Approximation of smallest linear tree grammar*

Artur Jeż†1 and Markus Lohrey2

1 MPI Informatik, Saarbrücken, Germany / University of Wrocław, Poland
2 University of Siegen, Germany

Abstract

A simple linear-time algorithm for constructing a linear context-free tree grammar of size $\mathcal{O}(r^2 g \log n)$ for a given input tree T of size n is presented, where g is the size of a minimal linear context-free tree grammar for T, and r is the maximal rank of symbols in T (which is a constant in many applications). This is the first example of a grammar-based tree compression algorithm with an approximation ratio polynomial in g. The analysis of the algorithm uses an extension of the recompression technique (used in the context of grammar-based string compression) from strings to trees.

1998 ACM Subject Classification F.4.2 Grammars and Other Rewriting Systems

Keywords and phrases Grammar-based compression, Tree compression, Tree-grammars

Digital Object Identifier 10.4230/LIPIcs.STACS.2014.445

1 Introduction

Grammar-based compression has emerged to an active field in string compression during the last 10 years. The principle idea is to represent a given string s by a small context-free grammar that generates only s; such a grammar is also called a *straight-line program* (SLP). For instance, the word $(ab)^{1024}$ can be represented by the SLP with the productions $A_0 \to ab$ and $A_i \to A_{i-1}A_{i-1}$ for $1 \le i \le 10$ (A_{10} is the start symbol). The size of this grammar is much smaller than the size (length) of the string $(ab)^{1024}$. In general, an SLP of size n (the size of an SLP is usually defined as the total length of all right-hand sides of productions) can produce a string of length $2^{\Omega(n)}$. Hence, an SLP can be seen indeed as a succinct representation of the generated word. The principle task of grammar-based string compression is to construct from a given input string s a small SLP that produces s. Unfortunately, finding a size-minimal SLP for a given input string is hard: Unless $\mathbb{P} = \mathbb{NP}$ there is no polynomial time grammar-based compressor, whose output SLP has size less than 8569/8568 times the size of a minimal SLP for the input string [4], and so there is no polynomial time grammar-based compressor $\mathcal{G}$ with an approximation ratio of less than 8569/8568. In general the approximation ratio for $\mathcal{G}$ is defined as the function $\alpha_{\mathcal{G}}$ with

$$\alpha_{\mathcal{G}}(n) = \max \frac{\text{size of the SLP produced by } \mathcal{G} \text{ with input } x}{\text{size of a minimal SLP for } x},$$

where the max is taken over all strings of length n (over an arbitrary alphabet). The best known polynomial time grammar-based compressors [4, 9, 17, 18] have an approximation ratio of $\mathcal{O}(\log(n/g))$, where g is the size of a smallest SLP for the input string (each of them works in linear time).

* The full version of this paper can be found at http://arxiv.org/abs/1309.4958.

† A. Jeż was partially supported by Polish National Science Centre (NCN) grant 2011/01/D/ST6/07164, 2011–2014.

At this point, the reader might ask, what makes grammar-based compression so attractive. There are actually several reasons: The output of a grammar-based compressor (an SLP) is a clean and simple object, which may simplify the analysis of a compressor or the analysis of algorithms that work on compressed data (see [13] for a survey). Moreover, there are grammar-based compressors which achieve very good compression ratios. For example RePair [12] performs very well in practice and was for instance used for the compression of web graphs [5]. Finally, the idea of grammar-based string compression can be generalized to other data types as long as suitable grammar formalisms are known for them. The last point is the most important one for this work. In [3], grammar-based compression was generalized from strings to trees (a tree in this paper is always a rooted ordered tree over a ranked alphabet, i.e., every node is labelled with a symbol and the rank of this symbol is equal to the number of children of the node). For this, context-free tree grammars were used. Context free tree grammars that produce only a single tree are also known as straight-line context-free tree grammars (SLCF tree grammars). Several papers deal with algorithmic problems on trees that are succinctly represented by SLCF tree grammars, see [13] for a survey. In [14], RePair was generalized from strings to trees, and the resulting algorithm TreeRePair achieved excellent results on real XML data trees. Other grammar-based tree compressors were developed in [15]. But none of these compressors has an approximation ratio polynomial in g: For instance, in [14] a series of trees is constructed, where the n-th tree t_n has size $\Theta(n)$, there exists an SLCF tree grammar for t_n of size $\mathcal{O}(\log n)$, but the grammar produced by TreeRePair for t_n has size $\Omega(n)$ (similar examples can be constructed for the compressors in [3, 15]).

In this paper, we give the first example of a grammar-based tree compressor, called TtoG, with an approximation ratio of $\mathcal{O}(\log n)$ assuming the maximal rank r of symbols is bounded; otherwise the approximation ratio becomes $\mathcal{O}(r^2 \log n)$. TtoG is based on the work [9] of the first author, where grammar-based string compressor with an approximation ratio of $\mathcal{O}(\log n)$ is presented. The crucial fact about this compressor is that in contrast to [4, 17, 18] it does not use the LZ77 factorization of a string (which makes the compressors from [4, 17, 18] not suitable for a generalization to trees, since LZ77 ignores the tree structure and no analogue of LZ77 for trees is known), but is based on the *recompression technique*. This technique was introduced in [7] and successfully applied for a variety of algorithmic problems for SLP-compressed strings [7, 8] and word equations [11, 10]. The basic idea is to compress a string using two operations: (i) block compressions, which replaces every maximal substring of the form a^ℓ for a letter a by a new symbol a_ℓ, and (ii) pair compression, which for a given partition $\Sigma_\ell \uplus \Sigma_r$ of the alphabet replaces every substring $ab \in \Sigma_\ell\Sigma_r$ by a new symbol c. It can be shown that the composition of block compression followed by pair compression (for a suitably chosen partition of the input letters) reduces the length of the string by a constant factor. Hence, the iteration of block compression followed by pair compression yields a string of length one after a logarithmic number of phases. By reversing the single compression steps, one obtains an SLP for the initial string. The term "recompression" refers to the fact, that for a given SLP $\mathbb{G}$, block compression and pair compression can be simulated on the SLP $\mathbb{G}$. More precisely, one can compute from $\mathbb{G}$ a new grammar $\mathbb{G}'$, which is not much larger than $\mathbb{G}$ such that $\mathbb{G}'$ produces the result of block compression (respectively, pair compression) applied to the string produced by $\mathbb{G}$. In [9], the recompression technique is used to bound the approximation ratio of the above compression algorithm based on block and pair compression.

In this work we generalize the recompression technique from strings to trees. The operations of block compression and pair compression can be directly applied to chains of unary nodes (nodes having only a single child) in a tree. But clearly, these two operations alone cannot reduce the size of the initial tree by a constant factor. Hence we need a third compression operation that we call leaf compression. It merges all children of node that are leafs into the node; the new label of the node determines the old label, the sequence of labels of the children that are leaves, and their positions in the sequence of all children of the node. Then, one can show that a single phase, consisting of block compression (that we call chain compression), followed by pair compression (that we call unary pair compression), followed by leaf compression reduces the size of the initial tree by a constant factor. As for strings, we obtain an SLCF tree grammar for the input tree by basically reversing the sequence of compression operations. The recompression approach again yield an approximation ratio of $\mathcal{O}(\log n)$ for our compression algorithm, but the analysis is technically more subtle.

Related work on grammar-based tree compression. We already mentioned that grammar-based tree compressors were developed in [3, 14, 15], but none of these compressors has a good approximation ratio. Another grammar-based tree compressors was presented in [1]. It is based on the BISECTION algorithm for strings and has an approximation ratio of $\mathcal{O}(n^{5/6})$. But this algorithm used a different form of grammars (elementary ordered tree grammars) and it is not clear whether the results from [1] can be extended to SLCF tree grammars, or whether the good algorithmic results for SLCF-compressed trees [13] can be extended to elementary ordered tree grammars. Let us finally mention [2], where trees are compressed by so called top trees. These are another hierarchical representation of trees. Upper bounds on the size of top trees are derived and compared with the size of the minimal dag (directed acyclic graph). More precisely, it is shown in [2] that the size of the top tree is larger than the size of the minimal dag by a factor of $\mathcal{O}(\log n)$. Since dags can be seen as a special case of SLCF tree grammars, our main result is stronger.

Computational model. To achieve a linear running time we employ RadixSort, see [6, Section 8.3], to obtain a linear-time grouping of symbols. To this end some assumption on the computational model and form of the input are needed: we assume that numbers of $\mathcal{O}(\log n)$ bits (where n is the size of the input tree) can be manipulated in time $\mathcal{O}(1)$ and that the labels of the input tree come from an interval $[1, .., n^c]$, where c is some constant.

1.1 Trees and SLCF tree grammars

Let us fix for every $i \geq 0$ a countably infinite set $\mathbb{F}_i$ of letters of rank i and let $\mathbb{F} = \bigcup_{i\geq 0} \mathbb{F}_i$ be their disjoint union. Symbols in $\mathbb{F}_0$ are called *constants*, while symbols in $\mathbb{F}_1$ are called *unary letters*. We also write $\mathrm{rank}(a) = i$ if $a \in \mathbb{F}_i$. A ranked alphabet F is a finite subset of $\mathbb{F}$. We also write F_i for $F \cap \mathbb{F}_i$ and $F_{\geq i}$ for $\bigcup_{j\geq i} F_i$. An *F-labelled tree* is a rooted, ordered tree whose nodes are labelled with elements from F, satisfying the condition that if a node v is labelled with a then it has exactly $\mathrm{rank}(a)$ children, which are linearly ordered (by the usual left-to-right order). We denote by $\mathcal{T}(F)$ the set of F-labelled trees. In the following we shall simply speak about trees when the ranked alphabet is clear from the context or unimportant. When useful, we identify an F-labelled tree with a term over F in the usual way. The size $|t|$ of the tree t is its number of nodes.

Fix a countable set $\mathbb{Y}$ with $\mathbb{Y} \cap \mathbb{F} = \emptyset$ of *(formal) parameters*, which are denoted by $y, y_1, y_2, \ldots$. For the purposes of building trees with parameters, we treat all parameters as constants, and so F-labelled trees with parameters from $Y \subseteq \mathbb{Y}$ (where Y is finite) are simply $(F \cup Y)$-labelled trees, where the rank of every $y \in Y$ is 0. However to stress the special role of parameters we write $\mathcal{T}(F, Y)$ for the set of F-labelled trees with parameters from Y. We identify $\mathcal{T}(F)$ with $\mathcal{T}(F, \emptyset)$. In the following we talk about *trees with parameters* (or even trees) when the ranked alphabet and parameter set is clear from the context or unimportant. The idea of parameters is best understood when we represent trees as terms: For instance $f(y_1, a, y_2, y_1)$ with parameters y_1 and y_2 can be seen as a term with variables y_1, y_2 and we can instantiate those variables later on. A *pattern* (or *linear tree*) is a tree $t \in \mathcal{T}(F, Y)$, that contains for every $y \in Y$ at most one y-labelled node. Clearly, a tree without parameters is a pattern. All trees in this paper will be patterns, and we will not mention this assumption explicitly in the following.

When we talk of a *subtree* u of a tree t, we always mean a full subtree in the sense that for every node of u all descendents of that node in t belong to u as well. In contrast, a *subpattern* v of t is obtained from a subtree u of t by replacing some of the subtrees of u by pairwise different parameters. In this way we obtain a pattern $p(y_1, \ldots, y_n)$ and we say that (i) the subpattern v is an *occurrence* of the pattern $p(y_1, \ldots, y_n)$ in t and (ii) $p(y_1, \ldots, y_n)$ is the pattern corresponding to the subpattern v (this pattern is unique up to renaming of parameters). This later terminology applies also to subtrees, since a subtree is a subpattern as well. To make this notions clear, consider for instance the tree $f(a(b(c)), a(b(d)))$ with $f \in \mathbb{F}_2$, $a, b \in \mathbb{F}_1$ and $c, d \in \mathbb{F}_0$. It contains one occurrence of the pattern $a(b(c))$ and two occurrences of the pattern $a(b(y))$.

A *chain pattern* is a pattern of the form $a_1(a_2(\ldots(a_k(y))\ldots))$ with $a_1, a_2, \ldots, a_k \in \mathbb{F}_1$. We write $a_1 a_2 \cdots a_k$ for this pattern and treat it as a string (even though this string still needs an argument on its right to form a proper term). In particular, we write a^ℓ for the chain pattern consisting of ℓ many a-labelled nodes and we write vw (for chain patterns v and w) for what should be $v(w(y))$. A *chain* in a tree t is an occurrence of a chain pattern in t. A chain s in t is *maximal* if there is no chain s' in t with $s \subsetneq s'$. A 2-chain is a chain consisting of only two nodes (which, most of the time, will be labelled with different letters). For $a \in \mathbb{F}_1$, an a-maximal chain is a chain such that (i) all nodes are labelled with a and (ii) there is no chain s' in t such that $s \subsetneq s'$ and all nodes of s' are labelled with a too. Note that an a-maximal chain is not necessarily a maximal chain. Consider for instance the tree $baa(c)$. The unique occurrence of the chain pattern aa is an a-maximal chain, but is not maximal. The only maximal chain is the unique occurrence of the chain pattern baa.

For the further consideration, fix a countable infinite set $\mathbb{N}_i$ of symbols of rank i with $\mathbb{N}_i \cap \mathbb{N}_j = \emptyset$ for $i \neq j$. Let $\mathbb{N} = \bigcup_{i \geq 0} \mathbb{N}_i$. Furthermore, assume that $\mathbb{F} \cap \mathbb{N} = \emptyset$. Hence, every finite subset $N \subseteq \mathbb{N}$ is a ranked alphabet. A *linear context-free tree grammar* (there exist also non-linear CF tree grammars, which we do not need for our purpose) or short *linear CF tree grammar* is a tuple $\mathbb{G} = (N, F, P, S)$ such that $N \subseteq \mathbb{N}$ (resp., $F \subseteq \mathbb{F}$) is a finite set of *nonterminals* (resp., *terminals*), $S \in N$ is the *start nonterminal* of rank 0, and P (the set of *productions*) is a finite set of pairs (A, t) (for which we write $A \to t$), where $A \in N$ and $t \in \mathcal{T}(F \cup N, \{y_1, \ldots, y_{\mathsf{rank}(A)}\})$ is a pattern, which contains exactly one y_i-labelled node for each $1 \leq i \leq \mathrm{rank}(A)$. To stress the dependency of A on its parameters we sometimes write $A(y_1, \ldots, y_{\mathrm{rank}(A)}) \to t$ instead of $A \to t$. Without loss of generality we assume that every nonterminal $B \in N \setminus \{S\}$ occurs in the right-hand side t of some production $(A \to t) \in P$, see [16, Theorem 5]. The derivation relation $\Rightarrow_{\mathbb{G}}$ on $\mathcal{T}(F \cup N, Y)$ is defined as follows: $s \Rightarrow_{\mathbb{G}} s'$ if and only if there is a production $(A(y_1, \ldots, y_\ell) \to t) \in P$ such that s' is obtained from s by replacing some subtree $A(t_1, \ldots, t_\ell)$ of s by t with each y_i replaced by t_i. Intuitively, we replace an A-labelled node by the pattern $t(y_1 \ldots, y_{\mathrm{rank}(A)})$ and thereby identify the j-th child of A with the unique y_j-labelled node of the pattern. Then $L(\mathbb{G}) = \{t \in \mathcal{T}(F) \mid S \Rightarrow_{\mathbb{G}}^* t\}$.

A *straight-line context-free tree grammar* (or *SLCF grammar* for short) is a linear CF tree grammar $\mathbb{G} = (N, F, P, S)$, where (i) for every $A \in N$ there is *exactly one* production

$(A \to t) \in P$ with left-hand side A, (ii) if $(A \to t) \in P$ and B occurs in t then $B < A$, where $<$ is a linear order on N, and (iii) S is the maximal nonterminal with respect to $<$. By (i) and (ii), every $A \in N$ derives exactly one tree from $\mathcal{T}(F, \{y_1, \ldots, y_{\mathsf{rank}(A)}\})$; we denote this tree by $\mathrm{val}(A)$ (like *value*). Moreover, we define $\mathrm{val}(\mathbb{G}) = \mathrm{val}(S)$, which is a tree from $\mathcal{T}(F)$. For an SLCF grammar $\mathbb{G} = (N, F, P, S)$ we can assume without loss of generality that for every production $(A \to t) \in P$ the parameters $y_1, \ldots, y_{\mathrm{rank}(A)}$ occur in t in the order $y_1, y_2, \ldots, y_{\mathrm{rank}(A)}$ from left to right. This can be ensured by a simple bottom-up rearranging procedure.

There is a subtle point, when defining the *size* $|\mathbb{G}|$ of the SLCF grammar $\mathbb{G}$: One possible definition could be $|\mathbb{G}| = \sum_{(A \to t) \in P} |t|$, i.e., the sum of all sizes of right-hand sides. However, consider for instance the rule $A(y_1, \ldots, y_\ell) \to f(y_1, \ldots, y_{i-1}, a, y_i, \ldots, y_\ell)$. It is in fact enough to describe the right-hand side as $(f, (i, a))$, as we have a as the i-th child of f. On the remaining positions we just list the parameters, whose order is known; see the above remark. In general, each right-hand side can be specified by listing for each node its children that are *not* parameters together with their positions in the list of all children. These positions are numbers between 1 and r (it is easy to show that our algorithm TtoG creates only nonterminals of rank at most r, see Lemma 1, and hence every node in a right-hand side has at most r children) and therefore fit into $\mathcal{O}(1)$ machine words. For this reason we define the size $|\mathbb{G}|$ as the total number of non-parameter nodes in all right-hand sides. If the size of a grammar is defined as the total number of all nodes (including parameters) in all right-hand sides, then the approximation ratio of TtoG is multiplied by an additional factor r.

Notational conventions. Our compression algorithm TtoG takes a tree T and applies to it local compression operations, which shrink the size of the tree. With T we always denote the current tree stored by TtoG, whereas n denotes the size of the initial input tree. The algorithm TtoG adds fresh letters to the tree. With F we always denote the set of letters occurring in the current tree T. The ranks of the fresh letters do not exceed the maximal rank of the original letters. To be more precise, if we add a letter a to F_i, then $F_{\geq i}$ was non-empty before this addition. By r we denote the maximal rank of the letters occurring in the input tree. By the above remark, TtoG never introduces letters of rank larger than r.

2 Compression operations

Our compression algorithm TtoG is based on three local replacement rules applied to trees:

(i) a-maximal chain compression: For a unary letter a replace every a-maximal chain consisting of $\ell > 1$ nodes with a fresh unary letter a_ℓ (for all $\ell > 1$).

(ii) (a, b)-pair compression: For two unary letters $a \neq b$ replace every occurrence of ab by a single node labelled with a fresh unary letter c (which identifies the pair (a, b)).

(iii) $(f, i_1, a_1 \ldots, i_\ell, a_\ell)$-leaf compression: For $f \in F_{\geq 1}$, $\ell \geq 1$, $a_1, \ldots, a_\ell \in F_0$ and $0 < i_1 < i_2 < \cdots < i_\ell \leq \mathrm{rank}(f) =: m$ replace every occurrence of $f(t_1, \ldots, t_m)$, where $t_{i_j} = a_j$ for $1 \leq j \leq \ell$ and t_i is a non-constant for $i \notin \{i_1, \ldots, i_\ell\}$, by $f'(t_1, \ldots, t_{i_1-1}, t_{i_1+1}, \ldots, t_{i_\ell-1}, t_{i_\ell+1}, \ldots, t_m)$, where f' is a fresh letter of rank $\mathrm{rank}(f) - \ell$ (which identifies $(f, i_1, a_1 \ldots, i_\ell, a_\ell)$).

Note that each of these operations shrinks the size of the current tree. Operations (i) and (ii) apply only to unary letters and are direct translations of the operations used in the recompression-based algorithm for constructing a grammar for a given string [9]. On the other hand, (iii) is a new and designed specifically to deal with trees.

Every application of one of our compression operations can be seen as the ‘backtracking’ of a production of the grammar that we construct: When we replace a^ℓ by a_ℓ, we introduce the new nonterminal $a_\ell(y)$ with the production $a_\ell(y) \to a^\ell(y)$. When we replace all occurrences of the chain ab by c, the new production is $c(y) \to a(b(y))$. Finally, for $(f, i_1, a_1 \ldots, i_\ell, a_\ell)$-leaf compression the production is $f'(y_1, \ldots, y_{\mathrm{rank}(f)-\ell}) \to f(t_1, \ldots, t_{\mathrm{rank}(f)})$, where $t_{i_j} = a_j$ for $1 \le j \le \ell$ and every t_i with $i \notin \{i_1, \ldots, i_\ell\}$ is a parameter (and the left-to-right order of the parameters in the right-hand side is $y_1, \ldots, y_{\mathrm{rank}(f)-\ell}$). All these productions are for nonterminals of rank at most r, which implies:

▶ **Lemma 1.** *The rank of nonterminals defined by* TtoG *is at most* r*.*

During the analysis of the approximation ratio of TtoG we also consider the nonterminals of a smallest grammar generating the given input tree. To avoid confusion between these nonterminals and the nonterminals of the grammar produced by TtoG, we insist on calling the fresh symbols introduced by TtoG (a_ℓ, c, and f' above) *letters* and add them to the set F of current letters, so that F always denotes the set of letters in the current tree T. In particular, whenever we talk about nonterminals, productions, etc. we mean the ones of the smallest grammar we consider. Nevertheless, the above productions for the new letters form the grammar returned by our algorithm TtoG and we need to estimate their size. In order not to mix the notation, we shall call the size of the rule for a new letter a the *representation cost* for a and say that *a represents* the subpattern it replaces in T. For instance, the representation cost of a_ℓ with $a_\ell(y) \to a^\ell(y)$ is ℓ, the representation cost of c with $c(y) \to a(b(y))$ is 2, and the representation cost of f' with $f'(y_1, \ldots, y_{\mathrm{rank}(f)-\ell}) \to f(t_1, \ldots, t_{\mathrm{rank}(f)})$ is $\ell + 1$. A crucial part of the analysis of TtoG is the reduction of the representation cost for letters a_ℓ: Note that instead of representing $a^\ell(y)$ directly by $a_\ell(y) \to a^\ell(y)$, we can introduce new unary letters representing some shorter chains in a^ℓ and build longer chains using the smaller ones as building blocks. For instance, the rule $a_8(y) \to a^8(y)$ can be replaced by the rules $a_8(y) \to a_4(a_4(y))$, $a_4(y) \to a_2(a_2(y))$ and $a_2(y) \to a(a(y))$. This yields a total representation cost of 6 instead of 8. Our algorithm employs a particular strategy for representing a-maximal chains, which yields the total cost stated in the following lemma:

▶ **Lemma 2** (cf. [9, Lemma 2])**.** *Given a list $\ell_1 < \ell_2 < \cdots < \ell_k$ we can represent the letters $a_{\ell_1}, a_{\ell_2}, \ldots, a_{\ell_k}$ that replace the chain patterns $a^{\ell_1}, a^{\ell_2}, \ldots, a^{\ell_k}$ with a total cost of $\mathcal{O}(k + \sum_{i=1}^{k} \log(\ell_i - \ell_{i-1}))$, where $\ell_0 = 0$.*

The important property of the compression operations is that we can perform many of them independently in an arbitrary order without influencing the outcome. Since different a-maximal chains and b-maximal chains do not overlap (regardless of whether $a = b$ or not) we can perform a-maximal chain compression for all unary letters a occurring in T in an arbitrary order (assuming that the new letters do not occur in T). We call the resulting tree ChainCmp(T), and denote the corresponding procedure also *chain compression*.

A similar observation applies to leaf compressions: We can perform $(f, i_1, a_1 \ldots, i_\ell, a_\ell)$-leaf compression for all $f \in F_{\geq 1}$, $0 < i_1 < i_2 < \cdots < i_\ell \le \mathrm{rank}(f) =: m$, and $(a_1, a_2, \ldots, a_\ell) \in F_0^\ell$ in an arbitrary order (again assuming that the fresh letters do not occur in the T). We denote the resulting tree with LeafCmp(T) and call the corresponding procedure also *leaf compression*.

The situation is more subtle for unary pair compression: observe that in a chain abc we can compress ab or bc but we cannot do both in parallel (and the outcome depends on the order of the operations). However, as in the case of string compression [9], independent (or parallel) (a, b)-pair compressions are possible when we take a and b from disjoint subalphabets

F_1^{up} and F_1^{down}, respectively. In this case for each unary letter we can tell whether it should be the parent node or the child node in the compression step and the result does not depend on the order of the considered 2-chains, as long as new letters are outside $F_1^{\text{up}} \cup F_1^{\text{down}}$. Hence, we denote with UNARYCMP($F_1^{\text{up}}, F_1^{\text{down}}, T$) the result of doing (a, b)-pair compression for all $a \in F_1^{\text{up}}$ and $b \in F_1^{\text{down}}$ (in an arbitrary order). The corresponding procedure is also called $(F_1^{\text{up}}, F_1^{\text{down}})$-*compression*.

3 The algorithm TTOG

In a single phase of the algorithm TTOG, chain compression, $(F_1^{\text{up}}, F_1^{\text{down}})$-compression and leaf compression are executed in this order (for an appropriate choice of the partition $F_1^{\text{up}}, F_1^{\text{down}}$).

The intuition behind this approach is as follows: If the tree t in question does not have any unary letters, then leaf compression on its own reduces the size of t by half, as it effectively reduces all constant nodes, i.e. leaves of the tree, and more than half of nodes are leaves. On the other end of the spectrum is the situation in which all nodes (except for the unique leaf) are labelled with unary letters. In this case our instance is in fact a string. Chain compression and unary pair compression correspond to the operations of block compression and pair compression, respectively, from the earlier work of the first author on string compression [9], where it is shown that block compression followed by pair compression reduces the size of the string by a constant factor 3/4 (for an appropriate choice of the partition $F_1^{\text{up}}, F_1^{\text{down}}$ of the letters occurring in the string). The in-between cases are a mix of those two extreme scenarios and for each of them the size of the instance drops by a constant factor in one phase as well, see Lemma 4. We need the following lemma, which is a modification of [9, Lemma 4]. Recall that F always denotes the set of letters occurring in T.

Algorithm 1 TTOG: Creating an SLCF tree grammar for the input tree T

```
1: while $|T| > 1$ do
2:     $T \leftarrow$ CHAINCMP($T$)
3:     compute a partition $F_1 = F_1^{\text{up}} \uplus F_1^{\text{down}}$     ▷ Lemma 3
4:     $T \leftarrow$ UNARYCMP($F_1^{\text{up}}, F_1^{\text{down}}, T$)
5:     $T \leftarrow$ LEAFCMP($T$)
6: return constructed grammar
```

▶ **Lemma 3.** *Assume that (i) T does not contain an occurrence of a chain pattern aa for some $a \in F_1$ and (ii) the symbols in T form an interval of numbers. Then, in time $\mathcal{O}(|T|)$ one can find a partition $F_1 = F_1^{up} \uplus F_1^{down}$ such that the number of occurrences of chain patterns from $F_1^{up} F_1^{down}$ in T is at least $(n_1 - 3c + 2)/4$, where n_1 is the number of nodes in T with a unary label and c is the number of maximal chains in T. In the same running time we can provide for each $ab \in F_1^{up} F_1^{down}$ occurring in T a lists of pointers to all occurrences of ab in T.*

A single iteration of the main loop of TTOG is called a *phase*. A single phase can be implemented in time linear to the size of the current T. The main idea is that RADIXSORT is used for effective grouping in linear time and finding a partition is a simple modification of [9, Lemma 4]. The main property of a single phase is:

▶ **Lemma 4.** *In each phase, $|T|$ is reduced by a constant factor.*

Since each phase needs linear time, the contributions of all phase give a geometric series and we get:

▶ **Theorem 5.** TTOG *runs in linear time.*

4 Size of the grammar produced by TtoG: recompression

4.1 Normal form

We want to compare the size of the grammar produced by TtoG with the size of a smallest SLCF grammar for the input tree T. For this, we first transform the minimal grammar into a so called handle grammar and show that this increases the grammar size by a factor of $\mathcal{O}(r)$, where r is the maximal rank of symbols from $\mathbb{F}$ occurring in T. Then, we compare the size of a minimal handle grammar for T with the size of the output of TtoG.

A *handle* is a pattern $t(y) = f(w_1(\gamma_1), w_2(\gamma_2), \dots, w_{i-1}(\gamma_{i-1}), y, w_{i+1}(\gamma_{i+1}), \dots, w_\ell(\gamma_\ell))$, where $\operatorname{rank}(f) = \ell$, every γ_j is either a constant symbol or a nonterminal of rank 0, every w_j is a chain pattern, and y is a parameter. Note that $a(y)$ for a unary letter a is a handle. Since handles have one parameter only, for handles $h_1, h_2, \dots, h_\ell$ we write $h_1 h_2 \cdots h_\ell$ for the tree $h_1(h_2(\dots(h_\ell(y))))$ and treat it as a string, similarly to chains patterns. We say that an SLCF grammar $\mathbb{G}$ is a *handle grammar* (or simply "$\mathbb{G}$ is handle") if the following conditions hold:

(H1) $N \subseteq \mathbb{N}_0 \cup \mathbb{N}_1$

(H2) For $A \in N \cap \mathbb{N}_1$ the unique rule for A is of the form $A(y) \to u(B(v(C(w(y)))))$ or $A(y) \to u(B(v(y)))$ or $A(y) \to u(y)$, where u, v, and w are (perhaps empty) sequences of handles and $B, C \in N_1$. We call B the *first* and C the *second* nonterminal in the rule for A.

(H3) For $A \in N \cap \mathbb{N}_0$ the rule for A is of the (similar) form $A \to u(B(v(C)))$ or $A \to u(B(v(c)))$ or $A \to u(C)$ or $A \to u(c)$, where u and v are (perhaps empty) sequences of handles, c is a constant, $B \in N_1$, $C \in N_0$, and $j, k < i$. Again we speak of the first and second nonterminal in the rule for A.

Note that the representation of the rules for nonterminals from $\mathbb{N}_0$ is not unique. Take for instance the rule $A \to f(B, C)$, which can be written as $A \to a(C)$ for the handle $a(y) = f(B, y)$ or as $A \to b(B)$ for the handle $b(y) = f(y, C)$. For nonterminals from $\mathbb{N}_1$ this problem does not occur, since there is a unique occurrence of the parameter y in the right-hand side. For a given SLCF grammar we can find an equivalent handle grammar of similar size:

▶ **Lemma 6.** *Let $\mathbb{G}$ be an SLCF grammar. Then there exists a handle grammar $\mathbb{G}'$ such that $\operatorname{val}(\mathbb{G}') = \operatorname{val}(\mathbb{G})$ and $|\mathbb{G}'| = \mathcal{O}(r|\mathbb{G}|)$, where r is the maximal rank of the letters used in $\mathbb{G}$.*

For the proof one first applies the main result of [16] to make $\mathbb{G}$ monadic (i.e., $N \subseteq \mathbb{N}_0 \cup \mathbb{N}_1$). The resulting grammar can be easily transformed into a handle grammar by considering for each nonterminal $A \in N \cap \mathbb{N}_1$ the path from the root to the unique occurrence of the parameter in the right-hand side of A.

4.2 Intuition and invariants

For a given input tree T we start with a smallest handle grammar $\mathbb{G}$ generating T. In the following, by g we always denote the size of this initial minimal handle grammar. With each occurrence of a letter from $\mathbb{F}$ in $\mathbb{G}$'s rules we associate 2 *credits*. During the run of TtoG we appropriately modify $\mathbb{G}$, so that $\operatorname{val}(\mathbb{G}) = T$ (where T always denotes the current tree in TtoG). In other words, we perform the compression steps of TtoG also on $\mathbb{G}$. We always maintain the invariant that every occurrence of a letter from $\mathbb{F}$ in $\mathbb{G}$'s rules has two credits. To this end, we *issue* some new credits during the modifications, and we have to do a precise bookkeeping on the amount of issued credit. On the other hand, if we do a compression step in $\mathbb{G}$, then we remove some occurrences of letters. The credit associated with these occurrences is then *released* and can be used to pay for the representation cost of the new letters introduced by the compression step. For unary pair compression and leaf compression, the released credit indeed suffices to pay the representation cost for the fresh letters, but for chain compression the released credit does not suffice. Here we need some extra amount that will be estimated separately. At the end, we bound the size of the grammar produced by TtoG as the sum of the initial credit assigned to $\mathbb{G}$ (at most $2g$) plus the total amount of issued credit plus the extra cost estimated in Section 4.6. We emphasize that the modification of $\mathbb{G}$ is not performed by TtoG, but is only a mental experiment done for the purpose of analyzing TtoG.

An important difference between our algorithm and the string compression algorithm from the earlier paper of the first author [9] is that we add new nonterminals to $\mathbb{G}$ during its modification. All such nonterminals will have rank 0 and we shall denote the set of such currently used nonterminals by $\widetilde{N_0}$. To simplify notation, we denote with m always the number of nonterminals of the current grammar $\mathbb{G}$, and we denote its nonterminals by $A_1, \ldots, A_m$. We assume that $i < j$ if A_i occurs in the right-hand side of A_j, and that A_m is the start nonterminal. With α_i we always denote the current right-hand side of A_i, i.e., the productions of $\mathbb{G}$ are $A_i \to \alpha_i$ for $1 \le i \le m$.

Suppose a compression step, for simplicity say an (a, b)-pair compression, is applied to T. We should also reflect it in $\mathbb{G}$. The simplest solution would be to perform the same compression on each of the rules of $\mathbb{G}$, hoping that in this way all occurrences of ab in $\mathrm{val}(\mathbb{G})$ will be replaced by c. However, this is not always the case. For instance, the 2-chain ab may occur 'between' a nonterminal and a unary letter: consider a grammar $A_1(y) \to a(y)$ and $A_2 \to A_1(b(c))$ and a 2-chain ab. Then it it occurs in $\mathrm{val}(A_2)$ but this occurrence is 'between' A_1 and b in the rule for A_2. This intuitions are made precise in Section 4.3. To deal with this problem, we modify the grammar, so that such bad cases no longer occur. Similar problems occur also when we want to replace an a-maximal chain or perform leaf compression. Solutions to those problems are similar and are given in Section 4.4 and Section 4.5, respectively.

To ensure that $\mathbb{G}$ is handle and to estimate the amount of issued credit, we show that the grammar preserves the following invariants, where n_0 (resp. n_1) is the initial number of nonterminals from N_0 (resp., N_1) in $\mathbb{G}$ and g is the initial size of $\mathbb{G}$.

(I1) $\mathbb{G}$ is handle.

(I2) $\mathbb{G}$ has nonterminals $N_0 \cup N_1 \cup \widetilde{N_0}$, where $\widetilde{N_0}, N_0 \subseteq \mathbb{N}_0$, $|N_0| \le n_0$ and $N_1 \subseteq \mathbb{N}_1$, $|N_1| \le n_1$.

(I3) The number of occurrences of nonterminals from N_0, N_1 and $\widetilde{N_0}$ in $\mathbb{G}$ are at most g, $n_0 + 2n_1$ and $(n_0 + 2n_1)(r - 1)$, respectively

(I4) The rules for $A_i \in \widetilde{N_0}$ are of the form $A_i \to wA_j$ or $A_i \to wc$, where w is a string of unary symbols, $A_j \in N_0 \cup \widetilde{N_0}$ and c is a constant.

It is easy to show that (I1)–(I4) hold for the initial handle grammar $\mathbb{G}$ when we set $\widetilde{N_0} = \emptyset$. The only non-trivial condition is that the number of occurrences of nonterminals from N_1 is at most $n_0 + 2n_1$. However, in a rule for $A_i \in N_0$ there is at most one occurrence of a nonterminal from N_1, namely the first nonterminal in this rule (all other nonterminals are parts of handles and so they are from N_0). Similarly in a rule for $A_i \in N_1$ there are at most two occurrences of nonterminals from N_1.

4.3 ($F_1^{\mathsf{up}}, F_1^{\mathsf{down}}$)-compression

We begin with some definitions that help to classify which 2-chains are easy and which hard to compress.

For a non-empty tree or pattern t its *first* letter is the letter that labels the root of t. For a pattern $t(y)$ which is not a parameter its *last* letter is the label of the node above the one labelled with y. A chain pattern ab has a *crossing occurrence* in a nonterminal A_i if one of the following holds:

(C1) $a(A_j)$ is a subpattern of α_i and the first letter of $\mathrm{val}(A_j)$ is b
(C2) $A_j(b)$ is a subpattern of α_i and the last letter of $\mathrm{val}(A_j)$ is a
(C3) $A_j(A_k)$ is a subpattern of α_i, the last letter of $\mathrm{val}(A_j)$ is a and the first letter of $\mathrm{val}(A_k)$ is b.

A chain pattern ab is *crossing* if it has a crossing occurrence in any nonterminal and *non-crossing* otherwise. Unless explicitly written, we use this notion only in case $a \neq b$.

When every chain pattern $ab \in F_1^{\mathrm{up}} F_1^{\mathrm{down}}$ is noncrossing, simulating $(F_1^{\mathrm{up}}, F_1^{\mathrm{down}})$-compression on $\mathbb{G}$ is easy: It is enough to apply $(F_1^{\mathrm{up}}, F_1^{\mathrm{down}})$-compression to each right-hand side of $\mathbb{G}$. We denote the resulting grammar with UnaryCmp($\mathbb{G}$).

Algorithm 2 Pop($F_1^{\mathrm{up}}, F_1^{\mathrm{down}}, \mathbb{G}$)

```
1: for $i \leftarrow 1 \,..\, m-1$ do
2:     if the first symbol of $\alpha_i$ is $b \in F_1^{\mathrm{down}}$ then
3:         if $\alpha_i = b$ then
4:             replace each $A_i$ $\mathbb{G}$ rules by $b$
5:         else remove this leading $b$ from $\alpha_i$
6:             replace each $A_i$ in $\mathbb{G}$ rules by $bA_i$
7:     do symmetric actions for the last symbol
```

To distinguish between the nonterminals, grammar, etc. before and after the application of UnaryCmp (or, in general, any procedure) we use 'primed' symbols, i.e. A_i', $\mathbb{G}'$, T' for the nonterminals, grammar and tree, respectively, after the compression step and 'unprimed' symbols (i.e. A_i, $\mathbb{G}$, T) for the ones before.

It is left to assure that indeed all occurrences of chain patterns from $F_1^{\mathrm{up}} F_1^{\mathrm{down}}$ are noncrossing. Consider for instance the grammar with the rules $A_1(y) \to a(y)$ and $A_2 \to A_1(b(c))$. The pattern ab has a crossing occurrence. To deal with crossing occurrences we change the grammar. In our example, we replace A_1 with a, leaving only $A_2 \to ab(c)$, which does not contain a crossing occurrence of ab.

In general, suppose that some $ab \in F_1^{\mathrm{up}} F_1^{\mathrm{down}}$ is crossing because of (C1). Let $a(A_i)$ be a subpattern of some right-hand side and let $\mathrm{val}(A_i) = b(t')$. Then it is enough to modify the rule for A_i so that $\mathrm{val}(A_i) = t'$ and replace each occurrence of A_i in a right-hand side by $b(A_i)$. We call this action *popping-up b from A_i*. The similar operation of popping down a letter a from $A_i \in N \cap \mathbb{N}_1$ is symmetrically defined (note that both pop operations apply only to unary letters). By Lemma 7 below, popping up and down removes all crossing occurrences of ab. Note that the popping up and popping down can be performed for many letters in parallel: The procedure Pop (Algorithm 2) 'uncrosses' all occurrences of patterns from the set $F_1^{\mathrm{up}} F_1^{\mathrm{down}}$, assuming that F_1^{up} and F_1^{down} are disjoint subsets of F_1. Then, $(F_1^{\mathrm{up}}, F_1^{\mathrm{down}})$-compression can be simulated on $\mathbb{G}$ by first uncrossing all 2-chains from $F_1^{\mathrm{up}} F_1^{\mathrm{down}}$ followed by $(F_1^{\mathrm{up}}, F_1^{\mathrm{down}})$-compression.

▶ **Lemma 7.** *Let* $\mathbb{G}$ *satisfy (I1)–(I4) and* $\mathbb{G}' = \text{UnaryCmp}(F_1^{up}, F_1^{down}, \text{Pop}(F_1^{up}, F_1^{down}, \mathbb{G}))$. *Then* $\mathrm{val}(\mathbb{G}') = \text{UnaryCmp}(F_1^{up}, F_1^{down}, \mathrm{val}(\mathbb{G}))$ *and* $\mathbb{G}'$ *satisfies (I1)–(I4).* $\mathcal{O}(g + (n_0 + n_1)r)$ *credits are issued in the construction of* $\mathbb{G}'$*, where* r *is the maximal rank of letters in* $\mathbb{G}$*. The issued credits and the credits released by* UnaryCmp *cover the representation cost of fresh letters as well as their credits.*

Since by Lemma 4 we apply $\mathcal{O}(\log n)$ many $(F_1^{\text{up}}, F_1^{\text{down}})$-compressions (for different sets F_1^{up} and F_1^{down}) to $\mathbb{G}$, we obtain:

▶ **Corollary 8.** *(F_1^{up}, F_1^{down})-compression issues in total $\mathcal{O}((g + (n_0 + n_1)r) \log n)$ credits during all modifications of $\mathbb{G}$.*

4.4 Chain compression

Our notations and analysis for chain compression is similar to those for $(F_1^{\text{up}}, F_1^{\text{down}})$-compression. In order to simulate chain compression on $\mathbb{G}$ we want to apply chain compression to the right-hand sides of $\mathbb{G}$. This works as long as there are no crossing chains: A unary letter *a has a crossing chain* in a rule $A_i \to \alpha_i$ if aa has a crossing occurrence in α_i, otherwise it has no crossing chain. As for $(F_1^{\text{up}}, F_1^{\text{down}})$-compression, when there are no crossing chains, we apply chain compression to the right-hand sides of $\mathbb{G}$. We denote with ChainCmp($\mathbb{G}$) the resulting grammar.

Crossing chains are eliminated by a procedure similar to Pop: Suppose for instance that a has a crossing chain because $a(A_i)$ is a subpattern in a right-hand side and $\text{val}(A_i)$ begins with a. Popping up a does not solve the problem, since after popping, $\text{val}(A_i)$ might still begin with a. Thus, we keep on popping up until the first letter of $\text{val}(A_i)$ is not a. In order to do this in one step we need some notation: We say that a^ℓ is an *a-prefix* of a tree (or pattern) t if $t = a^\ell(t')$ and the first letter of t' is not a (here t' might be the trivial pattern y). Similarly, we say that a^ℓ is an *a-suffix* of a pattern $t(y)$ if $t = t'(a^\ell(y))$ for a pattern $t'(y)$ and the last letter of t' is not a (again, t' might be the trivial pattern y). In this terminology, we have to pop-up (resp. pop-down) the whole a-prefix (resp., a-suffix) of $\text{val}(A_i)$ from of A_i in one step. This is achieved by a procedure RemCrChs, which is similar to Pop. So chain compression is done by first running RemCrChs and then ChainCmp on the right-hand sides of $\mathbb{G}$. We obtain:

▶ **Lemma 9.** *Let $\mathbb{G}$ satisfy (I1)–(I4) and $\mathbb{G}' = \text{ChainCmp}(\text{RemCrChs}(\mathbb{G}))$. Then $\text{val}(\mathbb{G}') = \text{ChainCmp}(\text{val}(\mathbb{G}))$ and $\mathbb{G}'$ satisfies (I1)–(I4). $\mathcal{O}(g + (n_0 + n_1)r)$ credits are issued in the construction of $\mathbb{G}'$ and these credits are used to pay the credits for the fresh letters introduced by ChainCmp (but not their representation cost).*

Since by Lemma 4 we apply $\mathcal{O}(\log n)$ many chain compressions to $\mathbb{G}$, we get:

▶ **Corollary 10.** *Chain compression issues in total $\mathcal{O}((g + (n_0 + n_1)r) \log n)$ credits during all modifications of $\mathbb{G}$.*

The representation cost for the new letters a_ℓ introduced by chain compression is addressed in Section 4.6.

4.5 Leaf compression

In order to simulate leaf compression on $\mathbb{G}$ we perform similar operations as for $(F_1^{\text{up}}, F_1^{\text{down}})$-compression: Ideally we would like to apply leaf compression to each right-hand side of $\mathbb{G}$. However, in some cases this does not return the appropriate result. We say that the pair (f, a) is a *crossing parent-leaf pair* in $\mathbb{G}$, if $f \in F_{\geq 1}$, $a \in F_0$, and one of the following holds:

(L1) $f(t_1, \ldots, t_\ell)$ is a subtree of some right-hand side of $\mathbb{G}$, where for some j we have $t_j = A_k$ and $\text{val}(A_k) = a$.

(L2) For some $A_i \in N_1$, $A_i(a)$ is a subtree of some right-hand side of $\mathbb{G}$ and the last letter of $\text{val}(A_i)$ is f.

(L3) For some $A_i \in N_1$ and $A_k \in N_0 \cup \widetilde{N_0}$, $A_i(A_k)$ is a subtree of some right-hand side of $\mathbb{G}$, the last letter of $\text{val}(A_i)$ is f, and $\text{val}(A_k) = a$.

When there is no crossing parent-leaf pair, we can apply leaf compression to each right-hand side of a rule; denote the resulting grammar with LeafCmp($\mathbb{G}$). If there is a crossing parent-leaf pair, we uncross them all by a generalisation of Pop, called GenPop, which pops up letters from F_0 and pops down letters from $F_{\geq 1}$. The latter requires some generalisation: If we want to pop down a letter of rank > 1, we need to pop a whole handle. This adds new nonterminals to $\mathbb{G}$ as well as a large number of new letters and hence a large amount of credit, so we need to be careful. There are two crucial details:

- When we pop down a whole handle $h = f(t_1, \dots, t_k, y, t_{k+1}, \dots, t_\ell)$, we add to the set $\widetilde{N}_0$ fresh nonterminals for all trees t_i that are non-constants, replace these t_i in h by their corresponding nonterminals and then pop down the resulting handle. In this way the issued credit is reduced and no new occurrence of nonterminals from $N_0 \cup N_1$ is created.
- We do not pop down a handle from every nonterminal, but do it only when it is needed, i.e., if for $A_i \in N_1$ one of the cases (L2) or (L3) holds. This allows preserving (I5). Note that when the last symbol in the rule for A_i is not a handle but another nonterminal, this might cause a need for recursive popping. So we perform the whole popping down in a depth-first-search style.

So, for leaf compression we can proceed as in the case of $(F_1^{\text{up}}, F_1^{\text{down}})$-compression and chain compression: We first uncross all parent-leaf pairs and then compress each right-hand side independently.

▶ **Lemma 11.** *Let $\mathbb{G}$ satisfy (I1)–(I4) and $\mathbb{G}' =$ LeafCmp(GenPop($\mathbb{G}$)). Then $\text{val}(\mathbb{G}') =$ LeafCmp(val($\mathbb{G}$)) and $\mathbb{G}'$ satisfies (I1)–(I4). $\mathcal{O}(g + (n_0 + n_1)r)$ credits are issued in the construction of $\mathbb{G}'$. The issued credit and the credit released by* LeafCmp *cover the representation cost of fresh letters as well as their credit.*

Since by Lemma 4 we apply $\mathcal{O}(\log n)$ many leaf compressions to $\mathbb{G}$, we obtain:

▶ **Corollary 12.** *Leaf compression issues in total $\mathcal{O}(((n_0 + n_1)r + g) \log n)$ credits during all modifications of $\mathbb{G}$.*

4.6 Calculating the total cost of representing letters

The issued credit of (which is $\mathcal{O}(((n_0 + n_1)r + g) \log n)$ by Corollaries 8, 10, and 12) is enough to pay the 2 credits for every letter introduced during popping, whereas the released credit covers the representation cost for the new letters introduced by $(F_1^{\text{up}}, F_1^{\text{down}})$-compression and leaf compression. However, the released credit *does not* cover the representation cost for letters created during chain compression. The appropriate analysis is similar to [9]. The idea is as follows: Firstly, we define a scheme of representing letters introduced by chain compression based on the grammar $\mathbb{G}$ and the way $\mathbb{G}$ is changed by chain compression (the $\mathbb{G}$-*based representation*). Then, we show that for this scheme the representation cost is bounded by $\mathcal{O}((g + (n_0 + n_1)r) \log n)$. Lastly, it is proved that the actual representation cost of letters introduced by chain compression during the run of TtoG (the TtoG-*based representation*, whose cost is given by Lemma 2) is smaller than the $\mathbb{G}$-based one. Hence, it is bounded by $\mathcal{O}((g + (n_0 + n_1)r) \log n)$, too. Adding this to the issued credit, we obtain the main result of the paper:

▶ **Corollary 13.** *The total representation cost of the letters introduced by* TtoG *(and hence the size of the grammar produced by* TtoG*) is $\mathcal{O}((g + (n_0 + n_1)r) \log n) \leq \mathcal{O}(g \cdot r \cdot \log n)$, where g is the size of a minimal handle grammar for the input tree T and r the maximal rank of symbols in T.*

Together with Lemma 6 we get:

▶ **Corollary 14.** *The size of the grammar produced by* TtoG *is* $\mathcal{O}(g\, r^2 \log n)$*, where* g *is the size of a minimal SLCF grammar for the input tree* T *and* r *is the maximal rank of symbols in* T*.*

Acknowledgements. The first author would like to thank P. Gawrychowski for introducing him to the topic of compressed data and discussions, as well as S. Maneth and S. Böttcher for the question of applicability of the recompression-based approach to the tree case.

References

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

2 P. Bille, I. Gørtz, G. Landau, and O. Weimann. Tree compression with top trees. In *Proc. ICALP 2013 (1)*, LNCS 7965, pp.160–171. Springer, 2013.

3 G.Busatto, M. Lohrey, and S. Maneth. Efficient memory representation of XML document trees. *Information Systems*, 33(4–5):456–474, 2008.

4 M. Charikar, E. Lehman, D. Liu, R. Panigrahy, M. Prabhakaran, A. Sahai, and A. Shelat. The smallest grammar problem. *IEEE Trans. Inf. Theory*, 51(7):2554–2576, 2005.

5 F. Claude and G. Navarro. Fast and compact web graph representations. *ACM Trans. Web*, 4(4), 2010.

6 T. Cormen, C. Leiserson, R. Rivest, and C. Stein. *Introduction to Algorithms (3. ed.)*. MIT Press, 2009.

7 A. Jeż. Compressed membership for NFA (DFA) with compressed labels is in NP (P). In *Proc. STACS 2012*, vol. 14 of *LIPIcs*, pp.136–147, Leibniz-Zentrum für Informatik, 2012.

8 A. Jeż. Faster fully compressed pattern matching by recompression. In *Proc. ICALP 2012 (1)*, LNCS 7391, pp.533–544. Springer, 2012.

9 A. Jeż. Approximation of grammar-based compression via recompression. In *Proc. CPM 2013*, LNCS 7922, pp.165–176. Springer, 2013. full version at http://arxiv.org/abs/1301.5842.

10 A. Jeż. One-variable word equations in linear time. In *Proc. ICALP 2013 (2)*, LNCS 7966, pp.324–335. Springer, 2013.

11 A. Jeż. Recompression: a simple and powerful technique for word equations. In *Proc. STACS 2013*, volume 20 of *LIPIcs*, pp.233–244, Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2013.

12 N. Jesper Larsson and A. Moffat. Offline dictionary-based compression. In *Proc. DCC 1999*, pp.296–305. IEEE Computer Society Press, 1999.

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

14 M. Lohrey, S. Maneth, and R. Mennicke. XML tree structure compression using RePair. *Inf. Syst.*, 38(8):1150–1167, 2013.

15 M. Lohrey, S. Maneth, and E. Nöth. XML compression via DAGs. In *Proc. ICDT 2013*, pp.69-80, ACM, 2013.

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

17 W. Rytter. Application of Lempel-Ziv factorization to the approximation of grammar-based compression. *Theor. Comput. Sci.*, 302(1-3):211–222, 2003.

18 H. Sakamoto. A fully linear-time approximation algorithm for grammar-based compression. *J. Discrete Algorithms*, 3(2-4):416–430, 2005.