that the root node of the right-hand side of a nonterminal in N_1 does not have a right child.

An hdag grammar h can be decompressed into a tree $\mathrm{eval}(h)$ as follows. We construct the regular tree grammar $\mathsf{RT}(h)$: For every production $A \to t$ of h with $A \in N_1$ we add to $\mathsf{RT}(h)$ the new production $A \to \mathrm{fcns}^{-1}(t')$, where t' is obtained from t by applying N_2-productions as long as possible. Then, $\mathsf{RT}(h)$ is an ordinary regular tree grammar. Finally, $\mathsf{eval}(h) = \mathrm{eval}(\mathsf{RT}(h))$. One can show that $\mathrm{hdag}(t)$ (seen as a grammar) is a size-minimal hdag grammar with $\mathrm{eval}(\mathrm{hdag}(t)) = t$.

3.1 The sizes of dag, bdag, and hdag

We want to compare the node size and the edge size of $\mathrm{dag}(t)$, $\mathrm{bdag}(t)$, and $\mathrm{hdag}(t)$ for an unranked tree t.

3.1.1 The number of nodes

In this section we consider the number of *nodes* in the dag and bdag of an unranked tree. We show that $\#_N(\mathrm{dag}(t)) \leq \#_N(\mathrm{bdag}(t))$.

Example 4. *Consider the tree $t_n = f(a, a, \ldots, a)$ consisting of n nodes, where $n \geq 2$. Then $\#_N(dag(t)) = 2$ and $\#_N(bdag(t)) = n$, while $|dag(t)| = |bdag(t)| = n - 1$. Note that dags with multiplicities on edges, as defined in [4], can store a tree such as t_n in size $O(\log n)$.*

Lemma 2. *Let t be an unranked tree. Then*

$$\#_N(dag(t)) \leq \#_N(bdag(t)).$$

Proof. The lemma follows from Lemma 1 and the obvious fact that the number of different subtrees of t (i.e., $|\mathrm{dag}(t)|$) is at most the number of different sibling sequences in t: $\mathrm{sibseq}(u) = \mathrm{sibseq}(v)$ implies $t/u = t/v$. □

Lemma 3. *There exists a family of trees $(t_n)_{n \geq 2}$ such that $\#_N(t_n) = \#_N(bdag(t)) = n$ and $\#_N(dag(t)) = 2$.*

Proof. Take the family of trees t_n from Example 4. □

Let us remark that the node size of the hdag can be larger than the node size of the bdag and the node size of the dag. When writing $\mathrm{dag}(t)$ as a grammar, the number of nodes in all right-hand sides can be larger than the node size of $\mathrm{dag}(t)$. On the other hand, when writing $\mathrm{dag}(t)$ as a grammar, the number of edges in all right-hand sides is exactly the number of edges of $\mathrm{dag}(t)$.

3.1.2 The number of edges

Recall from the previous section that the number of nodes of the (minimal) dag is always at most the number of nodes of the bdag, and that the gap can be maximal ($O(1)$ versus $|t|$). For the number of edges, the situation is different. We show that $\frac{1}{2}|\mathrm{bdag}(t)| \leq |\mathrm{dag}(t)| \leq |\mathrm{bdag}(t)|^2$ and that these bounds are sharp. In fact, we show the three inequalities $|\mathrm{hdag}(t)| \leq \min(|\mathrm{dag}(t)|, |\mathrm{bdag}(t)|)$, $|\mathrm{bdag}(t)| \leq 2|\mathrm{hdag}(t)|$, and $|\mathrm{dag}(t)| \leq |\mathrm{hdag}(t)|^2$ which imply

$$\frac{1}{2}|\mathrm{bdag}(t)| \leq |\mathrm{dag}(t)| \leq |\mathrm{bdag}(t)|^2.$$

Before we prove these bounds we need some definitions. Recall that the nodes of $\mathrm{bdag}(t)$ are in 1-1-correspondence with the different sibling sequences of t. In the following, let

$$\mathrm{sib}(t) = \{\mathrm{sibseq}(v) \mid v \text{ a node of } t\}$$

be the set of all sibling sequences of t. To count the size (number of edges) of $\mathrm{bdag}(t)$ we have to count for each sibling sequence $w \in \mathrm{sib}(t)$ the number of outgoing edges in $\mathrm{bdag}(t)$. We denote this number with $e(w)$; it can be computed as follows, where $w = s_1 s_2 \cdots s_m$ ($m \geq 1$) and the s_i are trees:

- $e(w) = 0$, if $m = 1$ and $|s_1| = 0$,
- $e(w) = 1$, if either $m = 1$ and $|s_1| \geq 1$ (then w has only a left child) or $m \geq 2$ and $|s_1| = 0$ (then w has only a right child),
- $e(w) = 2$, otherwise.

With this definition we obtain:

LEMMA 4. *For every $t \in \mathcal{T}(\Sigma)$, we have*

$$|bdag(t)| = \sum_{w \in sib(t)} e(w).$$

The size of the hdag can be computed similarly: In the following, we always view $\mathrm{dag}(t)$ as a regular tree grammar. Let N be the set of nonterminals of this grammar and let S be the start nonterminal. Recall that every right-hand side of $\mathrm{dag}(t)$ has the form $f(\alpha_1, \ldots, \alpha_n)$, where every α_i belongs to $\Sigma \cup N$. Let $\mathrm{sib}(\mathrm{dag}(t))$ be the set of all sibling sequences that occur in the right-hand sides of $\mathrm{dag}(t)$. Thus, for every right-hand side $f(\alpha_1, \ldots, \alpha_n)$ of $\mathrm{dag}(t)$, the sibling sequences $f(\alpha_1, \ldots, \alpha_n)$ (a sibling sequence of length 1) and $\alpha_i \alpha_{i+1} \cdots \alpha_n$ ($1 \leq i \leq n$) belong to $\mathrm{sib}(\mathrm{dag}(t))$. For such a sibling sequence w we define $e(w)$ as above. Then we have:

LEMMA 5. *For every $t \in \mathcal{T}(\Sigma)$, we have*

$$|hdag(t)| = \sum_{w \in sib(dag(t))} e(w).$$

For $w = s_1 \cdots s_m \in \mathrm{sib}(t)$ let $\tilde{w}$ the string that results from w by replacing every non-singleton tree $s_i \notin \Sigma$ by the unique nonterminal of $\mathrm{dag}(t)$ that derives to s_i. Actually, we should write $\tilde{w}_t$ instead of $\tilde{w}$, since the latter also depends on the tree t. But the tree t will be always clear from the context. Here are a few simple statements:

- For every $w \in \mathrm{sib}(t)$, the sibling sequence $\tilde{w}$ belongs to $\mathrm{sib}(\mathrm{dag}(t))$, except for the length-1 sequence $\tilde{w} = S$ that is obtained from the length-1 sequence $w = t \in \mathrm{sib}(t)$.
- For every $w \in \mathrm{sib}(t)$, $e(\tilde{w})$ is a word over $N \cup \Sigma$.
- For every $w \in \mathrm{sib}(t)$, $e(\tilde{w}) \leq e(w)$.
- The mapping $w \mapsto \tilde{w}$ is an injective mapping from $\mathrm{sib}(t) \setminus \{t\}$ to $\mathrm{sib}(\mathrm{dag}(t))$.

Using this mapping, the sums in Lemma 4 and 5 can be related as follows:

LEMMA 6. *For every $t \in \mathcal{T}(\Sigma)$, we have*

$$hdag(t) = \sum_{w \in sib(dag(t))} e(w) = |N| + \sum_{w \in sib(t)} e(\tilde{w}).$$

PROOF. By Lemma 5 it remains to show the second equality. The only sibling sequences in $\mathrm{sib}(\mathrm{dag}(t))$ that are not of the form $\tilde{w}$ for $w \in \mathrm{sib}(t)$ are the sequences (of length 1) that consist of the whole right-hand side $f(\alpha_1, \ldots, \alpha_m)$ of a nonterminal $A \in N$. For such a sibling sequence u we have $e(u) = 1$ (since it has length 1 and $f(\alpha_1, \ldots, \alpha_m)$ is not a single symbol). Hence, we have

$$\begin{aligned} \sum_{w \in \mathrm{sib}(\mathrm{dag}(t))} e(w) &= |N| + \sum_{w \in \mathrm{sib}(t) \setminus \{t\}} e(\tilde{w}) \\ &= |N| + \sum_{w \in \mathrm{sib}(t)} e(\tilde{w}), \end{aligned}$$

where the last equality follows from $e(\tilde{t}) = e(S) = 0$. □

THEOREM 1. *For every $t \in \mathcal{T}(\Sigma)$, we have*

$$|hdag(t)| \leq \min(|dag(t)|, |bdag(t)|).$$

PROOF. Since $\mathrm{hdag}(t)$ is obtained from $\mathrm{dag}(t)$ by sharing repeated sibling sequences, we immediately get $|\mathrm{hdag}(t)| \leq |\mathrm{dag}(t)|$. It remains to show $|\mathrm{hdag}(t)| \leq |\mathrm{bdag}(t)|$. By Lemma 4 and 6 we have to show

$$|N| + \sum_{w \in \mathrm{sib}(t)} e(\tilde{w}) \leq \sum_{w \in \mathrm{sib}(t)} e(w),$$

where N is the set of nonterminals of $\mathrm{dag}(t)$. To see this, note that:

- $e(\tilde{w}) \leq e(w)$ for all $w \in \mathrm{sib}(t)$, and
- for every nonterminal $A \in N$ there must exist a sibling sequence $w \in \mathrm{sib}(t)$ such that $\tilde{w}$ starts with A. For this sequence we have $e(w) = e(\tilde{w}) + 1$ (note that the right-hand side of A does not belong to Σ, hence w starts with a tree of size at least 1).

Choose for every $A \in N$ a sibling sequence $w_A \in \mathrm{sib}(t)$ such that $\tilde{w}_A$ starts with A. Let $R = \mathrm{sib}(t) \setminus \{w_A \mid A \in N\}$. We get

$$\begin{aligned} |N| + \sum_{w \in \mathrm{sib}(t)} e(\tilde{w}) &= |N| + \sum_{A \in N} e(\tilde{w}_A) + \sum_{w \in R} e(\tilde{w}) \\ &= \sum_{A \in N} (e(\tilde{w}_A) + 1) + \sum_{w \in R} e(\tilde{w}) \\ &\leq \sum_{A \in N} e(w_A) + \sum_{w \in R} e(w) \\ &= \sum_{w \in \mathrm{sib}(t)} e(w). \end{aligned}$$

This proves the theorem. □

THEOREM 2. *For every $t \in \mathcal{T}(\Sigma)$, we have*

$$|dag(t)| \leq |hdag(t)|^2.$$

PROOF. Let $f_i(\alpha_{i,1}, \ldots, \alpha_{i,n_i})$ for $1 \leq i \leq k$ be the right hand sides of $\mathrm{dag}(t)$ viewed as a grammar. W.l.o.g. assume that $1 \leq n_1 \leq n_2 \leq \cdots \leq n_k$. Every $\alpha_{i,j}$ is either from Σ or is a nonterminal. Moreover, all the trees $f_i(\alpha_{i,1}, \ldots, \alpha_{i,n_i})$ are pairwise different. We have $|\mathrm{dag}(t)| = \sum_{i=1}^{k} n_i$.

Recall that we compute $\mathrm{hdag}(t)$ by taking the minimal dag of the forest consisting of the binary encodings of the trees $f_i(\alpha_{i,1}, \ldots, \alpha_{i,n_i})$. The binary encoding of $f_i(\alpha_{i,1}, \ldots, \alpha_{i,n_i})$ has the form $f_i(t_i, \square)$, where t_i is a chain of $n_i - 1$ many right pointers. Let d be the minimal dag of the forest consisting

of all chains t_i. Since all the trees $f_i(\alpha_{i,1},\dots,\alpha_{i,n_i})$ are pairwise distinct, we have $|\mathrm{hdag}(t)| = k + |d|$. Since chain t_i consists of n_i many nodes, we have $|d| \geq \max\{n_i \mid 1 \leq i \leq k\} - 1 = n_k - 1$. Hence, we have to show that $\sum_{i=1}^{k} n_i \leq (k + n_k - 1)^2$. We have

$$\sum_{i=1}^{k} n_i \leq k \cdot n_k \leq (k-1)n_k + n_k^2 \leq (k - 1 + n_k)^2,$$

which concludes the proof. □

Consider the tree s_n from Figure 2. We have $|\mathrm{dag}(s_n)| = |s_n| = n^2$ and $|\mathrm{hdag}(s_n)| = |\mathrm{bdag}(s_n)| = 3n - 2$. This shows that up to a constant factor, the bound in Theorem 2 is sharp.

Next let us bound $|\mathrm{bdag}(t)|$ in terms of $|\mathrm{hdag}(t)|$:

THEOREM 3. *For every $t \in \mathcal{T}(\Sigma)$, we have*

$$|bdag(t)| + n \leq 2|hdag(t)|,$$

where n is the number of non-leaf nodes of $dag(t)$.

PROOF. We use the notations introduced before Theorem 1. Note that $n = |N|$ is the number of non-terminals of the regular grammar corresponding to $\mathrm{dag}(t)$. By Lemma 4 we have $|\mathrm{bdag}(t)| = \sum_{w \in \mathrm{sib}(t)} e(w)$. By Lemma 6 we have $|\mathrm{hdag}(t)| = |N| + \sum_{w \in \mathrm{sib}(t)} e(\tilde{w})$. Hence, we have to show that

$$|N| + \sum_{w \in \mathrm{sib}(t)} e(\tilde{w}) \;\geq\; \frac{1}{2} \sum_{w \in \mathrm{sib}(t)} e(w) \;+\; \frac{1}{2}|N|.$$

In order to prove this, we show the following for every sibling sequence $w \in \mathrm{sib}(t)$: Either $e(\tilde{w}) \geq \frac{1}{2}e(w)$ or $e(\tilde{w}) = 0$, $e(w) = 1$. In the latter case, the sibling sequence w consists of a single tree s of size at least one (i.e., s does not consist of a single node), and $\tilde{w}$ consists of a single nonterminal $A \in N$. So, let $w = t_1 \cdots t_n \in \mathrm{sib}(t)$ and let $\tilde{w} = \alpha_1 \cdots \alpha_n$ with $\alpha_i \in \Sigma \cup N$. We consider the following four cases:

Case 1. $n > 1$ and $t_1 = \alpha_1 \in \Sigma$. We have $e(w) = e(\tilde{w}) = 1$.

Case 2. $n > 1$ and $|t_1| \geq 1$. We have $e(w) = 2$ and $e(\tilde{w}) = 1$.

Case 3. $n = 1$ and $t_1 = \alpha_1 \in \Sigma$. We have $e(w) = e(\tilde{w}) = 0$.

Case 4. $n = 1$ and $|t_1| \geq 1$. We have $e(w) = 1$, $e(\tilde{w}) = 0$, and $\tilde{w}$ consists of a single nonterminal $A \in N$. □

For the tree t_n from Figure 1 we have $|\mathrm{bdag}(t_n)| = |t_n| = 2n$, $|\mathrm{hdag}(s_n)| = |\mathrm{dag}(t_n)| = n + 1$, and $n = |N| = 2$. Hence, Theorem 3 is optimal.

From Theorems 1, 2, and 3 we immediately get:

COROLLARY 1. *For every $t \in \mathcal{T}(\Sigma)$, we have*

$$\frac{1}{2}|bdag(t)| \leq |dag(t)| \leq |bdag(t)|^2.$$

3.2 Hybrid dag to SLT grammar

The definition of a hdag grammar suggests that unfolding it into the original tree needs to be done in two phases. We now show that this is not the case: every hdag grammar can be transformed (with little space overhead) into a "one-parameter straight-line linear context-free tree grammar" (for short, 1-SLT grammar), see [16]. During a bottom-up pass, a 1-SLT grammar can be unfolded in time linear in the size of input grammar plus the size of the output tree. There are several other advantages of 1-SLT grammars. For instance, tree automata can be executed in polynomial time over 1-SLT grammars [14]. It was shown in [16] that every SLT grammar can be transformed in polynomial time into a 1-SLT grammar. Consider the hdag grammar

$$\begin{array}{lcl} S & \to & f(B(\Box, C), \Box) \\ B & \to & f(C, \Box) \\ C & \to & A(\Box, A) \\ A & \to & g(a(\Box, b), \Box) \end{array}$$

It is derived from the tree t in Figure 3, where every g-labeled node has a second b-labeled child. In the right-hand side $A(\Box, A)$, the nonterminal $A \in N_1$ appears at a leaf and at a unary node. In an SLT grammar each nonterminal has a fixed rank, which is either 0 or 1 for a 1-SLT grammar. An occurrence of a nonterminal of rank r in a right-hand side has exactly r children. Thus, we need to introduce a copy $\hat{A}$ of rank 1 of the nonterminal A of the hdag grammar (the original A will have rank 0 in the 1-SLT grammar). In a 1-SLT grammar, the production for a rank-1 nonterminal X is of the form $X(y) \to t$, where t is a tree over nonterminals and terminals which contains exactly one occurrence of y at a leaf. When this production is applied inside a tree, then we replace every subtree $X(t')$ by t in which the tree t' is substituted for the unique occurrence of y. In the hdag, this substitution point of $X \in N_1$ is fixed as the right child of the right-hand side's root node. In our example, we introduce for A the copy $\hat{A}$ with the production

$$\hat{A}(y) \to g(a(\Box, b), y)$$

(we keep the original production $A \to g(a(\Box, b), \Box)$ of the hdag grammar). Clearly, this process may double the size of the hdag when transforming it into a 1-SLT grammar. However, observe that every N_1-production of a hdag grammar has the form $A \to f(\zeta, \Box)$. Thus, we may introduce a new nonterminal for the tree ζ and represent ζ only once. In our example, we replace the hdag production $A \to g(a(\Box, b), \Box)$ in the 1-SLT grammar by the productions

$$\begin{array}{lcl} \hat{A}(y) & \to & g(Z, y) \\ A & \to & g(Z, \Box) \\ Z & \to & a(\Box, b). \end{array}$$

In this way, only at most two edges are added when we build 1-SLT grammar productions for a nonterminal in N_1. The hdag productions for nonterminals in N_2 can be taken over without change into the 1-SLT grammar; we only have to replace every occurrence of a nonterminal $A \in N_1$ in the right-hand side by $\hat{A}$ if that occurrence has a (right) child. In our example, the production $C \to A(\Box, A)$ is replaced by $C \to \hat{A}(A)$. We obtain the following lemma.

LEMMA 7. *Given a hdag grammar $h = (N_1, N_2, \Sigma, S, P)$, a 1-SLT grammar G_h can be constructed in time $O(|h|)$ such that $eval(G_h) = fcns(eval(h))$ and $|G_h| \leq |h| + 2|N_1|$.*

Lemma 7 implies that results for 1-SLT grammars carry over to hdags. For instance, finite tree automata [14] (with sibling constraints [16]) and tree-walking automata can be evaluated in polynomial time over 1-SLT grammars and hence over hdags.

3.3 Using the reverse encoding

Instead of using the fcns encoding of a tree, one may also use the *last child previous sibling encoding* (lcps). Just

like fcns, lcps is a map from $\mathcal{T}(\Sigma)^*$ to $\mathcal{B}(\Sigma)$ and is defined as follows. For the empty word ε let $\mathrm{lcps}(\varepsilon) = \Box$ (the empty binary tree). If $n \geq 1$, $t_1, \ldots, t_n \in \mathcal{T}(\Sigma)$ and $t_n = f(u_1, \ldots, u_m)$ with $m \geq 0$, then $\mathrm{lcps}(t_1 t_2 \cdots t_n) = f(\mathrm{lcps}(t_1, \ldots, t_{n-1}), \mathrm{lcps}(u_1 \cdots t_m))$.

EXAMPLE 5. *Let* $t_1 = f(a_1, a_2, a_3)$ *and* $t_2 = g(b_1, b_2)$. *Then* $lcps(t_1 t_2) = g(f(\Box, a_3(a_2(a_1, \Box), \Box)), b_2(b_1, \Box))$.

Let $\mathrm{rbdag}(t) = \mathrm{dag}(\mathrm{lcps}(t))$ and

$$\mathrm{rhdag}(t) = \mathrm{dag}(\mathrm{lcps}(t_1), \ldots, \mathrm{lcps}(t_n)),$$

where $\mathrm{dag}(t)$ is seen as the regular tree grammar with productions $A_1 \to t_1, \ldots, A_n \to t_n$. Trivially, in all results from Section 3.1.2 we can replace hdag by rhdag. The reason to consider the lcps encoding is that $\mathrm{rbdag}(t)$ and $\mathrm{rhdag}(t)$ are smaller for trees that have repeated tree *begin* sequences. Empirically, as we show in Section 6.3, this is actually quite common and for most trees t in our corpus $|\mathrm{rbdag}(t)| < |\mathrm{bdag}(t)|$ and $|\mathrm{rhdag}(t)| < |\mathrm{hdag}(t)|$.

Clearly, there are also trees t where $|\mathrm{hdag}(t)| < |\mathrm{rhdag}(t)|$. This raises the question whether there is a scheme which combines the best of both approaches. Obviously one can construct both $\mathrm{hdag}(t)$ and $\mathrm{rhdag}(t)$ of a tree t and discard the larger of both. Yet a scheme which combines both approaches by sharing both suffixes and prefixes of children sequences, faces the problem that the resulting minimal object is not necessarily unique. This can easily be seen by considering trees in which repeated prefixes and suffixes of children sequences overlap. Also it is not clear how a minimal such object can be constructed efficiently. A (non-optimal) approach we have considered was to first share repeated prefixes and then share repeated suffixes. Yet the results in compression achieved were not significantly better than for the rhdag. Moreover, this approach can be further generalized by sharing arbitrary factors of sibling sequences. This is the topic of the next section.

4. DAG AND STRING COMPRESSION

As for the hdag, consider the forest $\mathrm{fcns}(\mathrm{dag}(t))$ of the binary encodings of the right-hand sides of $\mathrm{dag}(t)$ (seen as a grammar) for an unranked tree t. In the construction of the hdag we build the minimal dag of this forest. Therefore we only share repeated suffixes, i.e., "right branching" trees in the binary encoding. Such trees can in fact be considered as *strings*. We now want to generalize the sharing of suffixes. Instead of only sharing suffixes, we now apply an arbitrary grammar-based string compressor to (a concatenation) of these strings. Such a compressor infers a small straight-line context-free grammar (in short, SL grammar) for the given string. For a string grammar G, we define its size $|G|$ as the sum of the lengths of the strings in the right-hand sides of G's productions. As we will see, it is not difficult to transform this SL grammar into a 1-SLT grammar for the fcns-encoding of the initial tree with only little space overhead. This is done as follows: Let $N = \{A_1, \ldots, A_k\}$ be the set of nonterminals of $d = \mathrm{dag}(t)$ and let

$$A_i \to f_i(\alpha_{i,1}, \ldots, \alpha_{i,n_i}) \tag{4}$$

be the production for A_i. Every α_i is from $N \cup \Sigma$ and we have $n_i \geq 1$. For every symbol $\alpha \in N \cup \Sigma$ let $\hat{\alpha}$ be a copy of α. The idea is that $\hat{\alpha}$ represents a copy of α that appears at positions in the fcns encoding having exactly one child (a right child), whereas the original α will only appear in leaf positions. This distinction is necessary since in an SLT-grammar every nonterminal has a fixed rank (see also the discussion in Section 3.2, where we have already used nonterminals $\hat{A}$). Let w_i be the string $w_i = \hat{\alpha}_{i,1} \cdots \hat{\alpha}_{i,n_i-1} \alpha_{i,n_i}$. Note that we also have to rename occurrences of terminals $\alpha \in \Sigma$ that are not the last symbol of $\alpha_{i,1} \cdots \alpha_{i,n_i}$ by $\hat{\alpha}$. Consider for instance the right-hand side $f(a, b, a, b)$ with the fcns encoding $f(a(\Box, b(\Box, a(\Box, b(\Box)))), \Box)$. Then our string compressor should not introduce the same nonterminal for the two occurrences of ab in the string $abab$, since the first occurrence corresponds to the pattern $a(\Box, b(y))$ in the fcns encoding, while the second occurrence is a full subtree. We now apply any grammar-based string compressor (e.g. RePair [11] or Sequitur [21]) to the string $W_d = w_1 \$_1 w_2 \$_2 w_3 \cdots \$_{k-1} w_k$ where the $\$_i$ are distinct new separator symbols that do not appear in any of the w_i. Let G_s denote the SL grammar for W_d produced by the string compressor. We may assume that every nonterminal of G_s, except for the start nonterminal S, appears more than once in a right-hand side of G_s. This implies that the start production of G_s has the form $S \to v_1 \$_1 v_2 \$_2 v_3 \cdots \$_{k-1} v_k$. We now transform G_s into a 1-SLT grammar G follows: Similar to the hdag to SLT transformation from Section 3.2, we put for the dag production (4) the following three productions into the 1-SLT grammar, where v_i' results from v_i by replacing every occurrence of $\hat{\alpha}$ ($\alpha \in \Sigma$) by α:[1]

$$\begin{aligned} A_i &\to f_i(Z, \Box) \\ \hat{A}_i(y) &\to f_i(Z, y) \\ Z &\to \mathrm{fcns}(v_i') \end{aligned}$$

The total number of edges to non-$\Box$ nodes in these three productions is $|v_i| + 2$. For each production $X \to w$ of the string grammar G_s with $X \neq S$ we add to our 1-SLT grammar G the production $X(y) \to \mathrm{fcns}(wy)$ or $X \to \mathrm{fcns}(w)$ depending on whether X produces in G_s a string ending with a symbol $\hat{\alpha}$ or not. Note that the number of edges to non-$\Box$ nodes in the right-hand side of this production is at most $|w|$. For the size our final 1-SLT grammar G we obtain

$$|G| \;\leq\; \sum_{i=1}^{k} (|v_i| + 2) + \sum_{\substack{(B \to w) \in G_s \\ B \neq S}} |w| \;=\; 2|N| + |G_s'|,$$

where G_s' results from G_s by removing all symbols $\$_i$ from the start production.

EXAMPLE 6. *Consider the grammar* $dag(t)$ *consisting of the productions*

$$\begin{aligned} A_1 &\to f(a, A_2, A_3, A_4, A_2, A_3, c) \\ A_2 &\to g(a, a, a) \\ A_3 &\to h(a, a, b) \\ A_4 &\to f(A_2, A_3). \end{aligned}$$

A grammar-based string compressor applied to the string

$$\hat{a}\hat{A}_2\hat{A}_3\hat{A}_4\hat{A}_2\hat{A}_3 c \;\$_1\; \hat{a}\hat{a}a \;\$_2\; \hat{a}\hat{a}b \;\$_3\; \hat{A}_2 A_3$$

may return the SL grammar with start production

$$S' \to \hat{a}B\hat{A}_4 Bc \;\$_1\; Ca \;\$_2\; Cb \;\$_3\; \hat{A}_2 A_3$$

[1]Note that $\mathrm{fcns}(v_i')$ is a chain of right pointers. An occurrence of a G_s-nonterminal in v_i is written in $\mathrm{fcns}(v_i)$ as a unary node, i.e., we omit the left $\Box$-labeled child, see also Example 6.

and the productions $B \to \hat{A}_2\hat{A}_3$ and $C \to \hat{a}\hat{a}$. Our final 1-SLT grammar is:

$$\begin{array}{lcl} A_1 & \to & f(a(\Box, B(\hat{A}_4(B(c)))), \Box) \\ \hat{A}_2(y) & \to & g(C(a), y) \\ A_3 & \to & h(Z, \Box) \\ \hat{A}_3(y) & \to & h(Z, y) \\ Z & \to & C(b) \\ \hat{A}_4(y) & \to & f(\hat{A}_2(A_3), y) \\ B(y) & \to & \hat{A}_2(\hat{A}_3(y)) \\ C(y) & \to & a(\Box, a(\Box, y)). \end{array}$$

Note that our construction may compress $\mathrm{dag}(t)$ exponentially. For instance, for the tree $f(a, a, \ldots, a)$ (for which the dag has the same number of edges) with $2^n + 1$ many a-leaves we apply a grammar compressor to the string $\hat{a} \cdots \hat{a}a$ with 2^n-many $\hat{a}$'s. The string compressor may produce the string grammar

$$\begin{array}{lcll} S' & \to & A_1 A_1 a & \\ A_i & \to & A_{i+1} A_{i+1} & \text{for } 1 \le i \le n-3 \\ A_{n-2} & \to & \hat{a}\hat{a} & \end{array}$$

of size $2n - 1$ (actually, RePair would produce such a grammar). Our final 1-SLT grammar has the start production

$$S \to f(A_1(A_1(a)), \Box)$$

and hence in total the size $2n + 2$. Hence we obtain a 1-SLT grammar for the fcns encoding of $f(a, a, \ldots, a)$ of size $O(n)$.

5. SUBTREE EQUALITY CHECK

In the previous sections we have discussed five different formalisms for the compact representation of unranked trees: (1) dag, (2) binary dag, (3) hybrid dag, (4) combination of dag and SL grammar-based string compression as described in the previous section (we refer to this representation as SL-grammar compressed dag below), and (5) SLT grammars (e.g. produced by BPLEX or TreeRepair). As mentioned in Section 3.2, tree automata can be evaluated in polynomial time for 1-SLT grammars, hence the same holds for the above five formalisms (since the can be translated to 1-SLTs). In this section we consider another important processing primitive: *subtree equality check*. Consider a program which realizes two independent node traversals of an unranked tree, using one of (1)–(5) as memory representation. At a given moment we wish to check if the subtrees at the two nodes of the traversals coincide. How expensive is this check? As it turns out, the formalisms behave quite differently for this task. The dags (1)–(3) as well as SL-grammar compressed days (4) allow efficient equality check. Essentially, for an appropriate representation of the two nodes, this test can be performed in constant time. This is because we merely need to check whether subtrees are generated by the same (tree) nonterminal. For SLT grammars such a check is much more expensive. Note that we cannot unfold the subtrees and check node by node, as this can take exponential time. For SLT grammars a polynomial time algorithm is known, based on Plandowski's result [22]. A new, fine difference between bdags (2) and hdags (3) on one hand and (1), (4) and (5) on the other hand is that we can also check equality of sibling sequences in constant time, using the same argument as above, but now for sequence nonterminals. This is not possible (without other additional data structures) for the other formalisms.

In what follows we identify a pre-order number p with the node in t that it represents, and simply speak of "the node p". Given an unranked tree t and two numbers $1 \le p, q \le \#_N(t)$ we denote by $\mathrm{SubtrEQ}_t(p, q)$ the problem that returns true if the subtrees rooted at nodes p and q of t are equal, and returns false otherwise.

THEOREM 4. *Let t be an unranked tree with N nodes. Given $g = dag(t)$ or an SL-grammar compressed dag g (this includes the hdag) or $g = bdag(t)$, we can, after $O(|g|)$ time preprocessing, answer $SubtrEQ_t(p, q)$ for any $1 \le p, q \le N$ in time $O(\log N)$.*

PROOF. First, consider $g = \mathrm{dag}(t)$. Let $x_1, \ldots, x_N$ be an enumeration of the nodes of t in preorder. Let y_p be the unique node of g such that $\mathrm{eval}_g(y_p)$ is the subtree of t rooted at node x_p. Then it suffices to show that node y_p can be computed from p in time $O(\log N)$ (after $O(|g|)$ time preprocessing). For this, we use techniques from [3]. Let $g = (V, \gamma, \lambda)$. We obtain an SL string grammar G' for the preorder traversal of t by taking V as the set of nonterminals and taking the rule $v \to f v_1 \cdots v_n$ for the node v, where $\gamma(v) = v_1 \cdots v_n$ and $\lambda(v) = f$. The nodes of t correspond to the leaves of the derivation tree T_t of G'. Note that the dag node y_p is simply the label of the penultimate node on the path from the root of T_t to the p-th leaf (from the left to right) of T_t. This label can be computed in time $O(\log N)$ after $O(|g|)$ time preprocessing using the techniques from [3] (for this, G' has to be transformed into a grammar where all right-hand sides have length 2). In [3], a compact representation of the path from the root to the p-th leaf of the derivation tree is computed. This compact representation consists of the sequence of at most $\log N$ many light edges on the path, where "light" refers to the heavy path decomposition of the derivation tree. From this information, the label of the penultimate node on the path can be computed in constant time.

For a string-compressed dag G essentially the same procedure as for the dag applies. The only difference is that the sequence $v_1 \cdots v_n$ of children of the dag-node v is represented by an SL string grammar. But producing an SL string grammar for the preorder traversal of t is again straightforward.

Finally, for $g = \mathrm{bdag}(t) = (V, \gamma, \lambda)$ we can proceed similarly. Again we easily obtain an SL string grammar for the preorder traversal of t. The nodes of the grammar are again the (non-$\Box$ labeled) nodes of g and for every $v \in V$ with $\lambda(v) = f \neq \Box$ and $\gamma(v) = v_1 v_2$ we introduce the production $v \to f v_1' v_2'$, where $v_i' = \varepsilon$ if $\lambda(v_i) = \Box$ and $v_i' = v_i$ otherwise. For a given preorder number $1 \le p \le N$ we can compute in time $O(\log N)$ (as for the dag) the label $y_p \in V$ of the penultimate node on the path from the root of the derivation tree to the p-th leaf. But in contrast to the dag, $\mathrm{SubtrEQ}_t(p, q)$ is not equivalent to $y_p = y_q$. Instead, $\mathrm{SubtrEQ}_t(p, q)$ is equivalent to the following conditions: (i) $\lambda(y_p) = \lambda(y_q)$ and (ii) either y_p and y_q do not have left children, or there left children coincide. Since these checks only require constant time, we obtain the desired time complexity. □

We observe that for (5), i.e., for general SLT grammars, a result such as the one of Theorem 4 is not known. To our knowledge, the fastest known way of checking $\mathrm{SubtrEQ}_t(p, q)$ for a given SLT grammar G for t works as follows: From G we can again easily build an SL string grammar G' for the preorder traversal of t, see, e.g. [5, 18]. Assume that the

File	Edges	mD	aC	mC	dag	bdag	rbdag	hdag	rhdag	DS	TR
1998statistics	28305	5	22.4	50	1377	2403	2360	1292	1243	561	501
catalog-01	225193	7	3.1	2500	8554	6990	10303	4555	6421	4372	3965
catalog-02	2240230	7	3.1	25000	32394	52392	56341	27457	29603	27242	26746
dblp	3332129	5	10.1	328858	454087	677389	681744	358603	362571	149964	156412
dictionary-01	277071	7	4.4	733	58391	77554	75247	47418	46930	32139	22375
dictionary-02	2731763	7	4.4	7333	545286	681130	653982	414356	409335	267944	167927
EnWikiNew	404651	4	3.9	34974	35075	70038	70016	35074	35055	9249	9632
EnWikiQuote	262954	4	3.7	23815	23904	47710	47690	23903	23888	6328	6608
EnWikiVersity	495838	4	3.8	43593	43693	87276	87255	43691	43676	7055	7455
EnWikTionary	8385133	4	3.8	726091	726221	1452298	1452270	726219	726195	81781	84107
EXI-Array	226521	8	2.3	32448	95584	128009	128011	95563	95563	905	1000
EXI-factbook	55452	4	6.8	265	4477	5081	2928	3847	2355	1808	1392
EXI-Invoice	15074	6	3.7	1002	1073	2071	2068	1072	1069	96	108
EXI-Telecomp	177633	6	3.6	9865	9933	19808	19807	9933	9932	110	140
EXI-weblog	93434	2	11.0	8494	8504	16997	16997	8504	8504	44	58
JSTgene.chr1	216400	6	4.8	6852	9176	14606	14114	7901	7271	3943	4208
JSTsnp.chr1	655945	7	4.6	18189	23520	40663	37810	22684	19532	9809	10327
medline	2866079	6	2.9	30000	653604	740630	381295	466108	257138	177638	123817
NCBIgene.chr1	360349	6	4.8	3444	16038	14356	10816	11466	7148	6283	5166
NCBIsnp.chr1	3642224	3	9.0	404692	404704	809394	809394	404704	404704	61	83
sprot39.dat	10903567	5	4.8	86593	1751929	1437445	1579305	1000376	908761	335756	262964
SwissProt	2977030	4	6.7	50000	1592101	1453608	800706	1304321	682276	278915	247511
treebank	2447726	36	2.3	2596	1315644	1454520	1244853	1250741	1131208	1121566	528372

Table 1: The XML documents in Corpus I, their characteristics, and their compressed sizes.

subtree of t rooted in p (resp., q) consists of m (resp., n) nodes. Then we have to check whether the substring of eval(G') from position p to position $p+m-1$ is equal to the substring from position q to position $q+n-1$. Using Plandowski's result [22], this can be checked in time polynomial in the size of G' and hence in time polynomial in the size of the SLT grammar G. Note that more efficient alternatives than Plandowski's algorithm exist, see, e.g. [13] for a survey, but all of them require at least quadratic time in the size of the SL grammar.

In the context of XML document trees, it is also interesting to check equivalence of two sibling sequences. Given an unranked tree t and two numbers $1 \leq p, q \leq \#_N(t)$ we denote by $\text{SibSeqEQ}_t(p,q)$ the problem that returns true if the sibling sequences at nodes p and q of t are equal, and returns false otherwise. For the binary and hybrid dag this problem can be solved again very efficiently:

THEOREM 5. *Let t be an unranked tree with N nodes. Given $g = bdag(t)$ or $g = hdag(t)$ we can, after $O(|g|)$ time preprocessing, answer $SibSeqEQ_t(p,q)$ for any $1 \leq p, q \leq N$ in time $O(\log N)$.*

PROOF. The proof is similar to the one of Theorem 4. For the bdag, we can use exactly the same algorithm that was used for the dag in the proof of Theorem 4. For the hdag, we compute again an SL string grammar for the preorder traversal of t; the nonterminals of this grammar are the nodes of the hdag. Using [3] we compute the path from the root of the derivation tree to the p-th (resp.q-th) leaf. For $x \in \{p, q\}$ let $A_x \to \alpha_x$ and $B_x \to \beta_x$ be the last two rules that were applied on the path from the root to the x-th leaf, where A_x appears in β_x. Then the sibling sequences for p and q are the same if and only if (i) $A_p = A_q$ (this means that the subtrees rooted in p and q are the same), and (ii) either also $B_p = B_q$ or A_p is the last symbol in β_p as well as in β_q. Once the two paths are computed this can be easily checked in constant time. □

For dag(t) it seems to be more expensive to answer a query $\text{SibSeqEQ}_t(p,q)$. After locating p and q in the dag, we need to compare all right siblings. In the worst case, this takes time $O(|\text{dag}(t)|)$. For an SL-grammar compressed dag, the best solution we are aware of uses again an equality check for SL-grammar compressed strings.

6. EXPERIMENTS

In this section we empirically compare the sizes of different dags of unranked trees, namely dag, bdag, rbdag, hdag, and rhdag. We also include a comparison with SL-grammar compressed dags with RePair [11] as the string compressor, as explained in Section 4, and with TreeRepair [15]. We are interested in the tree structure only, hence we did not compare with XML file compressors like Xmill [12] or XQueC [1].

6.1 Corpora

We use three corpora of XML files for our tests. For each XML document we consider the unranked tree of its element nodes; we ignore all other nodes such as texts, attributes, etc. One corpus (*Corpus I*) consists of XML documents that have been collected from the web, and which have often been used in the context of XML compression research, e.g., in [4, 5, 15]. Each of these files is listed in Table 1 together with the following characteristics: number of edges, maximum depth (mD), average number of children of a node (aC), and maximum number of children of a node (mC). Precise references to the origin of these files can be found in [15]. The second corpus (*Corpus II*) consists of all well-formed XML document trees with more than 10000 edges and a depth of at least four that were in the *University of Amsterdam XML*

Corpus	Edges	mD	aC	mC
I	$1.9 \cdot 10^6$	6.6	5.7	$8 \cdot 10^4$
II	79465	7.9	6.0	2925
III	1531	18	1.5	13.2

Table 2: Document characteristics, average values.

Corpus	Parse	dag	hdag	DS	TR
I	35	43	46	48	175
II	85	105	120	117	310
III	6.9	8.7	9.2	10.0	14.8

Table 3: Cumulative Running times (in seconds).

Web Collection[2]. We decided on fixing a minimum size because there is no necessity to compress documents of very small size, and we chose a minimum depth because our subject is tree compression rather than list compression. Note that out of the over 180,000 documents of the collection, only 1100 fit our criteria and are part of Corpus II (more than 27.000 were ill-formed and more than 140.000 had less than 10.000 edges). The documents in this corpus are somewhat smaller than those in Corpus 1, but otherwise have similar characteristics (such as maximal depth and average number of children) as can be seen in Table 2. The third corpus (*Corpus III*) consists of term rewriting systems[3]. These are stored in XML files, but, are rather atypical XML documents, because their tree structures are trees with small rank, i.e., there are no long sibling sequences. This can be seen in Table 2, which shows that the average number of children is only 1.5 for these files (on average).

6.2 Experimental setup

For the dag, bdag, rbdag, and hdag we built our own implementation. It is written in C++ and uses the Libxml XML parser. It should be mentioned that these are only rough prototypes and that our code is not optimized at all. The running times listed in Table 3 should be understood with this in mind. For the RePair-compressed dag we use Gonzalo Navarro's implementation of RePair[4]. This is called "DS" in our tables. For TreeRePair, called "TR" in the tables, we use Roy Mennicke's implementation[5] and run with max_rank=1, which produces 1-STL grammars. Our test machine features an Intel Core i5 with 2.5Ghz and 4GB of RAM. Our dag implementations are done using g++ version 4.6.3 (with O3-switch) and Libxml 2.6 for XML parsing.

6.3 Comparison

Consider first Corpus 1 and the numbers shown in Table 1. The most interesting file, concerning the effectiveness of the hybrid dag and of the reverse binary encoding, is certainly the medline file. Just like dblp, it contains bibliographic data. In particular, it consists of MedlineCitation elements; such elements have ten children, the last of which varies greatly (it is a MeshHeadingList node with varying children lists) and thus cannot be shared in the dag. This is perfect for the reverse hybrid dag, which first eliminates repeated subtrees, thus shrinking the number of edges to 653,604, and then applies the last child/previous sibling encoding before building a dag again. This last step shrinks the number of edges to impressive 257,138. In contrast, the reverse binary dag has a size of 381,295. Thus, for this document really the combination of both ways of sharing, subtrees and reversed sibling sequences, is essential. We note that in the context of the first attempts to apply dag compression to XML [4] the medline files were particularly pathological cases where dag compression failed. We now have new explanations for this: using *reverse* (last child/previous sibling) encoding slashes the size of the dag by almost one half. And using hybrid dags again brings an improvement of more than 30%. The dblp document is similar, but does not make use of optional elements at the end of long sibling lists. Thus, the reverse dags are not smaller for dblp, but the hybrid dag is indeed more than 20% smaller than the dag. The treebank document, which is a ranked tree and does not contain long lists, gives hardly any improvement of hybrid dag over dag, but the reverse hybrid dag is somewhat smaller than the dag (by 5%). For treebank, TreeRePair is unchallenged and produces a grammar that is less than half the size of DS.

Next, consider the accumulated numbers for the three corpora in Table 4. For Corpus I, the reverse hdag is smaller than the dag by around 38% while the hdag is only around 25% smaller than the dag. As noted in Section 3.3, the somewhat surprising outcome that the reverse binary encoding enables better compression results from the custom that in many XML-documents optional elements are listed last. This means that there are more common sibling prefixes than suffixes, hence the reverse schemes perform better. When we transform hdags into SLT grammars (according to Section 3.2), then we get a modest size increase of about 1–2%. For the term rewriting systems of Corpus III, the hdag improves about 10% over the dag. Represented as grammars, however, this improvement disappears and in fact we obtain an accumulated size that is slightly larger than the dag. Note that for this corpus, also TreeRePair (TR) is not much smaller than the dag. Compared to the dag, TreeRePair shares tree patterns (=connected subgraphs). Hence, the trees in Corpus III do not contain many repeated tree patterns which are not already shared by the dag. When we compare DS with TR, then we see on corpora I and II that TreeRePair grammars are on average around 34% smaller than DS, while on Corpus III it is only 23% smaller. On very flat files, such as the EXI documents in Table 1, DS is about as good as TreeRePair. For combined dag and string compression we also experimented with another grammar-based string compressor: Sequitur [21], but found the combined sizes to be larger than with RePair. Concerning running times (see Table 3) note that the dag-variants stay close to the parsing time, while TreeRePair needs considerably more time. Hence, dags should be used when light-weight compression is preferred.

7. CONCLUSION AND FUTURE WORK

We compare the sizes of five different formalisms for compactly representing unranked trees: (1) dag, (2) binary dag, (3) hybrid dag, (4) combination of dag and SL grammar-based string compression, and (5) SLT grammars (e.g. produced by BPLEX or TreeRepair). For the comparison of (1)–(3) we prove precise bounds: (i) the size of the binary

[2]http://data.politicalmashup.nl/xmlweb/
[3]http://www.termination-portal.org/wiki/TPDB
[4]http://http://www.dcc.uchile.cl/~gnavarro/software/
[5]http://code.google.com/p/treerepair/

Corpus	Input	dag	bdag	rbdag	hdag	G(hdag)	rhdag	G(rhdag)	DS	TR
I	43021	7815	9292	8185	6270	6323	5220	5285	2523	1671
II	90036	13510	15950	14671	10884	11109	9806	10039	5162	3957
III	2095	354	391	390	319	362	320	364	324	310

Table 4: Accumulated sizes (in thousand edges). $G(X)$ stands for the grammar size of X.

dag of a tree is bounded by twice the size of the hybrid dag of the tree and (ii) the size of the unranked dag of a tree is bounded by the square of the size of the hybrid dag of the tree. As a corollary we obtain that the size of the dag is at least of the size of the binary dag, and at most the square of the size of the binary dag. We also prove that for (1)–(4), checking equality of the subtrees rooted at two given nodes of these structures, can be carried out in $O(\log N)$ time, where N is the number of nodes of the tree. One advantage of binary and hybrid dags, is that they also support the efficient checking of equality of (ending) sibling sequences in $O(\log N)$ time.

Our experiments over two large XML corpora and one corpus consisting of term rewriting systems show that dags and binary dags are the quickest to construct. Out of the dags (1)–(3), the reverse hdag (which uses a last child/previous sibling encoding) gives the smallest results. On our XML corpora, using the reverse binary encoding instead of the standard first child/next sibling encoding gives a compression improvement of more than 20%. As a practical yardstick we observe: For applications where sibling sequence check is important, or where the uniqueness of the compressed structures is important, the hybrid dag is a good choice. If strong compression is paramount, then structures (4) and (5) are appropriate. The advantage of (4) over (5) is its support of efficient subtree equality test. In future work we would like to apply our compression within other recent compression schemes in databases, such as for instance factorized databases [2].

Acknowledgments

The first and third author were supported by the DFG grant LO 748/8. The second author was supported by the DFG grant INST 268/239 and by the Engineering and Physical Sciences Research Council project "Enforcement of Constraints on XML streams" (EPSRC EP/G004021/1).

8. REFERENCES

[1] A. Arion, A. Bonifati, I. Manolescu, and A. Pugliese. XQueC: A query-conscious compressed XML database. *ACM Trans. Internet Techn.*, 7(2), 2007.

[2] N. Bakibayev, D. Olteanu, and J. Zavodny. Fdb: A query engine for factorised relational databases. *PVLDB*, 5(11):1232–1243, 2012.

[3] P. Bille, G. M. Landau, R. Raman, K. Sadakane, S. R. Satti, and O. Weimann. Random access to grammar-compressed strings. In *SODA*, pages 373–389, 2011.

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

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

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

[7] A. P. Ershov. On programming of arithmetic operations. *Commun. ACM*, 1(8):3–9, 1958.

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

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

[10] C. Koch. Efficient processing of expressive node-selecting queries on XML data in secondary storage: A tree automata-based approach. In *VLDB*, pages 249–260, 2003.

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

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

[13] M. Lohrey. Algorithmics on SLP-compressed strings: a survey. *Groups Complexity Cryptology*, 4:241–299, 2013.

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

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

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

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

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

[19] C. Meinel and T. Theobald. *Algorithms and Data Structures in VLSI Design: OBDD - Foundations and Applications*. Springer, 1998.

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

[21] C. G. Nevill-Manning and I. H. Witten. Identifying hierarchical strcture in sequences: A linear-time algorithm. *J. Artif. Intell. Res. (JAIR)*, 7:67–82, 1997.

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

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

[24] D. Suciu. Typechecking for semistructured data. In *DBPL*, pages 1–20, 2001.