Algorithmica manuscript No.
(will be inserted by the editor)

Constant-Time Tree Traversal and Subtree Equality Check for Grammar-Compressed Trees

Markus Lohrey · Sebastian Maneth · Carl Philipp Reh

Received: date / Accepted: date

Abstract A linear space data structure for grammar-compressed trees is presented which allows to carry out tree traversal operations and subtree equality checks in constant time. A traversal step consists of moving to the parent or to the ith child of a node.

Keywords Grammar-compressed trees · tree straight-line programs · algorithms for compressed trees

1 Introduction

Context-free grammars that produce single strings are a widely studied compact string representation, known as *straight-line programs* (SLPs). For instance, the string $(ab)^{1024}$ can be represented by the SLP with the rules $A_0 \to ab$ and $A_i \to A_{i-1}A_{i-1}$ for $1 \leq i \leq 10$ (A_{10} is the start symbol). In general, an SLP of size m can produce a string of length exponential in m. Besides grammar-based compressors (e.g. LZ78, RePair, or BISECTION, see [7] for more details) that derive an SLP from a given string, also algorithmic problems on SLP-compressed strings such as pattern match-

Markus Lohrey
Universität Siegen, Germany
Tel.: +49-271-740-2826
Fax: +49-271-740-2839
E-mail: lohrey@eti.uni-siegen.de

Sebastian Maneth
University of Edinburgh, UK
Tel.: +44 131 651 5642
E-mail: smaneth@inf.ed.ac.uk

Carl Philipp Reh
Universität Siegen, Germany
Tel.: +49-271-740-3233
E-mail: reh@eti.uni-siegen.de

ing, indexing, and compressed word problems have been investigated thoroughly, see [17] for a survey.

Motivated by applications where large tree structures occur, like XML processing, SLPs have been extended to node-labeled ranked ordered trees [4,5,12,15,19]. In those papers, straight-line linear context-free tree grammars are used. Such grammars produce a single tree and are also known as tree straight-line programs (TSLPs). TSLPs generalize dags (directed acyclic graphs), which are widely used as compact tree representation. Whereas dags only allow to share repeated subtrees, TSLPs can also share repeated internal tree patterns (i.e., connected subgraphs). The grammar-based tree compressor from [12] produces for every tree of size n (for a fixed set of node labels) a TSLP of size $\mathscr{O}(\frac{n}{\log n})$ and height $O(\log n)$, which is worst-case optimal. Implementations of grammar-based tree compressors which have proven to work well in practise include TreeRePair [19] and BPLEX[5].

Various querying problems on TSLP-compressed trees such as XPath querying and evaluating tree automata are studied in [18,20,22].

In this paper we study the problem of navigating in a TSLP-represented tree: given a TSLP $\mathscr{G}$ for a tree t, the task is to precompute in time $O(|\mathscr{G}|)$ an $O(|\mathscr{G}|)$-space data structure that allows to move from a node of t in time $O(1)$ to its parent node or to its ith child and to return in time $O(1)$ the node label of the current node. Here the nodes of t are represented in space $O(|\mathscr{G}|)$ in a suitable way. Such a data structure has been developed for string SLPs in [10]; it allows to move from left to right over the string produced by the SLP requiring time $O(1)$ per move. We first extend the data structure from [10] so that the string can be traversed in a two-way fashion, i.e., in one step we can move either to the left or right neighboring position in constant time. This data structure is then used to navigate a TSLP-represented tree.

TSLPs are typically used to compress ranked trees, i.e., trees where the maximal number r of children of a node is bounded by a constant. For instance, the above mentioned bound $\mathscr{O}(\frac{n}{\log n})$ from [12] assumes that r is constant. In many applications, r is indeed bounded (e.g., $r = 2$ for the following two encodings of unranked trees). For unranked trees where r is unbounded, it is more realistic to require that the data structure supports navigation to

(*i*) the parent node,
(*ii*) the first child,
(*iii*) the right sibling, and
(*iv*) the left sibling.

We can realize these operations by using a suitable constant-rank encoding of unranked trees. Two folklore binary tree encodings of an unranked tree t with maximal rank r are:

- First-child/next-sibling encoding $\text{fcns}(t)$: the left (resp. right) child of a node in $\text{fcns}(t)$ is the first child (resp., right sibling) in t. This coding is popular for XML, see, e.g., [25]; it is mentioned already in Paragraph 2.3.2 of Knuth's first book [16]. Under this encoding we can support $O(1)$ time navigation for (*ii*)–(*iv*) of above. The parent move (*i*) however, requires $O(r)$ time.
- Binary encoding: we define the binary encoding $\text{bin}(t)$ by adding for every node v of rank $s \leq r$ a binary tree of depth $\lceil \log s \rceil$ with s many leaves, whose root is v and

whose leaves are the children of v. This introduces at most $2s$ many new binary nodes (labeled by a new symbol). Thus $|\mathrm{bin}(t)| \leq 3|t|$. This encoding has also been used in the context of XML, see [25]. Every navigation step in the original tree can be simulated by $O(\log r)$ many navigation steps in $\mathrm{bin}(t)$.

Our second main result concerns subtree equality checks. This is the problem of checking for two given nodes of a tree whether the subtrees rooted at these two nodes are identical. We extend our data structure for tree navigation such that subtree equality checks can be carried out in time $O(1)$. The problem of checking equality of subtrees occurs in several different contexts, see for instance [6] for details. Typical applications are common subexpression detection, unification, and non-linear pattern matching. For instance, checking whether the pattern $f(x, f(y,y))$ is matched at a certain tree node needs a constant number of navigation steps and a single subtree equality check.

Related Work

The ability to navigate efficiently in a tree is a basic prerequisite for most tree querying procedures. For instance, the DOM representation available in web browsers through JavaScript provides tree navigation primitives (see, e.g., [9]). Tree navigation has been intensively studied in the context of succinct tree representations. Here, the goal is to represent a tree by a bit string, whose length is asymptotically equal to the information-theoretic lower bound. For instance, for ordered trees with n nodes the information-theoretic lower bound is $2n + o(n)$ and there exist succinct representations (e.g., the balanced parentheses representation) that encode an ordered tree of size n by a bit string of length $2n + o(n)$. In addition there exist such encodings that allow to navigate in the tree in constant time (and support many other tree operations), see e.g. [23] for a survey. Recently, grammatical formalisms for the compression of unranked trees have been proposed as well. In [2] the authors consider so called top dags as a compact tree representation. Top dags can be seen as a slight variant of TSLPs for unranked trees. It is shown in [2] that for every tree of size n the top dag has size $O(\frac{n}{\log^{0.19} n})$. Recently, this bound was improved to $O\left(\frac{n \cdot \log\log n}{\log n}\right)$ in [11]; it remains an open problem whether this bound can be improved to the information-theoretic limit $O\left(\frac{n}{\log n}\right)$. Moreover, also the navigation problem for top dags is studied in [2]. The authors show that a single navigation step in t can be carried out in time $O(\log|t|)$ in the top dag. Nodes are represented by their preorder numbers, which need $O(\log|t|)$ bits. In [3] an analogous result has been shown for unranked trees that are represented by string SLPs for the balanced parentheses representation of the tree. This covers also TSLPs: from a TSLP $\mathcal{G}$ one can easily compute in linear time an SLP for the balanced parentheses representation of the tree represented by $\mathcal{G}$. In some sense our results are orthogonal to the results of [3]:

- We can navigate, determine node labels, and check equality of subtrees in time $O(1)$, but our representation of tree nodes needs space $O(|\mathcal{G}|)$.
- Bille et al. [3] can navigate and execute several other tree queries (e.g. lowest common ancestor computations) in time $O(\log|t|)$, but their node representation (preorder numbers) only need space $O(\log|t|) \leq O(|\mathcal{G}|)$.

An implementation of navigation over TSLP-compressed trees is given in [21]. Their worst-case time per navigation step is $O(h)$ where h is the height of the TSLP. The authors demonstrate that on XML trees, full traversals take about 5–7 longer than over succinct trees (based on an implementation by Sadakane) while using 3–15 times less space; thus, their implementation provides a competitive space/time trade-off.

Checking equality of subtrees is trivial for minimal dags, since every subtree is uniquely represented. For so called SL grammar-compressed dags (which can be seen as TSLPs with certain restrictions) it was shown in [4] that equality of subtrees can be checked in time $O(\log|t|)$ for given preorder numbers.

2 Preliminaries

Let $\mathbb{N}$ denote the set $\{0,1,2,\dots\}$ of non-negative integers. For an alphabet Σ we denote by Σ^* the set of all strings over Σ including the empty string ε. For a string $w = a_1 \cdots a_n$ ($a_i \in \Sigma$) we denote by $\mathrm{alph}(w)$ the set of symbols $\{a_1,\dots,a_n\}$ occurring in w. Moreover, let $|w| = n$, $w[i] = a_i$ and $w[i:j] = a_i \cdots a_j$ where $w[i:j] = \varepsilon$, if $i > j$. Let $w[:i] = w[1:i]$ and $w[i:] = w[i:n]$.

2.1 Straight-Line Programs

A *straight-line program (SLP)* is a triple $\mathscr{P} = (N,\Sigma,\mathrm{rhs})$, where N is a finite set of *nonterminals*, Σ is a finite set of *terminals* ($\Sigma \cap N = \emptyset$), and $\mathrm{rhs} : N \to (N \cup \Sigma)^*$ is a mapping such that the binary relation $\{(A,B) \in N \times N \mid B \in \mathrm{alph}(\mathrm{rhs}(A))\}$ is acyclic. This condition ensures that every nonterminal $A \in N$ produces a unique string $\mathrm{val}_{\mathscr{P}}(A) \in \Sigma^*$. It is obtained from the string A by repeatedly replacing nonterminals B by $\mathrm{rhs}(B)$, until no nonterminal occurs in the string. We also write $A \to \alpha$ if $\mathrm{rhs}(A) = \alpha$ and call it a *rule* of $\mathscr{P}$. Usually, an SLP has a start nonterminal as well, but for our purpose it is convenient to consider SLPs without a start nonterminal.

Let $A \in N$ with $\mathrm{rhs}(A) = X_1 \cdots X_n$ and $X_1,\dots,X_n \in (N \cup \Sigma)$. The *derivation tree of $\mathscr{P}$ rooted in A* consists of a root node u labeled A. Let $i \in \{1,\dots,n\}$. If $X_i \in N$, then the ith child of u is the root node of a copy of the derivation tree of $\mathscr{P}$ rooted in X_i. If $X_i \in \Sigma$, then the ith child of u is a single leaf node labeled X_i.

The *size* of the SLP $\mathscr{P}$ is $|\mathscr{P}| = \sum_{A \in N} |\mathrm{rhs}(A)|$, i.e., the total length of all right-hand sides. A simple induction shows that for every SLP $\mathscr{P}$ of size m and every nonterminal A, $|\mathrm{val}_{\mathscr{P}}(A)| \leq 3^{\lceil m/3 \rceil}$ [7, proof of Lemma 1]. On the other hand, it is straightforward to define an SLP $\mathscr{P}$ of size $2m$ such that $|\mathrm{val}_{\mathscr{P}}(S)| \geq 2^m$ for a nonterminal S of the SLP. Hence, an SLP can be seen as a compressed representation of the string it generates, and it can achieve (at most) exponential compression ratios.

2.2 Tree Straight-Line Programs

A ranked alphabet is a set Σ such that every $a \in \Sigma$ has an associated rank $\mathrm{rank}(a) \in \mathbb{N}$. Let $\Sigma_i = \{a \in \Sigma \mid \mathrm{rank}(a) = i\}$. Fix ranked alphabets $\mathscr{F}$ and $\mathscr{N}$ of *terminal symbols* and *nonterminal symbols* such that for every $i \geq 0$, $\mathscr{F}_i$ and $\mathscr{N}_i$ are countably infinite.

Moreover, let $\mathscr{X} = \{x_1, x_2, \dots\}$ be the set of *parameters*. We assume that the three sets $\mathscr{F}$, $\mathscr{N}$, and $\mathscr{X}$ are pairwise disjoint. A *labeled tree* $t = (V, E, \lambda)$ is a finite, directed, and ordered tree t with set of nodes V, set of edges $E \subseteq V \times \mathbb{N} \times V$, and labeling function $\lambda : V \to \mathscr{F} \cup \mathscr{N} \cup \mathscr{X}$. We require for every node $v \in V$ that if $\lambda(v) \in \mathscr{F}_k \cup \mathscr{N}_k$, then v has k distinct children $u_1, \dots, u_k$, i.e., $(v, i, u) \in E$ if and only if $1 \le i \le k$ and $u = u_i$. A leaf of t is a node with zero children. We require that every node v with $\lambda(v) \in \mathscr{X}$ is a leaf of t. The *size* of t is $|t| = |V|$. We denote trees by their usual term notation, e.g. $a\langle b, c\rangle$ denotes the tree with an a-labeled root node that has a first child labeled b and a second child labeled c. We use the bracket symbols $\langle$ and $\rangle$ in terms in order to distinguish them from brackets that arise from function applications. We define $\mathscr{T}$ as the set of all labeled trees. Let $\text{labels}(t) = \{\lambda(v) \mid v \in V\}$. For $\mathscr{L} \subseteq \mathscr{F} \cup \mathscr{N} \cup \mathscr{X}$ we let $\mathscr{T}(\mathscr{L}) = \{t \in \mathscr{T} \mid \text{labels}(t) \subseteq \mathscr{L}\}$. The tree $t \in \mathscr{T}$ is *linear* if there do not exist different leaves that are labeled with the same parameter.

We now define a particular form of context-free tree grammars (see [8] for more details on context-free tree grammars) with the property that exactly one tree is derived. A *tree straight-line program (TSLP)* is a triple $\mathscr{G} = (N, S, \text{rhs})$, where

- $N \subseteq \mathscr{N}$ is a finite set of *nonterminals*,
- $S \in \mathscr{N}_0 \cap N$ is the *start nonterminal*, and
- $\text{rhs} : N \to \mathscr{T}(\mathscr{F} \cup N \cup \mathscr{X})$ is a mapping such that the following hold:
  (1) for every $A \in N$, the tree $\text{rhs}(A)$ is linear and if $A \in \mathscr{N}_k$ $(k \ge 0)$ then $\mathscr{X} \cap \text{labels}(\text{rhs}(A)) = \{x_1, \dots, x_k\}$.
  (2) The binary relation $\{(A, B) \in N \times N \mid B \in \text{labels}(\text{rhs}(A))\}$ is acyclic.

The conditions (1) and (2) ensure that from every nonterminal $A \in N \cap \mathscr{N}_k$ exactly one linear tree $\text{val}_{\mathscr{G}}(A) \in \mathscr{T}(\mathscr{F} \cup \{x_1, \dots, x_k\})$ is derived by applying the rules $A \to \text{rhs}(A)$ as rewrite rules in the usual sense. More generally, for every tree $t \in \mathscr{T}(\mathscr{F} \cup N \cup \{x_1, \dots, x_n\})$ we can derive the unique tree $\text{val}_{\mathscr{G}}(t) \in \mathscr{T}(\mathscr{F} \cup \{x_1, \dots, x_n\})$ by applying the rules of $\mathscr{G}$. Formally, we define this tree inductively as follows:

- $\text{val}_{\mathscr{G}}(a\langle t_1, \dots, t_k\rangle) = a\langle \text{val}_{\mathscr{G}}(t_1), \dots, \text{val}_{\mathscr{G}}(t_k)\rangle$ for $a \in \mathscr{F}$,
- $\text{val}_{\mathscr{G}}(A\langle t_1, \dots, t_k\rangle) = \text{val}_{\mathscr{G}}(\text{rhs}(A)[x_1/t_1, \dots, x_k/t_k])$ for $A \in N \cap \mathscr{N}_k$. Here, $\text{rhs}(A)[x_1/t_1, \dots, x_k/t_k]$ is the tree obtained from $\text{rhs}(A)$ by replacing the unique x_i-labeled leaf by the tree t_i.

The tree defined by $\mathscr{G}$ is $\text{val}(\mathscr{G}) = \text{val}_{\mathscr{G}}(S)$. The following example shows the derivation of $\text{val}(\mathscr{G})$ for a TSLP $\mathscr{G}$.

Example 1 Let $\mathscr{G} = (\{S, A, B, C, D, E, F\}, S, \text{rhs})$, $a \in \mathscr{F}_0$, $b \in \mathscr{F}_2$, and rhs be given by

$$\begin{aligned}
\text{rhs}(S) &= A\langle B\rangle \\
\text{rhs}(A) &= C\langle F, x_1\rangle \\
\text{rhs}(B) &= E\langle F\rangle \\
\text{rhs}(C) &= D\langle E\langle x_1\rangle, x_2\rangle \\
\text{rhs}(D) &= b\langle x_1, x_2\rangle \\
\text{rhs}(E) &= D\langle F, x_1\rangle \\
\text{rhs}(F) &= a.
\end{aligned}$$

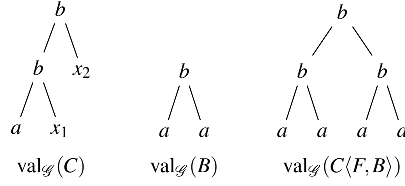

Fig. 1 Example derivations for the grammar $\mathscr{G}$ from Example 1.

The derivation of $\mathrm{val}(\mathscr{G}) = b\langle b\langle a,a\rangle, b\langle a,a\rangle\rangle$ from S is:

$$
\begin{aligned}
S &\to A\langle B\rangle \to C\langle F,B\rangle \to D\langle E\langle F\rangle,B\rangle \to b\langle E\langle F\rangle,B\rangle \to b\langle D\langle F,F\rangle,B\rangle \\
&\to b\langle b\langle F,F\rangle,B\rangle \to b\langle b\langle a,F\rangle,B\rangle \to b\langle b\langle a,a\rangle,B\rangle \to b\langle b\langle a,a\rangle,E\langle F\rangle\rangle \\
&\to b\langle b\langle a,a\rangle,D\langle F,F\rangle\rangle \to b\langle b\langle a,a\rangle,b\langle F,F\rangle\rangle \to b\langle b\langle a,a\rangle,b\langle a,F\rangle\rangle \\
&\to b\langle b\langle a,a\rangle,b\langle a,a\rangle\rangle.
\end{aligned}
$$

The *size* $|\mathscr{G}|$ of a TSLP $\mathscr{G} = (N,S,\mathrm{rhs})$ is defined as $|\mathscr{G}| = \sum_{A\in N} |\mathrm{rhs}(A)|$. For instance, the TSLP from Example 1 has size 18.

A TSLP $\mathscr{G} = (N,S,\mathrm{rhs})$ is *monadic* if $N \subseteq \mathscr{N}_0 \cup \mathscr{N}_1$, i.e., every nonterminal has rank at most one. From now on we only consider monadic TSLPs, which can be obtained by using the following result, shown in [20]:

Proposition 1 *From a given TSLP $\mathscr{G}$, where r and k are the maximal ranks of terminal and nonterminal symbols appearing in a right-hand side, one can construct in time $O(r\cdot k\cdot|\mathscr{G}|)$ a monadic TSLP $\mathscr{G}'$ such that* $\mathrm{val}(\mathscr{G}') = \mathrm{val}(\mathscr{G})$ *and* $|\mathscr{G}'| \in O(r\cdot|\mathscr{G}|)$.

2.3 Computational model

For the rest of the paper we use the word RAM model, where registers have a certain bit length w. Arithmetic operations and comparisons of registers can be carried out in time $O(1)$. The space of a data structure is measured by the number of registers. Our algorithms need the following register lengths w, where $\mathscr{G}$ is the input TSLP and $t = \mathrm{val}(\mathscr{G})$.

- For navigation (Section 4) we need a bit length of $w = O(\log|\mathscr{G}|)$, since we only have to store numbers of length at most $|\mathscr{G}|$.
- For equality checks (Section 5) we need a bit length of $w = O(\log|t|) \le O(|\mathscr{G}|)$, which is the same assumption as in [2,3].

3 Two-Way Traversal in SLP-Compressed Strings

In [10] the authors present a data structure of size $O(|\mathscr{P}|)$ for storing an SLP $\mathscr{P}$ that allows to produce, for any nonterminal X of $\mathscr{P}$, the string $\mathrm{val}_{\mathscr{P}}(X)$ with time delay of $O(1)$ per symbol. That is, the symbols of $\mathrm{val}_{\mathscr{P}}(X)$ are produced from left to right and for each symbol constant time is needed.

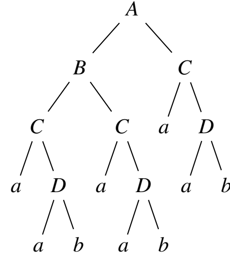

Fig. 2 Derivation tree for the SLP $\mathscr{H}$ rooted in A from Example 2 below.

In a first step, we enhance the data structure from [10] for traversing SLP-compressed strings in such a way that both operations of moving to the left and right symbol are supported in constant time. For this, we assume that the SLP $\mathscr{P} = (N, \Sigma, \mathrm{rhs})$ has the property that

$$|\mathrm{rhs}(X)| = 2 \text{ for each } X \in N. \tag{1}$$

Every SLP $\mathscr{P}$ with $|\mathrm{val}(\mathscr{P})| \geq 2$ can be transformed in linear time into an SLP $\mathscr{P}'$ with property (1) and so that $\mathrm{val}(\mathscr{P}') = \mathrm{val}(\mathscr{P})$: first, we replace all occurrences of nonterminals B with $|\mathrm{rhs}(B)| \leq 2$ by $\mathrm{rhs}(B)$. If $\mathrm{rhs}(S) = A \in N$, then we redefine $\mathrm{rhs}(S) = \mathrm{rhs}(A)$. Finally, for every $A \in N$ such that $\mathrm{rhs}(A) = \alpha_1 \cdots \alpha_n$ with $n \geq 3$ and $\alpha_1, \ldots, \alpha_n \in (N \cup \Sigma)$ we introduce new nonterminals $A_2, \ldots, A_{n-1}$ with rules $A_i \to \alpha_i A_{i+1}$ for $2 \leq i \leq n-2$, and $A_{n-1} \to \alpha_{n-1}\alpha_n$, and redefine $\mathrm{rhs}(A) = \alpha_1 A_2$. It should be clear that $|\mathscr{P}'| \leq 2 \cdot |\mathscr{P}|$.

Recall the definition of a derivation tree of an SLP from Section 2.1. For instance, Figure 2 shows the derivation tree of $\mathscr{H}$ rooted in A for the SLP $\mathscr{H}$ from Example 2. Note that the positions in $\mathrm{val}_{\mathscr{P}}(X)$ correspond to root-leaf paths in the (binary) derivation tree of $\mathscr{P}$ rooted in the nonterminal X. We represent a root-leaf path by merging successive edges where the path moves in the same direction (left or right) towards the leaf. To formalize this idea, we define for every $\alpha \in N \cup \Sigma$ the strings $L(\alpha), R(\alpha) \in N^*\Sigma$ inductively as follows: for $a \in \Sigma$ let

$$L(a) = R(a) = a.$$

For $A \in N$ with $\mathrm{rhs}(A) = \alpha\beta$ $(\alpha, \beta \in N \cup \Sigma)$ let

$$L(A) = A\,L(\alpha) \text{ and } R(A) = A\,R(\beta). \tag{2}$$

Note that for every $A \in N$, the string $L(A)$ has the form $A_1 A_2 \cdots A_n a$ with $A_i \in N$, $A_1 = A$, and $a \in \Sigma$. We define $\omega_L(A) = a$. The terminal $\omega_R(A)$ is defined analogously by referring to the string $R(A)$.

Example 2 Let $\mathscr{H} = (\{S, A, B, C, D\}, \{a, b\}, \mathrm{rhs})$, where rhs is given by

$$S \to AB,\ A \to BC,\ B \to CC,\ C \to aD,\ D \to ab.$$

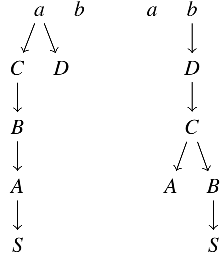

Fig. 3 The left shows the tries $T_L(a)$, $T_L(b)$, and the right shows $T_R(a)$, $T_R(b)$, for the SLP from Example 2.

Then we have

$$\begin{aligned}
L(S) &= SABCa \\
L(A) &= ABCa \\
L(B) &= BCa \\
L(C) &= Ca \\
L(D) &= Da \\
R(S) &= SBCDb \\
R(A) &= ACDb \\
R(B) &= BCDb \\
R(C) &= CDb \\
R(D) &= Db.
\end{aligned}$$

Moreover, $\omega_L(X) = a$ and $\omega_R(X) = b$ for all $X \in \{S,A,B,C,D\}$.

We store all strings $L(A)$ (for $A \in N$) in $|\Sigma|$-many tries: fix $a \in \Sigma$ and let $w_1, \ldots, w_n$ be all strings $L(A)$ such that $\omega_L(A) = a$. Let v_i be the string w_i reversed. Then, $a, v_1, \ldots, v_n$ is a prefix-closed set of strings (except that the empty string is missing) that can be stored in a trie $T_L(a)$. Formally, the nodes of $T_L(a)$ are the strings $a, v_1, \ldots, v_n$, where each node is labeled by its last symbol (so the root is labeled with a), and there is an edge from aw to awA for all appropriate $w \in N^*$ and $A \in N$. The tries $T_R(a)$ are defined in the same way by referring to the strings $R(A)$. Note that the total number of nodes in all tries $T_L(a)$ ($a \in \Sigma$) is exactly $|N| + |\Sigma|$. In fact, every $\alpha \in N \cup \Sigma$ occurs exactly once as a node label in the forest $\{T_L(a) \mid a \in \Sigma\}$.

Example 3 (Example 2 continued) The tries $T_L(a)$, $T_L(b)$, $T_R(a)$, and $T_R(b)$ for the SLP from Example 2 are shown in Figure 3.

Next, we define two alphabets L and R as follows:

$$L = \{(A, \ell, \alpha) \mid \alpha \in \mathrm{alph}(L(A)) \setminus \{A\}\} \tag{3}$$

$$R = \{(A, r, \beta) \mid \beta \in \mathrm{alph}(R(A)) \setminus \{A\}\}. \tag{4}$$

Note that the sizes of these alphabets are quadratic in the size of $\mathscr{H}$. On the alphabets L and R we define the partial operations ℓ-reduce : $L \to L$ and r-reduce : $R \to R$ as follows: let $(A, \ell, \alpha) \in L$. Hence we can write $L(A)$ as $Au\alpha v$ for some strings u and v. If $u = \varepsilon$, then ℓ-reduce(A, ℓ, α) is undefined. Otherwise, we can write u as $u'B$ for

some $B \in N$. Then we define ℓ-reduce$(A,\ell,\alpha) = (A,\ell,B)$. The definition of r-reduce is analogous: if $(A,r,\alpha) \in R$, then we can write $R(A)$ as $Au\alpha v$ for some strings u and v. If $u = \varepsilon$, then r-reduce(A,r,α) is undefined. Otherwise, we can write u as $u'B$ for some $B \in N$ and define r-reduce$(A,r,\alpha) = (A,r,B)$.

Example 4 (Example 2 continued) The sets L and R are:

$$\begin{aligned} L = \{&(S,\ell,A),(S,\ell,B),(S,\ell,C),(S,\ell,a),(A,\ell,B),\\ &(A,\ell,C),(A,\ell,a),(B,\ell,C),(B,\ell,a),(C,\ell,a),(D,\ell,a)\}\\ R = \{&(S,r,B),(S,r,C),(S,r,D),(S,r,b),(A,r,C),(A,r,D),\\ &(A,r,b),(B,r,C),(B,r,D),(B,r,b),(C,r,D),(C,r,b),(D,r,b)\}. \end{aligned}$$

For instance, ℓ-reduce$(S,\ell,a) = (S,\ell,C)$ and r-reduce$(B,r,D) = (B,r,C)$ whereas ℓ-reduce(S,ℓ,A) is undefined.

An element (A,ℓ,α) can be represented by a pair (v_1,v_2) of different nodes in the forest $\{T_L(a) \mid a \in \Sigma\}$, where v_1 (resp. v_2) is the unique node labeled with α (resp., A). Note that v_1 and v_2 belong to the same trie and that v_2 is below v_1. This observation allows us to reduce the computation of the mapping ℓ-reduce to a so-called *next link query*: from the pair (v_1,v_2) we have to compute the unique child v of v_1 such that v is on the path from v_1 to v_2. If v is labeled with B, then ℓ-reduce$(A,\ell,\alpha) = (A,\ell,B)$, which is represented by the pair (v,v_2). Clearly, the same remark applies to the map r-reduce. The following result is mentioned in [10], see Section 6 for a discussion.

Proposition 2 *A trie T can be represented in space $O(|T|)$ such that any next link query can be answered in time $O(1)$. Moreover, this representation can be computed in time $O(|T|)$ from T.*

We represent a path in the derivation tree of $\mathscr{P}$ rooted in X by a sequence of triples

$$\gamma = (A_1,\delta_1,A_2)(A_2,\delta_2,A_3)\cdots(A_{n-1},\delta_{n-1},A_n)(A_n,\delta_n,a) \in (L \cup R)^+$$

such that $n \geq 1$ and the following properties hold:

- $A_1 = X$, $a \in \Sigma$
- $\delta_i = \ell$ if and only if $\delta_{i+1} = r$ for all $1 \leq i \leq n-1$.

We call such a sequence a *valid X-sequence for $\mathscr{P}$* in the following, or briefly a valid sequence if X is not important and $\mathscr{P}$ is clear from the context. Note that a valid X-sequence γ indeed defines a unique path in the derivation tree rooted at X that ends in a leaf that is labeled with the terminal symbol a if γ ends with (A,δ,a). This path, in turn, defines a unique position in the string $\mathrm{val}_{\mathscr{P}}(X)$ that we denote by $\mathrm{pos}(\gamma)$.

We now define a procedure right (see Algorithm 1) that takes as input a valid X-sequence γ and returns a valid X-sequence γ' such that $\mathrm{pos}(\gamma') = \mathrm{pos}(\gamma)+1$ in case the latter is defined and otherwise returns “undefined”. It is based on the obvious fact that in order to move in a full binary tree from a leaf to the next leaf (where “next” refers to the natural left-to-right order on the leaves) one has to repeatedly move to parent nodes as long as right-child edges are traversed (in the opposite direction);

Algorithm 1: right(γ)

```
Input: valid sequence γ
Output: valid sequence γ' with pos(γ') = pos(γ) + 1 if it exists, and "undefined" otherwise
(A, δ, a) := pop(γ)
if δ = ℓ then
|   γ := expand-right(γ, A, a)
else
|   if γ = ε then
|   |   return "undefined"
|   else
|   |   (A', ℓ, A) := pop(γ)
|   |   γ := expand-right(γ, A', A)
|   end
end
return γ
```

Algorithm 2: expand-right(γ, A, α)

```
Input: sequence γ, A ∈ N, α ∈ N ∪ Σ such that γ(A, ℓ, α) is a prefix of a valid sequence
Output: valid sequence γ, representing the next position
let rhs(A) = α1 α2
if α ≠ α1 then
|   (A, ℓ, B) := ℓ-reduce(A, ℓ, α)
|   push(γ, (A, ℓ, B))
|   let rhs(B) = β1 β2
|   push(γ, (B, r, β2))
|   if β2 ∈ N then
|   |   push(γ, (β2, ℓ, ωL(β2)))
|   end
else
|   if γ = ε then
|   |   push(γ, (A, r, α2))
|   else
|   |   (B, r, A) := pop(γ)
|   |   push(γ, (B, r, α2))
|   end
|   if α2 ∈ N then
|   |   push(γ, (α2, ℓ, ωL(α2)))
|   end
end
return γ
```

when this is no longer possible, the current node is the left child of its parent p. One now moves to the right child of p and from here repeatedly to left children until a leaf is reached. Each of these four operations can be implemented in constant time on valid sequences, using the fact that consecutive edges to left (resp., right) children are merged into a single triple from L (resp., R) in our representation of paths. We use the valid sequence γ as a stack with the operations pop (which returns the right-most triple of γ, which is thereby removed from γ) and push (which appends a given triple on the right end of γ). The procedure right uses the procedure expand-right (see Algorithm 2) that takes as input a (non-valid) sequence γ of triples, $A \in N$, and a symbol $\alpha \in N \cup \Sigma$ such that $\gamma(A, \ell, \alpha)$ is a prefix of a valid sequence. The sequence γ

has to be treated as a global variable in order to obtain an $O(1)$-time implementation (to make the presentation clearer, we pass γ as a parameter to expand-right).

In a completely analogous way we can define a procedure left that takes as input a valid X-sequence γ and returns a valid X-sequence γ' such that $\mathrm{pos}(\gamma') = \mathrm{pos}(\gamma) - 1$ in case the latter is defined, and otherwise returns “undefined”. The details are left to the reader.

4 Traversal in TSLP-Compressed Trees

In this section, we extend the traversal algorithm from the previous section from SLPs to TSLPs. We only consider monadic TSLPs. If the TSLP is not monadic, then we transform it into a monadic TSLP using Proposition 1. Furthermore, we say that a *monadic* TSLP $\mathscr{G} = (N, S, \mathrm{rhs})$ is in *normal form*, if each of its right-hand sides $\mathrm{rhs}(A)$, where $A \in N$, uses one of the following four forms (we write x for the parameter x_1):

(a) $B\langle C\rangle$ for $B, C \in N$ (and A has rank 0)
(b) $B\langle C\langle x\rangle\rangle$ for $B, C \in N$ (and A has rank 1)
(c) $a \in \mathscr{F}_0$ (and A has rank 0)
(d) $f\langle A_1, \ldots, A_{i-1}, x, A_{i+1}, \ldots, A_n\rangle$ for $A_1, \ldots, A_{i-1}, A_{i+1}, \ldots, A_n \in N$, $f \in \mathscr{F}_n$, $n \geq 1$ (and A has rank 1).

Let us write N_x ($x \in \{a, b, c, d\}$) for the set of all nonterminals whose right-hand side is of the above type (x). Let $N_1 = N_a \cup N_b$ and $N_2 = N_c \cup N_d$.

Lemma 1 *A monadic TSLP $\mathscr{G}$ can be transformed in linear time into a TSLP $\mathscr{G}'$ that is in* normal form *such that* $\mathrm{val}(\mathscr{G}) = \mathrm{val}(\mathscr{G}')$.

This transformation is done in a similar way as the transformation of SLPs at the beginning of Section 3: remove nonterminals $A \in \mathscr{N}_1$ with $\mathrm{rhs}(A) = x_1$ (resp., $\mathrm{rhs}(A) = B\langle x_1\rangle, B \in \mathscr{N}$) by replacing in all right-hand sides every subtree $A\langle t\rangle$ by t (resp., $B\langle t\rangle$) and iterate this as long as possible. Similarly, we eliminate nonterminals $A \in \mathscr{N}_0 \setminus \{S\}$ with $\mathrm{rhs}(A) \in \mathscr{N}$. If $\mathrm{rhs}(S) = B \in \mathscr{N}_0$ then redefine $\mathrm{rhs}(S) := \mathrm{rhs}(B)$. Finally, right-hand sides of size at least two are split top-down into new nonterminals with right-hand sides of the above types (a), ..., (d). In case the right-hand side contains the parameter x_1 (in a unique position), we decompose along the path to x_1. For example, the rule

$$Z \to h\langle f\langle A, a\rangle, f\langle A, g\langle x\rangle\rangle, B\langle A\rangle\rangle$$

is replaced by

$$\begin{aligned} Z &\to C\langle D\langle x\rangle\rangle \\ C &\to h\langle E, x, F\rangle \\ D &\to G\langle H\langle x\rangle\rangle \\ E &\to G\langle J\rangle \\ F &\to B\langle A\rangle \\ G &\to f\langle A, x\rangle \\ H &\to g\langle x\rangle \\ J &\to a. \end{aligned}$$

The resulting TSLP has size at most $2|\mathscr{G}|$. The derivation tree of a TSLP can be conveniently defined, similar to the definition for SLPs; in this derivation tree, the ith child of a node labeled by a nonterminal X corresponds to the ith nonterminal in the tree t, where $A \to t$ is a rule of the TSLP. Thus, every node of the derivation tree is labeled by a nonterminal, i.e., it is an "abstract derivation tree" in which terminal nodes are not represented.

For the rest of the section we fix a monadic TSLP $\mathscr{G} = (N, S, \mathrm{rhs})$ in normal form and define $N_a, N_b, N_c, N_d, N_1, N_2$ as above. Note that if we start with a nonterminal $A \in N_a$ and then replace nonterminals from N_1 by their right-hand sides repeatedly, we obtain a tree that consists of nonterminals from N_d followed by a single nonterminal from N_c. After replacing these nonterminals by their right-hand sides, we obtain a *caterpillar tree* (CP-tree) which is composed of right-hand sides of the form (d) followed by a single constant from $\mathscr{F}_0$. Hence, there is a unique path of terminal symbols from $\mathscr{F}$, and we call this path the *spine path* of A. All other nodes of the caterpillar tree are leaves and labeled with nonterminals of rank zero to which we can apply again the TSLP rules. The size of a caterpillar tree and therefore a spine path can be exponential in the size of the TSLP.

For the TSLP $\mathscr{G}$ we define the *derived SLP* $\mathscr{P} = (N_1, N_2, \mathrm{rhs}_1)$ as follows: if $A \in N_1$ with $\mathrm{rhs}(A) = B\langle C\rangle$ or $\mathrm{rhs}(A) = B\langle C\langle x\rangle\rangle$, then $\mathrm{rhs}_1(A) = BC$. The triple alphabets L and R from (3) and (4) refer to this SLP $\mathscr{P}$. Moreover, we define M as the set of triples (A, k, A_k) such that $A \in N_d$, $\mathrm{rhs}(A) = f\langle A_1, \ldots, A_{i-1}, x, A_{i+1}, \ldots, A_n\rangle$ and $k \in \{1, \ldots, n\} \setminus \{i\}$.

Note that the nodes of the tree $\mathrm{val}(\mathscr{G})$ can be identified with the nodes of $\mathscr{G}$'s derivation tree that are labeled with a nonterminal from N_2 (every nonterminal from N_2 has a unique occurrence of a terminal symbol on its right-hand side).

A *valid sequence* for $\mathscr{G}$ is a sequence

$$\gamma = (A_1, e_1, A_2)(A_2, e_2, A_3)\cdots(A_{n-1}, e_{n-1}, A_n)(A_n, e_n, A_{n+1}) \in (L \cup R \cup M)^*$$

such that $n \geq 0$ (note that $e_1, \ldots, e_n \in \{\ell, r\} \uplus \mathbb{N}$) and the following hold:

- if $S \in N_a$ then $n \geq 1$.
- If $n \geq 1$ then $A_1 = S$ and $A_{n+1} \in N_2$.
- If $e_i, e_{i+1} \in \{\ell, r\}$ then $e_i = \ell$ if and only if $e_{i+1} = r$.

Such a valid sequence represents a path in the derivation tree of the TSLP $\mathscr{G}$ from the root to an N_2-labeled node, and hence represents a node of the tree $\mathrm{val}(\mathscr{G})$. Note that in case $S \in N_c$, the empty sequence is valid too and represents the root of the single-node tree $\mathrm{val}(\mathscr{G})$. Moreover, if the last triple (A_n, e_n, A_{n+1}) belongs to M, then we must have $A_{n+1} \in N_c$, i.e., $\mathrm{rhs}(A_{n+1}) \in \mathscr{F}_0$.

Example 5 Consider the monadic TSLP $\mathscr{G}$ with nonterminals S, A, B, C, D, E, F, G, H, J and the following rules

$$\begin{aligned} S &\to A\langle B\rangle \\ A &\to C\langle D\langle x\rangle\rangle \\ B &\to C\langle E\rangle \\ C &\to f\langle F,x\rangle \\ D &\to f\langle x,F\rangle \\ E &\to D\langle F\rangle \\ F &\to G\langle H\rangle \\ G &\to J\langle J\langle x\rangle\rangle \\ H &\to a \\ J &\to g\langle x\rangle. \end{aligned}$$

It is in normal form and produces the tree shown in Figure 4. We have

$$N_1 = \{S,A,B,E,F,G\} \text{ and } N_2 = \{C,D,J\}.$$

The SLP $\mathscr{P}$ consists of the rules

$$\begin{aligned} S &\to AB \\ A &\to CD \\ B &\to CE \\ E &\to DF \\ F &\to GH \\ G &\to JJ \end{aligned}$$

The terminal symbols are C, D, H, and J. The triple set M is

$$M = \{(C,1,F),(D,2,F)\}.$$

A valid sequence is for instance $(S,\ell,A)(A,r,D)(D,2,F)(F,\ell,J)$. It represents the circled g-labeled node in Figure 4.

From a valid sequence γ we can clearly compute in time $O(1)$ the label of the tree node represented by γ. Let us denote this terminal symbol with label(γ): if γ ends with the triple (A,e,B), then we have $B \in N_2$ and label(γ) is the unique terminal symbol in rhs(B). If $\gamma = \varepsilon$, then $S \in N_c$ and label(γ) is the unique terminal symbol in rhs(S).

Using valid sequences of $\mathscr{G}$ it is easy to carry out a single navigation step in constant time. Let us fix a valid sequence γ. We consider the following possible navigation steps: move to the parent node (if it exists) and move to the ith child (if it exists). Consider for instance the navigation to the ith child (similar arguments can be used to navigate to the parent node). If γ is empty or ends with a triple from $M \cup (N \times \{\ell,r\} \times N_c)$, then γ represents a leaf node of val$(\mathscr{G})$; hence the ith child does not exist. Otherwise let $\beta \neq \varepsilon$ be the maximal suffix of γ, which belongs to $(L \cup R)^*$. Then β represents a path in the derivation tree of the string SLP $\mathscr{P}$ that is rooted in a certain nonterminal $A \in N_a$ and that leads to a certain nonterminal $B \in N_d$. This path corresponds to a node of val$_{\mathscr{G}}(A)$ that is located on the spine path of A. We can now apply our SLP-navigation algorithms left and right to the sequence β in

Algorithm 3: parent(γ)

```
Input: valid sequence γ representing a node u of val(𝒢)
Output: valid sequence γ representing the parent of u if it exists, and "undefined" otherwise
if γ ∈ {ε} ∪ L then
  | return "undefined"
end
if γ belongs to α·M·L or α·M for some α ∈ (L∪R∪M)* then
  | return α                               //Note that α is a valid sequence.
end
let γ = α·β, where α ∈ {ε} ∪ (L∪R∪M)*·M and β ∈ (L∪R)+
return α·left(β)                           //β ∉ L, hence left(β) ≠ undefined.
```

Algorithm 4: child(γ, i)

```
Input: valid sequence γ representing a node u of val(𝒢), positive integer i
Output: valid sequence representing the ith child of u if it exists, and "undefined" otherwise
if i > r, where r is the rank of label(γ) then
  | return "undefined"
end
let γ = α·β, where α ∈ {ε} ∪ (L∪R∪M)*·M and β ∈ (L∪R)+
let β end with the triple (C,δ,B)                          //Note that B ∈ N_d.
let rhs(B) = f⟨A_1,...,A_{j-1},x_1,A_{j+1},...,A_n⟩
if i = j then
  | return α·right(β)
end
return α·β·(B,i,A_i)·root(A_i)
```

order to move up or down on the spine path. More precisely, let β end with the triple (C,δ,B) (we must have $\delta \in \{\ell, r\}$ and $B \in N_d$). We can now distinguish the following cases, where $f\langle A_1,\dots,A_{j-1},x,A_{j+1},\dots,A_n\rangle$ is the right-hand side of B:

(i) if $i \neq j$, then the ith child is obtained by appending to γ the triple $(B,i,A_i) \in M$ followed by the path that represents the root of A_i (which consists of at most one triple).
(ii) If $i = j$, then the ith child of the current node is obtained by moving down on the spine path. Thus, we replace the suffix β by right(β).

Algorithm 3 shows the pseudo code for moving to the parent node, and Algorithm 4 shows the pseudo code for moving to the i-th child. If this node does not exist, then "undefined" is returned. To make the code more readable we denote the concatenation operator for sequences of triples (or sets of triple sequences) by "$\cdot$". We make use of the procedures left and right from Section 3, applied to the SLP $\mathscr{P}$ derived from our TSLP $\mathscr{G}$. Note that in Algorithm 3 we apply in the final case the procedure left to the maximal suffix β from $(L\cup R)^+$ of the current valid sequence γ (and similarly for Algorithm 4). To provide an $O(1)$ time implementation we do not copy the sequence β and pass it to left (which is not possible in constant time) but apply left directly to γ. The right-most triple from M in γ (if it exists) works as a left-end marker. Algorithm 4 uses the procedure root(A) (with A of rank 0). This procedure is not shown explicitly: it simply returns ε if $A \in N_2$, and otherwise returns

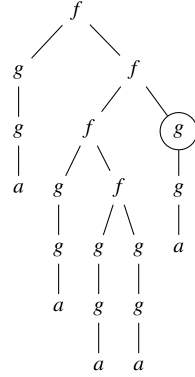

Fig. 4 The tree produced from the TSLP in Example 5

$(A, \ell, \omega_L(A))$ (recall ω_L from Page 7). Hence, it returns the representation of the root node of $\text{val}_{\mathscr{G}}(A)$.

The following theorem summarizes the results of this section.

Theorem 1 *Given a monadic TSLP $\mathscr{G}$ we can compute in linear time on a word RAM with register length $O(\log|\mathscr{G}|)$ a data structure of size $O(|\mathscr{G}|)$ that allows to carry out the following computations in time $O(1)$, where γ is a valid sequence that represents the tree node v:*

- *compute the valid sequence for the parent node of v, and*
- *compute the valid sequence for the ith child of v.*

5 Subtree Equality Checking

Consider a monadic TSLP $\mathscr{G} = (N, S, \text{rhs})$ in *normal form*, which again means that for every nonterminal $A \in N$, $\text{rhs}(A)$ has one of the following four forms:

(a) $B\langle C\rangle$ for $B, C \in N$ (and A has rank 0)
(b) $B\langle C\langle x\rangle\rangle$ for $B, C \in N$ (and A has rank 1)
(c) $a \in \mathscr{F}_0$ (and A has rank 0)
(d) $f\langle D_1, \ldots, D_{i-1}, x, D_{i+1}, \ldots, D_n\rangle$ for $D_1, \ldots, D_{i-1}, D_{i+1}, \ldots, D_n \in N$, $f \in \mathscr{F}_n$, $n \geq 1$ (and A has rank 1)

Let $t = \text{val}(\mathscr{G})$ be the tree produced by $\mathscr{G}$. The goal of this section is to extend the navigation algorithm from the previous section such that for two nodes of t (represented by valid sequences) we can test in $O(1)$ time whether the subtrees rooted at the two nodes are equal. For this we will allow polynomial time preprocessing of the TSLP $\mathscr{G}$. We make use of the following algorithmic result from [5]:

Proposition 3 *For two given TSLPs $\mathscr{G}_1$ and $\mathscr{G}_2$ it can be checked in polynomial time whether or not* $\text{val}(\mathscr{G}_1) = \text{val}(\mathscr{G}_2)$.

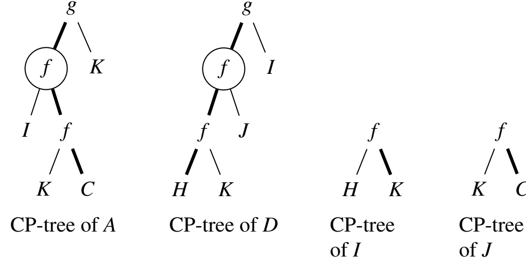

Fig. 5 Caterpillar trees of A, D, I and J of the TSLP $\mathscr{G}$ from Example 6. Thick lines represent spine paths. The subtrees at the two nodes marked by circles derive equal trees.

We also make use of some algorithmic facts about SLPs, which are collected in the following proposition, see [17] for details:

Proposition 4 *There are polynomial time algorithms that compute, for a given SLP* $\mathscr{P} = (N, \Sigma, \mathrm{rhs})$*,* $X, Y \in N$*,* $1 \le i, j \le |\mathrm{val}_{\mathscr{P}}(X)|$ *and* $i < j$*:*

- *the symbol* $\mathrm{val}_{\mathscr{P}}(X)[i]$*,*
- *an SLP for the string* $\mathrm{val}_{\mathscr{P}}(X)[i:j]$*, and*
- *the length of the longest common prefix of* $\mathrm{val}_{\mathscr{P}}(X)$ *and* $\mathrm{val}_{\mathscr{P}}(Y)$*.*

To simplify the following constructions, we assume that $\mathscr{G}$ is *reduced* in the sense that $\mathrm{val}_{\mathscr{G}}(A) \neq \mathrm{val}_{\mathscr{G}}(B)$ for all $A, B \in N$ with $A \neq B$. This can be checked using Proposition 3 since we allow polynomial time preprocessing. In case there are two nonterminals that produce the same trees we simply replace one of them by the other. The next step is to characterize the equal subtrees of a TSLP in a suitable way: for that, we reuse some of the notations introduced in the previous section. Recall that in the beginning of Section 4 we derived an SLP $\mathscr{P} = (N_1, N_2, \mathrm{rhs}_1)$ from $\mathscr{G}$. It was defined by $\mathrm{rhs}_1(A) = BC$ if $A \in N_1$ with $\mathrm{rhs}(A) = B\langle C\rangle$ or $\mathrm{rhs}(A) = B\langle C\langle x\rangle\rangle$. So, for every $A \in N_a$ we have

$$\mathrm{val}_{\mathscr{P}}(A) = A_1 A_2 \cdots A_n A_{n+1} \in N_d^* N_c$$

for some $n \geq 1$. Let $\ell(A) = n+1$ (this is the length of the spine path of A) and $i, j \in \{1, \ldots, n+1\}$ set $A[i:j] = A_i A_{i+1} \cdots A_j$, $A[i:] = A_i \cdots A_n A_{n+1}$, $A[:i] = A_1 \cdots A_i$ and $A[i] = A_i$. The subtree produced by the spine path of A at depth i is denoted by $A_{\triangleright}(i)$ and defined by

$$A_{\triangleright}(i) = \mathrm{val}_{\mathscr{G}}(A_i \langle A_{i+1} \langle \cdots A_n \langle A_{n+1}\rangle \cdots \rangle\rangle).$$

Let $A \in N_a$. We define $s(A)$ as the smallest number $i \geq 2$ such that $A_{\triangleright}(i) = \mathrm{val}_{\mathscr{G}}(D)$ for some nonterminal $D \in N$ of rank zero. This unique nonterminal D is denoted by A'. Moreover, let

$$r_A = \mathrm{rhs}(A[s(A)-1]).$$

By definition, $A[s(A)-1] \in N_d$, so r_A is of the form $f\langle D_1, \ldots, D_{j-1}, x, D_{j+1}, \ldots, D_m\rangle$ for some $f \in \mathscr{F}_m$, $D_1, \ldots, D_{j-1}, D_{j+1}, \ldots, D_m \in N$. A convenient notation is to treat r_A as a function $r_A : \mathscr{T}(\mathscr{F} \cup N) \to \mathscr{T}(\mathscr{F} \cup N)$ that substitutes x for an actual tree:

$$r_A(t) = f\langle D_1, \ldots, D_{j-1}, t, D_{j+1}, \ldots, D_m\rangle$$

With these notations, we have

$$A_{\triangleright}(s(A)-1) = \mathrm{val}_{\mathscr{G}}(A[s(A)-1](A')) = \mathrm{val}_{\mathscr{G}}(r_A(A')). \tag{5}$$

Note that $s(A)$, r_A, and A' are well-defined since $A_{\triangleright}(n+1) = \mathrm{val}_{\mathscr{G}}(A[n+1])$ and $A[n+1]$ has rank zero.

Example 6 We give an example of a TSLP $\mathscr{G}$ that has two non-trivially equal subtrees in A and D respectively. The rules for the caterpillar tree that is derived from A are:

$$\begin{aligned} A &\to B\langle C\rangle \\ B &\to B_1\langle B_2\langle x\rangle\rangle \\ B_1 &\to g\langle x, K\rangle \\ B_2 &\to B_3\langle B_4\langle x\rangle\rangle \\ B_3 &\to f\langle I, x\rangle \\ B_4 &\to f\langle K, x\rangle \end{aligned}$$

Additionally, we have rules that generate a caterpillar tree from D:

$$\begin{aligned} D &\to E\langle H\rangle \\ E &\to E_1\langle E_2\langle x\rangle\rangle \\ E_1 &\to g\langle x, I\rangle \\ E_2 &\to E_3\langle E_4\langle x\rangle\rangle \\ E_3 &\to f\langle x, J\rangle \\ E_4 &\to f\langle x, K\rangle \end{aligned}$$

Finally, the rules for deriving caterpillar trees from I and J and some terminal rules are added:

$$\begin{aligned} I &\to L\langle K\rangle \\ L &\to f\langle H, x\rangle \\ J &\to M\langle C\rangle \\ M &\to f\langle K, x\rangle \\ C &\to c \\ K &\to k \\ H &\to h \end{aligned}$$

The reader can check that this TSLP is reduced. See Figure 5 for a visualization. The SLP $\mathscr{P}$ that is derived from $\mathscr{G}$ has the following rules:

$$\begin{aligned} A &\to BC \\ B &\to B_1B_2 \\ B_2 &\to B_3B_4 \\ D &\to EH \\ E &\to E_1E_2 \\ E_2 &\to E_3E_4 \\ I &\to LK \\ J &\to MC \end{aligned}$$

We have $\mathrm{val}_{\mathscr{P}}(A) = B_1B_3B_4C$ and $\mathrm{val}_{\mathscr{P}}(D) = E_1E_3E_4H$. Also, we have

$$A_{\triangleright}(1) = \mathrm{val}_{\mathscr{G}}(B_1\langle B_3\langle B_4\langle C\rangle\rangle\rangle) = g\langle f\langle f\langle h,k\rangle, f\langle k,c\rangle\rangle, k\rangle \text{ and}$$
$$D_{\triangleright}(1) = \mathrm{val}_{\mathscr{G}}(E_1\langle E_3\langle E_4\langle H\rangle\rangle\rangle) = g\langle f\langle f\langle h,k\rangle, f\langle k,c\rangle\rangle, f\langle h,k\rangle\rangle.$$

The equal subtrees, marked by circles in Figure 5, are

$$\begin{aligned} A_{\triangleright}(2) &= \mathrm{val}_{\mathscr{G}}(B_3\langle B_4\langle C\rangle\rangle) \\ &= f\langle f\langle h,k\rangle, f\langle k,c\rangle\rangle \\ &= \mathrm{val}_{\mathscr{G}}(E_3\langle E_4\langle H\rangle\rangle) \\ &= D_{\triangleright}(2). \end{aligned}$$

Moreover, $s(A) = 3$, $s(D) = 3$, $A' = J$, $D' = I$, $r_A = \mathrm{rhs}(A[2]) = f\langle I,x\rangle$, and $r_D = \mathrm{rhs}(D[2]) = f\langle x,J\rangle$.

Lemma 2 *For every nonterminal $A \in N_a$, we can compute $s(A), r_A$ and A' in polynomial time.*

Proof We first compute the position $s(A)$ as follows. Let $B_1,\dots,B_k$ be a list of all nonterminals of rank zero. We can compute the size $n_i = |\mathrm{val}_{\mathscr{G}}(B_i)|$ by a simple bottom-up computation. Let us assume w.l.o.g. that $n_1 \le n_2 \le \cdots \le n_k$. Note that there can be only one position $1 \le s_i \le \ell(A)$ $(1 \le i \le k)$ on the spine path of A such that the tree $A_{\triangleright}(s_i)$ has size n_i. This position (if it exists) can be found in polynomial time using binary search. Note that for a given position $1 \le s \le \ell(A)$ we can compute the size of the tree $A_{\triangleright}(s)$ in polynomial time by first computing a TSLP for this tree (using the second statement in Proposition 4) and then compute the size of the produced tree bottom-up. Once the positions $s_1,\dots,s_k$ are computed, we can compute in polynomial time TSLPs for the trees $t_i = A_{\triangleright}(s_i)$. The position $s(A)$ is the smallest $s_i \ge 2$ such that t_i is equal to one of the trees $\mathrm{val}_{\mathscr{G}}(B_j)$, and the latter can be checked in polynomial time by Proposition 3. This also yields the nonterminal A'. Finally, the right-hand side r_A can be computed by computing in polynomial time the symbol $B \in N_2$ in $\mathrm{val}_{\mathscr{P}}(A)$ at position $s(A)-1$ by the first statement of Proposition 4. Then, $r_A = \mathrm{rhs}(B)$.

For the following proof, let us first state a few simple observations: let $A,B \in N_a$ and $1 \le i < \ell(A)$, $1 \le j < \ell(B)$. The definition of $A_{\triangleright}(i)$ implies

$$A_{\triangleright}(i) = \mathrm{val}_{\mathscr{G}}(A[i]\langle A_{\triangleright}(i+1)\rangle). \tag{6}$$
$$\text{If } A_{\triangleright}(i+1) = B_{\triangleright}(j+1) \text{ and } A[i] = B[j], \text{ then } A_{\triangleright}(i) = B_{\triangleright}(j). \tag{7}$$
$$\text{If } A_{\triangleright}(i) = B_{\triangleright}(j) \text{ and } A[i] = B[j], \text{ then } A_{\triangleright}(i+1) = B_{\triangleright}(j+1). \tag{8}$$

Non-trivial equalities arise when right-hand sides do not place their variables in the same position. If for instance $A_1 \to f\langle x,D\rangle$ and $A_2 \to f\langle E,x\rangle$, then $\mathrm{val}_{\mathscr{G}}(A_1\langle E\rangle) = \mathrm{val}_{\mathscr{G}}(A_2\langle D\rangle)$. Here, the nonterminals D and E fill in for each other.

Proposition 5 *Let $D,E \in N_d$, $D \ne E$ and $t,u \in \mathscr{T}(\Sigma)$. If $\mathrm{val}_{\mathscr{G}}(D\langle t\rangle) = \mathrm{val}_{\mathscr{G}}(E\langle u\rangle)$, then there exist $A,B \in N$ of rank zero such that $\mathrm{val}_{\mathscr{G}}(A) = t$ and $\mathrm{val}_{\mathscr{G}}(B) = u$.*

Proof Let us assume that

$$\mathrm{rhs}(D) = f\langle D_1,\ldots,D_{i-1},x,D_{i+1},\ldots,D_n\rangle \text{ and}$$
$$\mathrm{rhs}(E) = g\langle E_1,\ldots,E_{j-1},x,E_{j+1},\ldots,E_m\rangle.$$

Since $\mathscr{G}$ is reduced, we have $\mathrm{val}_{\mathscr{G}}(D\langle t\rangle) = \mathrm{val}_{\mathscr{G}}(E\langle u\rangle)$ if and only if $f = g$ (and thus $m = n$), $i \neq j$, $D_k = E_k$ for $k \in \{1,\ldots,m\}\setminus\{i,j\}$, $t = \mathrm{val}_{\mathscr{G}}(E_i)$ and $u = \mathrm{val}_{\mathscr{G}}(D_j)$. Note that if $i = j$, then, since $\mathscr{G}$ is reduced, we would obtain $D = E$, which contradicts the assumption.

Lemma 3 *For all $A,B \in N_a$ and all $1 \le i < s(A)$, $1 \le j < s(B)$, the following two conditions are equivalent:*

(i) $A_{\triangleright}(i) = B_{\triangleright}(j)$
(ii) $\mathrm{val}_{\mathscr{P}}(A[i : s(A)-2]) = \mathrm{val}_{\mathscr{P}}(B[j : s(B)-2])$ *and* $r_A(A') = r_B(B')$.

Proof To obtain (i) from (ii), note that by (5) $\mathrm{val}_{\mathscr{G}}(r_A(A')) = \mathrm{val}_{\mathscr{G}}(r_B(B'))$ implies $A_{\triangleright}(s(A)-1) = B_{\triangleright}(s(B)-1)$. Repeated application of equation (7) implies $A_{\triangleright}(i) = B_{\triangleright}(j)$. Now assume that (i) holds, so $A_{\triangleright}(i) = B_{\triangleright}(j)$. By induction on i and j, we show that $\mathrm{val}_{\mathscr{P}}(A[i : s(A)-2]) = \mathrm{val}_{\mathscr{P}}(B[j : s(B)-2])$ and $r_A(A') = r_B(B')$.

Case 1. $i = s(A)-1$. Then equation (5) becomes

$$A_{\triangleright}(i) = \mathrm{val}_{\mathscr{G}}(r_A(A')) = \mathrm{val}_{\mathscr{G}}(A[s(A)-1]\langle A'\rangle).$$

By equation (6) we have $B_{\triangleright}(j) = \mathrm{val}_{\mathscr{G}}(B[j]\langle B_{\triangleright}(j+1)\rangle)$. Therefore, we obtain

$$\mathrm{val}_{\mathscr{G}}(A[s(A)-1]\langle A'\rangle) = \mathrm{val}_{\mathscr{G}}(B[j]\langle B_{\triangleright}(j+1)\rangle).$$

Now there are two cases: either $A[s(A)-1] = B[j]$ in which case $\mathrm{val}_{\mathscr{G}}(A') = B_{\triangleright}(j+1)$, or $A[s(A)-1] \neq B[j]$ in which case we obtain from Proposition 5 that there is a nonterminal of rank zero that expands to $B_{\triangleright}(j+1)$. In both cases, there is a nonterminal of rank zero that expands to $B_{\triangleright}(j+1)$. Since $j+1 \le s(B)$, we must have $j+1 = s(B)$, $B_{\triangleright}(j+1) = \mathrm{val}_{\mathscr{G}}(B')$, and $r_B = \mathrm{rhs}(B[j])$. Therefore we have $\mathrm{val}_{\mathscr{P}}(A[i : s(A)-2]) = \varepsilon = \mathrm{val}_{\mathscr{P}}(B[j : s(B)-2])$ and $\mathrm{val}_{\mathscr{G}}(r_A(A')) = \mathrm{val}_{\mathscr{G}}(r_B(B'))$. The latter implies $r_A(A') = r_B(B')$ since $\mathscr{G}$ is reduced.

Case 2. $j = s(B)-1$. This case is symmetric to Case 1.

Case 3. $i < s(A)-1$ and $j < s(B)-1$. We claim that $A[i] = B[j]$. Assume that $A[i] \neq B[j]$. From $A_{\triangleright}(i) = B_{\triangleright}(j)$ we obtain $\mathrm{val}_{\mathscr{G}}(A[i]\langle A_{\triangleright}(i+1)\rangle) = \mathrm{val}_{\mathscr{G}}(B[j]\langle B_{\triangleright}(j+1)\rangle)$ by equation (6). Proposition 5 implies that there are nonterminals of rank zero that expand to $A_{\triangleright}(i+1)$ and $B_{\triangleright}(j+1)$, respectively. This contradicts $i+1 < s(A)$ as well as $j+1 < s(B)$. Hence we have $A[i] = B[j]$. Because $A_{\triangleright}(i) = B_{\triangleright}(j)$ it follows from equation (8) that $A_{\triangleright}(i+1) = B_{\triangleright}(j+1)$. We can now conclude with induction.

Consider a valid sequence $\gamma \in (L \cup R \cup M)^*$ for $\mathscr{G}$. We can uniquely factorize γ as

$$\gamma = \gamma_1(A_1,k_1,B_1)\gamma_2(A_2,k_2,B_2)\cdots\gamma_{n-1}(A_{n-1},k_{n-1},B_{n-1})\gamma_n, \tag{9}$$

where $\gamma_i \in (L \cup R)^*$ $(1 \le i \le n)$ and $(A_i,k_i,B_i) \in M$ $(1 \le i \le n-1)$. To simplify the notation, let us set $B_0 = S$ (the start nonterminal of $\mathscr{G}$). Hence, every γ_i is either

empty or a valid B_{i-1}-sequence for the SLP $\mathscr{P}$, and we have defined the position $\mathrm{pos}(\gamma_i)$ in the string $\mathrm{val}_{\mathscr{P}}(B_{i-1})$ according to Section 3. It is easy to modify our traversal algorithms from the previous section such that for every $1 \leq i \leq n$ we store in the sequence γ also the nonterminal B_{i-1} and the number $\mathrm{pos}(\gamma_i)$ right after γ_i (if $\gamma_i \neq \varepsilon$), i.e., just before (A_i, k_i, B_i). The number $\mathrm{pos}(\gamma_i)$ has to be incremented (resp., decremented) each time one moves down (resp., up) in the spine path for B_{i-1}. We do not explicitly write these nonterminals and positions in valid sequences in order to not complicate the notation.

We would like to use Lemma 3 for equality checks. To achieve this, we have to assume that $\mathrm{pos}(\gamma_i) < s(B_{i-1})$ in (9) for every $1 \leq i \leq n$ with $\gamma_i \neq \varepsilon$ (this corresponds to the assumptions $1 \leq i < s(A)$ and $1 \leq j < s(B)$ in Lemma 3). This requires a modification of the traversal algorithms from the previous section as follows: assume that the current valid sequence is γ from (9). As remarked above, we store the numbers $\mathrm{pos}(\gamma_i)$ right after each γ_i. Assume that the final number $\mathrm{pos}(\gamma_n)$ has reached the value $s(B_{n-1}) - 1$ and we want to move to the ith child of the current node. We proceed as in Algorithm 4 with one exception: in case (ii) (Section 4) we would increase $\mathrm{pos}(\gamma_i)$ to $s(B_{n-1})$. To avoid this, we start a new valid sequence for the root of the tree $\mathrm{val}_{\mathscr{G}}(B'_{n-1})$. Note that by the definition of B'_{n-1}, this is exactly the tree rooted at the ith child of the node represented by γ. So, we can continue the traversal in the tree $\mathrm{val}_{\mathscr{G}}(B'_{n-1})$. Therefore, we continue with the sequence $\gamma \mid \mathrm{root}(B'_{n-1})$, where $\mid$ is a separator symbol, and the root-function is defined at the end of Section 4. The navigation to the parent node can be easily adapted as well. The only new case that we have to add to Algorithm 3 is for $\gamma = \alpha \mid \beta$, where $\beta \in (L \cup R)^+$. In that case, we compute $\beta' = \mathrm{left}(\beta)$ and return $\alpha \mid \beta'$ if β' is not undefined, and α otherwise. Thus, the separator symbol $\mid$ is treated in the same way as triples from M.

Let us now consider two sequences γ_1 and γ_2 (that may contain the separator symbol $\mid$ as explained in the previous paragraph). Let v_i be the node of $\mathrm{val}(\mathscr{G})$ represented by γ_i and let t_i be the subtree of $\mathrm{val}(\mathscr{G})$ rooted in v_i. We want to check in time $O(1)$ whether $t_1 = t_2$. We can first compute in time $O(1)$ the labels $\mathrm{label}(\gamma_1)$ and $\mathrm{label}(\gamma_2)$ of the nodes v_1 and v_2, respectively. In case one of these labels belongs to $\mathscr{F}_0$ (i.e., one of the nodes v_1, v_2 is a leaf) we can easily determine whether $t_1 = t_2$. Hence, we can assume that neither v_1 nor v_2 is a leaf. In particular we can assume that $\gamma_1 \neq \varepsilon \neq \gamma_2$ (recall that ε is a valid sequence only in case $\mathrm{val}(\mathscr{G})$ consists of a single node) and that neither γ_1 nor γ_2 ends with a triple from M. Let us factorize γ_i as $\gamma_i = \alpha_i \beta_i$, where β_i is the maximal suffix of γ_i that belongs to $(L \cup R)^*$. Hence, we have $\beta_1 \neq \varepsilon \neq \beta_2$.

Assume that β_i is a valid C_i-sequence of $\mathscr{P}$, and that $n_i = \mathrm{pos}(\beta_i)$. Thus, the suffix β_i represents the n_ith leaf of the derivation tree of $\mathscr{P}$ with root C_i. Recall that we store C_i and n_i at the end of the sequence γ_i. Hence, we have constant time access to C_i and n_i. We have $C_i \in N_a$ and $n_i < s(C_i)$. With the notation introduced before, we obtain $t_i = C_{i\triangleright}(n_i)$. Since $n_1 < s(C_1)$ and $n_2 < s(C_2)$, Lemma 3 implies that $t_1 = t_2$ if and only if the following two conditions hold:

(i) $\mathrm{val}_{\mathscr{P}}(C_1[n_1 : s(C_1) - 2]) = \mathrm{val}_{\mathscr{P}}(C_2[n_2 : s(C_2) - 2])$ and
(ii) $r_{C_1}(C'_1) = r_{C_2}(C'_2)$.

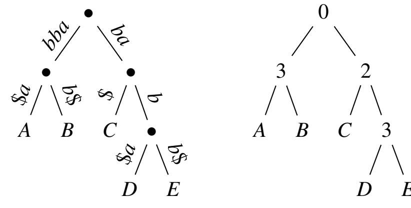

Fig. 6 The Patricia tree (left) and the modified Patricia tree (right) for Example 7

Condition (ii) can be checked in time $O(1)$, since we can precompute in polynomial time r_A and A' for every $A \in N_a$ by Lemma 2, and the length of each right-hand side is bounded by the maximal rank of the grammar, which is a constant. Now, let us concentrate on condition (i). First, we check whether $s(C_1) - n_1 = s(C_2) - n_2$. If not, then the lengths of $\mathrm{val}_{\mathscr{P}}(C_1[n_1 : s(C_1) - 2])$ and $\mathrm{val}_{\mathscr{P}}(C_2[n_2 : s(C_2) - 2])$ differ and we cannot have equality. Hence, assume that $k := s(C_1) - 1 - n_1 = s(C_2) - 1 - n_2$. Let ℓ be the length of the longest common suffix of $\mathrm{val}_{\mathscr{P}}(C_1[: s(C_1) - 2])$ and $\mathrm{val}_{\mathscr{P}}(C_2[: s(C_2) - 2])$. Then, it remains to check whether $k \leq \ell$. Clearly, in space $O(|\mathscr{G}|)$ we cannot store explicitly all these lengths ℓ for all $C_1, C_2 \in N_a$. Instead, we precompute in polynomial time a modified Patricia tree for the set of strings $w_A := \mathrm{val}_{\mathscr{P}}(A[: s(A) - 2])^{\mathrm{rev}}\$$ ($ is a new symbol that is appended in order to make the set of strings prefix-free and w^{rev} is the string w reversed) for $A \in N_a$. Then, we need to compute in time $O(1)$ the length of the longest common prefix for two of these strings w_A and w_B. Recall that the Patricia tree for a set of strings $w_1, \ldots, w_n$ is obtained from the trie for the prefixes of the w_i by eliminating nodes with a single child. But instead of labeling edges of the Patricia tree with factors of the w_i, we label every internal node with the length of the strings that lead from the root to the node. Let us give an example instead of a formal definition.

Example 7 Consider the strings $w_A = abba\$$, $w_B = abbb\$$, $w_C = ba\$$, $w_D = baba\$$ and $w_E = babb\$$. Figure 6 shows their Patricia tree (left) and the modified Patricia tree (right).

Since our modified Patricia tree has $|N_a|$ many leaves (one for each $A \in N_a$) and every internal node has at least two children, we have at most $2|N_a| - 1$ many nodes in the tree and every internal node is labeled with a $(\log|t|)$-bit number (note that the length of every string $\mathrm{val}_{\mathscr{P}}(A)$ $(A \in N_a)$ is bounded by $|t|$). Hence, on the word RAM model we can store the modified Patricia tree in space $O(|\mathscr{G}|)$. Finally, the length of the longest common prefix of two string w_A and w_B can be obtained by computing the lowest common ancestor of the two leaves corresponding to the strings w_A and w_B in the modified Patricia tree. The number stored in the lowest common ancestor is the length of the longest common prefix of w_A and w_B. Using a data structure for computing lowest common ancestors in time $O(1)$ [1,24] we obtain an $O(1)$-time implementation of subtree equality checking. Finally, from Proposition 4 it follows that the modified Patricia tree for the strings w_A $(A \in N_a)$ can be precomputed in polynomial time.

The following theorem summarizes the main result of this section.

Theorem 2 *Given a monadic TSLP $\mathscr{G}$ for a tree $t = \mathrm{val}(\mathscr{G})$ we can compute in polynomial time on a word RAM with register length $O(\log|t|)$ a data structure of size $O(|\mathscr{G}|)$ that allows to carry out the following computations in time $O(1)$, where γ and γ' are valid sequences (as modified in this section) that represent the tree nodes v and v', respectively:*

- *compute the valid sequence for the parent node of v (if it exists),*
- *compute the valid sequence for the ith child of v (if it exists), and*
- *check whether the subtrees rooted in v and v' are equal.*

6 Discussion

We have presented a linear-space data structure for grammar-compressed ranked trees, that provides constant-time traversal operations over the represented tree, and constant-time equality check for subtrees of the represented tree. The solution is based on the ideas of [10] and the fact that next link queries can be answered in time $O(1)$ (after linear time preprocessing). It would be interesting to develop an efficient implementation of the technique. Next link queries can be implemented in many different ways. The solution given in [10] is based on a variant of the lowest common ancestor algorithm due to Schieber and Vishkin [24] (described in [13]). Another solution is to use *level-ancestor queries* (together with depth-queries), as are available in implementations of succinct tree data structures (e.g., the one of Navarro and Sadakane [23]). Yet another alternative is to store the first-child/next-sibling encoded binary tree of the original tries. The first-child/next-sibling encoding is defined for ordered trees. In our situation, we have to answer next-link queries for the tries $T_L(a)$ and $T_R(a)$ for $a \in \Sigma$, which are unordered. Hence we order the children of a node in an arbitrary way. Then the next link of v_1 above v_2 is equal to the *lowest common ancestor* of v_2 and the last child of v_1 in the original tree. This observation allows to use simple and efficient lowest common ancestor data structures like for instance the one of Bender and Farach-Colton [1].

Recall that we use polynomial time preprocessing to build up our data structure for subtree equality checks. It remains to come up with a more precise time bound. We already argued that the problem is at least as difficult as checking equality of SLP-compressed strings, for which the best known algorithm is quadratic. It would be interesting to show that our preprocessing has the same time complexity as checking equality of SLP-compressed strings.

References

1. M. A. Bender and M. Farach-Colton. The LCA problem revisited. In *Proceedings of LATIN 2000*, volume 1776 of Lecture Notes in Computer Science, pages 88–94. Springer, 2000.
2. P. Bille, I. L. Gørtz, G. M. Landau, and O. Weimann. Tree compression with top trees. *Information and Computation*, 243:166–177, 2015.
3. P. Bille, G. M. Landau, R. Raman, K. Sadakane, S. R. Satti, and O. Weimann. Random access to grammar-compressed strings and trees. *SIAM Journal on Computing*, 44(3):513–539, 2015.
4. M. Bousquet-Mélou, M. Lohrey, S. Maneth, and E. Noeth. XML compression via DAGs. *Theory of Computing Systems*, 57(4):1322–1371, 2014.

5. G. Busatto, M. Lohrey, and S. Maneth. Efficient memory representation of XML document trees. *Information Systems*, 33(4–5):456–474, 2008.
6. J. Cai and R. Paige. Using multiset discrimination to solve language processing problems without hashing. *Theoretical Computer Science*, 145(1&2):189–228, 1995.
7. M. Charikar, E. Lehman, A. Lehman, D. Liu, R. Panigrahy, M. Prabhakaran, A. Sahai, and A. Shelat. The smallest grammar problem. *IEEE Transactions on Information Theory*, 51(7):2554–2576, 2005.
8. H. Comon, M. Dauchet, R. Gilleron, F. Jacquemard, D. Lugiez, C. Löding, S. Tison, and M. Tommasi. Tree automata techniques and applications, 2007. Available at `http://tata.gforge.inria.fr/`
9. O. Delpratt, R. Raman, and N. Rahman. Engineering succinct DOM. In *Proceedings of EDBT*, volume 261 of ACM International Conference Proceeding Series, pages 49–60, ACM, 2008.
10. L. Gasieniec, R. M. Kolpakov, I. Potapov, and P. Sant. Real-time traversal in grammar-based compressed files. In *Proceedings of DCC 2005*, page 458, IEEE Computer Society, 2005. Long version availabe at `http://www.csc.liv.ac.uk/~leszek/papers/dcc05.ps.gz`
11. L. Hübschle-Schneider and Rajeev Raman. Tree compression with top trees revisited. In *Proceedings of SEA 2015*, volume 9125 of Lecture Notes in Computer Science, pages 15–27. Springer, 2015. Long version available at `http://arxiv.org/abs/1506.04499`
12. D. Hucke, M. Lohrey, and E. Noeth. Constructing small tree grammars and small circuits for formulas. In *Proceedings of FSTTCS 2014*, volume 29 of *LIPIcs*, pages 457–468. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2014.
13. D. Gusfield. Algorithms on Strings, Trees, and Sequences - Computer Science and Computational Biology. Cambridge University Press, 1997.
14. A. Jeż. Faster fully compressed pattern matching by recompression. *ACM Transactions on Algorithms*, 11(3):20:1–20:43, 2015.
15. A. Jeż and M. Lohrey. Approximation of smallest linear tree grammars. In *Proceedings of STACS 2014*, volume 25 of *LIPIcs*, pages 445–457. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2014.
16. D. Knuth. The Art of Computer Programming, Vol I: Fundamental Algorithms. Addison-Wesley, 1968.
17. M. Lohrey. Algorithmics on SLP-compressed strings: A survey. *Groups Complexity Cryptology*, 4(2):241–299, 2012.
18. M. Lohrey and S. Maneth. The complexity of tree automata and XPath on grammar-compressed trees. *Theoretical Computer Science*, 363(2):196–210, 2006.
19. M. Lohrey, S. Maneth, and R. Mennicke. XML tree structure compression using RePair. *Information Systems*, 38(8):1150–1167, 2013.
20. M. Lohrey, S. Maneth, and M. Schmidt-Schauß. Parameter reduction and automata evaluation for grammar-compressed trees. *Journal of Computer and Systems Science*, 78(5):1651–1669, 2012.
21. S. Maneth and T. Sebastian. Fast and tiny structural self-indexes for XML. *CoRR*, abs/1012.5696, 2010.
22. S. Maneth and T. Sebastian. XPath node selection over grammar-compressed trees. In *Proceedings of TTATT 2013*, Electronic Proceedings in Theoretical Computer Science 134, pages 38–48, 2013.
23. G. Navarro and K. Sadakane. Fully functional static and dynamic succinct trees. *ACM Transactions on Algorithms*, 10(3):16, 2014.
24. B. Schieber and U. Vishkin. On Finding Lowest Common Ancestors: Simplification and Parallelization. *SIAM Journal on Computing*, 17(6):1253–1262, 1988.
25. T. Scwentick. Automata for XML - A survey. *Journal of Computer and Systems Science*, 73(3):289–315, 2007.