View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by Dagstuhl Research Online Publication Server

LZ77 Factorisation of Trees*

Paweł Gawrychowski[1] and Artur Jeż[2]

1 University of Haifa, Israel
2 University of Wrocław, Poland

Abstract

We generalise the fundamental concept of LZ77 factorisation from strings to trees. A tree is represented as a collection of edge-disjoint fragments that either consist of one node or has already occurred earlier (in the BFS order). Similarly as for strings, such a collection uniquely determines the tree, so by minimising the number of fragments we obtain a compressed representation of the tree. We show that our generalisation has several useful properties of the standard LZ77 factorisation: it can be computed in polynomial time and its simpler variant in linear time; its size is not larger than the smallest grammar for a tree; it can be transformed (in linear time) into a tree grammar of size $\mathcal{O}(rg\log(n/(rg)))$, where n is the size of the tree, g the size of the smallest grammar for this tree and r the maximal arity of the nodes in the tree, which matches a recent bound of Jeż and Lohrey [STACS 2014], but with a simpler and more modular proof.

1998 ACM Subject Classification F.4.2 Grammars and Other Rewriting Systems, F.2.2 Nonnumerical Algorithms and Problems, E.4 Data compaction and compression

Keywords and phrases Tree grammars, Grammar compression, LZ77, SLP, Tree compression

Digital Object Identifier 10.4230/LIPIcs.FSTTCS.2016.35

1 Introduction

Many popular compression text methods are based on the concept of block compression, where the input text is divided into blocks, each block being either a single letter or a longer block defined using the already compressed prefix (e.g. its arbitrary substring). Examples of such methods include LZ77, LZ78 and LZW algorithms. Another closely connected and particularly clean method is grammar compression, where we represent the input text as a CFG deriving exactly one word. Such a grammar is usually called a straight-line program (SLP). It is known that grammar compression captures most if not all block compression methods with small or no overhead in the size of the representation, yet it is much easier to operate on. This makes it particularly attractive given that the compressed data is often not only stored, but also accessed and processed on demand, and decompressing it defeats the purpose of having a small compressed representation in the first place. Thus, a recent trend is a design of algorithms working directly on a compressed representation of the data. Because grammar compression has an inductive definition, it is well-suited for such processing. In fact, similar algorithms for block compression routinely transform the the compressed representation into an equivalent SLP and only afterwards process it. The connection between grammar and block compression is used also in the other direction, for instance: computing a smallest SLP is NP-hard [8, 29, 7], but one can transform the LZ77 representation into an SLP, in this way we obtain an approximation algorithm with currently best known approximation ratio [25, 8, 19].

* This work was supported under National Science Centre, Poland project number 2014/15/B/ST6/00615.

© Paweł Gawrychowski Artur Jeż;
licensed under Creative Commons License CC-BY
36th IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS 2016).
Editors: Akash Lal, S. Akshay, Saket Saurabh, and Sandeep Sen; Article No. 35; pp. 35:1–35:15

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

However, often the data has a more complex structure than a one-dimensional text. In particular, it is often the case that it has a natural hierarchical structure, which can be represented as a tree. Such a tree can be of course flattened and then treated as the input text, but this disregards the structure of the original data and hence might make processing the compressed representation (provably) difficult [10]. Hence it is desirable to design compression methods that retain the tree structure.

The folklore tree compression preserving the tree structure are directed acyclic graphs (DAGs), in which identical subtrees are shared. They yield up to exponential compression, processing DAGs is usually easy. The sharing mechanism of DAGs is limited, and it does not generalise the SLP: text a^n, when treated as a tree, is incompressible using DAGs. Tree SLP (*TSLP*) are a generalisation of SLP to the case of trees, which allow sharing of more general substructures. Moreover, DAG are subclass of TSLP and TSLP can be exponentially smaller than DAG. As in the case of SLP [8, 29], determining the size of the smallest TSLP is NP-hard, but there are approximation algorithms for this problem [17, 20] as well as heuristical ones [23, 5, 6].

TSLP became a *de facto* standard of tree compression in computer science, as several types of queries can be computed on TSLP in polynomial time [24] and there are various applications of TSLPs in other subfields. Most notably, they were used in problems related to context unification [13, 14, 15, 21], culminating in the recent proof that context unification is in PSPACE [18]. Moreover, studying matching problems for TSLP became an area on its own [11, 12, 27]. Other applications of TSLP (and SLP) can be found in a recent survey [22].

There are other approaches at tree grammar compression: Akutsu [2] generalised SLP to trees in a different (and incomparable) way. Bille et al. [3] proposed *top trees*, which are a variant of TSLP for unranked and unlabelled trees. Bojańczyk and Walukiewicz [4] proposed the model of forest algebras, which is designed specifically to handle unranked trees (while their model does not mention grammars but rather expressions, it can be easily trimmed to yield an TSLP-like formalism).

There were attempts at defining a variant of block compression designed for trees, but none of them became standard nor did they have ties to tree grammar compression. In particular, so far exploiting the connection between the grammar and block compression, so popular in text compression, did not occur for trees.

Our contribution. We propose a model of block compression for trees, inspired by both the notion of LZ77 and TSLPs. Similarly to LZ77, it represents the tree as a collection of trees (with “holes” that are placeholders for another trees), called *fragments*; each fragment is either an individual node or has already appeared earlier (in the BFS order) in the tree. Such factorisation can be compactly represented: each fragment is encoded by the BFS-addresses of the root and all the holes of its earlier occurrence. We give an $\mathcal{O}(n^3)$ algorithm constructing the smallest such representation for the general case, and a specialised linear algorithm for the case when fragments have at most one hole. Such factorisations are related to TSLPs in a similar way as LZ77 are related to SLPs: it is not difficult to see each TSLP can be transformed into a factorisation without size increase. We prove that a factorisation can be transformed into a TSLP defining the same tree yielding an approximation algorithm for the smallest TSLP problem, the approximation ratio matches the best known [20].

2 Notions, definitions, basic results

Trees. We consider Σ-labeled rooted trees: The children of every node are ordered (by the usual left-to-right order) and each node is labeled with a letter from a fixed *ranked*

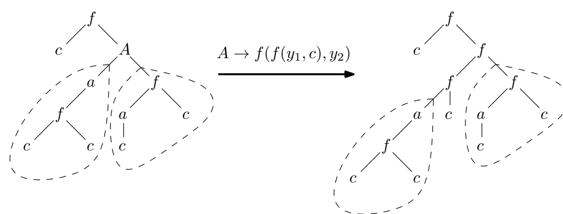

Figure 1 A tree $f(c, A(a(f(c,c)), f(a(c), c)))$ and the result of applying a rule $A(y_1, y_2) \to f(f(y_1, c), y_2)$.

alphabet Σ, i.e. every $a \in \Sigma$ has arity $\mathrm{ar}(a)$. The label of a node u is denoted by $\mathit{label}(u)$ and determines its number of children $\mathrm{ar}(\mathit{label}(u))$. By $\mathit{children}(u)$ we denote the children of u, and by $\mathit{parent}(u)$ its parent. We assume integer alphabet, that is, $\Sigma = \{1, 2, \ldots, n\}$, where n is the size of the tree; thus, node labels can be sorted in $\mathcal{O}(n)$ time using RadixSort. By r we denote the maximal arity of nodes in the tree. We often employ a BFS-order on the nodes of the tree, denoted by $\leq_{BFS}$ or $\leq$, when this causes no confusion; it is a linear order. By *BFS-addresses* (or *addresses*) we denote the BFS-order numbers.

We consider also trees with *holes* that represent missing subtrees. The holes are represented by labels $y, y_1, y_2, \ldots$ from $\mathbb{Y}$ that is disjoint with Σ. For the purposes of building trees holes have arity 0. Labels of different holes are different. As holes represent missing trees, we can substitute trees (or with trees with holes) for them: we delete the appropriate hole and replace it with the appropriate tree.

LZ77. As we model our tree factorisation on the LZ77 factorisations, let us recall the notion of LZ77 for strings: An LZ77 factorisation of a word w is a representation $w = f_1 f_2 \cdots f_\ell$, where each f_i (called *phrase*) is either a single letter or it already occurred in the text, formally: $f_i = w[j \mathinner{.\,.} j + |f_i| - 1]$ for some $j \leq |f_1 \cdots f_{i-1}|$. If for some f_i as in the latter case it holds that $j + |f_i| > |f_1 \cdots f_{i-1}|$ (informally, the whole phrase occurred earlier), this factorisation is called *self-referencing*; it is called *non-self-referencing* otherwise. The self-referencing factorisations are in some cases more succinct than the non-self referencing ones, with the trivial example being string a^k. Each f_i can be represented as a single letter or as a pair $(j, |f_i|)$, where j is as in the explanation above, yielding a *compressed representation* of a LZ77 factorisation. The (compressed) *size* of a factorisation $f_1 f_2 \cdots f_\ell$ is ℓ. Smallest LZ77 factorisation of a given text can be computed in linear time.

Tree grammars. Tree grammars extend CFG to trees: they have nonterminals (finite set N), terminals (ranked alphabet Σ), rules (P) and a starting symbol (S). Nonterminals generate trees with holes: each nonterminal A has an *arity* $\mathrm{ar}(A)$ and it generates a tree with $\mathrm{ar}(A)$ holes. Thus in the rules we treat A as $\mathrm{ar}(A)$-ary function symbol and in the rule $A \to t$ the tree t has $\mathrm{ar}(A)$ holes, labeled with $y_1, \ldots, y_{\mathrm{ar}(A)}$. To stress the dependence on holes, we sometimes write the rule as $A(y_1, \ldots, y_{\mathrm{ar}(A)}) \to t$.

We apply a rule $A \to t$ as follows: given a tree s we can rewrite its subtree $A(t_1, \ldots, t_{\mathrm{ar}(A)})$ by t with each y_i replaced by t_i. Intuitively, we replace an A-labeled node by $t(y_1, \ldots, y_{\mathrm{ar}(A)})$ and identify the j-th child of A with the unique y_j-labeled node of t, see Fig. 1.

The *size* of a rule $A(y_1, \ldots, y_{\mathrm{ar}(A)}) \to t$ is $|t| - \mathrm{ar}(A)$, i.e. the number of non-holes nodes in t. This makes sense, as we can assume that BFS order of holes' labels in t is $y_1, \ldots, y_{\mathrm{ar}(A)}$ and all of them occur in t, so we do not have to list them when describing t. The size of a tree grammar is the sum of sizes of its rules.

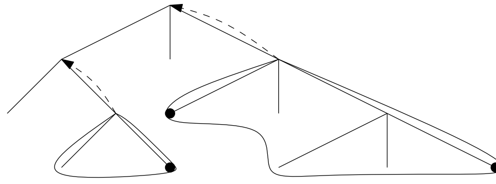

Figure 2 BFS-factorisation: for simplicity, the node labels were suppressed, only one label of each arity is used. The fragments are enclosed in blobs, the holes are represented as full dots, other nodes are free. Note that nodes 7, 11, 14 are free nodes and holes in their fragments. The dashed arrows are definitions.

In *Tree SLP* (or *TSLP* for short) we additionally insist that for $A \in N$ there is *exactly one* production $A \to t$ and that nonterminals are linearly ordered such that S is the smallest nonterminal and if $A \to t$ and B occurs in t then $B > A$. So, TSLP produces a unique tree.

Factorisations of trees. A *fragment* F is a connected subtree of a tree T in which each node *either* has the same label a as in T (and then has $\mathrm{ar}(a)$ children) *or* is labeled with some $y \in \mathbb{Y}$, in which case it is a leaf (in F) and it is called a *hole*. The *arity* of a fragment is its total amount of holes, a fragment of arity k is called a *k-fragment*. We require that a fragment has at least 2 non-hole nodes. A *free node* is defined similarly as a fragment, but it has exactly one non-hole node, i.e. it has a non-hole root and all its children are holes. When referring to a free node rooted in v we call it simply v.

A *factorisation* $\mathcal{F}$ of a tree T, denoted by $(T, \mathcal{F})$, is a collection of edge-disjoint fragments and free nodes, such that each node of T is either a non-hole node in some fragment or a free node. Such a factorisation is a *BFS-factorisation* if for each fragment $F \in \mathcal{F}$, which is rooted in $v \in T$, there is an isomorphic fragment F' of T, called the *definition* of F, rooted in $v' <_{BFS} v$; note that F' is not necessarily in $\mathcal{F}$ and that F and F' may overlap. See Fig. 2 for an example. Neither a factorisation, nor the BFS-factorisation of a tree is unique, similarly to the LZ77 factorisation of a string.

For $v \in F$ the corresponding $v' \in F'$ is the *definition* of v, denoted by $\mathit{def}[v]$, a link to this node is a *definition link*. The *arity* of a factorisation is the maximal arity of the fragments in it. An ℓ-factorisation is a factorisation of arity at most ℓ.

Note that nodes are shared, but in such a case in the “lower” fragment (or free node) this node is a root and in the “upper” a hole. In this way nodes in the tree can have two definitions: one as a definition of a hole and one as a definition of a root. We store only the latter but use a term “definition of a hole”, as it is useful in description and analysis.

We use the BFS order because the condition $\mathit{def}[v] < v$ required for the root of the fragment is inherited by other nodes in this fragment. Thus, given a fragment, we can extend it by a node above the root (at the hole) as long as the labels are the same.

▶ **Lemma 1.** *Let F be a fragment in a BFS-factorisation of a tree, D a definition of F and u, v be two nodes in F. Then $\mathit{def}[u] < u$ if and only if $\mathit{def}[v] < v$.*

In case of TSLP, the maximal arity of nonterminals in a grammar usually multiplicatively affects the running time and in general makes the reasoning about TSLP harder. Luckily, it is known that any TSLP can be converted to a one that uses only nonterminals of arity 0 and 1, with a polynomial size increase [24]. Similar result can be also obtained for the tree factorisations: a factorisation can be converted to a 1-factorisation, with a polynomial size increase:

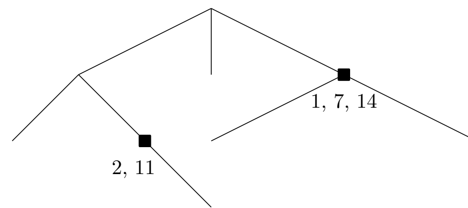

Figure 3 Compressed BFS-factorisation of the tree from Fig. 2. The fragments are depicted by square boxes, the addresses of their definitions and holes are beneath the boxes.

▶ **Lemma 2.** *In any factorisation an ℓ-fragment can be (effectively) replaced with at most $(\ell-1)(r-2)$ 0-fragments, $\ell-1$ free nodes and $(2\ell-1)$ 1-fragments. If the original factorisation was a BFS one, also the obtained one is.*

Compressed representation. The *compressed BFS-factorisation* for a BFS-factorisation $(T, \mathcal{F})$ is given by a tree: each fragment and free node is represented as a single node; an edge between two fragments or free nodes means that they share a node in T (in other words: a hole in one of them is a root in the other). Each fragment-node stores addresses (in T) of its root and holes. See Fig. 3.

A compressed BFS-factorisation is *valid* if it corresponds to some BFS-factorisation of a tree; not every compressed BFS-factorisation is valid, as it may use undefined addresses. Still, a valid compressed BFS-factorisation defines a unique tree, which can be computed by iteratively adding the nodes in the BFS order.

▶ **Lemma 3.** *A valid compressed BFS-factorisation corresponds to exactly one BFS-factorisation of a tree, which can be computed in $\mathcal{O}(|T|)$ time, where T is the resulting tree.*

For the purpose of storing the compressed factorisation, we count the storage size of the factorisation in terms of memory cells rather than bits and assume that we can fit one address in $\mathcal{O}(1)$ memory cells. Note that this is also the assumption for LZ77.

▶ **Lemma 4.** *The compressed BFS-factorisation of $(T, \mathcal{F})$ uses $\mathcal{O}(|\mathcal{F}|)$ memory cells.*

In the full version of this paper, we consider also *weighted size* of factorisations: the weight of an i-fragment is w_i and of a free node is 1; we assume $1 \leq \mathrm{w}_0 \leq \mathrm{w}_1 \leq \dots$. The (weighted) *size* of the factorisation is the sum of weights of its fragments. To streamline the presentation we do not consider the weights, though our algorithms work also in weighted case.

Comparison with the text model. A text can be seen as a tree using only nodes of arity 1, and the compressed BFS-factorisation of such a tree corresponds with a compressed LZ77 factorisation of this text: each LZ77 phrase can be extended to the right with an extra hole added at the end. The sizes and the descriptions of two such factorisations are similar.

In case of text we can distinguish between self-referencing and non-self-referencing factorisation. In case of trees we use only the self-referencing variant, as a clear and useful definition of a non-self-referencing one is elusive.

Comparison with tree grammars. In case of strings, the size of the LZ77 is a lower bound for the size of the grammar generating the same string [25, 8]; the same holds for trees; this observation is the first step in the approximation algorithm for the smallest tree grammar problem, presented in Section 4.

▶ **Lemma 5.** *The size of the smallest compressed BFS-factorisation for a tree T is not larger than the size of the smallest TSLP for T.*

3 Computing BFS-factorisations

In this section we first present a polynomial time dynamic programming algorithm SimpleFactors, which computes an optimal BFS-factorisation (of unbounded arity). Then, we show how to modify it, so that it computes optimal k-BFS-factorisations. Finally, using additional insight, we give a specialised algorithm for 1-BFS-factorisations that runs in linear time. We focus on describing how to calculate the number of fragments in an optimal BFS-factorisation, reconstructing the corresponding factorisation is straightforward.

Unbounded arity. Let T_u denote the tree rooted in u and $opt(u)$ the number of fragments in an optimal factorisation of T_u into fragments occurring earlier in the whole tree T (so not necessarily in T_u). To compute $opt(u)$ SimpleFactors constructs a table $T[u, v]$, which stores the smallest size of a factorisation of T_u, such that v is the definition of u (or $T[u, v] = +\infty$, if $label(u) \neq label(v)$). We fill $T[u, v]$ in a reversed BFS order, thus when calculating $T[u, v]$ the $T[u', v']$ and $opt(u')$ are known for every $u' \in children(u)$ and $v' < u'$.

Let $children(u) = (u_1, u_2, \dots, u_d)$ and $children(v) = (v_1, v_2, \dots, v_d)$. Then $T[u, v]$ is 1 plus the number of fragments in subtrees rooted in u_i, for each i. If u_i is a hole this is $opt(u_i)$; otherwise this is $T[u_i, v_i] - 1$ (the '-1' is for the fragment rooted in u_i in $T[u_i, v_i]$, which is not counted in $T[u, v]$). We take the minimum of those two numbers separately for each i. Thus $T[u, v] = 1 + \sum_{i=1}^{d} \min(T[u_i, v_i] - 1, opt(u_i))$. Similarly, $opt(u) = \min(1 + \sum_{i=1}^{d} opt(u_i), \min_{v<u} T[u, v])$, where the (outer) minimum represents the two possibilities: there is a free node or a fragment rooted in u.

▶ **Theorem 6.** SimpleFactors *computes the size of the smallest BFS-factorisation (of unbounded arity) using quadratic time and space.*

Bounded arity. We modify SimpleFactors into BoundedFactors, which computes smallest k-BFS-factorisation, by introducing another parameter. In particular, we again allow factorisations of T_u into fragments that occur earlier in T, not necessarily in T_u.

Let $T[u, v, h]$ be the size of an optimal k-factorisation for T_u such that $v < u$ is a definition of u and the fragment rooted in u has h holes. Let $children(u) = (u_1, u_2, \dots, u_d)$ and $children(v) = (v_1, v_2, \dots, v_d)$. Any factorisation of T_u, when restricted to T_{u_i}, is either a free node with some fragments attached or an optimal factorisation of T_{u_i} of a cost $T[u_i, v_i, h_i]$; moreover, the sum of h_i over all children is h, i.e. $h_1 + h_2 + \dots + h_d = h$, where $h_i = 1$ if u_i is a hole. Thus BoundedFactors fills the table $T[u, v, h]$ similarly as SimpleFactors the $T[u, v]$ but computes an appropriate min-plus product of $T[u_i, v_i, h_i] - 1$ instead of taking a simple minimum; then $opt(u)$ is computed from $T[u, v, h]$ as previously. A careful analysis yields $\mathcal{O}(n^2 \min(k^2, n))$ bound on the running time.

▶ **Theorem 7.** BoundedFactors *computes the size of the smallest k-BFS-factorisation and runs in $\mathcal{O}(n^2 \min(k^2, n))$ time and $\mathcal{O}(n^2 k)$ space, where n is the size of the tree and k the maximal arity of fragments in the k-BFS-factorisation.*

1-BFS-factorisations

BoundedFactors runs in quadratic time in case of 1-BFS-factorisations. We now present a specialised version of BoundedFactors, called 1-Factors, which achieves linear time.

▶ **Theorem 8.** *The algorithm* 1-Factors *runs in linear time and computes the size of the smallest* 1*-BFS-factorisation.*

Outline. Each factorisation of T_u has one of the following *types*: (F1) it is a single 0-fragment; (F2) the root u is a free node; (F3) there is a 1-fragment rooted in u such that its hole is a leaf. 1-Factors processes the nodes in a BFS-reversed order and for each node u computes for T_u the optimal factorisation of each type and takes the smallest among them.

Computing the F1 factorisations of all trees boils down to grouping the subtrees $\{T_u\}_{u \in T}$ into isomorphism classes, which is explained in detail later on. F2 factorisation is easy to calculate, as we already know the optimal factorisation trees rooted in children of u. Lastly, we need to compute F3 factorisations only for nodes that do not have an F1 factorisation, as it is always smaller than F3.

▶ **Lemma 9.** *For every u, the size of F1 factorisations of T_u (if feasible) is at most the size of F3 factorisation of T_u.*

Call the nodes that have an F1 factorisation or are leaves *bottom part* and the remaining nodes the *top part*.

▶ **Lemma 10.** *Top part of the tree is a subtree rooted in the root of T; it can be computed in linear time.*

Now, for F3 factorisations, we would like to also group 1-fragments into isomorphism classes, but there are $\mathcal{O}(n^2)$ of them. Thus we limit their amount: we show that it is enough to consider maximal (in an appropriate sense) 1-fragments and prove that there are $\mathcal{O}(n)$ of them. Furthermore, using additional insight we can compute in linear time a superset of all 1-factors used in an optimal 1-factorisation. Then it remains to calculate their actual cost, to see whether they should be used in the 1-factorisation. It turns out that those candidates are in some sense consecutive and we can process them in amortised constant time.

F1 factorisations. Two trees T_u and T_v are *isomorphic*, denoted by $T_u \equiv T_v$, if $label(u) = label(v)$ and the subtrees rooted at the corresponding children of u and v are also isomorphic. Grouping subtrees of T into isomorphism classes in linear time is already folklore [1].

▶ **Lemma 11.** *Given a tree T we can calculate in total time $\mathcal{O}(|T|)$ for every node u:* (1) *a number $id(u) \in \{1, 2, \ldots, |T|\}$, such that $T_u \equiv T_v \iff id(u) = id(v)$* (2) *a node $v < u$ such that $id(u) = id(v)$, whenever such v exists.*

F3 factorisations. If F' is a definition of a fragment F then trees rooted in a node and its definition are mostly isomorphic: define for every node u a number $siblings\text{-}id(u)$, such that $siblings\text{-}id(u) = siblings\text{-}id(v)$ if and only if the following conditions hold: (1) $label(parent(u)) = label(parent(v))$; (2) u is the k-th child of $parent(u)$ and v is the k-th child of $parent(v)$; (3) for every $i \in \{1, \ldots, \mathrm{ar}(parent(u))\} \setminus \{k\}$, the trees rooted in i-th child of $parent(u)$ and the i-th child of $parent(v)$ are isomorphic. These can be efficiently computed, by appropriate grouping of nodes and its siblings by RadixSort.

▶ **Lemma 12.** *Given a tree T we can calculate in total $\mathcal{O}(|T|)$ time for every node u its number $siblings\text{-}id(u) \in \{1, 2, \ldots, |T|\}$.*

Consider a 1-fragment F rooted at u_1 and a hole at u_{k+1}: the path from u_1 to u_{k+1}, denoted as $u_1, \ldots, u_{k+1}$, is the *spine* of F; a spine is earlier than other if its hole is earlier.

Given a path $u_1, \dots, u_{k+1}$ we call it a *spine* without explicitly mentioning the 1-fragment. The *length* of the spine $u_1, \dots, u_{k+1}$ is k. A spine $u_1, \dots, u_{k+1}$ is *isomorphic* to a spine $v_1, \dots, v_{k+1}$ if and only if $\mathit{siblings\text{-}id}(u_i) = \mathit{siblings\text{-}id}(v_i)$ for $i = 2, \dots, k+1$. Then 1-fragments are isomorphic if and only if their spines are. This characterisation is used to find isomorphism classes of 1-fragments. Given such classes, we can easily determine, whether a fragment F can be used in a BFS-factorisation (i.e. whether it has an earlier definition).

▶ **Lemma 13.** *1-fragments are isomorphic if and only if their spines are. In particular, a 1-fragment F can be used in a 1-BFS-factorisation if and only if there is an earlier spine isomorphic to the spine of F.*

A subsequence of a spine is also a spine, thus we extend the spines up as far as possible: a spine $u_1, \dots, u_{k+1}$ is *up-maximal* if and only if it has an earlier isomorphic spine and $\mathit{parent}(u_1), u_1, \cdots, u_{k+1}$ does not. A 1-fragment is up-maximal when its spine is and 1-factorisation is up-maximal if each of its 1-fragment is. Any optimal 1-BFS-factorisation can be converted into an up-maximal one without increasing its size; thus we consider only up-maximal 1-factorisations.

Up-maximal spines can be compactly represented: for every node v let $\mathit{spine}(v)$ be its highest ancestor such that the path starting at $\mathit{spine}(v)$ and ending at v is an up-maximal spine; if there is no such an ancestor then $\mathit{spine}(v) = v$. All $\mathit{spine}(v)$ and the lengths of the corresponding spines can be calculated in linear-time using a suffix tree of a tree [28].

▶ **Lemma 14.** *Given a tree T, we can find in $\mathcal{O}(|T|)$ total time for every node v its $\mathit{spine}(v)$ as well as the length of the spine beginning at $\mathit{spine}(v)$ and a hole at v.*

Now, F3 factorisation are computed as follows: we process the tree in the reversed BFS order. When we are in a non-leaf node u we *activate* it, a node u is *deactivated* when we processed $\mathit{spine}(u)$. When we are in node u from a top part, we choose from its active descendent v with minimal $\mathit{opt}(v)$, if such v exists. We call such a query a minad (*minimal active descendent*) query, and denote by $\mathrm{minad}(u)$. Then the cost of the smallest F3 factorisation of T_u is $1 + \mathit{opt}(\mathrm{minad}(u))$, or undefined, if $\mathrm{minad}(u)$ is not defined. Note that $\mathit{opt}(\mathrm{minad}(u))$ is already known, as $\mathrm{minad}(u)$ is processed. The data structure presented below answers minad queries in amortised $\mathcal{O}(1)$ time.

Data structure for minad queries. We exploit the structure of spines in the top part of the tree. Firstly, the top part can be represented as union of vertex disjoint paths: A *snake* originates in a node from a top part; if u is on a snake and it has a unique son u' that is not in a bottom part then u' is also on a snake; each snake is maximal: it cannot be extended neither up nor down. A spine of a node in a top part is contained in a single snake, perhaps except the hole.

▶ **Lemma 15.** *Snakes are vertex-disjoint paths, all snakes can be constructed in linear time.*

Let u be in a top part and let T_u have an F4 factorisation. Then there are: an optimal up-maximal F4 factorisation of T_u and a snake $u_1, u_2, \dots, u_s$ such that $u = u_i$ and the hole of fragment rooted in u is either at some u_j, where $j > i$, or at a child of u_s.

For a snake $u_1, \dots, u_s$ we ask queries $\mathrm{minad}(u_s), \mathrm{minad}(u_{s-1}), \dots, \mathrm{minad}(u_1)$ in this order. Our data structures are constructed for a snake and updated when we process consecutive nodes of a snake. These data structures are called $S1$ and $S2$ later on, the former stores the active children of u_s and the other the active nodes from the snake. A node v is represented in these data structures as a pair $(i, \mathit{opt}(v))$, where $\mathit{spine}(v) = u_i$. These data structures

support three operations (on a set of elements S): insert(x',y') inserts a new pair (x',y') into S such that $x' \leq x$ for all $(x,y) \in S$; remove(x') removes from S all pairs (x,y) such that $x \geq x'$; min returns the pair $(x,y) \in S$ with the smallest value of y. A data structure supporting those operations is standard.

▶ **Lemma 16.** *A set S of pairs (x,y) can be maintained in linear space, so that* insert*,* remove*,* min *operations take amortised constant time.*

This data structure is used in our setting as follows: when we begin to process a snake $u_1,\ldots,u_s$, we insert into S_1 each child u of u_s, whenever $spine(u) \neq u$ (so $spine(u) = u_j$ for some j); this can be done in linear time using RadixSort. We also initialize S_2 as empty. After processing u_i we create a pair $(j, opt(u_i))$, where $spine(u_i) = u_j$, insert it into S_2 and remove from S_1, S_2 elements with first coordinate less or equal to i. To calculate minad(u_i) we take minima from S_1, S_2 and return the minimum of their second coordinates.

Note that insert(x',y') assumes $x' \leq x$ for all $(x,y) \in S$. This is guaranteed for S_1 as we sort the pairs by their first coordinate before inserting, for S_2 this requires a simple proof.

4 Application: approximation of the smallest TSLP

In the string case, the algorithm transforming the LZ77 representation into a grammar [25, 8, 26, 19] (with a $\mathcal{O}(\log(n/\ell))$ size increase, where ℓ is the size of the LZ77 representation and n the length of the text) has profound implications. On one hand, this is still the best approximation algorithms for the smallest SLP construction, in particular it returns a grammar at most quadratic in size of the smallest one, which is for instance usually good enough for computational complexity considerations. On the other hand, this algorithm is used as a first step in algorithms processing LZ77 compressed text, see for instance [16].

In this section we show a similar result for our factorisations of trees: We give an algorithm that transform a tree factorisation of size ℓ into a TSLP of size $\mathcal{O}(\ell r + \ell r \log(n/\ell g))$, where r is the maximal arity on nodes in the tree and n the size of the tree. In particular, the approximation ratio is $\mathcal{O}(rg\log(n/(rg)))$, which matches a recent best known bound [20]; we call our algorithm FactToG. This algorithm generalises the approach of [19] from the string case to tree case.

▶ **Theorem 17.** FactToG *runs in linear time and returns a TSLP of size $\mathcal{O}(rg + rg\log(n/rg))$, where n is the size of the input tree T, g the size of the smallest TSLP for T and r the maximal arity of nodes in T.*

4.1 Idea

FactToG repeatedly applies two operations: *leaf compression* and *unary nodes compression*. Given a tree T leaf compression deletes all leaves in T and relabels their parents in a consistent way: the new label f' of the node v uniquely determines the previous label of v (say f), on which positions v had leaf-children (say $i_1, i_2, \ldots, i_\ell$) and what were their labels (say c_1, …, c_ℓ). The "reverse" of leaf compression corresponds to a TSLP rule of a form

$$f'(y_1,\ldots,y_{i_1-1},y_{i_1+1},\ldots,y_{i_2-1},y_{i_2+1},\ldots,y_{i_\ell-1},y_{i_\ell+1},\ldots) \to \\ f(y_1,\ldots,y_{i_1-1},c_1,y_{i_1+1},\ldots,y_{i_2-1},c_2,y_{i_2+1},\ldots,y_{i_\ell-1},c_\ell,y_{i_\ell+1},\ldots) \ . \quad \text{(1a)}$$

Unary nodes compression chooses some neighbouring unary nodes (say labelled with a, b) deletes the lower of them and relabels the remaining one (say with c); the new label uniquely

determines the original labels. The corresponding TSLP rule is

$$c(y) \to a(b(y)) \ . \tag{1b}$$

These operations are iterated until a single-node tree is obtained.

We want to perform several unary nodes compression in parallel. As pairs of unary nodes may overlap, we choose a set of non-overlapping pairs. For appropriate choice the leaf compression followed by (parallel) unary nodes compression reduces the size of the tree by a constant factor. We call this a *phase* of FactToG. One phase can be implemented in linear time, assuming that RadixSort for grouping nodes; this yields a total linear running time.

As we end FactToG when the tree is reduced to a single node, the TSLP rules produced on the way yield an TSLP generating the input tree. It remains to bound its size in terms of the smallest TSLP for the input tree, thus yielding a bound on the approximation ratio. To this end we use the 1-BFS-factorisations: firstly, we show that the size of the smallest 1-BFS-factorisation is $\mathcal{O}(rg)$, where r is the maximal arity of nodes in the input tree and g the size of the smallest TSLP for it. Then we show that if T has a 1-factorisation of size m then rules introduced during one phase of FactToG have size $\mathcal{O}(m)$ and that the resulting tree also has a 1-factorisation of size m. As there are $\mathcal{O}(\log n)$ phases, this yields $\mathcal{O}(rg \log n)$ bound on the size of the TSLP, better estimations yield the actual bound in Theorem 17.

The bound on the size of the produced rules is rather straightforward, but the construction of a 1-factorisation of the resulting tree is involved. To this end ensure that when we perform a compression operation on a fragment, we perform the same operation also on its definition. In particular, each compression operation should be performed wholly within or wholly outside a fragment (or definition of a fragment). As a result, way the factorisation of the resulting tree is “inherited” from the original tree.

To guarantee this property on one hand we choose the pairs of unary nodes to compress with the help of the factorisation, and on the other we modify the factorisation by “popping” the nodes that are compressed with nodes outside the fragments: we remove such a node from the fragment and make it a free node. This sometimes introduces new fragments, but they are of specific form and are under control. We call the (restrictions of) fragments that were present in the input *original fragments* and the created ones the *introduced fragments*.

Due to compression operations we cannot guarantee that we keep a BFS-factorisation. Instead, we store an order “$<$” on the nodes and require that $def[v] < v$ (as well as some other technical conditions). The actual order is obtained in a natural way: it is the original BFS order restricted to the nodes in the current tree.

4.2 Representation and basic operations

Factorisation. The factorisation depends on an order $<$, we refer to the factorisation as $(T, \mathcal{F}, <)$ and call it a *proper factorisation* if (1) $def[v] < v$; (2) $parent[v] < v$ and (3) a hole of a fragment is not a root of the same fragment. The first condition ensures that a proper factorisation is a generalisation of a BFS-factorisation, second ensures that the order on the vertices is meaningful and the third that there are no useless fragments. Initially $<$ is the BFS order on the input tree and clearly for BFS-order the 1-BFS-factorisation is proper. We often process the tree by considering nodes in the stored order, then we say that we *traverse the tree*.

Each node v in a fragment (except for the hole), has a definition link $def[v]$ to its definition. We also store bitflags telling whether a node is a root or a hole of a fragment. A $(T, \mathcal{F}, <)$ stores the order $<$.

Popping nodes. During the algorithm we modify the factorisation by *popping* nodes:

- *Popping a unary node up.* When a root v of a fragment is a unary node, we make v a free node and its unique child a new root.
- *Popping a unary node down.* When a parent v of a hole is a unary node, we perform symmetrical operations: we make v a free node and make it a hole of this fragment.
- *Popping a nullary node.* When a whole 0-fragment consist of a single nullary node v, we turn it into a free node.
- *Popping a node down.* When v is a parent of a hole we remove the current hole from the fragment, make v a new hole, make v a free node and create new 0-fragments rooted in each child of v, except the ex-hole. This generalises the popping of a unary node down.

When the variant is clear from the context, we simply refer to *popping a node v*. A simple case inspection shows that popping nodes turns a proper factorisation into a proper one.

▶ **Lemma 18.** *Popping a node in a proper factorisation results in a proper factorisation.*

4.3 Compression operations and the algorithm

When we perform the compression operations, both the fragment and its definition should be modified in the same way. We now explain how to ensure this.

Leaf compression Both affected nodes (the deleted leaf and its parent) should either be both within a fragment or both outside it, the same applies to the definition of the fragment. The following conditions ensure this:

(L1) A leaf is neither a hole nor a definition of a hole.
(L2) A leaf is not a whole fragment.

Ensuring (L1–L2) is done by PairLeaves: if a leaf is a root of a fragment (so it violates (L2)), we turn it into a free node (so pop it). To ensure (L1) we look at each hole, if it violates (L1) then we pop its parent from the fragment; in this way we may create 0-fragments violating (L2), we pop nodes from such fragments as well.

▶ **Lemma 19.** PairLeaves *applied to a proper factorisation returns a proper factorisation satisfying (L1–L2). It introduces at most r free nodes per* 1*-fragment and* 1 *per* 0*-fragment.*

Then we perform the compression CompLeaves: traverse the tree, if the node v has a leaf child, then change its label: If this node is within a fragment, copy the label from the definition; if it is a free node then assign it a new label. If the read node is a leaf then delete it. The new order is a restriction of the order to the nodes that were not deleted.

▶ **Lemma 20.** CompLeaves *applied to a proper factorisation satisfying (L1–L2) returns a proper factorisation. The size of created rules is twice the number of removed free leaves.*

Unary nodes compression. Since we replace pairs of nodes with new ones, this compression boils down to pairing (of unary nodes): for each node we should determine, whether it is a top (top) or bottom one ($bottom$) in a pair or perhaps that it is unpaired ($none$). We construct a pairing satisfying the following properties, which are a slight modifications of conditions for a similar algorithm for in the string case [19]:

(P1) There are no two neighbouring unary nodes that are both unpaired.
(P2) If a root of a fragment is unary then it is not paired with its parent, if a node above the hole is unary then it is paired with its parent;
(P3) v and $def[v]$ are paired in the same way (so either both are unpaired, both are top-nodes in a pair or both bottom nodes in a pair).

(P1) guarantees that compression of chosen pairs decreases the length of each sequence of unary nodes by a constant factor, (P3) says that the fragment is paired exactly in the same way as its definition and (P2) ensures that pairing for a fragment is done within this fragment, in particular that we can inherit the factorisation after the compression.

The computation of the pairing is technical and it is deferred to Section 4.4, let us first give the procedure CompUnary that uses this pairing: We traverse the tree. If the read node v is *top* and within a fragment then we replace its label with the label of its definition; if it is *top* and a free node then we assign it a fresh label. If the read node v is a *bottom* node, then let u be its unique child: we change father of u to current father of v and delete v.

▶ **Lemma 21.** CompUnary *applied to a proper factorisation satisfying (P1)–(P3) returns a proper factorisation. The size of introduced rules is at most twice the number of free unary nodes removed from the tree.*

FactToG first computes an optimal 1-factorisation. Then it transforms this factorisation into a TSLP in phases, until the tree is reduced into a single node. In each phase it first computes the pairing for the unary nodes and replaces those pairs and then modifies the factorisation so that the leaves compression can be applied and applies the leaves compression.

4.4 Pairing unary nodes

Removing circular references. Imagine a unary root of a fragment v has $def[v] = parent[v]$. Then no pairing satisfying (P1–P3) can be devised, as all nodes in this fragment should be paired in the same way (i.e. all as top or all as bottom nodes). However, all nodes in this fragment are labeled with the same (unary) label, thus we can pop the root and change the definition of this fragment: each node has the definition two nodes up, instead of one.

Such circular reference can be more complex: it could be that $def^{\ell}[v] = parent[v]$ for some $\ell > 1$, which again makes the pairing impossible. Thus, in some sense, there are many sequences of nodes involved in this circular reference. The solution is in the same spirit as above: we isolate (in some fragment) the unary nodes with such a circular reference and create a new 1-fragment from them. Then we change their definition so that it goes two nodes up. The details of the procedure and the actual properties guaranteed after it are left for the full version.

▶ **Lemma 22.** UnaryPreproc *runs in linear time and pops $\mathcal{O}(1)$ nodes per fragment.*

Pairing of unary nodes. For the pairing, we first pair the free nodes, so that if a fragment has a unary root then the two free nodes above are paired and if the parent of its hole is unary then the two nodes below are paired. Then we traverse the tree: for a node v we copy the pairing from its definition, with some exceptions: If v is a root and its definition is paired as *bottom* then we pop v and set its status as *none*, similarly, when v is father of a hole and its definition is *top* then we pop it and set its pairing as *none*. In this way we may violate (P1) (as neighbour of v is also paired as *none*), by case inspection and $\mathcal{O}(1)$ additional pops we can find appropriate pairing.

▶ **Lemma 23.** PairUnary *applied to a proper factorisation returns a proper factorisation and a pairing satisfying (P1)–(P3). It introduces $\mathcal{O}(1)$ new free nodes per fragment.*

4.5 Analysis

Size of the factorisation. Using Lemma 5 and known properties of the TSLP we can show that the smallest 1-factorisation for T has $\mathcal{O}(g)$ 1-fragments and $\mathcal{O}(rg)$ 0-fragments and free nodes, where r is the maximal arity of nodes in T.

Size of the tree. One phase of FactToG reduces the size of the tree by a constant factor: if there are no unary nodes in the tree then we remove all leaves and so halve the size. If there are only unary nodes, then by (P1) there are no two consecutive unpaired ones, so the size of the tree also drops by 1/4. The general argument is a mix of the two above.

▶ **Lemma 24.** FactToG *applied to a tree T returns a tree of size at most $\frac{3|T|}{4}$.*

Running time. The construction of the 1-BFS-factorisation takes linear time. All sub-procedures run in linear time, combined with Lemma 24 this yields a total linear running time.

Size of rules and popped letters. The size of new rules is covered by the number of removed free nodes, see Lemma 20, 21, so it is enough to count the number of introduced free nodes. From Lemmata 19 and 22 this is $\mathcal{O}(r)$ per 1-fragment and $\mathcal{O}(1)$ per 0-fragment per phase.

Introduced fragments. We now bound the number of letters popped from introduced fragments. On one hand we bound the number of introduced 0-fragments and on the other we give an amortised analysis for introduced 1-fragments.

When we introduce a 0-fragment, we *associate* it with the 1-fragment of which it used to be part of. It can be shown that introduced 1-fragments have no associated 0-fragments, furthermore, when new 0-fragment are associated with an original 1-fragment F then F has no fragments associated with it. Thus at any point there are $\mathcal{O}(rg)$ introduced 0-fragments.

We also associate introduced 1-fragments with original 1-fragments; this association is involved and described in the full version. We can amortise the number of nodes they pop.

▶ **Lemma 25.** *At any point there are $\mathcal{O}(rg)$ introduced 0-fragments. Introduced 1-fragments pop in total amortised $\mathcal{O}(g)$ nodes per phase.*

Size of the constructed TSLP. Combining the lemmata above we get an upper bound of $\mathcal{O}(rg + rg\log(n))$ on the size of the constructed TSLP. In a more detailed analysis we separately consider the computation before and after the moment in which the current tree has size rg. We apply the above analysis to the first part: there are only $\mathcal{O}(\log(n/(rg)))$ phases and so the size of constructed part of SLP is $\mathcal{O}(rg\log(n/rg))$. In the second step we use a simple fact [20] that any reasonable grammar for a tree of size rg has size $\mathcal{O}(rg)$, which yields the total size $\mathcal{O}(rg + rg\log(n/rg))$ and so also the claimed approximation ratio.

5 Open problems/Future work

String LZ77 can be constructed in LOGSPACE and is used for instance in approximation of the construction of the smallest SLP in the streaming model [9]. Can a similar construction be performed also for the LZ77 for trees?

In case of strings, the LZ77 compressed representation can be directly translated into an SLP [8, 25], yielding an approximation algorithm for SLP construction. Can a similar construction be carried out also for the tree variant of LZ77? This would allow to translate several known algorithms for TSLP to the case of tree factorisations.

References

1 Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, 1974.

2 Tatsuya Akutsu. A bisection algorithm for grammar-based compression of ordered trees. *Inf. Process. Lett.*, 110(18–19):815–820, 2010.

3 Philip Bille, Inge Li Gørtz, Gad M. Landau, and Oren Weimann. Tree compression with top trees. *Infmormation and Computation*, 243:166–177, 2015. doi:10.1016/j.ic.2014.12.012.

4 Mikołaj Bojańczyk and Igor Walukiewicz. Forest algebras. In Jörg Flum, Erich Grädel, and Thomas Wilke, editors, *Logic and Automata: History and Perspectives [in Honor of Wolfgang Thomas].*, volume 2 of *Texts in Logic and Games*, pages 107–132. Amsterdam University Press, 2008.

5 Mireille Bousquet-Mélou, Markus Lohrey, Sebastian Maneth, and Eric Nöth. XML compression via directed acyclic graphs. *Theory Comput. Syst.*, 57(4):1322–1371, 2015. doi:10.1007/s00224-014-9544-x.

6 Giorgio Busatto, Markus Lohrey, and Sebastian Maneth. Efficient memory representation of XML document trees. *Information Systems*, 33(4–5):456–474, 2008.

7 Katrin Casel, Henning Fernau, Serge Gaspers, Benjamin Gras, and Markus L. Schmid. On the complexity of grammar-based compression over fixed alphabets. In Ioannis Chatzigiannakis, Michael Mitzenmacher, Yuval Rabani, and Davide Sangiorgi, editors, *ICALP*, volume 55 of *LIPIcs*, pages 122:1–122:14. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2016. doi:10.4230/LIPIcs.ICALP.2016.122.

8 Moses Charikar, Eric Lehman, Ding Liu, Rina Panigrahy, Manoj Prabhakaran, Amit Sahai, and Abhi Shelat. The smallest grammar problem. *IEEE Transactions on Information Theory*, 51(7):2554–2576, 2005. doi:10.1109/TIT.2005.850116.

9 Travis Gagie and Paweł Gawrychowski. Grammar-based compression in a streaming model. In Adrian Horia Dediu, Henning Fernau, and Carlos Martín-Vide, editors, *LATA*, volume 6031 of *LNCS*, pages 273–284. Springer, 2010. doi:10.1007/978-3-642-13089-2_23.

10 Moses Ganardi, Danny Hucke, Markus Lohrey, and Eric Noeth. Tree compression using string grammars. In Evangelos Kranakis, Gonzalo Navarro, and Edgar Chávez, editors, *LATIN*, volume 9644 of *LNCS*, pages 590–604. Springer, 2016. doi:10.1007/978-3-662-49529-2_44.

11 Adria Gascón, Guillem Godoy, and Manfred Schmidt-Schauß. Context matching for compressed terms. In *Proceedings of the Twenty-Third Annual IEEE Symposium on Logic in Computer Science, LICS 2008, 24-27 June 2008, Pittsburgh, PA, USA*, pages 93–102. IEEE Computer Society, 2008. doi:10.1109/LICS.2008.17.

12 Adria Gascón, Guillem Godoy, and Manfred Schmidt-Schauß. Unification with singleton tree grammars. In Ralf Treinen, editor, *RTA*, volume 5595 of *LNCS*, pages 365–379. Springer, 2009. doi:10.1007/978-3-642-02348-4_26.

13 Adria Gascón, Guillem Godoy, Manfred Schmidt-Schauß, and Ashish Tiwari. Context unification with one context variable. *J. Symb. Comput.*, 45(2):173–193, 2010. doi:10.1016/j.jsc.2008.10.005.

14 Adria Gascón, Manfred Schmidt-Schauß, and Ashish Tiwari. Two-restricted one context unification is in polynomial time. In Stephan Kreutzer, editor, *CSL*, volume 41 of *LIPIcs*, pages 405–422. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2015. doi:10.4230/LIPIcs.CSL.2015.405.

15 Adria Gascón, Ashish Tiwari, and Manfred Schmidt-Schauß. One context unification problems solvable in polynomial time. In *LICS*, pages 499–510. IEEE, 2015. doi:10.1109/LICS.2015.53.

16 Paweł Gawrychowski. Pattern matching in Lempel-Ziv compressed strings: fast, simple, and deterministic. In Camil Demetrescu and Magnús M. Halldórsson, editors, *ESA*, volume 6942 of *LNCS*, pages 421–432. Springer, 2011. doi:10.1007/978-3-642-23719-5_36.

17 Danny Hucke, Markus Lohrey, and Eric Noeth. Constructing small tree grammars and small circuits for formulas. In Venkatesh Raman and S. P. Suresh, editors, *FSTTCS*, volume 29 of *LIPIcs*, pages 457–468. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2014. doi:10.4230/LIPIcs.FSTTCS.2014.457.

18 Artur Jeż. Context unification is in PSPACE. In Elias Koutsoupias, Javier Esparza, and Pierre Fraigniaud, editors, *ICALP*, volume 8573 of *LNCS*, pages 244–255. Springer, 2014. doi:10.1007/978-3-662-43951-7_21.

19 Artur Jeż. A *really* simple approximation of smallest grammar. *Theoretical Computer Science*, 616:141–150, 2016. doi:10.1016/j.tcs.2015.12.032.

20 Artur Jeż and Markus Lohrey. Approximation of smallest linear tree grammar. In Ernst W. Mayr and Natacha Portier, editors, *STACS*, volume 25 of *LIPIcs*, pages 445–457. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2014. doi:10.4230/LIPIcs.STACS.2014.445.

21 Jordi Levy, Manfred Schmidt-Schauß, and Mateu Villaret. On the complexity of bounded second-order unification and stratified context unification. *Logic Journal of the IGPL*, 19(6):763–789, 2011. doi:10.1093/jigpal/jzq010.

22 Markus Lohrey. Algorithmics on SLP-compressed strings: A survey. *Groups Complexity Cryptology*, 4(2):241–299, 2012.

23 Markus Lohrey, Sebastian Maneth, and Roy Mennicke. XML tree structure compression using RePair. *Inf. Syst.*, 38(8):1150–1167, 2013. doi:10.1016/j.is.2013.06.006.

24 Markus Lohrey, Sebastian Maneth, and Manfred Schmidt-Schauß. Parameter reduction and automata evaluation for grammar-compressed trees. *J. Comput. Syst. Sci.*, 78(5):1651–1669, 2012. doi:10.1016/j.jcss.2012.03.003.

25 Wojciech Rytter. Application of Lempel-Ziv factorization to the approximation of grammar-based compression. *Theor. Comput. Sci.*, 302(1-3):211–222, 2003. doi:10.1016/S0304-3975(02)00777-6.

26 Hiroshi Sakamoto. A fully linear-time approximation algorithm for grammar-based compression. *J. Discrete Algorithms*, 3(2-4):416–430, 2005. doi:10.1016/j.jda.2004.08.016.

27 Manfred Schmidt-Schauß. Linear compressed pattern matching for polynomial rewriting (extended abstract). In Rachid Echahed and Detlef Plump, editors, *TERMGRAPH*, volume 110 of *EPTCS*, pages 29–40, 2013. doi:10.4204/EPTCS.110.5.

28 Tetsuo Shibuya. Constructing the suffix tree of a tree with a large alphabet. In *Algorithms and Computation*, pages 225–236. Springer, 1999.

29 James A. Storer and Thomas G. Szymanski. The macro model for data compression. In Richard J. Lipton, Walter A. Burkhard, Walter J. Savitch, Emily P. Friedman, and Alfred V. Aho, editors, *STOC*, pages 30–39. ACM, 1978.