Grammar-Based Tree Compression*

Markus Lohrey

Universität Siegen
lohrey@eti.uni-siegen.de

Abstract. This paper gives a survey on recent progress in grammar-based compression for trees. Also algorithms that directly work on grammar-compressed trees will be surveyed.

1 Introduction

Trees are an omnipresent data structure in computer science. Large trees occur for instance in XML processing or automated deduction. For certain appliations it is important to work with compact tree representations. A widely studied standard compact tree representation is the *dag* (*directed acyclic graph*), see e.g. [5, 10–12, 38]. A dag is basially a folded tree, where nodes may share children. The tree represented by a dag is obtained by unfolding the dag. One of the nice things about dags is that every tree has a unique minimal (or smallest) dag that can be computed in linear time [10]. In the minimal dag of a tree t, isomorphic subtrees of t are represented only once. Figure 1 shows a tree and its minimal dag (we consider ranked ordered trees, where every node is labelled with a symbol, whose rank determines the number of children of the node). In [38], dags are used to obtain a universal (in the information theoretic sense) compressor for binary trees under certain distributions. Dags can achieve exponential compression in the best case: The minimal dag of a full binary tree of height n is a linear chain of length n.

In recent years, another compact tree representation that generalizes dags has been studied: Tree straight-line programs, briefly TSLPs. Whereas dags can only share repeated complete subtrees, TSLPs can also share repeated occurrences of subtrees with gaps (i.e., subtrees, where some smaller subtrees are removed). A TSLP can be seen as a very restricted context-free tree grammar that produces exactly one tree. It consists of rewrite rules (productions) of the form $A(x_1, \ldots, x_k) \to t(x_1, \ldots, x_k)$. Here, A is a nonterminal of rank k and $x_1, \ldots, x_k$ are parameters that are replaced by concrete trees in the application of this rule. The nodes of the tree $t(x_1, \ldots, x_k)$ are labelled with terminal symbols (the node labels of the tree produced by the TSLP), nonterminal symbols and the parameters $x_1, \ldots, x_k$. There is a distinguished start nonterminal S of rank 0. To produce a single tree, it is required that (i) for every nonterminal A there is exactly one rule with A on the left-hand side, and (ii) that from a nonterminal A one cannot reach A by more than one rewrite step. Finally, it is

* This research is supported by the DFG-project LO 748/10-1.

required that every parameter x_i appears at most once in the right-hand side of A (linearity). Dags can be seen as TSLPs, where every nonterminal has rank zero. As for dags, TSLPs allow exponential compression in the best case, but due to the ability to share also internal patterns, one can easily come up with examples where the minimal dag is exponentially larger than the smallest TSLP, see Section 2.

TSLPs generalize straight-line programs for words (SLPs). These are context-free grammars that produce a single word. There exist several grammar-based string compressors that produce (a suitable encoding of) an SLP for an input word. Prominent examples are LZ78, RePair, Sequitur, and BiSection. Theoretical results on the compression ratio of these algorithmis can be found in [8]. Over the last couple of years, the idea of grammar-based compression has been extended from words to trees. In Section 3 we will discuss several grammar-based tree compressors based on TSLPs.

SLPs and TSLPs are a simple and mathematically clean data structure. These makes them well-suited for the development of efficient algorithms on compressed objects. The goal of such algorithms is to manipulate and analyze compressed objects and thereby beat a naive decompress-and-compute strategy, where the uncompressed object is first computed and then analyzed. A typical example for this is pattern matching. Here, we have a large text, which is stored in compressed form and want to locate occurrences of a pattern (which is usually given in explicit form) in the text. But algorithms on compressed objects can be also useful for problems, where we do not directly deal with compression. In many algorithms huge intermediate data structures have to be stored, which are the main bottleneck in the computation. An obvious potential solution in such a situation is to store these intermediate data structures in a compressed way.

A survey on algorithms that work on SLP-compressed words can be found in [24], which contains a section on algorithms on TSLP-compressed trees as well. In Section 4 we give a more detailed and up-to-date survey on algorithms for trees that are represented by TSLPs.

2 Tree straight-line programs

For background on trees and tree grammars see [9]. Here, trees are rooted, ordered and node-labelled. Every node has a label from a finite alphabet Σ. Moreover, with every symbol $a \in \Sigma$ a natural number (the rank of a) is associated. Symbols of rank zero are called *constants* and symbols of rank one are *unary*. If a tree node v is labelled with a symbol of rank n, then v has exactly n children, which are linearly ordered. Such trees can conveniently be represented as terms. The size $|t|$ of a tree t is the number of nodes of t. Here is an example:

Example 1. Let f be a symbol of rank 2, h a symbol of rank 1, and a a symbol of rank 0 (a constant). Then the term $h(f(h(f(h(h(a)),a)),h(f(h(h(a)),a))))$ corresponds to the tree of size 14, shown in Figure 1.

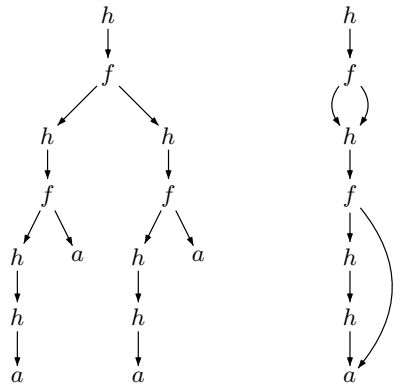

Fig. 1. A node-labelled tree and its minimal dag

A tree straight-line program (TSLP for short and also called SLCF tree grammar in [25, 28] or SLT grammar in [27]) over the terminal alphabet Σ (which is a ranked alphabet in the above sense) is a tuple $\mathcal{G} = (N, \Sigma, S, P)$, such that

- N is a finite set of ranked symbols (the nonterminals) with $N \cap \Sigma = \emptyset$,
- $S \in N$ has rank 0 (the initial nonterminal),
- and P is a finite set of productions of the form $A(x_1, \ldots, x_n) \to t$ where A is a nonterminal of rank n and t is a tree built up from the ranked symbols in $\Sigma \cup N$ and the parameters $x_1, \ldots, x_n$ which are considered as symbols of rank 0 (i.e., constants). Every x_i is required to appear exactly once in t. Moreover, it is required that every nonterminal occurs on the left-hand side of exactly one production, and that the relation $\{(A, B) \in N \times N \mid (A(x_1, \ldots, x_n) \to t) \in P, B \text{ occurs in } t\}$ is acyclic.

A TSLP $\mathcal{G}$ generates a tree $\mathrm{val}(\mathcal{G})$ in the natural way. During the derivation process, the parameters $x_1, \ldots, x_n$ are instantiated with concrete trees. Instead of giving a formal definition, let us consider an example.

Example 2. Let S, A, B, C be nonterminals, let S be the start nonterminal and let the TSLP $\mathcal{G}$ consist of the following productions:

$$\begin{aligned} S &\to A(B(a), B(a)) \\ A(x_1, x_2) &\to C(C(x_1, a), C(x_2, a)) \\ C(x_1, x_2) &\to h(f(x_1, x_2)) \\ B(x) &\to h(h(x)) \end{aligned}$$

Then $\mathrm{val}(G)$ is the tree from Example 1. It can be derived as follows:

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

The size of a TSLP $\mathcal{G} = (N, \Sigma, S, P)$ is defined as the total number of all nodes in right-hand sides of P, where nodes labelled with a parameter are not counted (see [17] for a discussion of this). Hence, the size of the TSLP in Example 2 is 14. It is easy to show that the size of tree $\mathrm{val}(\mathcal{G})$ is bounded by $2^{O(|\mathcal{G}|)}$.

The following result from [28] turned out to be very useful for algorithmic problems on trees that are represented by TSLPs:

Theorem 1 ([28]). *From a given TSLP $\mathcal{G} = (N, \Sigma, S, P)$, where every $A \in N$ has rank at most k and every $\sigma \in \Sigma$ has rank at most r, one can compute in time $O(k \cdot r \cdot |\mathcal{G}|)$ a TSLP $\mathcal{H}$ of size $O(r \cdot |\mathcal{G}|)$ such that (i) $\mathrm{val}(\mathcal{G}) = \mathrm{val}(\mathcal{H})$ and every nonterminal of $\mathcal{H}$ has rank at most one.*

This result is sharp in the sense that transforming a TSLP into an equivalent TSLP where every nonterminal has rank zero involves an exponential blow-up in the size of the TSLP. For instance, the tree $t_n = f^{2^n}(a)$ with 2^n many occurrences of the unary symbol f can be produced by a TSLP of size $O(n)$ ($S \to A_n(a)$, $A_i(x) \to A_{i-1}(A_{i-1}(x))$ for $1 \leq i \leq n$, and $A_0(x) \to f(x)$) but the minimal dag for t_n is t_n itself. Note that a dag can be transformed into a TSLP, where every nonterminal has rank zero: The nodes of the dag are the nonterminals of the TSLP, and if a σ-labelled node v has the children $v_1, \ldots, v_n$ in the dag (from left to right), then we introduce the production $v \to \sigma(v_1, \ldots, v_n)$. Similarly, a TSLP, where every nonterminal has rank zero, can be transformed into a dag of the same size.

In [25, 27, 28], also *non-linear* TSLPs were studied. A non-linear TSLP may contain productions of the form $A(x_1, \ldots, x_k) \to t$, where a parameter x_i occurs several times in t. Non-linear TSLPs can achieve double exponential compression: The non-linear TSLP with the productions $S \to A_n(a)$, $A_i(x) \to A_{i-1}(A_{i-1}(x))$ for $1 \leq i \leq n$, and $A_0(x) \to f(x, x)$ produces a full binary tree of height 2^n and hence has $2^{2^n+1} - 1$ many nodes.

TSLPs generalize SLPs, which produce words instead of trees. In SLPs, symbols do not have a rank, and the productions are simply of the form $A \to w$, where w consists of terminal symbols and nonterminals. It is required again that every nonterminal occurs on the left-hand side of exactly one production, and that the relation $\{(A, B) \mid (A \to w)$ is a production and nonterminal B occurs in $w\}$ is acyclic. More details on SLPs can be found in [24].

3 Constructing small TSLPs

Efficient algorithms that generate for a given input tree t a linear TSLP $\mathcal{G}$ with $\mathrm{val}(\mathcal{G}) = t$ are described in [6, 26]. The algorithm from [26], called *TreeRePair*, is an extension of the grammar-based string compressor RePair [20] to trees. On a collection of XML skeleton trees (where the data values were removed) the compression ratio of TreeRePair (measured in the size of the computed TSLP divided by the number of edges of the input tree) was about 2.8 %, whereas for the same data set the compression ratio achieved by the minimal dag (number of edges of the minimal dag divided by the number of edges of the input tree) is about 12%, see [26].

Altough TreeRePair works very well on real XML data, its performance is quite poor from a theoretical viewpoint: In [26], a familiy of (binary) trees t_n ($n \geq 1$) is constructed, such that (i) t_n has size $O(n)$, (ii) a TSLP of size $O(\log n)$ for t_n exists, but (iii) the TSLP for t_n computed by TreeRePair has size $\Omega(n)$.

For a tree t let $\mathsf{opt}(t)$ be the size of a smallest TSLP for the tree t. Similarly, for a word s let $\mathsf{opt}(s)$ be the size of a smallest SLP for the word s. It was shown in [8] that unless $\mathsf{P} = \mathsf{NP}$ there is no polynomial time algorithm that computes for a given word s an SLP of size less than $8569/8568 \cdot \mathsf{opt}(s)$. The same result holds also for trees: Simply encode a word by the tree consisting of unary nodes and a single leaf. A TSLP for this tree is basically an SLP for the original word. For SLPs the best known polynomial time grammar-based compressors achieve an approximation ratio of $O(\log(\frac{n}{\mathsf{opt}(s)}))$, i.e., the size of the computed SLP for an input word s of length n is bounded by $O(\mathsf{opt}(s) \cdot \log(\frac{n}{\mathsf{opt}(s)}))$ [8, 16, 32, 33]. Recently, this bound has been also shown for trees:

Theorem 2 ([17]). *From a given tree t of size n, one can compute in linear time a TSLP $\mathcal{G}$ of size $O(r \cdot g + r \cdot g \cdot \log(\frac{n}{r \cdot g}))$ such that $\mathrm{val}(\mathcal{G}) = t$. Here, $g = \mathsf{opt}(t)$ and r is the maximal rank of a node label in t.*

The algorithm from [17] uses three different types of compression operations that are executed repeatedly on the current tree in the following order as long as the tree has size at least two. At the same time we build up the TSLP for the input tree.

Chain compression: For every unary symbol a, we replace every maximal occurrence of a pattern $a^n(x)$ (maximal means that the parent node of the topmost a-node is not labelled with a and also the unique child of the deepest a-node is not labelled with a) by a single tree that is labelled with a fresh unary symbol a_n. We call such maximal patterns *maximal a-chains*. Moreover, we add to the TSLP productions that generate from the nonterminal $a_n(x)$ the a-chain $a^n(x)$. These productions basically form an SLP for a^n. If $a^{n_1}(x), a^{n_2}(x), \ldots, a^{n_k}(x)$ are all maximal a-chains in the current tree with $n_1 < n_2 < \cdots < n_k$, then the total size of all productions needed to produce these chains can be bounded by $O(k + \sum_{i=1}^{k} \log(n_i - n_{i-1})$ with $n_0 = 0$.

Pair compression: After the chain compression step, there do not exist occurrences of a pattern $a(a(x))$ in the current tree for a unary symbol a. Let Σ_1 be

the set of all unary symbols that appear in the current tree. We first compute a partition $\Sigma_1 = \Sigma_{0,1} \cup \Sigma_{1,1}$. Then, every occurrence of a pattern $a(b(x))$ with $a \in \Sigma_{0,1}$ and $b \in \Sigma_{1,1}$ is replaced by a single node labelled with the fresh unary symbol $c_{a,b}$. Moreover, we introduce the TSLP-production $c_{a,b}(x) \to a(b(x))$. The partition $\Sigma_1 = \Sigma_{0,1} \cup \Sigma_{1,1}$ is chosen such that the number of occurrences of a pattern $a(b(x))$ with $a \in \Sigma_{0,1}$ and $b \in \Sigma_{1,1}$ is large. More precisely, one can choose the partition such that there are at most $(n_1 - c + 2)/4$ such occurrences, where n_1 is the number of unary nodes in the current tree and c is the number of maximal chains consisting of unary nodes.

Leaf compression: We eliminate all leaves of the current tree as follows: Let v be an f-labelled node such that v has at least one leaf among its children. Let $n \geq 1$ be the rank of f, and let $1 \leq i_1 < i_2 < \cdots < i_k \leq n$ be the positions of the leaves among the children of v. Let a_j be the label (a constant) of the i_j^{th} child of v. Then we remove all children of v, which are leaves, and replace the label f of v by the fresh symbol $f_{i_1,a_1,\ldots,i_k,a_k}$, which has rank $n - k$. Moreover, we add to the TSLP the production $f_{i_1,a_1,\ldots,i_k,a_k} \to f(x_1, \ldots, x_{i_1-1}, a_1, x_{i_1+1}, \ldots x_{i_2-1}, a_2, x_{i_2+1}, \ldots, x_{i_k-1}, a_k, x_{i_k+1}, \ldots, x_n)$.

Chain compression and pair compression are the two compression steps in Jeż's string compressor from [16]. They allow to shrink chains of unary nodes. Intuitively, if there are no long chains of unary nodes in the tree, then there must be many leaves and leaf compression will shrink the size of the tree substantially. More precisely, it can be shown that in a single phase, consisting of chain compression, followed by pair compression, followed by leaf compression, the size of the tree drops by a constant factor. This allows to come up with a linear bound on the running time. To bound the size of the produced TSLP and, in particular, to compare it with the size of a smallest TSLP for the input tree, Jeż's recompression technique is used in [17].

To the knowledge of the author, there is no algorithm for computing a small non-linear TSLP for a given input tree and thereby achieves a reasonable approximation ratio. This raises the question of whether the size of a smallest non-linear TSLP can be approximated in polynomial time up to a factor of say $\log n$ (assuming reasonable assumptions from complexity theory).

It is well known that for every word $w \in \Sigma^*$ there exists an SLP for w of size $O(\frac{n}{\log_\sigma n})$, where $\sigma = |\Sigma|$. Examples of grammar-based compressors that achieve this bound are for instance LZ78 or BiSection [18]. A simple information theoretic argument shows that the bound $O(\frac{n}{\log_\sigma n})$ is optimal. By the following result from [15] the same bound holds also for binary trees and TSLPs.

Theorem 3 ([15]). *From a given tree t of size n, where every terminal symbol has rank at most 2, one can compute in linear time a TSLP $\mathcal{G}$ of size $O(\frac{n}{\log_\sigma n})$ such that $\mathrm{val}(\mathcal{G}) = t$. Here, σ is the number of different node labels that appear in t.*

In [15], only the bound $O(n \log n)$ on the running time is stated. The linear time algorithm will appear in a long version of [15]. Let us briefly sketch the linear

time algorithm for a binary trees t with node labels from a set Σ ($|\Sigma| = \sigma$). The algorithm works in two steps:

Step 1. We decompose the tree t into $O(\frac{n}{\log_\sigma n})$ many clusters (connected subgraphs) of size at most $c \cdot \log_\sigma n$ for a constant c that will be chosen later. Each cluster is a full subtree of t with at most two full subtrees of t removed from it. Hence, we can write such a cluster as a tree $u(x_1, \ldots, x_k)$ with $k \geq 2$, where every parameter x_i appears exactly once. We replace each cluster $u(x_1, \ldots, x_k)$ in t by a single node labelled with a nonterminal A_u of rank at most two, and introduce the production $A_u(x_1, \ldots, x_k) \to u(x_1, \ldots, x_k)$. Note that the resulting tree s has size $O(\frac{n}{\log_\sigma n})$. We add the production $S \to s$, where S is the start nonterminal. With some care, this first step can be done in linear time.

Step 2. The TSLP we obtain from the previous step has size $O(n)$, so nothing is gained. We now compute in linear time (using [10]) the minimal dag for the forest consisting of all cluster trees $u(x_1, \ldots, x_k)$. Recall that each such tree has size at most $c \cdot \log_\sigma n$. Hence, to bound the size of the minimal dag of this forest, one only has to count the number of binary trees of size at most $c \cdot \log_\sigma n$, where every node is labelled with a symbol from $\Sigma \cup \{x_1, x_2\}$. By choosing the constant c suitably, we can (using the formula for the number of binary trees of size m, which is given by the Catalan numbers) bound this number by $\sqrt{n}$. The minimal dag for the cluster trees together with the start production $S \to s$ translates into a TSLP for t of size $\sqrt{n} + O(\frac{n}{\log_\sigma n}) = O(\frac{n}{\log_\sigma n})$.

Theorem 3 can be generalized to trees of higher rank. Then the constant hidden in the big-O-notation depends on the maximal rank of a terminal symbol, but the precise dependence is not analyzed in [15].

A simple information theoretic argument shows that the average size of a minimal TSLP for a uniformly chosen tree of size n with labels from an alphabet of size σ is $\Omega(\frac{n}{\log_\sigma n})$ and hence, by Theorem 3, $\Theta(\frac{n}{\log_\sigma n})$. In [11] it is shown that the average size of the minimal dag of a uniformly chosen binary tree with n unlabelled nodes is $\Theta(\frac{n}{\sqrt{\log n}})$. In [3] this result is extended to node-labelled unranked trees.

With some additional effort, one can ensure that the TSLP $\mathcal{G}$ in Theorem 3 has height $O(\log n)$. This has an interesting application for the problem of transforming arithmetical expressions into circuits (i.e., dags). Let $\mathbb{S} = (S, +, \cdot)$ be a (not necessarily commutative) semiring. Thus, $(S, +)$ is a commutative monoid with identity element 0, $(S, \cdot)$ is a monoid with identity element 1, and $\cdot$ left and right distributes over $+$. An *arithmetical expression* is just a labelled binary tree where internal nodes are labelled with the semiring operations $+$ and $\cdot$, and leaf nodes are labelled with variables $y_1, y_2, \ldots$ or the constants 0 and 1. An *arithmetical circuit* is a (not necessarily minimal) dag whose internal nodes are labelled with $+$ and $\cdot$ and whose leaf nodes are labelled with variables or the constants 0 and 1. The *depth* of a circuit is the length of a longest path from the root node to a leaf. An arithmetical circuit evaluates to a multivariate noncommutative polynomial $p(y_1, \ldots, y_n)$ over $\mathbb{S}$, where $y_1, \ldots, y_n$ are the variables occurring at the leaf nodes. Two arithmetical circuits are equivalent

if they evaluate to the same polynomial. Brent [4] has shown that every arithmetical expression of size n over a commutative ring can be transformed into an equivalent circuit of depth $O(\log n)$ and size $O(n)$ (the proof easily generalizes to semirings). Using Theorem 3 one can refine the size bound to $O(\frac{n \cdot \log m}{\log n})$, where m is the number of different variables in the formula:

Theorem 4 ([15]). *A given arithmetical expression F of size n having m different variables can be transformed in time $O(n)$ into an arithmetical circuit C of depth $O(\log n)$ and size $O(\frac{n \cdot \log m}{\log n})$ such that over every semiring, C and F evaluate to the same noncommutative polynomial (in m variables).*

To show Theorem 4 one first transforms the arithmetical expression into a TSLP of size $O(\frac{n}{\log_m n}) = O(\frac{n \cdot \log m}{\log n})$. Then one transforms the TSLP into a circuit that evaluates to the same polynomial (over any semiring) as the TSLP. Only for this second step, one has to use the semiring structure.

There are also some other tree compressors that use grammar formalisms slightly different from TSLPs. In [1] so called elementary ordered tree grammars are used, and a polynomial time compressor with an approximation ratio of $O(n^{5/6})$ is presented. Also the *top dags* from [2] can be seen as a variation of TSLPs for unranked trees. In [2] it was shown that for every tree t of size n the top dag has size $O(\frac{n}{\log^{0.19} n})$. An extension of TSLPs to higher order tree grammars was proposed in [19].

4 Algorithmic problems for TSLP-compressed trees

Let us now consider algorithmic problems for TSLP-compressed trees. Probably the most basic question is whether two trees, both given by TSLPs, are equal.

Theorem 5 ([6, 34]). *For two given TSLPs $\mathcal{G}$ and $\mathcal{H}$ it can be checked in polynomial time, whether* $\mathrm{val}(\mathcal{G}) = \mathrm{val}(\mathcal{H})$.

For the proof of Theorem 5 one constructs in polynomial time from a TSLP $\mathcal{G}$ an SLP $\mathcal{G}'$ such that $\mathrm{val}(\mathcal{G}')$ represents a depth-first left-to-right transversal of the tree $\mathrm{val}(\mathcal{G})$. For this, $\mathcal{G}'$ contains $k+1$ nonterminals $A_{0,1}, A_{1,2}, A_{2,3}, \ldots, A_{k-1,k}$, $A_{k,0}$ for a rank-k nonterminal A of $\mathcal{G}$. Intuitively, $A_{0,1}$ produces the part of the traversal of $\mathrm{val}_{\mathcal{G}}(A)$ from the root of $\mathrm{val}_{\mathcal{G}}(A)$ to the position of the first parameter, $A_{i,i+1}$ $(1 \leq i \leq k-1)$ produces the part of the traversal from the position of the i^{th} parameter to the position of the $(i+1)^{\text{th}}$ parameter, and $A_{k,0}$ produces the part of the traversal from the position of the k^{th} parameter back to the root. For the TSLP from Example 2 we obtain the following SLP:

$$S_{0,0} \to A_{0,1}B_{0,1}aB_{1,0}A_{1,2}B_{0,1}aB_{1,0}A_{2,0}$$
$$A_{0,1} \to C_{0,1}C_{0,1}, \quad A_{1,2} \to C_{1,2}aC_{2,0}C_{1,2}C_{0,1}, \quad A_{2,0} \to C_{1,2}aC_{2,0}C_{2,0}$$
$$C_{0,1} \to hf, \quad C_{1,2} \to \varepsilon, \quad C_{2,0} \to \varepsilon$$
$$B_{0,1} \to hh, \quad B_{1,0} \to \varepsilon$$

For a ranked tree, its depth-first left-to-right transversal uniquely represents the tree. Therefore, for two TSLPs $\mathcal{G}$ and $\mathcal{H}$ we have $\mathrm{val}(\mathcal{G}) = \mathrm{val}(\mathcal{H})$ if and only if $\mathrm{val}(\mathcal{G}') = \mathrm{val}(\mathcal{H}')$. Hence, equality of trees that are represented by linear TSLPs can be reduced to checking equality of SLP-compressed words, which can be checked in polynomial time by a famous result of Plandowski [31] (which has been indendently shown in [14, 30]). In [34], Theorem 5 is shown by a direct extension of Plandowski's algorithm for SLPs.

It is open whether Theorem 5 can be extended to non-linear TSLPs. For these, the best upper bound on the equivalence problem is PSPACE [6] and no good lower bound is known.

In [13], Theorem 5 has been extended to the unification problem. Unification is a classical problem in logic and deduction. One considers trees s and t with distinguished variables (these variables should be not confused with the parameters in TSLPs), which label leaf nodes. The trees s and t are unifiable if there exists a substitution that maps every variable x appearing in s or t to a variable-free tree (also called ground term) such that $\sigma(s) = \sigma(t)$. Here, $\sigma(s)$ (resp., $\sigma(t)$) denotes the tree that is obtained by replacing every x-labelled leaf of s (resp., t) by the tree $\sigma(x)$. The following result has been shown in [13]:

Theorem 6 ([13]). *For two given TSLPs $\mathcal{G}$ and $\mathcal{H}$ (where some of the terminal symbols of rank 0 are declared as variables) it can be checked in polynomial time, whether $\mathrm{val}(\mathcal{G})$ and $\mathrm{val}(\mathcal{H})$ are unifiable.*

In fact, the representation of the most general unifier of $\mathrm{val}(\mathcal{G})$ and $\mathrm{val}(\mathcal{H})$ in terms of TSLPs for the variables is computed in [13] in polynomial time.

In [36], the authors studied the compressed submatching problem: The input consists of TSLPs $\mathcal{G}$ (the pattern TSLP) and $\mathcal{H}$, where some of the terminal symbols of rank 0 appearing in $\mathcal{G}$ are declared as variables, and it is asked whether there exists a substitution σ such that $\sigma(\mathrm{val}(\mathcal{G}))$ is a subtree of $\mathrm{val}(\mathcal{H})$. Whereas the complexity of the general compressed submatching problem is still open (the best upper bound is NP), Schmidt-Schauß proved in [36]:

Theorem 7 ([36]). *Compressed submatching can be solved in polynomial time, if (i) no variable appears more than once in the tree produced by the pattern TSLP (i.e., this tree is linear) or (ii) all nonterminals in the pattern TSLP have rank zero (i.e., the pattern TSLP is in fact a dag).*

So far, we considered ordered trees, where the children of a node are linearly ordered. Deciding isomorphism of unordered trees, where the children are not ordered is more difficult than for ordered trees. For explicitly given unordered trees, isomorphism can be decided in logspace by a result of Lindell [22]. For unordered trees that are given by dags, one can solve the isomorphism problem by a simple partition refinement algorithm [29]. Recently this result has been extended to unordered trees that are represented by TSLPs [27]:

Theorem 8 ([27]). *For two given TSLPs $\mathcal{G}$ and $\mathcal{H}$ it can be checked in polynomial time, whether $\mathrm{val}(\mathcal{G})$ and $\mathrm{val}(\mathcal{H})$ are isomorphic as unordered trees.*

For non-linear TSLPs it was shown in [27] that the problem whether val($\mathcal{G}$) and val($\mathcal{H}$) are isomorphic as unordered trees is PSPACE-hard and in EXPTIME.

In [25, 28], the problem of evaluating tree automata over TSLP-compressed input trees was considered. A tree automaton runs on a ranked input tree bottom-up and thereby assigns states to tree nodes. Transitions are of the form $(q_1, \ldots, q_n, f, q)$, where f is a node label of rank n and $q_1, \ldots, q_n, q$ are states of the tree automaton. Then a run of the tree automaton is a mapping ρ from the tree nodes to states that is consistent with the set of transitions in the following sense: If a tree node is labelled with the symbol f (of rank n) and $v_1, \ldots, v_n$ are the children of v in that order, then $(\rho(v_1), \ldots, \rho(v_n), f, \rho(v))$ must be a transition of the tree automaton. A tree automaton accepts a tree if there is a run that assigns a final state to the root of the tree (every tree automaton has a distinguished set of final states). The problem of checking whether an explicitly given tree is accepted by a tree automaton that is part of the input is complete for the class LogCFL (which is contained in the parallel class NC^2) [23]. For a fixed tree automaton this problem belongs to NC^1 [23]. For TSLP-compressed trees we have:

Theorem 9 ([28]). *It is* P*-complete to check for a given TSLP $\mathcal{G}$ and a given tree automaton $\mathcal{A}$, whether $\mathcal{A}$ accepts* val($\mathcal{G}$)*.*

The polynomial time algorithm works in two steps:

Step 1. Using Theorem 1 the input TSLP $\mathcal{G}$ is transformed in polynomial time into a TSLP $\mathcal{H}$ such that val($\mathcal{G}$) = val($\mathcal{H}$) and every nonterminal of $\mathcal{H}$ has rank at most one.

Step 2. For a TSLP $\mathcal{G}$, where every nonterminal has rank 0 or 1, it is easy to evaluate a tree automaton $\mathcal{A}$ on val($\mathcal{G}$). Bottom-up on the structure of the TSLP, one computes for every nonterminal A of rank 0 the set of states to which $\mathrm{val}_{\mathcal{G}}(A)$ can evaluate (i.e., those states that may appear in a run at the root), whereas for a nonterminal A of rank 1 one computes a binary relation on the set of states of $\mathcal{A}$. This relation contains a pair (q_1, q_2) if and only if the following holds: There is a mapping from the nodes of $\mathrm{val}_{\mathcal{G}}(A)$ to the states of $\mathcal{A}$ such that (i) the above condition of a run is satisfied, (ii) to the unique parameter-labelled node of $\mathrm{val}_{\mathcal{G}}(A)$ the state q_1 is assigned, and (iii) to the root the state q_2 is assigned. It is easy to compute this information for a nonterminal A assuming it has been computed for all nonterminals in the right-hand side of A.

In [28], also a generalization of Theorem 9 to tree automata with sibling constraints is shown. In this model, transitions can depend on (dis)equalities between the children of the node to which the transition is applied to.

The problem, whether a given tree automaton accepts the tree val($\mathcal{G}$), where $\mathcal{G}$ is a given non-linear TSLP was shown to be PSPACE-complete in [25]. In fact, PSPACE-hardness already holds for a fixed tree automaton.

Several other algorithmic problems for TSLP-compressed input trees are studied in [7, 13, 21, 35, 37, 36].

References

1. T. Akutsu. A bisection algorithm for grammar-based compression of ordered trees. *Information Processing Letters*, 110(18-19):815–820, 2010.
2. P. Bille, I. L. Gørtz, G. M. Landau, and O. Weimann. Tree compression with top trees. In *Proceedings of ICALP (1) 2013*, volume 7965 of *Lecture Notes in Computer Science*, pages 160–171. Springer, 2013.
3. M. Bousquet-Mélou, M. Lohrey, S. Maneth, and E. Noeth. XML compression via DAGs. *Theory of Computing Systems*, 2014. DOI 10.1007/s00224-014-9544-x.
4. R. P. Brent. The parallel evaluation of general arithmetic expressions. *Journal of the Association for Computing Machinery*, 21(2):201–206, 1974.
5. P. Buneman, M. Grohe, and C. Koch. Path queries on compressed XML. In *Proceedings of VLDB 2003*, pages 141–152. Morgan Kaufmann, 2003.
6. G. Busatto, M. Lohrey, and S. Maneth. Efficient memory representation of XML document trees. *Information Systems*, 33(4–5):456–474, 2008.
7. A. G. Carles Creus and G. Godoy. One-context unification with STG-compressed terms is in NP. In *Proceedings of RTA 2012*, volume 15 of *LIPIcs*, pages 149–164. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2012.
8. 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.
9. H. Comon, M. Dauchet, R. Gilleron, F. Jacquemard, D. Lugiez, C. Löding, S. Tison, and M. Tommasi. Tree automata techniques and applications. Available on: `http://tata.gforge.inria.fr/`, 2007.
10. P. J. Downey, R. Sethi, and R. E. Tarjan. Variations on the common subexpression problem. *Journal of the Association for Computing Machinery*, 27(4):758–771, 1980.
11. P. Flajolet, P. Sipala, and J.-M. Steyaert. Analytic variations on the common subexpression problem. In *Proceedings of ICALP 1990*, volume 443 of *Lecture Notes in Computer Science*, pages 220–234. Springer, 1990.
12. M. Frick, M. Grohe, and C. Koch. Query evaluation on compressed trees (extended abstract). In *Proceedings of LICS 2003*, pages 188–197. IEEE Computer Society Press, 2003.
13. A. Gascón, G. Godoy, and M. Schmidt-Schauß. Unification and matching on compressed terms. *ACM Transactions on Computational Logic*, 12(4):26, 2011.
14. Y. Hirshfeld, M. Jerrum, and F. Moller. A polynomial algorithm for deciding bisimilarity of normed context-free processes. *Theoretical Computer Science*, 158(1&2):143–159, 1996.
15. 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.
16. A. Jeż. Approximation of grammar-based compression via recompression. In *Proceedings of CPM 2013*, volume 7922 of *Lecture Notes in Computer Science*, pages 165–176. Springer, 2013.
17. 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.
18. J. C. Kieffer and E. hui Yang. Grammar-based codes: A new class of universal lossless source codes. *IEEE Transactions on Information Theory*, 46(3):737–754, 2000.

19. N. Kobayashi, K. Matsuda, and A. Shinohara. Functional programs as compressed data. In *Proceedings of PEPM 2012*, pages 121–130. ACM Press, 2012.
20. N. J. Larsson and A. Moffat. Offline dictionary-based compression. In *Proceedings of DCC 1999*, pages 296–305. IEEE Computer Society Press, 1999.
21. J. Levy, M. Schmidt-Schauß, and M. Villaret. The complexity of monadic second-order unification. *SIAM Journal on Computing*, 38(3):1113–1140, 2008.
22. S. Lindell. A logspace algorithm for tree canonization (extended abstract). In *Proceedings of STOC 1992*, pages 400–404. ACM Press, 1992.
23. M. Lohrey. On the parallel complexity of tree automata. In *Proceedings of RTA 2001*, volume 2051 in Lecture Notes in Computer Science, pages 201–215. Springer, 2001.
24. M. Lohrey. Algorithmics on SLP-compressed strings: A survey. *Groups Complexity Cryptology*, 4(2):241–299, 2012.
25. M. Lohrey and S. Maneth. The complexity of tree automata and XPath on grammar-compressed trees. *Theoretical Computer Science*, 363(2):196–210, 2006.
26. M. Lohrey, S. Maneth, and R. Mennicke. XML tree structure compression using RePair. *Information Systems*, 38(8):1150–1167, 2013.
27. M. Lohrey, S. Maneth, and F. Peternek. Compressed tree canonization. Technical report, arXiv.org, 2015. http://arxiv.org/abs/1502.04625. An extended abstract will appear in Proceedings of ICALP 2015.
28. M. Lohrey, S. Maneth, and M. Schmidt-Schauß. Parameter reduction and automata evaluation for grammar-compressed trees. *Journal of Computer and System Sciences*, 78(5):1651–1669, 2012.
29. M. Lohrey and C. Mathissen. Isomorphism of regular trees and words. *Information and Computation*, 224:71–105, 2013.
30. K. Mehlhorn, R. Sundar, and C. Uhrig. Maintaining dynamic sequences under equality-tests in polylogarithmic time. In *Proceedings of SODA 1994*, pages 213–222. ACM/SIAM, 1994.
31. W. Plandowski. Testing equivalence of morphisms on context-free languages. In *Proceedings of ESA 1994*, volume 855 of *Lecture Notes in Computer Science*, pages 460–470. Springer, 1994.
32. W. Rytter. Application of Lempel-Ziv factorization to the approximation of grammar-based compression. *Theoretical Computer Science*, 302(1–3):211–222, 2003.
33. H. Sakamoto. A fully linear-time approximation algorithm for grammar-based compression. *Journal of Discrete Algorithms*, 3(2-4):416–430, 2005.
34. M. Schmidt-Schauß. Polynomial equality testing for terms with shared substructures. Technical Report Report 21, Institut für Informatik, J. W. Goethe-Universität Frankfurt am Main, 2005.
35. M. Schmidt-Schauß. Matching of compressed patterns with character-variables. In *Proceedings of RTA 2012*, volume 15 of *LIPIcs*, pages 272–287. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2012.
36. M. Schmidt-Schauß. Linear compressed pattern matching for polynomial rewriting (extended abstract). In *Proceedings of TERMGRAPH 2013*, volume 110 of *EPTCS*, pages 29–40, 2013.
37. M. Schmidt-Schauß, D. Sabel, and A. Anis. Congruence closure of compressed terms in polynomial time. In *Proceedings of FroCos 2011*, volume 6989 of *Lecture Notes in Computer Science*, pages 227–242. Springer, 2011.
38. J. Zhang, E.-H. Yang, and J. C. Kieffer. A universal grammar-based code for lossless compression of binary trees. *IEEE Transactions on Information Theory*, 60(3):1373–1386, 2014.