Balancing Straight-Line Programs

Moses Ganardi, Markus Lohrey
Universität Siegen
Siegen, Germany
Email: {ganardi,lohrey}@eti.uni-siegen.de

Artur Jeż
University of Wrocław
Wrocław, Poland
aje@cs.uni.wroc.pl

***Abstract*—We show that a context-free grammar of size m that produces a single string w of length n (such a grammar is also called a string straight-line program) can be transformed in linear time into a context-free grammar for w of size $\mathcal{O}(m)$, whose unique derivation tree has depth $\mathcal{O}(\log n)$. This solves an open problem in the area of grammar-based compression, improves many results in this area and greatly simplifies many existing constructions. Similar results are stated for two formalisms for grammar-based tree compression: top dags and forest straight-line programs. These balancing results can be all deduced from a single meta theorem stating that the depth of an algebraic circuit over an algebra with a certain finite base property can be reduced to $\mathcal{O}(\log n)$ with the cost of a constant multiplicative size increase. Here, n refers to the size of the unfolding (or unravelling) of the circuit. In particular, this results applies to standard arithmetic circuits over (non-commutative) semirings. A long version of the paper can be found in [1].**

***Keywords*-grammar-based compression; balancing; straight-line programs; random access problem**

I. Introduction

Grammar-based string compression: In *grammar-based compression* a combinatorial object is compactly represented using a grammar of an appropriate type. In many grammar-based compression formalisms such a grammar can be exponentially smaller than the object itself. A well-studied example of this general idea is grammar-based string compression using context-free grammars that produce only one string, which are also known as *straight-line programs*. Since the term "straight-line programs" is used in the literature for different kinds of objects (e.g. arithmetic straight-line programs) and we will also deal with different types of straight-line programs, we use the term *string straight-line program*, SSLP for short. Grammar-based string compression is tightly related to dictionary based compression: the famous LZ78 algorithm can be viewed as a particular grammar-based compressor, the number of phrases in the LZ77-factorization is a lower bound for the smallest SSLP for a string [2], and an LZ77-factorization of length m can be converted to an SSLP of size $\mathcal{O}(m \cdot \log n)$ where n is the length of the string [2]–[5]. For various other aspects of grammar-based string compression see [3], [6].

Balancing string straight-line programs: The two important measures for an SSLP are size and depth. To define these measures, it is convenient to assume that all right-hand sides of the grammar have length two (as in Chomsky normal form). Then, the size $|\mathcal{G}|$ of an SSLP $\mathcal{G}$ is the number of variables (nonterminals) of $\mathcal{G}$ and the depth of $\mathcal{G}$ ($\mathrm{depth}(\mathcal{G})$ for short) is the depth of the unique derivation tree of $\mathcal{G}$. It is straightforward to show that any string s of length n can be produced by an SSLP of size $\mathcal{O}(n)$ and depth $\mathcal{O}(\log n)$. A more difficult problem is to balance a given SSLP: assume that the SSLP $\mathcal{G}$ produces a string of length n. Several authors have shown that one can restructure $\mathcal{G}$ in time $\mathcal{O}(|\mathcal{G}| \cdot \log n)$ into an equivalent SSLP $\mathcal{H}$ of size $\mathcal{O}(|\mathcal{G}| \cdot \log n)$ and depth $\mathcal{O}(\log n)$ [2], [3], [5].

Finding SSLPs of small size and small depth is important in many algorithmic applications. A prominent example is the *random access problem for grammar-compressed strings*: For a given SSLP $\mathcal{G}$ that produces the string s of length n and a given position $p \in [1, n]$ one wants to access the p-th symbol in s. As observed in [7] one can solve this problem in time $\mathcal{O}(\mathrm{depth}(\mathcal{G}))$ (assuming arithmetic operations on numbers from the interval $[0, n]$ use constant time). Using sophisticated data structures, the authors of [7] computed from $\mathcal{G}$ a data structure of size $\mathcal{O}(|\mathcal{G}|)$ (measured in words of bit length $\log n$) which allows to access every position in time $\mathcal{O}(\log n)$. Alternatively, one can obtain $\mathcal{O}(\log n)$ access time using one of the known SSLP balancing procedures [2], [3], but this increases the size to $\mathcal{O}(|\mathcal{G}| \cdot \log n)$ [7].

Our main result for string straight-line programs states that SSLP balancing is in fact possible with a constant blow-up in size.

Theorem I.1. *Given an SSLP $\mathcal{G}$ producing a string of length n one can construct in linear time an equivalent SSLP $\mathcal{H}$ of size $\mathcal{O}(|\mathcal{G}|)$ and depth $\mathcal{O}(\log n)$.*

As a corollary we obtain a very simple and clean algorithm for the random access problem with access time $\mathcal{O}(\log n)$ that uses a data structure of size $\mathcal{O}(m)$ (in words of bit length $\log n$). We can also obtain an algorithm for the random access problem with running time $\mathcal{O}(\log n / \log \log n)$ using $\mathcal{O}(m \cdot \log^{\epsilon} n)$ words; previously this bound was only shown for balanced SSLPs [8]. Section V contains a list of further applications of Theorem I.1, which include the following problems on SSLP-compressed strings: rank and

2575-8454/19/$31.00 ©2019 IEEE
DOI 10.1109/FOCS.2019.00073

IEEE computer society

select queries [8], subsequence matching [9], computing Karp-Rabin fingerprints [10], computing runs, squares, and palindromes [11], real-time traversal [12], [13] and range-minimum queries [14]. In all these applications we either improve existing results or significantly simplify existing proofs by replacing depth($\mathcal{G}$) by $\mathcal{O}(\log n)$ in time/space bounds.

Computational model: Our balancing procedure involves (simple) arithmetic on lengths, i.e., numbers of order n. Thus the linear running time can be achieved assuming that machine words have $\Omega(\log n)$ bits. Otherwise the running time increases by a multiplicative $\log n$ factor. Note that such an assumption is realistic and standard in the field: machine words of bit length $\Omega(\log n)$ are needed, say, for indexing positions in the represented string. On the other hand, our procedure works in the pointer model regime.

Balancing forest straight-line programs and top dags: Grammar-based compression has been generalized from strings to ordered ranked node-labelled trees. In fact, the representation of a tree t by its smallest directed acyclic graph (DAG) is a form of grammar-based tree compression. This DAG is obtained by merging nodes where the same subtree of t is rooted. It can be seen as a regular tree grammar that produces only t. A drawback of DAG-compression is that the size of the DAG is lower-bounded by the height of the tree t. Hence, for deep narrow trees (like for instance caterpillar trees), the DAG-representation cannot achieve good compression. This can be overcome by representing a tree t by a linear context-free tree grammar that produces only t. Such grammars are also known as *tree straight-line programs* in the case of ranked trees [15]–[17] and *forest straight-line programs* in the case of unranked trees [18]. The latter are tightly related to *top dags* [18]–[22], which are another tree compression formalism, also akin to grammars. Our balancing technique works similarly for those types of compression:

Theorem I.2. *Given a top dag/forest straight-line program/tree straight-line program $\mathcal{G}$ producing the tree t one can compute in time $\mathcal{O}(|\mathcal{G}|)$ a top dag/forest straight-line program/tree straight-line program $\mathcal{H}$ for t of size $\mathcal{O}(|\mathcal{G}|)$ and depth $\mathcal{O}(\log |t|)$.*

For top dags, this solves an open problem from [19], where it was proved that from a tree t of size n, whose minimal DAG has size m (measured in number of edges in the DAG), one can construct in linear time a top dag for t of size $\mathcal{O}(m \cdot \log n)$ and depth $\mathcal{O}(\log n)$. It remained open whether one can get rid of the additional factor $\log n$ in the size bound. For the specific top dag constructed in [19], it was shown in [20] that the factor $\log n$ in the size bound $\mathcal{O}(m \cdot \log n)$ cannot be avoided. On the other hand, our results yield another top dag of size $\mathcal{O}(m)$ and depth $\mathcal{O}(\log n)$. To see this note that one can easily convert the minimal DAG of t into a top dag of roughly the same size, which can then be balanced. This also gives an alternative proof of a result from [21], according to which one can construct in linear time a top dag of size $\mathcal{O}(n/\log_\sigma n)$ and depth $\mathcal{O}(\log n)$ for a given tree of size n containing σ many different node labels.

Balancing circuits over algebras: Our balancing results for SSLPs, top-dags, forests straight-line programs and tree straight-line programs are all instances of a general balancing result that applies to a large class of circuits over algebraic structures. To see the connection between circuits and straight-line programs, consider SSLPs as an example. An SSLP is the same thing as a bounded fan-in circuit over a free monoid. The circuit gates compute the concatenation of their inputs and correspond to the variables of the SSLP. In general, for any algebra one can define straight-line programs, which coincide with the classic notion of a circuit.

The definition of a class of algebras, to which our general balancing technique applies, uses *unary linear term functions*, which were also used for instance in the context of efficient parallel evaluation of expression trees [23]. Fix an algebra $\mathcal{A}$ (a set together with finitely many operations of possibly different arities). For some of our applications we have to allow multi-sorted algebras that have several carrier sets (think for instance of a vector space, where the two carrier sets are an abelian group and a field of scalars). A unary linear term function is a unary function on $\mathcal{A}$ that is computed by a term (or algebraic expression) that contains a single variable x (which stands for the function argument) and, moreover, x occurs exactly once in the term. For instance, a unary linear term function over a commutative ring is of the form $x \mapsto ax + b$ for ring elements a, b. A *subsumption base* for an algebra $\mathcal{A}$ is, roughly speaking, a finite set $C(\mathcal{A})$ of unary linear term functions that are described by terms with parameters such that every unary linear term function can be obtained from one of the terms in $C(\mathcal{A})$ by instantiating the parameters. In the above example for a commutative ring the set $C(\mathcal{A})$ consists of the single term $ax + b$, where a and b are the parameters.

Our general balancing result needs one more concept, namely the *unfolded size* of a circuit $\mathcal{G}$. It can be conveniently defined as follows: we replace in $\mathcal{G}$ every input gate by the number 1, and we replace every internal gate by an addition gate. The unfolded size of $\mathcal{G}$ is the number to which this additive circuit evaluates too. In other words, this is the size of the tree obtained by unravelling $\mathcal{G}$ into a tree. Note that the size of this unfolding can be exponential in the circuit size. Now we can state the general balancing result:

Theorem I.3 (Informal statement)**.** *Let $\mathcal{A}$ be a multi-sorted algebra with a finite number of operations (of arbitrary arity) such that $\mathcal{A}$ has a finite subsumption base. Given a circuit $\mathcal{G}$ over $\mathcal{A}$ whose unfolded size is n, one can compute in time $\mathcal{O}(|\mathcal{G}|)$ a circuit $\mathcal{H}$ evaluating to the same element of $\mathcal{A}$ such that $|\mathcal{H}| \in \mathcal{O}(|\mathcal{G}|)$ and depth$(\mathcal{H}) \in \mathcal{O}(\log n)$.*

Theorems I.1 and I.2 are immediate corollaries of Theorem I.3. Theorem I.3 can be also applied to not necessarily commutative semirings, as every semiring has a finite subsumption base. Hence, for every semiring circuit one can reduce with a linear size blow-up the depth to $\mathcal{O}(\log n)$, where n is the size of the circuit unfolding.

Note that in the depth bound $\mathcal{O}(\log n)$ in our balancing result for string straight-line programs (Theorem I.1), n refers to the length of the produced string. A string straight-line program can be viewed as a circuit for a non-commutative semiring circuit that produces a single monomial (the symbols in the string correspond to the non-commuting variables). If one considers arbitrary circuits over non-commutative semirings (that produce a sum of more than one monomial), depth reduction is not possible in general by a result of Kosaraju [24]. For circuits over commutative semirings depth reduction is possible by a seminal result of Valiant, Skyum, Berkowitz and Rackoff [25]: for any commutative semiring, every circuit of size m and formal degree d can be transformed into an equivalent circuit of depth $\mathcal{O}(\log m \log d)$ and size polynomial in m and d. This result led to many further investigations on depth reduction for bounded degree circuits over various classes of commutative as well as non-commutative semirings [26]. If one drops the restriction to bounded degree circuits, then depth reduction gets even harder. For general Boolean circuits, the best known result states that every Boolean circuit of size m is equivalent to a Boolean circuit of depth $\mathcal{O}(m/\log m)$ [27].

Proof strategy: The proof of Theorem I.1 consists of two main steps (the general result Theorem I.3 is shown similarly). Take an SSLP $\mathcal{G}$ for the string s of length n and let m be the size of $\mathcal{G}$. We consider the derivation tree t for $\mathcal{G}$; it has size $\mathcal{O}(n)$. The SSLP $\mathcal{G}$ can be viewed as a DAG for t of size m. We decompose this DAG into node-disjoint paths such that each path from the root to a leaf intersects $\mathcal{O}(\log n)$ paths from the decomposition (Section II). Each path from the decomposition is then viewed as a string of integer-weighted symbols, where the weights are the lengths of the strings derived from nodes that branch off from the path. For this weighted string we construct an SSLP of linear size that produces all suffixes of the path in a weight-balanced way (Section III). Plugging these SSLPs together yields the final balanced SSLP.

Some of the concepts of our construction can be traced back to the area of parallel algorithms: the path decomposition for DAGs from Section II is related to the centroid path decomposition of trees [28], where it is the key technique in several parallel algorithms on trees. Moreover, the SSLP of linear size that produces all suffixes of a weighted string with (Section III) can be seen as a weight-balanced version of the optimal prefix sum algorithm.

For the general result Theorem I.3 we need another ingredient: when the above construction is used for circuits over algebras, the corresponding procedure produces a tree straight-line program for the unfolding of the circuit. We show that if the underlying algebra $\mathcal{A}$ has a finite subsumption base, then one can compute from a tree straight-line program an equivalent circuit over $\mathcal{A}$. Moreover, the size and depth of this circuit are linearly bounded in the size and depth of the tree straight-line program. This construction was used before for the special cases of semirings and regular expressions [29], [30].

II. The symmetric centroid decomposition of a DAG

We start with a new decomposition of a DAG (directed acyclic graph) into disjoint paths. We believe that this decomposition might have further applications. For trees, several decompositions into disjoint paths with the additional property that every path from the root to a leaf only intersects a logarithmic number of paths from the decomposition exist. Examples are the heavy path decomposition [31] and centroid decomposition [28]. These decompositions can be also defined for DAGs but a technical problem is that the resulting paths are no longer disjoint and form, in general, a subforest of the DAG, see e.g. [7].

Our new path decomposition can be seen as a symmetric form of the centroid decomposition of [28]. Consider a DAG $\mathcal{D} = (V, E)$ with node set V and the set of multi-edges E, i.e., E is a finite subset of $V \times \mathbb{N} \times V$ such that $(u, d, v) \in E$ implies that for ever $1 \leq i < d$ there exists v' with $(u, i, v') \in E$. Intuitively, (u, d, v) is the d-th outgoing edge of u. We assume that there is a single root node $r \in V$, i.e., r is the unique node with no incoming edges. Hence, all nodes are reachable from r. A path from $u \in V$ to $v \in V$ is a sequence of edges $(v_0, d_1, v_1), (v_1, d_2, v_2), \ldots, (v_{p-1}, d_p, v_p)$ where $u = v_0$ and $v = v_p$. We also allow the empty path from u to u. With $\pi(u, v)$ we denote the number of paths from u to v, and for $V' \subseteq V$ let $\pi(u, V') = \sum_{v \in V'} \pi(u, v)$. Let $W \subseteq V$ be the set of sink nodes of $\mathcal{D}$, i.e., those nodes without outgoing edges, and let $n(\mathcal{D}) = \pi(r, W)$. This is the number of leaves in the tree obtained by unfolding $\mathcal{D}$ into a tree. With a node $v \in V$ we assign the pair $\lambda_{\mathcal{D}}(v) = (\lfloor \log_2 \pi(r, v) \rfloor, \lfloor \log_2 \pi(v, W) \rfloor)$. If $\lambda_{\mathcal{D}}(v) = (k, \ell)$, then $k, \ell \leq \lfloor \log_2 n(\mathcal{D}) \rfloor$ because $\pi(r, v)$ and $\pi(v, W)$ are both bounded by $n(\mathcal{D})$. Let us now define the edge set $E_{\text{scd}}(\mathcal{D})$ ("scd" stands for symmetric centroid decomposition) as

$$E_{\text{scd}}(\mathcal{D}) = \{(u, i, v) \in E \colon \lambda_{\mathcal{D}}(u) = \lambda_{\mathcal{D}}(v)\}.$$

Example II.1. Figure 1(left) shows the symmetric centroid path decomposition of a DAG. The numbers in a node v are the values $\pi(r, v)$ and $\pi(v, W)$ where r is the root and W consists of the two sink nodes. Edges that belong to a symmetric centroid path are drawn in red. Note that the 9 topmost nodes form a symmetric centroid path since $\lfloor \log_2 \pi(r, v) \rfloor = 0$ and $\lfloor \log_2 \pi(v, W) \rfloor = 5$ for each of

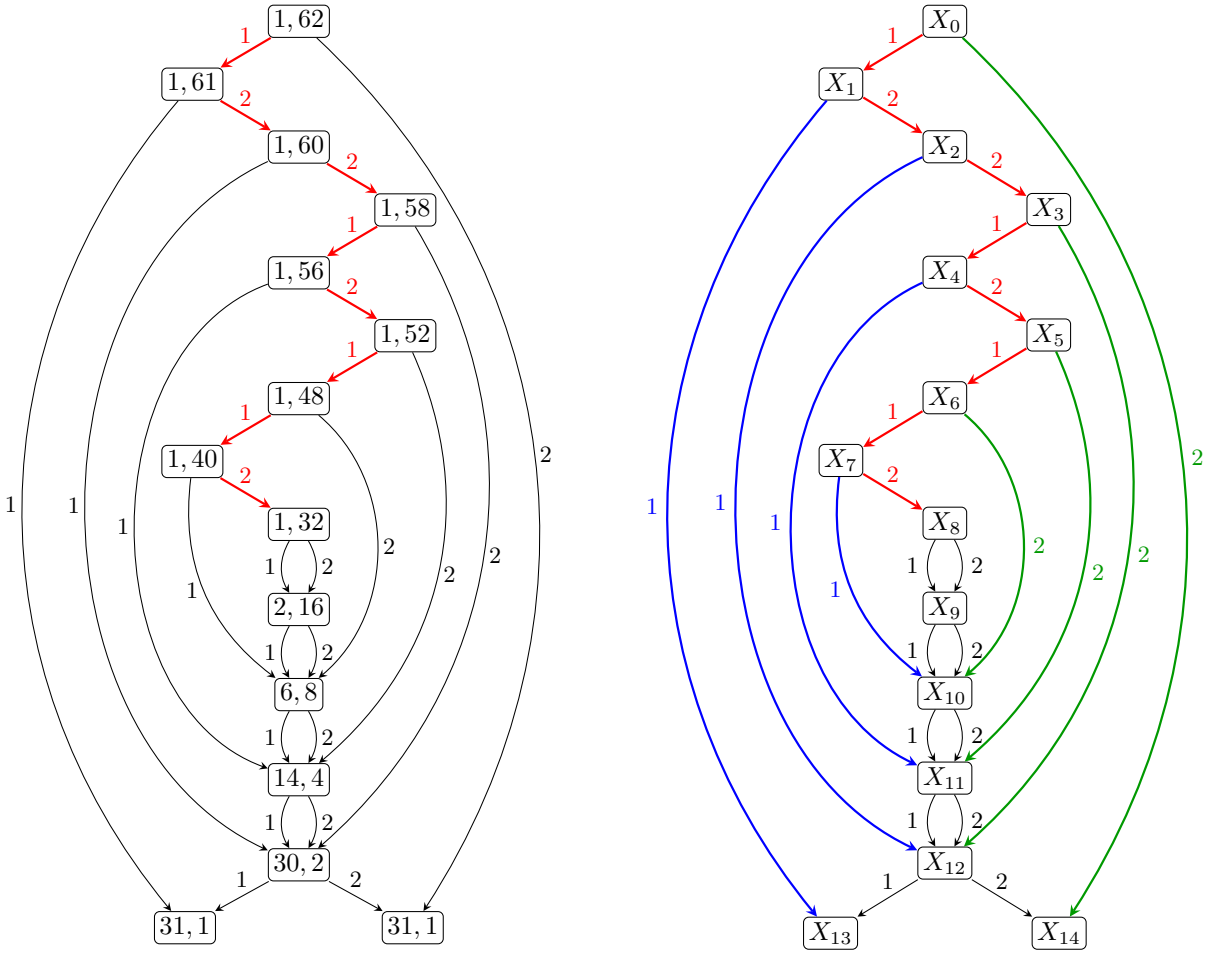

Figure 1. The DAG for an SSLP and its symmetric centroid path decomposition.

these nodes. In this example the symmetric centroid path decomposition consists of one path of length 8 (number of edges); all other nodes form symmetric centroid paths of length zero.

Lemma II.2. *Let $\mathcal{D} = (V, E)$ be a DAG with $n = n(\mathcal{D})$. Then every node has at most one outgoing and at most one incoming edge from $E_{scd}(\mathcal{D})$. Furthermore, every path from the root r to a sink node contains at most $2\log_2 n$ edges that do not belong to $E_{scd}(\mathcal{D})$.*

Proof: Consider a node $v \in V$ and assume that v has two different outgoing edges $(u, i, v), (u, j, w) \in E_{\text{scd}}(\mathcal{D})$. Hence, $\lambda_{\mathcal{D}}(u) = \lambda_{\mathcal{D}}(v) = \lambda_{\mathcal{D}}(w)$. Let $\lambda_{\mathcal{D}}(u) = (k, \ell)$. If W is the set of sinks, we get $\pi(u, W) \geq \pi(v, W) + \pi(w, W)$ (since we consider paths of multi-edges, this inequality also holds for $v = w$). W.l.o.g. assume that $\pi(w, W) \geq \pi(v, W)$ and thus $\pi(u, W) \geq 2\pi(v, W)$. We get

$$\lfloor \log_2 \pi(u, W) \rfloor \geq 1 + \lfloor \log_2 \pi(v, W) \rfloor = 1 + \lfloor \log_2 \pi(u, W) \rfloor,$$

where the last equality follows from $\lambda_{\mathcal{D}}(u) = \lambda_{\mathcal{D}}(v)$. This is a contradiction and proves the claim for outgoing edges. Incoming edges are treated similarly, this time using $\pi(r, v)$.

For the second claim of the lemma, consider a path $(v_0, d_1, v_1), (v_1, d_2, v_2), \ldots, (v_{p-1}, d_p, v_p)$, where v_0 is the root and v_p is a sink. Let $\lambda_{\mathcal{D}}(v_i) = (k_i, \ell_i)$. We must have $k_i \leq k_{i+1}$ and $\ell_i \geq \ell_{i+1}$ for all $0 \leq i \leq p-1$. Moreover, $k_0 = \ell_p = 0$ and $\ell_0, k_p \leq \lfloor \log_2 n \rfloor$. Consider now an edge $(v_i, d_i, v_{i+1}) \in E \setminus E_{\text{scd}}(\mathcal{D})$. Since $\lambda_{\mathcal{D}}(v_i) \neq \lambda_{\mathcal{D}}(v_{i+1})$, we have $k_i < k_{i+1}$ or $\ell_i > \ell_{i+1}$. Hence, there can be at most $2\lfloor \log_2 n \rfloor \leq 2\log_2 n$ edges from $E \setminus E_{\text{scd}}(\mathcal{D})$ on the path. ■

Lemma II.2 implies that the subgraph $(V, E_{\text{scd}}(\mathcal{D}))$ is a disjoint union of possibly empty paths, called *symmetric centroid paths* of $\mathcal{D}$. It is straight-forward to compute the edge set $E_{\text{scd}}(\mathcal{D})$ in time $\mathcal{O}(|\mathcal{D}|)$, where $|\mathcal{D}|$ is defined as the number of edges of the DAG: By traversing $\mathcal{D}$ in both directions (from the root to the sinks and from the sinks to the root) one can compute all pairs $\lambda_{\mathcal{D}}(v)$ for $v \in V$ in linear time. For this, we need the ability to add numbers of size at most $n = n(\mathcal{D})$ in constant time. This is in accordance with

our machine model (see the introduction). When we apply Lemma II.2 to string straight-line programs then $n(\mathcal{D})$ will be exactly the length of the string produced by the string straight-line program.

One can use Lemma II.2 in order to simplify the original proof of [7, Corollary V.1] (random access for grammar-compressed strings): in [7], the authors use the heavy-path decomposition of the derivation tree of an SSLP. In the SSLP (viewed as a DAG that defines the derivation tree), these heavy paths lead to a forest, called the heavy path forest [7]. The important property used in [7] is the fact that any path from the root of the DAG to a sink node contains only $\mathcal{O}(\log n)$ edges that do not belong to a heavy path, where n is the length of string produced by the SSLP. Using Lemma II.2, one can replace this heavy path forest by the decomposition into symmetric centroid paths. The fact that the latter is a disjoint union of paths in the DAG simplifies the technical details in [7] a lot. On the other hand, [7, Corollary V.1] follows directly from Theorem I.1, see Section IV.

III. String straight-line programs and suffixes of weighted strings

In the following, we define string straight-line programs formally. Given an alphabet of symbols Σ, Σ^* denotes the set of all finite strings over the alphabet Σ, including the empty string ε. The set of non-empty strings is denoted by $\Sigma^+ = \Sigma^* \setminus \{\varepsilon\}$. The length of a string w is denoted with $|w|$.

Let Σ be a finite alphabet of terminal symbols. A string straight-line program (SSLP for short) over the alphabet Σ is a triple $\mathcal{G} = (\mathcal{V}, \rho, S)$, where $\mathcal{V}$ is a finite set of variables, $S \in \mathcal{V}$ is the start variable, and $\rho\colon \mathcal{V} \to (\Sigma \cup \mathcal{V})^*$ (the right-hand side mapping) has the property that the binary relation $E(\mathcal{G}) = \{(X, Y) \in \mathcal{V} \times \mathcal{V}\colon Y \text{ occurs in } \rho(X)\}$ is acyclic. This allows to define for every string $w \in (\Sigma \cup \mathcal{V})^*$ a string $[\![w]\!]_{\mathcal{G}} \in \Sigma^*$. For this we extend ρ to a morphism by setting $\rho(a) = a$ for $a \in \Sigma$ and $\rho(\alpha_1 \cdots \alpha_n) = \rho(\alpha_1) \cdots \rho(\alpha_n)$ for $\alpha_1, \ldots, \alpha_n \in \Sigma \cup \mathcal{V}$. The acyclicity of $E(\mathcal{G})$ implies that the m-fold application $\rho^m(w)$ of ρ to $w \in (\Sigma \cup \mathcal{V})^*$, where $m = |\mathcal{V}|$ (any larger value for m would be also fine), is a string over the terminal alphabet Σ. We set $[\![w]\!]_{\mathcal{G}} = \rho^m(w) \in \Sigma^*$. We omit the subscript $\mathcal{G}$ if $\mathcal{G}$ is clear from the context. Finally, we define $[\![\mathcal{G}]\!] = [\![S]\!]$ (the string derived from $\mathcal{G}$).

An SSLP $\mathcal{G}$ can be seen as a context-free grammar that produces the single string $[\![\mathcal{G}]\!]$. Quite often, one assumes that every right-hand sides $\rho(X)$ is of the form $a \in \Sigma$ or YZ with $Y, Z \in \mathcal{V}$. This corresponds to the Chomsky normal form. For every SSLP $\mathcal{G}$ with $[\![\mathcal{G}]\!] \neq \varepsilon$ one can construct in linear time an equivalent SSLP in Chomsky normal form by replacing every right-hand side by a balanced binary derivation tree.

Fix an SSLP $\mathcal{G} = (\mathcal{V}, \rho, S)$. We define the size $|\mathcal{G}|$ of $\mathcal{G}$ as $\sum_{X \in \mathcal{V}} |\rho(X)|$. Let d be the length of a longest path in the DAG $(\mathcal{V}, E(\mathcal{G}))$ and $r = \max\{|\rho(X)|\colon X \in \mathcal{V}\}$. We define the depth of $\mathcal{G}$ as $\operatorname{depth}(\mathcal{G}) = d \cdot \lceil \log_2 r \rceil$. These definitions ensure that depth and size only increase by fixed constants when an SSLP is transformed into Chomsky normal form. Note that for an SSLP in Chomsky normal form, the definition of the depth simplifies to $\operatorname{depth}(\mathcal{G}) = d$.

A *weighted string* is a string $s \in \Sigma^*$ additionally equipped with a weight function $\|\cdot\|\colon \Sigma \to \mathbb{N} \setminus \{0\}$, which is extended to a homomorphism $\|\cdot\|\colon \Sigma^* \to \mathbb{N}$ by $\|a_1 a_2 \cdots a_n\| = \sum_{i=1}^{n} \|a_i\|$. If X is a variable in an SSLP $\mathcal{G}$, we also write $\|X\|$ for the weight of the string $[\![X]\!]_{\mathcal{G}}$ derived from X. Moreover, when we speak of suffixes of a string, we always mean non-empty suffixes.

Lemma III.1. *For every non-empty weighted string s of length n one can construct in linear time an SSLP $\mathcal{G}$ with the following properties:*

- *$\mathcal{G}$ contains at most $3n$ variables,*
- *all right-hand sides of $\mathcal{G}$ have length at most 4,*
- *$\mathcal{G}$ contains* suffix variables *$S_1, \ldots, S_n$ producing all suffixes of s, and*
- *every path from S_i to a terminal symbol a in the derivation tree of $\mathcal{G}$ has length $\leq 3 + 2(\log_2 \|S_i\| - \log_2 \|a\|)$.*

Proof: First, the presented algorithm never uses the fact that some letters of s may be equal. Thus it is more convenient to assume that letters in s are pairwise different—in this way the path from a variable S_i to a terminal symbol a in the last condition is defined uniquely.

For the sake of the inductive proof, the constructed SSLP will satisfy a slightly stronger and more technical variant of the last condition: every path from S_i to some terminal symbol a in the derivation tree of $\mathcal{G}$ has length at most $1 + 2(\lceil \log_2 \|S_i\| \rceil - \log_2 \|a\|)$. The trivial estimation $\lceil \log_2 \|S_i\| \rceil \leq 1 + \log_2 \|S_i\|$ then yields the last condition of the lemma.

We first show how to construct $\mathcal{G}$ with the desired properties and then prove that the construction can be done in linear time.

The case $n = 1$ is trivial. Now assume that $n \geq 2$ and let

$$s = a_1 \cdots a_k \, c \, b_1 \cdots b_m$$

where $c\, b_1 \cdots b_m$ is the shortest suffix of s such that $\lceil \log_2 \|c\, b_1 \cdots b_m\| \rceil = \lceil \log_2 \|s\| \rceil$. Clearly such a suffix exists (in the extreme cases it is the whole s or the last letter of s). Note that

$$\lceil \log_2 \|c\, b_1 \cdots b_m\| \rceil = \lceil \log_2 \|a_i \cdots a_k \, c \, b_1 \cdots b_m\| \rceil \quad (1)$$

for $1 \leq i \leq k+1$. Moreover, the following inequalities hold:

$$\lceil \log_2 \|c\, b_1 \cdots b_m\| \rceil \geq \lceil \log_2 \|b_1 \cdots b_m\| \rceil + 1 \quad (2)$$

$$\lceil \log_2 \|c\, b_1 \cdots b_m\| \rceil \geq \lceil \log_2 \|a_1 \cdots a_k\| \rceil + 1 \quad (3)$$

(here, we define $\log_2(0) = -\infty$). Inequality (2) is clear from the definition of $c\,b_1 \cdots b_m$, as $b_1 \cdots b_m$ satisfies $\lceil \log_2 \|b_1 \cdots b_m\| \rceil < \lceil \log_2 \|s\| \rceil = \lceil \log_2 \|c\,b_1 \cdots b_m\| \rceil$. If (3) does not hold then both $a_1 \cdots a_k$ and $c\,b_1 \cdots b_m$ have weights strictly more than $2^{\lceil \log_2 \|s\| \rceil - 1}$ and so their concatenation s has weight strictly more than $2^{\lceil \log_2 \|s\| \rceil} \geq \|s\|$, which is a contradiction.

Recall that the symbols $a_1, \ldots, a_k, c, b_1, \ldots, b_m$ are pairwise different by the convention from the first paragraph of the proof.

For the suffix $b_1 \cdots b_m$ we make a recursive call (if $m = 0$ we do nothing at this step) and include the produced SSLP in the output SSLP $\mathcal{G}$. Let $V_1, V_2, \ldots, V_m$ be the variables such that

$$[\![V_i]\!]_{\mathcal{G}} = b_i \cdots b_m.$$

By the inductive assumption, every path $V_i \xrightarrow{*} b_j$ in the derivation tree has length at most

$$1 + 2\lceil \log_2 \|V_i\| \rceil - 2 \log_2 \|b_j\|.$$

Add a variable V_0 with right-hand side cV_1 (or c if $m = 0$), which derives the suffix $c\,b_1 \cdots b_m$. The path from V_0 to c in the derivation tree has length 1, which is fine, and the path $V_0 \xrightarrow{*} b_j$ is one larger than the path $V_1 \xrightarrow{*} b_j$ and hence has length at most

$$\begin{aligned} 1 + 1 + 2\lceil \log_2 \|V_1\| \rceil - 2 \log_2 \|b_j\| \\ \leq\ 2\lceil \log_2 \|V_0\| \rceil - 2 \log_2 \|b_j\|, \end{aligned}$$

as $1 + \lceil \log_2 \|V_1\| \rceil \leq \lceil \log_2 \|V_0\| \rceil$ by (2).

Next we decompose the prefix $a_1 \cdots a_k$ into $\lfloor k/2 \rfloor$ many blocks of length two and, when k is odd, one block of length 1. We add to the output SSLP $\mathcal{G}$ new variables $X_1, \ldots, X_{\lfloor k/2 \rfloor}$ and define their right-hand sides by

$$\rho(X_i) = a_{2i-1} a_{2i}.$$

The number of variables in $\mathcal{G}$ is $\lfloor \frac{k}{2} \rfloor$. For ease of presentation, when k is odd, define $X_{\lceil k/2 \rceil} = a_k$; this is not a new variable, rather just a notational convention to streamline the presentation. Note that for k even $\lceil k/2 \rceil = \lfloor k/2 \rfloor$ and in this case $X_{\lceil k/2 \rceil}$ is already defined. Viewing $X_1 \cdots X_{\lceil k/2 \rceil}$ as a weighted string of length $\lceil k/2 \rceil$ over the alphabet $\{X_1, \ldots, X_{\lceil k/2 \rceil}\}$, we obtain inductively an SSLP $\mathcal{G}_X$ with at most $3\lceil k/2 \rceil$ variables and right-hand sides of length at most 4 (if $k = 0$ we do nothing at this step). Moreover, $\mathcal{G}_X$ contains variables $U_1, U_2, \ldots, U_{\lceil k/2 \rceil}$ with

$$[\![U_i]\!]_{\mathcal{G}_X} = X_i X_{i+1} \cdots X_{\lceil k/2 \rceil}$$

such that any path of the form $U_i \xrightarrow{*} X_j$ in the derivation tree of $\mathcal{G}_X$ has length at most

$$1 + 2\lceil \log_2 \|U_i\| \rceil - 2 \log_2 \|X_j\|.$$

By adding all variables and right-hand side definitions from $\mathcal{G}_X$ to $\mathcal{G}$ (where all symbols X_i are variables, except $X_{\lceil k/2 \rceil}$ when k is odd, in which case $X_{\lceil k/2 \rceil} = a_k$) we obtain

$$[\![U_i]\!]_{\mathcal{G}} = a_{2i-1} a_{2i} \cdots a_k$$

for all $1 \leq i \leq \lceil k/2 \rceil$. Any path $U_i \xrightarrow{*} a_j$ in the derivation tree of $\mathcal{G}$ has length at most

$$2 + 2\lceil \log_2 \|U_i\| \rceil - 2 \log_2 \|a_j\|. \tag{4}$$

Now, every suffix of s that includes some letter of $a_1 \cdots a_k$ (note that we already have variables for all other suffixes) can be defined by a right-hand side of the form $U_i c V_1$ or $a_{2i-2} U_i c V_1$ ($U_i c$ or $a_{2i-2} U_i c$ if $m = 0$). As in the statement of the lemma, denote those variables by $S_1, \ldots, S_k$. Let us next verify the condition on the path lengths for derivations from those variables. All paths $S_i \xrightarrow{*} c$ have length one. Now consider a path $S_i \xrightarrow{*} a_j$. If the path has length one then we are done. Otherwise, the path must be of the form $S_i \to U_l \xrightarrow{*} a_j$. Therefore, by (4) the path length is at most

$$\begin{aligned} &3 + 2\lceil \log_2 \|U_l\| \rceil - 2 \log_2 \|a_j\| \\ &\leq 3 + 2\lceil \log_2 \|U_1\| \rceil - 2 \log_2 \|a_j\| \\ &= 3 + 2\lceil \log_2 \|a_1 \cdots a_k\| \rceil - 2 \log_2 \|a_j\| \\ &\leq 1 + 2\lceil \log_2 \|c\,b_1 \cdots b_m\| \rceil - 2 \log_2 \|a_j\| \\ &= 1 + 2\lceil \log_2 \|S_i\| \rceil - 2 \log_2 \|a_j\|, \end{aligned}$$

where the second inequality follows from (3) and the equality at the end follows from (1).

Paths of the form $S_i \xrightarrow{*} b_j$ can be treated similarly: they are of the form $S_i \to V_1 \xrightarrow{*} b_j$, where the path $V_1 \xrightarrow{*} b_j$ is of length at most $1 + 2\lceil \log_2 \|V_1\| \rceil - 2 \log_2 \|b_j\|$ by the inductive assumption. Thus, the whole path is of length at most

$$\begin{aligned} &2 + 2\lceil \log_2 \|V_1\| \rceil - 2 \log_2 \|b_j\| \\ &= 2 + 2\lceil \log_2 \|b_1 \cdots b_m\| \rceil - 2 \log_2 \|b_j\| \\ &\leq 2\lceil \log_2 \|c\,b_1 \cdots b_m\| \rceil - 2 \log_2 \|b_j\| \\ &= 2\lceil \log_2 \|S_i\| \rceil - 2 \log_2 \|b_j\|, \end{aligned}$$

which follows from (2) and (1).

Let us now bound the number of variables of the SSLP $\mathcal{G}$. There are

- $\lfloor k/2 \rfloor$ variables X_i,
- at most $3(\lceil k/2 \rceil)$ variables from the recursive call for $X_1 \cdots X_{\lceil k/2 \rceil}$,
- at most $3m = 3(n-k-1)$ variables from the recursive call for $b_1 \cdots b_m$, and
- $1 + k$ new suffix variables for suffixes beginning at $a_1 \cdots a_k c$ (note that those beginning at $b_1 \cdots b_m$ are taken care of by the recursive call).

Therefore $\mathcal{G}$ contains at most

$$\begin{aligned} &\lfloor k/2 \rfloor + 3\lceil k/2 \rceil + 3(n-k-1) + 1 + k \\ &= 3n + 2\lceil k/2 \rceil - k - 2 < 3n \end{aligned}$$

variables. Also note that all right-hand sides of $\mathcal{G}$ have length at most four.

It remains to show that the construction works in linear time. To this end we need a small trick: we assume that when the algorithm is called on s, we supply the algorithm with the value $\|s\|$. More formally, the main algorithm applied to a string s computes $\|s\|$ in linear time by going through s and adding weights. Then it calls a subprocedure $\mathrm{main}'(s, \|s\|)$, which performs the actions described above. To find the appropriate symbol c, main' computes the weights of consecutive prefixes $s_1 s_2 \cdots s_i$, until it finds the first such that $\lceil \log_2 \|s\| \rceil > \lceil \log_2(\|s\| - \|s_1 \cdots s_i\|) \rceil$. Then $k = i - 1$ and so $a_1 \cdots a_k = s_1 \cdots s_{i-1}$, $c = s_i$, $b_1 \cdots b_m = s_{i+1} \cdots s_n$. Moreover, we can compute $\|a_1 \cdots a_k\|$ and $\|b_1 \cdots b_m\|$ for the recursive calls of main' in constant time.

Let $T(n)$ be the running time of main' on a string of length n. Then all operations of main', except the recursive calls, take at most $\alpha(k+1)$ time for some constant $\alpha \geq 1$, where s is represented as $a_1 \cdots a_k \, c \, b_1 \cdots b_m$. Thus $T(n)$ satisfies $T(1) = 1$ and

$$T(n) = T(\lceil k/2 \rceil) + T(n - k - 1) + \alpha(k+1).$$

We claim that $T(n) \leq 2\alpha n$. This is true for $n = 1$ and inductively for $n \geq 2$ we get

$$\begin{aligned} T(n) &\leq 2\alpha(\lceil k/2 \rceil) + 2\alpha(n - k - 1) + \alpha(k+1) \\ &\leq 2\alpha \frac{k+1}{2} + 2\alpha n - \alpha(k+1) \\ &= 2\alpha n. \end{aligned}$$

This concludes the proof of the lemma. ∎

IV. Balancing of string straight-line programs

We now prove Theorem I.1. Let $\mathcal{G} = (\mathcal{V}, \rho_\mathcal{G}, S)$ be an SSLP. W.l.o.g. we can assume that $\mathcal{G}$ is in Chomsky normal form (the case that $[\![G]\!] = \varepsilon$ is trivial). Note that the graph $(\mathcal{V}, E(\mathcal{G}))$ is a directed acyclic graph (DAG). We can assume that every variable is reachable from the start variable S. Consider a variable X with $\rho_\mathcal{G}(X) = YZ$. Then X has the two outgoing edges (X, Y) and (X, Z) in $(\mathcal{V}, E(\mathcal{G}))$. We replace these two edges by the triples $(X, 1, Y)$ and $(X, 2, Z)$. Hence, $\mathcal{D} := (\mathcal{V}, E(\mathcal{G}))$ becomes a DAG with multi-edges (triples from $\mathcal{V} \times \{1, 2\} \times \mathcal{V}$) as considered in Section II. Figure 1(right) shows the DAG $\mathcal{D}$ for an example SSLP; it is the same DAG as in Example II.1. The right-hand sides for the two sink variables X_{13} and X_{14} are terminal symbols. The start variable S is X_0.

We define for every $X \in \mathcal{V}$ the weight $\|X\|$ as the length of the string $[\![X]\!]_\mathcal{G}$. Moreover, for a string $w = X_1 X_2 \cdots X_n$ we define the weight $\|w\| = \sum_{i=1}^n \|X_i\|$. Note that $\|S\| = n$ is the length of the derived string $[\![\mathcal{G}]\!]$ and that this also the value $n(\mathcal{D})$ defined in Section II.

We compute in linear time the edges from the symmetric centroid decomposition of the DAG $\mathcal{D}$, see Lemma II.2. In Figure 1 these are the red edges. The second components of the node labels in Figure 1(left) are the weights of the corresponding variables from the right part of the figure. Hence, we have $\|X_0\| = 62$, $\|X_1\| = 61$, $\|X_2\| = 60$, $\|X_3\| = 58$, etc.

Every variable X of $\mathcal{G}$ will be also a variable of $\mathcal{H}$ and we will have $[\![X]\!]_\mathcal{G} = [\![X]\!]_\mathcal{H}$. In addition, $\mathcal{H}$ will contain auxiliary variables.

Consider a symmetric centroid path

$$(X_0, d_0, X_1), (X_1, d_1, X_2), \ldots, (X_{p-1}, d_{p-1}, X_p) \qquad (5)$$

in $\mathcal{D}$, where all X_i belong to $\mathcal{V}$ and $d_i \in \{1, 2\}$. Thus, for all $0 \leq i \leq p-1$, the right-hand side of X_i in $\mathcal{G}$ has the form $\rho_\mathcal{G}(X_i) = X_{i+1} X'_{i+1}$ (if $d_i = 1$) or $\rho_\mathcal{G}(X_i) = X'_{i+1} X_{i+1}$ (if $d_i = 2$) for some $X'_{i+1} \in \mathcal{V}$. Note that we can have $X'_i = X'_j$ for $i \neq j$. The right-hand side $\rho_\mathcal{G}(X_p)$ can be a terminal symbol or the concatenation of two variables. Note that the variables X'_i $(1 \leq i \leq p)$ and the variables in $\rho_\mathcal{G}(X_p)$ (if they exist) belong to other symmetric centroid paths. We will introduce $\mathcal{O}(p)$ many variables in the SSLP $\mathcal{H}$ to be constructed. Moreover, all right-hand sides of $\mathcal{H}$ have length at most four. By summing over all symmetric centroid paths, this yields the size bound $\mathcal{O}(|\mathcal{G}|)$ for $\mathcal{H}$.

We now define the right-hand sides of the variables $X_0, \ldots, X_p$ in $\mathcal{H}$. We write $\rho_\mathcal{H}$ for the right-hand side mapping of $\mathcal{H}$. For X_p we set $\rho_\mathcal{H}(X_p) = \rho_\mathcal{G}(X_p)$. For the variables $X_0, \ldots, X_{p-1}$ we have to "accelerate" the derivation somehow in order to get the depth bound $\mathcal{O}(\log n)$ at the end. For this, we apply Proposition III.1. Let $L_1 \cdots L_s$ be the subsequence obtained from $X'_1 X'_2 \cdots X'_p$ by keeping only those X'_i with $d_i = 2$ and let $R_1 \cdots R_t$ be the subsequence obtained from the reversed sequence $X'_p X'_{p-1} \cdots X'_1$ by keeping only those X'_i with $d_i = 1$. Take for instance the red symmetric centroid path consisting of the nodes $X_0, X_2, \ldots, X_8$ (hence, $p = 8$) from our running example in Figure 1. We have $L_1 \cdots L_s = X_{13} X_{12} X_{11} X_{10}$ (the target nodes of the blue edges) and $R_1 \cdots R_t = X_{10} X_{11} X_{12} X_{14}$ (the target nodes of the green edges).

Note that every string $[\![X_i]\!]$ $(0 \leq i \leq p-1)$ can be derived in $\mathcal{G}$ from a string $w_\ell X_p w_r$, where w_ℓ is a suffix of $[\![L_1 \cdots L_s]\!]$ and w_r is a prefix of $[\![R_1 \cdots R_t]\!]$. For instance, $[\![X_2]\!]$ can be derived from $(X_{12} X_{11} X_{10}) X_8 (X_{10} X_{11} X_{12})$ in our running example. We now apply Lemma III.1 to the sequence $L_1 \cdots L_s$ in order to get an SSLP $\mathcal{G}_\ell$ of size $\mathcal{O}(s) \leq \mathcal{O}(p)$ that contains variables $S_1 \ldots, S_s$ for the non-empty suffixes of $L_1 \cdots L_s$. Moreover, every path from a variable S_i to some L_j in the derivation tree has length at most $3 + 2\log_2 \|S_i\| - 2\log_2 \|L_j\|$, where $\|S_i\|$ is the weight of $[\![S_i]\!]_{\mathcal{G}_\ell}$. Analogously, we obtain an SSLP $\mathcal{G}_r$ of size $\mathcal{O}(t) \leq \mathcal{O}(p)$ that contains variables $P_1 \ldots, P_t$ for the non-empty prefixes of $R_1 \cdots R_t$. Moreover, every path from a variable P_i to some R_j in the derivation tree has length at most $3 + 2\log_2 \|P_i\| - 2\log_2 \|R_j\|$. We can then define every right-hand side $\rho_\mathcal{H}(X_i)$ $(0 \leq i \leq p-1)$ as $S_j X_p P_k$,

X_pP_k, or S_jX_p for suitable j and k. Moreover, we add all variables and right-hand side definitions of $\mathcal{G}_\ell$ and $\mathcal{G}_r$ to $\mathcal{H}$.

We make the above construction for all symmetric centroid paths of the DAG $\mathcal{D}$. This concludes the construction of $\mathcal{H}$. In our running example we set $\rho_{\mathcal{H}}(X_i) = \rho_{\mathcal{G}}(X_i)$ for $8 \le i \le 14$. Since we introduce $\mathcal{O}(p)$ many variables for every symmetric centroid path of length p and all right-hand sides of $\mathcal{H}$ have length at most four, we obtain the size bound $\mathcal{O}(|\mathcal{G}|)$ for $\mathcal{H}$.

It remains to show that the depth of $\mathcal{H}$ is $\mathcal{O}(\log n)$. Let us first consider the symmetric centroid path (5) and a path in the derivation tree of $\mathcal{H}$ from a variable X_i ($0 \le i \le p$) to a variable Y, where Y is

(a) a variable in $\rho_{\mathcal{G}}(X_p) = \rho_{\mathcal{H}}(X_p)$ or
(b) a variable X'_j for some $i < j \le p$.

In case (a), the path $X_i \xrightarrow{*} Y$ has length at most two. In case (b) the path $X_i \xrightarrow{*} Y$ is of the form $X_i \to S_k \xrightarrow{*} X'_j = Y$ or $X_i \to P_k \xrightarrow{*} X'_j = Y$. Here, $S_k \xrightarrow{*} X'_j$ (resp., $P_k \xrightarrow{*} X'_j$) is a path in $\mathcal{G}_\ell$ (resp., $\mathcal{G}_r$) and therefore has length at most $3 + 2\log_2 \|S_k\| - 2\log_2 \|Y\|$ (resp., $3 + 2\log_2 \|P_k\| - 2\log_2 \|Y\|$). In both cases, we can bound the length of the path $X_i \xrightarrow{*} Y$ by $4 + 2\log_2 \|X_i\| - 2\log_2 \|Y\|$.

Consider now a maximal path in the derivation tree of $\mathcal{H}$ that starts in the root S and ends in a leaf. We can factorize this path as

$$S = X_0 \xrightarrow{*} X_1 \xrightarrow{*} X_2 \xrightarrow{*} \cdots \xrightarrow{*} X_k \tag{6}$$

where all variables X_i belong to the original SSLP and every subpath $X_i \xrightarrow{*} X_{i+1}$ is of the form $X_i \xrightarrow{*} Y$ considered in the previous paragraph. The right-hand side of X_k is a single symbol from Σ. In the DAG $\mathcal{D}$ we have a corresponding path $X_i \xrightarrow{*} X_{i+1}$, which is contained in a single symmetric centroid path except for the last edge leading to X_{i+1}. By the above consideration, the length of the path (6) is bounded by

$$\sum_{i=0}^{k-1} (4 + 2\log_2 \|X_i\| - 2\log_2 \|X_{i+1}\|)$$
$$\le 4k + 2\log_2 \|S\| = 4k + 2\log_2 n.$$

By the second claim of Lemma II.2 we have $k \le 2\log_2 n$ which shows that the length of the path (6) is bounded by $6\log_2 n$.

V. Applications

There are several algorithmic applications of Theorem I.1 with a common underlying idea: let $\mathcal{G}$ be an SSLP of size m for a string s of length n. In many algorithms for SSLP-compressed strings the running time or space consumption depends on depth($\mathcal{G}$), which can be m in the worst case. Theorem I.1 shows that we can replace depth($\mathcal{G}$) by $\mathcal{O}(\log n)$. This is the best we can hope for since depth($\mathcal{G}$) $\ge \Omega(\log n)$ for every SSLP $\mathcal{G}$. Moreover, SSLPs that are produced by practical grammar-based compressors (e.g., LZ78 or RePair) are in general unbalanced in the sense that depth($\mathcal{G}$) $\ge \omega(\log n)$.

The time bounds in the following results refer to the RAM model, where arithmetic operations on numbers from the interval $[0, n]$ need time $\mathcal{O}(1)$. The size of a data structure is measured in the number of words of bit length $\log_2 n$.

A random access query for a string s takes a position $1 \le i \le |s|$ and returns the letter at position i in s. The following result was shown in [7] using several quite sophisticated data structures. It becomes a straightforward corollary of Theorem I.1 using the fact that random access queries for $[\![\mathcal{G}]\!]$ can be answered in time $\mathcal{O}(\text{depth}(\mathcal{G}))$.

Corollary V.1 (random access to grammar-compressed strings, cf. [7])**.** *From a given SSLP $\mathcal{G}$ of size m such that the string $s = [\![\mathcal{G}]\!]$ has length n, one can construct in time $\mathcal{O}(m)$ a data structure of size $\mathcal{O}(m)$ that allows to answer random access queries in time $\mathcal{O}(\log n)$.*

Proof: Using Theorem I.1 we compute in time $\mathcal{O}(m)$ an equivalent SSLP $\mathcal{H}$ for s of size $\mathcal{O}(m)$ and depth $\mathcal{O}(\log n)$. By a single pass over $\mathcal{H}$ we compute for every variable X of $\mathcal{H}$ the length of the string $[\![X]\!]$. Using these lengths one can descend in the derivation tree $[\![\mathcal{H}]\!]$ from the root to the i-th leaf node (which is labelled with the i-th symbol of s) in time $\mathcal{O}(\text{depth}(\mathcal{H})) \le \mathcal{O}(\log n)$. ∎

Remark V.2. It is easy to see that the balancing algorithm from Theorem I.1 can be implemented on a pointer machine, see [32] for a discussion of the pointer machine model. Thus, also the random access data structure from Corollary V.1 can be implemented on a pointer machine. In contrast, the random access data structure from [7] needs the RAM model (for the pointer machine model only preprocessing time and size $\mathcal{O}(m \cdot \alpha_k(m))$ for any fixed k, where α_k is the k-th inverse Ackermann function, is shown in [7]). On the other hand, recently, in [33], the $\mathcal{O}(m)$-space data structure from [7] has been modified so that it can be implemented on a pointer machine as well.

Using fusion trees [34] one can improve the time bound in Corollary V.1 to $\mathcal{O}(\log n / \log\log n)$ at the cost of an additional factor of $\mathcal{O}(\log^\epsilon n)$ in the size bound. The following result has been shown in [8, Theorem 2] under the assumption that the input SSLP has depth $\mathcal{O}(\log n)$. We can enforce this bound with Theorem I.1.

Corollary V.3. *Fix an arbitrary constant $\epsilon > 0$. From a given SSLP $\mathcal{G}$ of size m such that the string $s = [\![\mathcal{G}]\!]$ has length n, one can construct in time $\mathcal{O}(m \cdot \log^\epsilon n)$ a data structure of size $\mathcal{O}(m \cdot \log^\epsilon n)$ that allows to answer random access queries in time $\mathcal{O}(\log n / \log\log n)$.*

Proof: The proof is exactly the same as for [8, Theorem 2]. There, the author have to assume that the input SSLP has depth $\mathcal{O}(\log n)$, which we can enforce by Theorem I.1.

Roughly speaking, the idea in [8] is to reduce the depth of the SSLP to $\mathcal{O}(\log n/\log\log n)$ by expanding right-hand sides to length $\mathcal{O}(\log^\epsilon n)$. Then for each right-hand side a fusion tree is constructed, which allows to spend constant time at each variable during the navigation to the i-th symbol.

Let us also remark that the size bound for the computed data structure in [8] is given in bits, which yields $\mathcal{O}(m \cdot \log^{1+\epsilon} n)$ bits since numbers from $[0, n]$ have to be encoded with $\log_2 n$ bits. ∎

Given a string $s \in \Sigma^*$, a rank query gets a position $1 \leq i \leq |s|$ and a symbol $a \in \Sigma$ and returns the number of a's in the prefix of s of length i. A select query gets a symbol $a \in \Sigma$ and a number $1 \leq i \leq |s|$ and returns the position of the i-th a in s (if it exists).

Corollary V.4. *Fix an arbitrary constant $\epsilon > 0$. From a given SSLP $\mathcal{G}$ of size m such that the string $s = [\![\mathcal{G}]\!]$ has length n, one can construct in time $\mathcal{O}(m \cdot |\Sigma| \cdot \log^\epsilon n)$ a data structure of size $\mathcal{O}(m \cdot |\Sigma| \cdot \log^\epsilon n)$ that allows to answer rank and select queries in time $\mathcal{O}(\log n/\log\log n)$.*

Proof: Again we follow the proof [8, Theorem 2] but first apply Theorem I.1 in order to reduce the depth of the SSLP to $\mathcal{O}(\log n)$. ∎

Corollary V.4 improves [8, Theorem 2], where the query time is $\mathcal{O}(\log n)$ and the space is $\mathcal{O}(m \cdot |\Sigma| \cdot \log n)$.

Our balancing result also yields an improvement for the *compressed subsequence problem* [9]. Bille et al. [9] present an algorithm based on a *labelled successor* data structure. Given a string $s = a_1 \cdots a_n \in \Sigma^*$, a labelled successor query gets a position $1 \leq i \leq n$ and a symbol $a \in \Sigma$ and returns the minimal position $j > i$ with $a_i = a$ (or rejects if it does not exist). The following result is an improvement over [9], where the authors present two algorithms for the compressed subsequence problem: one with $\mathcal{O}(m + m \cdot |\Sigma|/w)$ preprocessing time and $\mathcal{O}(\log n \cdot \log w)$ query time, and another algorithm with $\mathcal{O}(m + m \cdot |\Sigma| \cdot \log w/w)$ preprocessing time and $\mathcal{O}(\log n)$ query time, where n, m, and w are as below.

Corollary V.5. *There is a data structure supporting labelled successor (and predecessor) queries on a string $s \in \Sigma^*$ of length n compressed by an SSLP of size m in the word RAM model with word size $w \geq \log_2 n$ using $\mathcal{O}(m + m \cdot |\Sigma|/w)$ space, $\mathcal{O}(m + m \cdot |\Sigma|/w)$ preprocessing time, and $\mathcal{O}(\log n)$ query time.*

Proof: In the preprocessing phase we first reduce the depth of the given SSLP to $\mathcal{O}(\log n)$ using Theorem I.1. We compute for every variable X the length of $[\![X]\!]$ in time and space $\mathcal{O}(m)$ as in the proof of Corollary V.1. Additionally for every variable X we compute a bitvector of length $|\Sigma|$ which encodes the set of symbols $a \in \Sigma$ that occur in $[\![X]\!]$. Notice that this information takes $\mathcal{O}(m \cdot |\Sigma|)$ bits and fits into $\mathcal{O}(m \cdot |\Sigma|/w)$ memory words. If $\rho(X) = YZ$ then the bitvector of X can be computed from the bitvectors of Y and Z by $\mathcal{O}(|\Sigma|/w)$ many bitwise OR operations. Hence in total all bitvectors can be computed in time $\mathcal{O}(m \cdot |\Sigma|/w)$.

A labelled successor query (for position i and symbol a) can now be answered in $\mathcal{O}(\log n)$ time in a straightforward way: First we compute the path $(X_0, X_1, \ldots, X_\ell)$ in the derivation tree from the root X_0 to the symbol at the i-th position. Then we follow the path starting from the leaf upwards to find the maximal k such that $\rho(X_k) = X_{k+1}Y$ and $[\![Y]\!]$ contains the symbol a, or reject if no such k exists. Finally, starting from Y we navigate in time $\mathcal{O}(\log n)$ to the leftmost leaf in the derivation tree rooted at Y which produces the symbol a. Thereby, the position of that leaf in the whole derivation tree can be computed using $\mathcal{O}(\log n)$ many additions. ∎

A *minimal subsequence occurrence* of a string $p = a_1 a_2 \cdots a_k$ in a string $s = b_1 b_2 \cdots b_l$ is given by two positions i, j with $1 \leq i \leq j \leq l$ such that p is a subsequence of $b_i b_{i+1} \cdots b_j$ (i.e., $b_i b_{i+1} \cdots b_j$ belongs to the language $\Sigma^* a_1 \Sigma^* a_2 \cdots \Sigma^* a_k \Sigma^*$) but p is neither a subsequence of $b_{i+1} \cdots b_j$ nor of $b_i \cdots b_{j-1}$. Following the proof of [9, Theorem 1] we obtain:

Corollary V.6. *Given an SSLP $\mathcal{G}$ of size m producing a string $s \in \Sigma^*$ of length n and a pattern $p \in \Sigma^*$ one can compute all minimal subsequence occurrences of p in s in space $\mathcal{O}(m + m \cdot |\Sigma|/w)$ and time $\mathcal{O}(m + m \cdot |\Sigma|/w + |p| \cdot \log n \cdot occ)$ where $w \geq \log n$ is the word size and occ is the number of minimal subsequence occurrences of p in s.*

Corollary V.6 improves [9, Theorem 1], which states the existence of two algorithms for the computation of all minimal subsequence occurrences with the following running times (the space bounds are the same as in Corollary V.6):

- $\mathcal{O}(m + m \cdot |\Sigma|/w + |p| \cdot \log n \cdot \log w \cdot \text{occ})$,
- $\mathcal{O}(m + m \cdot |\Sigma| \cdot \log w/w + |p| \cdot \log n \cdot \text{occ})$.

Let us list further applications of Theorem I.1 (recall that $\mathcal{G}$ is an SSLP of size m for a string s of length n):

Computing fingerprints of SSLP-compressed strings: Given two positions $i \leq j$ in s one wants to compute the Karp-Rabin fingerprint of the factor of s that starts at position i and ends at position j. In [10] it was shown that one can compute from $\mathcal{G}$ a data structure of size $\mathcal{O}(m)$ that allows to compute fingerprints in time $\mathcal{O}(\log n)$. First, the authors of [10] present a very simple data structure of size $\mathcal{O}(m)$ that allows to compute fingerprints in time $\mathcal{O}(\text{depth}(\mathcal{G}))$. With Theorem I.1, we can use this data structure to obtain an $\mathcal{O}(\log n)$-time solution. This simplifies the proof in [10] considerably.

Computing runs, squares, and palindromes in SSLP-compressed strings: It is shown in [11] that certain compact representations of the set of all runs, squares and palindromes in s (see [11] for precise definitions) can be computed in time $\mathcal{O}(m^3 \cdot \text{depth}(\mathcal{G}))$. We can improve the time bound to $\mathcal{O}(m^3 \cdot \log n)$.

Real time traversal for SSLP-compressed strings: One wants to output the symbols of s from left to right and thereby spend constant time per symbol. A solution can be found in [12]; a two-way version (where one can navigate in each step to the left or right neighboring position in s) can be found in [13]. The drawback of these solutions is that they need space $\mathcal{O}(\text{depth}(\mathcal{G}))$. With Theorem I.1 we can reduce this to space $\mathcal{O}(\log n)$.

Compressed range minimum queries: Range minimum data structure preprocesses a given string s of integers so that the following queries can be efficiently answered: given $i \leq j$, what is the minimum element in $s_i, \ldots, s_j$ (the substring of s from position i to j). We are interested in the variant of the problem, in which the input is given as an SSLP $\mathcal{G}$. It is known, that after a preprocessing taking $\mathcal{O}(|\mathcal{G}|)$ time, one can answer range minimum queries in time $\mathcal{O}(\log n)$ [14, Theorem 1.1]. This implementation extends the data structure for random access for SSLPs [7] with some additional information, which includes in particular adding standard range minimum data structures for subtrees leaving the heavy path and extending the original analysis. Using the balanced SSLP the same running time can be easily obtained, without the need of hacking into the construction of the balanced SSLP. To this end for each variable X we store the length ℓ_X of the derived string $[\![X]\!]$ as well the minimum value in $[\![X]\!]$. In the following, let $\mathrm{RMQ}(X, i, j)$ be the range minimum query called on $[\![X]\!]$ for the interval $[i, j]$. Given $\mathrm{RMQ}(X, i, j)$, with the right-hand side for X being $X \to YZ$ we proceed as follows:

- If the query asks about the minimum in the whole $[\![X]\!]$, i.e., $i = 1$ and $j = \ell_X$, then we return the minimum of $[\![X]\!]$; we call this case trivial in the following.
- If the whole range is within $[\![Y]\!]$, i.e., $j \leq \ell_Y$, then we call $\mathrm{RMQ}(Y, i, j)$.
- If the whole range is within $[\![Z]\!]$, i.e., $i > \ell_Y$, then we call $\mathrm{RMQ}(Z, i - \ell_Y, j - \ell_Y)$.
- Otherwise, i.e., when $i \leq \ell_Y$ and $j > \ell_Y$ and $(i, j) \neq (1, \ell_X)$, the range spans over the substrings generated by both nonterminals. Then we compute the queries for two substrings and take their minimum, i.e., we return the minimum of $\mathrm{RMQ}(Y, i, \ell_Y)$ and $\mathrm{RMQ}(Z, 1, j - \ell_Y)$.

To see that the running time is $\mathcal{O}(\text{depth}(\mathcal{G})) = \mathcal{O}(\log n)$ observe first that the cost of trivial cases can be charged to the function that called them. Thus it is enough to estimate the number of nontrivial recursive calls. In the second and third case there is only one recursive call for a variable that is deeper in the derivation tree of the SSLP. In the fourth case there are two calls, but two nontrivial calls are made at most once during the whole computation: if two nontrivial calls are made in the fourth case then one of them asks for the RMQ of a suffix of $[\![Y]\!]$ and the other call asks for the RMQ of a prefix of $[\![Z]\!]$. Moreover, every recursive call on a prefix of some string $[\![X']\!]$ leads to at most one nontrivial call, which is again on a prefix of some string $[\![X'']\!]$; and analogously for suffixes.

Lifshits' algorithm for compressed pattern matching [35]: The input consists of an SSLP $\mathcal{P}$ for a pattern p and an SSLP $\mathcal{T}$ for a text t and the question is whether p occurs in t. Lifshits' algorithm has a running time of $\mathcal{O}(|\mathcal{P}| \cdot |\mathcal{T}|^2)$. It was conjectured by Lifshits that the running time could be improved to $\mathcal{O}(|\mathcal{P}| \cdot |\mathcal{T}| \cdot \log |t|)$. This follows easily from Theorem I.1: the algorithm fills a table of size $|\mathcal{P}| \cdot |\mathcal{T}|$ and on each entry it calls a recursive subprocedure, whose running time is at most depth$(\mathcal{T})$. By Theorem I.1 we can bound the running time by $\mathcal{O}(\log |t|)$, which proves Lifshits' conjecture. Note, that in the meantime a faster algorithm for compressed pattern matching with running time $\mathcal{O}(|\mathcal{T}| \cdot \log |p|)$ was found [36].

Remark V.7 (smallest grammar problem). In the "smallest grammar problem" (for strings) for a given string w we want to construct a smallest SSLP defining w. The decision variant of this problem is NP-hard, the best known approximation lower bound is $\frac{8569}{8568}$ [3], and the best known approximation algorithms have an approximation ratio of $\mathcal{O}(\log n)$, where n is the length of the input string [2]–[5]. Except for [4], all these algorithms produce SSLPs of depth $\mathcal{O}(\log n)$. It was discussed in [4] that the reason for the lack of constant-factor approximation algorithms might be the fact that smallest SSLPs can have larger than logarithmic depth. Theorem I.1 refutes this approach.

VI. Balancing circuits over algebras

This section gives additional details on our general balancing Theorem I.3 for certain multi-sorted algebras. We first introduce the general framework.

Let us fix a finite set $\mathcal{S}$ of so-called *sorts*. In a moment, we will assign to each sort $i \in \mathcal{S}$ a set A_i (of elements of sort i). An *$\mathcal{S}$-sorted signature* is a set of symbols Γ and a mapping $\text{type}\colon \Gamma \to \mathcal{S}^+$ that assigns to each symbol from Γ a non-empty string over the alphabet $\mathcal{S}$. The number $|\text{type}(f)| - 1 \geq 0$ is also called the *rank* of f. If $\text{type}(f) = p \in \mathcal{S}$ then f is called a constant of sort p. In order to exclude pathological cases, we assume that Γ contains for every sort $p \in \mathcal{S}$ at least one constant of sort p.

A *Γ-algebra* is a tuple $\mathcal{A} = ((A_p)_{p \in \mathcal{S}}, (f^{\mathcal{A}})_{f \in \Gamma})$ where every A_p is a non-empty set (the universe of sort p or the set of elements of sort p) and for every $f \in \Gamma$ with $\text{type}(f) = p_1 p_2 \cdots p_n q$, $f^{\mathcal{A}}\colon \prod_{1 \leq j \leq n} A_{p_j} \to A_q$ is an n-ary function. We also say that Γ is the *signature* of $\mathcal{A}$.

Example VI.1. A well known example of a multi-sorted algebra is a vector space. More precisely, it can be formalized as a two-sorted algebra where $\mathcal{S} = \{v, s\}$. Here v stands for "vectors" and s stands for "scalars". The $\mathcal{S}$-sorted signature would be $\Gamma = \{\overline{0}, 0, 1, \oplus, \odot, +, \cdot\}$, where

- $\text{type}(\overline{0}) = v$ (the zero vector),

- type$(0) = s$ (the 0-element of the scalar field),
- type$(1) = s$ (the 1-element of the scalar field),
- type$(\oplus) = vvv$ (vector addition),
- type$(\odot) = svv$ (multiplication of a scalar by a vector),
- type$(+) = sss$ (addition in the field of scalars),
- type$(\cdot) = sss$ (multiplication in the field of scalars).

Let us fix $\mathcal{S}$, the $\mathcal{S}$-sorted signature Γ, and the Γ-algebra $\mathcal{A}$ for the further discussion. A *circuit* over $\mathcal{A}$ is basically a DAG $\mathcal{G} = (V, E)$ as in Section II. The nodes of $\mathcal{G}$ are called *gates* in the following, the root node $r \in V$ is called the *output gate*, and gates without outgoing edges are called *input gates*. In addition to Section II we assume that every gate $v \in V$ is labelled with a symbol $f_v \in \Gamma$. In order to evaluate $\mathcal{G}$ in the algebra $\mathcal{A}$, it has to be well-typed (with respect to $\mathcal{A}$) in the following sense: one can assign to every gate $v \in V$ a sort s_v such that for every gate v the following holds: if v is labelled with the symbol f_v and $(v, 1, v_1), \ldots, (v, d, v_d)$ are the outgoing edges of v in the circuit, then we must have type$(f_v) = s_{v_1} \cdots s_{v_d} s_v$. Clearly, if these sorts s_v exist then they are uniquely defined and it is straightforward to check whether a given circuit is well-typed. A well-typed circuit $\mathcal{G}$ over the Γ-sorted algebra $\mathcal{A}$ can be evaluated in the natural way. Thereby we assign to each gate v an element $a_v \in A_{s_v}$. If $v, v_1, \ldots, v_d$ are as above and the values $a_{v_1}, \ldots, a_{v_d}$ have been already computed then we set $a_v = f_v^{\mathcal{A}}(a_{v_1}, \ldots, a_{v_d})$. In particular, we start the evaluation process at the input gates of the circuit. The value computed by the circuit is then a_r. We define the *size of a circuit* as its number of edges and the *depth of a circuit* as the length of a longest path from the output gate r to an input gate. Also recall the definition of the *unfolded size* of a circuit, which was defined in the paragraph before Theorem I.3. In the worst-case, the unfolded circuit size is exponentially larger than the size of a circuit. In the following we implicitly assume that all circuits are well-typed with respect to the underlying algebra.

Example VI.2. Figure 2 shows a circuit over the vector space $\mathcal{A} := \mathbb{R}^2$ which is viewed as a Γ-sorted algebra as explained in Example VI.1. In order to get a circuit that evaluates to a non-trivial vector, we have to include additional constants of sort v (i.e, vectors) in the algebra. These constants are the symbols e_1 (for the unit vector $(1,0)^{\mathrm{T}}$) and e_2 (for the unit vector $(0,1)^{\mathrm{T}}$). Next to each gate, we write in Figure 2 the scalar/vector to which the gate evaluates to.

The only missing notion in order to understand Theorem I.3 is the notion of a subsumption base. Let us explain this notion in more detail. We start with tree-like expressions that are built from the symbols in Γ and so-called *auxiliary variables*. For this let us fix a finite set $\mathcal{Y}$ of auxiliary variables, where $\mathcal{Y} \cap \Gamma = \emptyset$. To every auxiliary variable $y \in \mathcal{Y}$ we assign a sort $s(y)$. We then define for every

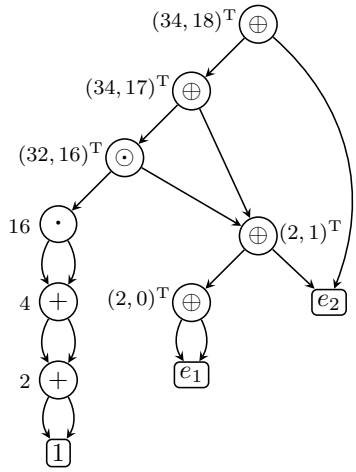

Figure 2. A circuit over the vector space $\mathbb{R}^2$, which evaluates to the vector $(34,18)^{\mathrm{T}}$.

sort $p \in \mathcal{S}$ the set $\mathcal{T}_p(\mathcal{Y})$ (*expression trees of sort p with auxiliary variables from $\mathcal{Y}$*) inductively as follows: (i) $y \in \mathcal{T}_{s(y)}(\mathcal{Y})$ for every $y \in \mathcal{Y}$ and (ii) if $f \in \Gamma$, type$(f) = p_1 \cdots p_n p$ and $t_1 \in \mathcal{T}_{p_1}(\mathcal{Y}), \ldots, t_n \in \mathcal{T}_{p_n}(\mathcal{Y})$ then $f(t_1, \ldots, t_n) \in \mathcal{T}_p(\mathcal{Y})$. Note that every constant of sort p belongs to $\mathcal{T}_p(\mathcal{Y})$ (formally, we have to identify $a()$ with a for a constant a). We also write $\mathcal{T}_p$ for $\mathcal{T}_p(\emptyset)$. Note that expressions from $\mathcal{T}_p$ can be identified with tree-like circuits, where every gate except of the output gate has a unique incoming edge. In particular, an expression $t \in \mathcal{T}_p$ evaluates to an element from A_p in the obvious way.

Next, we consider for every sort $p \in \mathcal{S}$ a special symbol $x_p \notin \Gamma \cup \mathcal{Y}$ which we call the *main variable of sort p*. We then consider for all sorts $p, q \in \mathcal{S}$ the set $\mathcal{C}_{p,q}(\mathcal{Y})$ (*parameterized contexts with input sort p and output sort q*) of all expressions that are obtained from an expression $t \in \mathcal{T}_q(\mathcal{Y})$ by replacing in t a unique occurrence of a constant of sort p by x_p. Let us emphasize that every expression from $\mathcal{C}_{p,q}(\mathcal{Y})$ contains a unique occurrence of the main variable x_p and contains no occurrence of a main variable $x_{p'}$ for $p' \neq p$. We write $\mathcal{C}_{p,q}$ for $\mathcal{C}_{p,q}(\emptyset)$ (*contexts with input sort p and output sort q*).

Consider $t \in \mathcal{C}_{p,q}$. We define a unary function $t^{\mathcal{A}} \colon A_p \to A_q$ as follows: for $a \in A_p$, $t^{\mathcal{A}}(a)$ is the result of evaluating the expression obtained from t by replacing the unique occurrence of x_p by a. We call $t^{\mathcal{A}}$ a *unary linear term function* (ULTF for short) over $\mathcal{A}$. The adjective "linear" is justified by the fact that x_p has a unique occurrence in t.

Example VI.3. Consider again the example with the vector space $\mathbb{R}^2$ and let $t = e_1 \oplus ((1+1) \odot (x_v \oplus e_2)) \in \mathcal{C}_{v,v}$ (recall that v is the sort of vectors). The corresponding ULTF is the affine mapping $x \mapsto 2x \oplus (1,2)^{\mathrm{T}}$ on $\mathbb{R}^2$.

As another example note that a ULTF, where the underlying algebra is a ring $\mathcal{R}$ (this is a one-sorted algebra), is

nothing else than a linear polynomial over $\mathcal{R}$ in a single variable x.

We would like to describe the set of all ULTFs by finitely many parameterized contexts. For this, consider two finite sets $\mathcal{Y}$ and $\mathcal{Z}$ of auxiliary variables (they can have a non-empty intersection). A $(\mathcal{Y},\mathcal{Z})$-*substitution* is a mapping σ that assigns to every auxiliary variable $y \in \mathcal{Y}$ an expression $\sigma(y) \in \mathcal{T}_{s(y)}(\mathcal{Z})$. Such a substitution can be applied to a parameterized context $t \in \mathcal{C}_{p,q}(\mathcal{Y})$ and naturally yields a parameterized context $\sigma(t) \in \mathcal{C}_{p,q}(\mathcal{Z})$ obtained by replacing in t every occurrence of an auxiliary variable $y \in \mathcal{Y}$ by the expression $\sigma(y)$. A $(\mathcal{Y},\emptyset)$-*substitution* is also called a $\mathcal{Y}$-*substitution*. In this case, we have $\sigma(t) \in \mathcal{C}_{p,q}$ and we can consider the ULTF $\sigma(t)^{\mathcal{A}} \colon A_p \to A_q$. In this sense we can view a parameterized context from $\mathcal{C}_{p,q}(\mathcal{Y})$ as a parameterized ULTF. We say that $s,t \in \mathcal{C}_{p,q}(\mathcal{Y})$ are *equivalent* (in $\mathcal{A}$) if they yield the same ULTF under every $\mathcal{Y}$-substitution.

Consider now $s \in \mathcal{C}_{p,q}(\mathcal{Y})$ and $t \in \mathcal{C}_{p,q}(\mathcal{Z})$ ($\mathcal{Y}$ and $\mathcal{Z}$ can be different sets of auxiliary variables). We say that s is *subsumed* by t if there is a $(\mathcal{Z},\mathcal{Y})$-substitution σ such that $\sigma(t)$ and s are equivalent. Intuitively, this means that t is more general than s. A *finite subsumption base for* $\mathcal{A}$ is a finite set C of parameterized contexts (i.e., C is a finite subset of $\bigcup_{p,q\in\mathcal{S}} \mathcal{C}_{p,q}(\mathcal{Z})$ for some sufficiently large set $\mathcal{Z}$ of auxiliary variables) such that every parameterized context $s \in \mathcal{C}_{p,q}(\mathcal{Y})$ (for $p,q,\mathcal{Y}$ arbitrary) is subsumed by some parameterized context from C. If a multi-sorted algebra has a finite subsumption base than the set of all ULFTs can be described by a finite set of parameterized contexts.

Example VI.4. Every (not necessarily commutative) semiring with a 0 and 1 has a finite subsumption base, consisting of the single expression $y_0 + y_1xy_2$ (y_0, y_1, y_2 are the auxiliary variables). If the semiring is commutative then $y_0 + y_1x$ forms a finite subsumption base. If we do not assume that the noncommutative semiring has a 0 and 1 then the eight expressions obtained from $y_0 + y_1xy_2$ by omitting some of the parameters y_0, y_1, y_2 form a finite subsumption base.

Example VI.5. The *free term algebra* $\mathcal{T}$ over Γ is defined by $\mathcal{T} = ((\mathcal{T}_p)_{p\in\mathcal{S}}, (f)_{f\in\Gamma})$, where every expression $t \in \mathcal{T}_p$ evaluates to itself. If Γ contains a symbol of rank at least one, then $\mathcal{T}$ has no finite subsumption base: If C were a finite subsumption base of $\mathcal{T}$, then every context could be obtained from some $t \in C$ by replacing the auxiliary parameters in t by expressions trees. But this replacement does not change the length of the path from the root of the context to its main variable. Hence, we would obtain a bound for the length of the path from the root to the main variable in a context, which clearly does not exist.

We now have introduced all necessary notions in order to understand Theorem I.3 according to which a circuit over a multi-sorted algebra that has a finite subsumption base can be balanced. Thereby the size of the circuits blows up only by a constant factor. Note again, that "balanced" in this context means that the depth of the constructed circuit is bounded by $\mathcal{O}(\log n)$ where n is the size of tree obtained by completely unfolding the circuit. The proof of Theorem I.3 follows to a large extent the arguments from the proof of Theorem I.1. In fact, the main tools from Sections II and III are also used in the proof of Theorem I.3. The two main steps in the proof of Theorem I.3 are the following:

- We start with a circuit $\mathcal{G}$ over a Γ-sorted algebra $\mathcal{A}$. By unfolding $\mathcal{G}$ into a tree we obtain an expression tree t over the sorted signature Γ. Let n be the size of t. In the first step we construct for the tree t a so-called tree straight-line program of size $|\mathcal{G}|$ and depth $\mathcal{O}(\log n)$. The proof follows the arguments for Theorem I.1. Let us emphasize that this first step is purely syntactic in the sense that the underlying algebra $\mathcal{A}$ is not relevant. Tree straight-line programs are an extension of string straight-line programs and produce expression trees over a sorted signature (in the next section we will introduce an extension of tree straight-line programs to so called unranked trees), see [16].
- Only in the second step we need the fact that $\mathcal{A}$ has a finite subsumption base. This allows to transform the tree straight-line program for t that we have constructed in the first step back into a circuit of the same size and depth (up to constant factors) as the tree straight-line program. Moreover, this circuit evaluates to the same element of $\mathcal{A}$ as the expression tree t (or the initial circuit $\mathcal{G}$). The arguments from this second step appeared implicitly also in [29], [30] (these papers do not define the concept of a finite subsumption base in full generality).

Since every free monoid has a singleton subsumption base, consisting of the expression y_1xy_2, we can deduce Theorem I.1 from Theorem I.3 (note that an SSLP is the same thing as a circuit over a free monoid). A minor technical detail is that one first has to convert the SSLP into Chomsky normal form. This ensures that the derivation tree of the SSLP (which corresponds to the tree t in the above proof outline) has (up to a factor of size two) the same size as the string produced by the SSLP.

VII. Balancing forest-straight-line programs

In this section we present a further application of the general balancing theorem (Theorem I.3). This application deals with unranked trees and forest-straight-line programs, see Theorem I.2. We start with the definition of so-called forest algebras.

A. Forest algebra

Let us fix a finite set Σ of node labels. In this section, we consider Σ-labelled rooted ordered trees, where "ordered"

means that the children of a node are totally ordered. Every node has a label from Σ. In contrast to the trees (expressions) from the previous section we make no rank assumption: the number of children of a node (also called its degree) *is not* determined by its node label. A *forest* is a (possibly empty) sequence of such trees. The size $|v|$ of a forest is the total number of nodes in v. The set of all Σ-labelled forests is denoted by $\mathcal{F}_0(\Sigma)$. Formally, $\mathcal{F}_0(\Sigma)$ can be inductively defined as the smallest set of strings over the alphabet $\Sigma \cup \{(,)\}$ such that

- $\varepsilon \in \mathcal{F}_0(\Sigma)$ (the empty forest),
- if $u, v \in \mathcal{F}_0(\Sigma)$ then $uv \in \mathcal{F}_0(\Sigma)$, and
- if $u \in \mathcal{F}_0(\Sigma)$ then $a(u) \in \mathcal{F}_0(\Sigma)$ (this is a forest consisting of a single tree whose root is labelled with a).

Let us fix a distinguished symbol $* \notin \Sigma$. The set of forests $u \in \mathcal{F}_0(\Sigma \cup \{*\})$ such that $*$ has a unique occurrence in u and this occurrence is at a leaf node is denoted by $\mathcal{F}_1(\Sigma)$. Elements of $\mathcal{F}_1(\Sigma)$ are called *forest contexts*. The symbol $*$ should be viewed as a placeholder for an arbitrary forest. Following [37], we define the *forest algebra* $\mathsf{F}(\Sigma)$ as the 2-sorted algebra

$$(\mathcal{F}_0(\Sigma), \mathcal{F}_1(\Sigma), \ominus_{00}, \ominus_{01}, \ominus_{10}, \oplus_0, \oplus_1, (a(*))_{a\in\Sigma}, \varepsilon, *)$$

with the following operations:

- $\ominus_{ij} \colon \mathcal{F}_i(\Sigma) \times \mathcal{F}_j(\Sigma) \to \mathcal{F}_{i+j}(\Sigma)$ $(i, j \in \{0, 1\}, i + j \leq 1)$ is a horizontal concatenation operator: for $u \in \mathcal{F}_i(\Sigma)$, $v \in \mathcal{F}_j(\Sigma)$ we set $u \ominus_{ij} v = uv$ (i.e., we concatenate the corresponding sequences of trees).
- $\oplus_i \colon \mathcal{F}_1(\Sigma) \times \mathcal{F}_i(\Sigma) \to \mathcal{F}_i(\Sigma)$ is a vertical concatenation operator: for $u \in \mathcal{F}_1(\Sigma)$ and $v \in \mathcal{F}_i(\Sigma)$, $u \oplus_i v$ is obtained by replacing in u the unique occurrence of $*$ by v.
- $\varepsilon \in \mathcal{F}_0(\Sigma)$ and $*, a(*) \in \mathcal{F}_1(\Sigma)$ $(a \in \Sigma)$ are constants of the forest algebra.

Note that $(\mathcal{F}_0(\Sigma), \ominus_{00}, \varepsilon)$ and $(\mathcal{F}_1(\Sigma), \oplus_1, *)$ are monoids. In the following we will omit the subscripts i, j in $\ominus_{ij}$ and $\oplus_i$, since they will be always clear from the context. Most of the time, we simply write uv instead of $u \ominus v$, $a(u)$ instead of $a(*) \oplus u$, and a instead of $a(\varepsilon)$. With these abbreviations, a forest $u \in \mathcal{F}(\Sigma)$ can be also viewed as an algebraic expression over the algebra $\mathsf{F}(\Sigma)$, which evaluates to itself.

Lemma VII.1. *Every forest algebra $\mathsf{F}(\Sigma)$ has a finite subsumption base.*

Proof sketch: We denote by x_0 and x_1 the main variables of sorts $\mathcal{F}_0(\Sigma)$ and $\mathcal{F}_1(\Sigma)$, respectively, and by $\sigma_1, \sigma_2, \ldots$ (resp., $\tau_1, \tau_2, \ldots$) auxiliary variables of sorts $\mathcal{F}_0(\Sigma)$ (resp., $\mathcal{F}_1(\Sigma)$). Let C be the finite set consisting of the following parameterized contexts:

- $\tau_1 \oplus x_0$,
- $\tau_1 \oplus x_1 \oplus \sigma_1$,
- $\tau_1 \oplus x_1 \oplus \tau_2$,
- $\tau_1 \oplus (\tau_2 \ominus (\tau_3 \oplus x_0))$,
- $\tau_1 \oplus ((\tau_2 \oplus x_0) \ominus \tau_3)$,
- $\tau_1 \oplus (\tau_2 \ominus (\tau_3 \oplus x_1 \oplus \sigma_1))$,
- $\tau_1 \oplus ((\tau_2 \oplus x_1 \oplus \sigma_1) \ominus \tau_3)$.

One can then show that C is a finite subsumption base for $\mathsf{F}(\Sigma)$; see [1] for details. ■

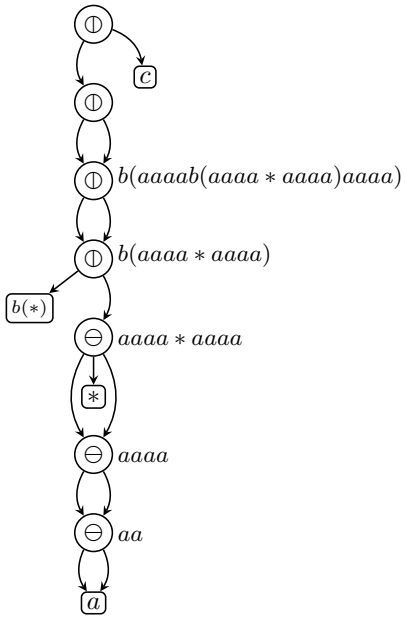

Figure 3. An FSLP.

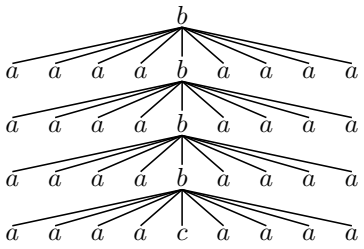

Figure 4. The forest to which the FSLP from Figure 3 evaluates to.

B. Forest straight-line programs

A *forest straight-line program* over Σ, FSLP for short, is a circuit over the forest algebra $\mathsf{F}(\Sigma)$ that evaluates to an element of $\mathcal{F}_0(\Sigma)$. Iterated vertical and horizontal concatenations allow to generate forests, whose depth and width is exponential in the size of the FSLP.

Example VII.2. Consider the circuit shown in Figure 3. It evaluates to the forest shown in Figure 4. Note that we use a $\ominus$-gate with three incoming edges. Since $\ominus$ is associative, this does not lead to problems.

Our main result for FSLPs is:

Theorem VII.3. *Given an FSLP $\mathcal{G}$ producing the forest u one can compute in time $\mathcal{O}(|\mathcal{G}|)$ an FSLP $\mathcal{H}$ for u of size $\mathcal{O}(|\mathcal{G}|)$ and depth $\mathcal{O}(\log |u|)$.*

Proof sketch: The case $u = \varepsilon$ is trivial. Let us now assume that $u \neq \varepsilon$. In a first step, one has to eliminate gates that are labelled with the constants ε and $*$. This is possible in linear time and increases the size of $\mathcal{G}$ only by a constant factor. Basically, its the same as bringing an SSLP into Chomsky normal form. After this preprocessing, $\mathcal{G}$ has the property that its unfolded size is linearly bounded in $|u|$. By Lemma VII.1 we can apply Theorem I.3 in order to get the FSLP $\mathcal{H}$ with the desired properties. Strictly speaking this proof only works for the case that the alphabet Σ of node labels is fixed, which ensures that we are working with a fixed forest algebra (note that Theorem I.3 deals with a fixed algebra $\mathcal{A}$). But with a little effort one can also allow the alphabet Σ to be part of the input, see the full version [1] for details (the crucial detail is that the finite subsumption base from the proof of Lemma VII.1 does not depend on the alphabet Σ). ∎

The analogous statements for tree straight-line programs and top dags from Theorem I.2 are shown in a similar way. Top dags are basically a minor syntactic variant of forest straight-line programs, but embedding top dags into our general algebraic framework is slightly more technical than for forest straight-line programs.

VIII. Open problems

For SSLPs one may require a strong notion of balancing. Let us say that an SSLP $\mathcal{G}$ is c-balanced if (i) the length of every right-hand side is at most c and (ii) if a variable Y occurs in $\rho(X)$ then $|[\![Y]\!]_{\mathcal{G}}| \leq |[\![X]\!]_{\mathcal{G}}|/2$. It is open, whether there is a constant c such that for every SSLP of size m there exists an equivalent c-balanced SSLP of size $\mathcal{O}(m)$.

Another important open problem is whether the query time bound in Corollary V.1 (random access to grammar-compressed strings) can be improved from $\mathcal{O}(\log n)$ to $\mathcal{O}(\log n/\log\log n)$. If we allow space $\mathcal{O}(m \cdot \log^{\epsilon} n)$ (for any small $\epsilon > 0$) then such an improvement is possible by Corollary V.3, but it is open whether query time $\mathcal{O}(\log n/\log\log n)$ can be achieved with space $\mathcal{O}(m)$. By the lower bound from [38] this would be an optimal random-access data structure for grammar-compressed strings.

Acknowledgements

Markus Lohrey has been supported by the DFG research project LO 748/10-1, Artur Jeż was supported under National Science Centre, Poland project number 2017/26/E/ST6/00191.

References

[1] M. Ganardi, A. Jeż, and M. Lohrey, "Balancing straight-line programs," *CoRR*, vol. abs/1902.03568, 2019. [Online]. Available: http://arxiv.org/abs/1902.03568

[2] W. Rytter, "Application of Lempel-Ziv factorization to the approximation of grammar-based compression," *Theoretical Computer Science*, vol. 302, no. 1–3, pp. 211–222, 2003.

[3] M. Charikar, E. Lehman, D. Liu, R. Panigrahy, M. Prabhakaran, A. Sahai, and A. Shelat, "The smallest grammar problem," *IEEE Transactions on Information Theory*, vol. 51, no. 7, pp. 2554–2576, 2005.

[4] A. Jeż, "Approximation of grammar-based compression via recompression," *Theoretical Computer Science*, vol. 592, pp. 115–134, 2015. [Online]. Available: https://doi.org/10.1016/j.tcs.2015.05.027

[5] ——, "A *really* simple approximation of smallest grammar," *Theoretical Computer Science*, vol. 616, pp. 141–150, 2016. [Online]. Available: http://dx.doi.org/10.1016/j.tcs.2015.12.032

[6] M. Lohrey, "Algorithmics on SLP-compressed strings: a survey," *Groups Complexity Cryptology*, vol. 4, no. 2, pp. 241–299, 2012.

[7] P. Bille, G. M. Landau, R. Raman, K. Sadakane, S. R. Satti, and O. Weimann, "Random access to grammar-compressed strings and trees," *SIAM Journal on Computing*, vol. 44, no. 3, pp. 513–539, 2015.

[8] D. Belazzougui, P. H. Cording, S. J. Puglisi, and Y. Tabei, "Access, rank, and select in grammar-compressed strings," in *Proceedings of the 23rd Annual European Symposium on Algorithms, ESA 2015*, ser. Lecture Notes in Computer Science, vol. 9294. Springer, 2015, pp. 142–154.

[9] P. Bille, P. H. Cording, and I. L. Gørtz, "Compressed subsequence matching and packed tree coloring," *Algorithmica*, vol. 77, no. 2, pp. 336–348, 2017. [Online]. Available: https://doi.org/10.1007/s00453-015-0068-9

[10] P. Bille, I. L. Gørtz, P. H. Cording, B. Sach, H. W. Vildhøj, and S. Vind, "Fingerprints in compressed strings," *Journal of Computer and System Sciences*, vol. 86, pp. 171–180, 2017.

[11] T. I, W. Matsubara, K. Shimohira, S. Inenaga, H. Bannai, M. Takeda, K. Narisawa, and A. Shinohara, "Detecting regularities on grammar-compressed strings," *Information and Computation*, vol. 240, pp. 74–89, 2015.

[12] L. Gasieniec, R. M. Kolpakov, I. Potapov, and P. Sant, "Real-time traversal in grammar-based compressed files," in *Proceedings of the 2005 Data Compression Conference, DCC 2005*. IEEE Computer Society, 2005, p. 458.

[13] M. Lohrey, S. Maneth, and C. P. Reh, "Constant-time tree traversal and subtree equality check for grammar-compressed trees," *Algorithmica*, vol. 80, no. 7, pp. 2082–2105, 2018.

[14] P. Gawrychowski, S. Jo, S. Mozes, and O. Weimann, "Compressed range minimum queries," *CoRR*, vol. abs/1902.04427, 2019. [Online]. Available: http://arxiv.org/abs/1902.04427

[15] G. Busatto, M. Lohrey, and S. Maneth, "Efficient memory representation of XML document trees," *Information Systems*, vol. 33, no. 4–5, pp. 456–474, 2008.

[16] M. Lohrey, "Grammar-based tree compression," in *Proceedings of the 19th International Conference on Developments in Language Theory, DLT 2015*, ser. Lecture Notes in Computer Science, vol. 9168. Springer, 2015, pp. 46–57.

[17] M. Lohrey, S. Maneth, and R. Mennicke, "XML tree structure compression using RePair," *Information Systems*, vol. 38, no. 8, pp. 1150–1167, 2013.

[18] A. Gascón, M. Lohrey, S. Maneth, C. P. Reh, and K. Sieber, "Grammar-based compression of unranked trees," in *Proceedings of 13th International Computer Science Symposium in Russia, CSR 2018*, ser. Lecture Notes in Computer Science, vol. 10846. Springer, 2018, pp. 118–131.

[19] P. Bille, I. L. Gørtz, G. M. Landau, and O. Weimann, "Tree compression with top trees," *Information and Computation*, vol. 243, pp. 166–177, 2015.

[20] P. Bille, F. Fernstrøm, and I. L. Gørtz, "Tight bounds for top tree compression," in *Proceedings of the 24th International Symposium on String Processing and Information Retrieval, SPIRE 2017*, ser. Lecture Notes in Computer Science, vol. 10508. Springer, 2017, pp. 97–102.

[21] B. Dudek and P. Gawrychowski, "Slowing down top trees for better worst-case compression," in *Proceedings of the Annual Symposium on Combinatorial Pattern Matching, CPM 2018*, ser. LIPIcs, vol. 105. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2018, pp. 16:1–16:8.

[22] L. Hübschle-Schneider and R. Raman, "Tree compression with top trees revisited," in *Proceedings of the 14th International Symposium on Experimental Algorithms, SEA 2015*, ser. Lecture Notes in Computer Science, vol. 9125. Springer, 2015, pp. 15–27.

[23] G. L. Miller and S. Teng, "Tree-based parallel algorithm design," *Algorithmica*, vol. 19, no. 4, pp. 369–389, 1997.

[24] S. R. Kosaraju, "On parallel evaluation of classes of circuits," in *Proceedings of the 10th Conference on Foundations of Software Technology and Theoretical Computer Science, FSTTCS 1990*, ser. Lecture Notes in Computer Science, vol. 472. Springer, 1990, pp. 232–237.

[25] L. G. Valiant, S. Skyum, S. Berkowitz, and C. Rackoff, "Fast parallel computation of polynomials using few processors," *SIAM Journal on Computing*, vol. 12, no. 4, pp. 641–644, 1983.

[26] E. Allender, J. Jiao, M. Mahajan, and V. Vinay, "Non-commutative arithmetic circuits: Depth reduction and size lower bounds," *Theoretical Computer Science*, vol. 209, no. 1-2, pp. 47–86, 1998.

[27] M. Paterson and L. G. Valiant, "Circuit size is nonlinear in depth," *Theoretical Computer Science*, vol. 2, no. 3, pp. 397–400, 1976.

[28] R. Cole and U. Vishkin, "The accelerated centroid decomposition technique for optimal parallel tree evaluation in logarithmic time," *Algorithmica*, vol. 3, pp. 329–346, 1988.

[29] M. Ganardi, D. Hucke, A. Jeż, M. Lohrey, and E. Noeth, "Constructing small tree grammars and small circuits for formulas," *Journal of Computer and System Sciences*, vol. 86, pp. 136–158, 2017. [Online]. Available: http://dx.doi.org/10.1016/j.jcss.2016.12.007

[30] M. Ganardi and M. Lohrey, "A universal tree balancing theorem," *ACM Transaction on Computation Theory*, vol. 11, no. 1, pp. 1:1–1:25, Oct. 2018.

[31] D. Harel and R. E. Tarjan, "Fast algorithms for finding nearest common ancestors," *SIAM Journal on Computing*, vol. 13, no. 2, pp. 338–355, 1984.

[32] R. E. Tarjan, *Data Structures and Network Algorithms*. Philadelphia, PA, USA: Society for Industrial and Applied Mathematics, 1983.

[33] P. Bille, I. L. Gørtz, P. Gawrychowski, G. M. Landau, and O. Weimann, "Top tree compression of tries," *CoRR*, vol. abs/1902.02187, 2019. [Online]. Available: http://arxiv.org/abs/1902.02187

[34] M. L. Fredman and D. E. Willard, "Surpassing the information theoretic bound with fusion trees," *Journal of Computer and System Sciences*, vol. 47, no. 3, pp. 424–436, 1993.

[35] Y. Lifshits, "Processing compressed texts: A tractability border," in *Proceedings of the 18th Annual Symposium on Combinatorial Pattern Matching, CPM 2007*, ser. Lecture Notes in Computer Science, vol. 4580. Springer, 2007, pp. 228–240. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-73437-6_24

[36] A. Jeż, "Faster fully compressed pattern matching by recompression," *ACM Transactions on Algorithms*, vol. 11, no. 3, pp. 20:1–20:43, Jan 2015. [Online]. Available: http://doi.acm.org/10.1145/2631920

[37] M. Bojańczyk and I. Walukiewicz, "Forest algebras," in *Proceedings of Logic and Automata: History and Perspectives [in Honor of Wolfgang Thomas]*., ser. Texts in Logic and Games, vol. 2. Amsterdam University Press, 2008, pp. 107–132.

[38] E. Verbin and W. Yu, "Data structure lower bounds on random access to grammar-compressed strings," in *Proceedings of the 24th Annual Symposium on Combinatorial Pattern Matching, CPM 2013*, ser. Lecture Notes in Computer Science, vol. 7922. Springer, 2013, pp. 247–258.