Grammar-based compression for strings and trees

DISSERTATION
zur Erlangung des Grades eines Doktors
der Naturwissenschaften

vorgelegt von
Danny Hucke, Dipl. Math. (FH)

eingereicht bei der Naturwissenschaftlich-Technischen Fakultät
der Universität Siegen
Siegen 2019

Betreuer und erster Gutachter
Prof. Dr. Markus Lohrey
Universität Siegen

Zweiter Gutachter
Prof. Dr. Johannes Fischer
Technische Universität Dortmund

Tag der mündlichen Prüfung
25.11.2019

Abstract

The goal of grammar-based compression is to represent a string by a small context-free grammar that produces only this string. Such a grammar is called a straight-line program (SLP). Grammar-based compression is a powerful tool to efficiently store data and process the compressed representation without decompressing it. In the first part of this work, we study the grammar-based compressors LZ78, BiSection, RePair, Greedy and LongestMatch, which are among the most popular compressors in this area. In the seminal work "The smallest grammar problem" by Charikar et al., the authors derived lower and upper bounds on the approximation ratios of several grammar-based compressors including the algorithms mentioned above. Unfortunately, for none of the compressors the presented bounds matched. Here, we close the gaps for LZ78 and BiSection. To be more precise, we show that the approximation ratio of LZ78 is $\Theta((n/\log n)^{2/3})$ and the approximation ratio of BiSection is $\Theta(\sqrt{n/\log n})$. For RePair, we improve the lower bound from $\Omega(\sqrt{\log n})$ to $\Omega(\log / \log\log n)$ and for Greedy from approximately 1.138 to approximately 1.348. The best known upper bound in both cases is $\mathcal{O}((n/\log n)^{2/3})$. Moreover, we improve a result of Arpe and Reischuk which relates grammar-based compression for arbitrary alphabets and binary alphabets.

In the second part of this work, we consider grammar-based compression for trees. The main principle is similar, because the goal is to represent a tree by a small linear context-free tree grammar that produces only this tree. Such a tree grammar is called a tree straight-line program (TSLP). As a main contribution, we present two algorithms that produce a TSLP of size $\mathcal{O}(n/\log_\sigma n)$ for any given tree with n nodes and σ many node labels, where we assume that the maximal number of children of a node in the tree is bounded by a constant. Additionally, the obtained TSLP has logarithmic depth. We show that those properties can be achieved in logarithmic space, or alternatively, in linear time. Similar results on the worst-case size of SLPs are well known.

We use our constructions for two applications: First, we apply TSLPs to the problem of transforming arithmetical formulas into equivalent circuits of size $\mathcal{O}((n \cdot \log m)/\log n)$ and depth $\mathcal{O}(\log n)$, where n is the size of the formula and m is number of different variables occurring in the formula. As a second application, we present a binary encoding of unlabeled binary trees based on grammar-based tree compression. We prove that this encoding is worst-case universal and thus asymptotically optimal for certain tree sources.

Acknowledgements

I would like to thank my advisor Markus Lohrey. He is a brilliant researcher and a patient mentor, who helped me to evolve from a student to a researcher. Although I had my problems with the city of Siegen at the beginning, I am very thankful for the opportunity he gave me and I would definitely make the decision to follow him from Leipzig to Siegen again.

I am also very grateful to Johannes Fischer for coexamining this thesis. Further, I want to thank all of my coauthors and colleagues for many fruitful discussions about various interesting topics. A special thanks in this context goes to Moses Ganardi, who had to sit next to me for years and endured so many questions from my side without being annoyed (as far as I noticed). At least it was not in vain, because I really like the many different results we achieved over the years together with Markus and the others.

Last but not least, I want to thank my wonderful family for the huge support over the years. My parents always let me be who I am and I know that I was not the easiest child you could have. I can only hope that my little daughter Luna thinks the same about me when she is an adult. A special thanks goes to my beautiful wife Christiane. She has to do the double work at home during my stays in Siegen, but she never complains and her loving support in all situations means the world to me.

Finally, this work is dedicated to my beloved grandma, whom I miss more than words can ever say.

Contents

Chapter 1

Introduction

The amount of digital data around the world is beyond imagination and with more and more people having access to the world wide web, the speed of producing data accelerates. According to a report of the International Data Corporation (IDC) from November 2018 [98], the size of the global datasphere was about 33 zettabytes in 2018 and it is predicted to grow to 175 zettabytes in 2025. For readers unfamiliar with this magnitude, one zettabyte is one trillion gigabytes, so if one would try to download 175 zettabytes at an average of 45 megabytes per second, then it would take one person one billion years to do it. But not only the pure amount of data grows massively, the importance of collecting, storing and analyzing digital data increases in almost all economic sectors. The hardware and software systems needed to store, protect and share data are complex, difficult to manage and therefore expensive. This motivates the goal of storing data as efficient and succinct as possible from the perspective of modern economy. In this work, we investigate a compression method called *grammar-based compression*, where a text is represented by a context-free grammar. This topic has become an active research field of lossless data compression during the past 25 years.

Before we start a detailed explanation of this approach, let us briefly revisit the path data compression has taken to make this work possible. Probably the first instance of data compression is the famous Morse code from the 19th century, where common letters in the english language have short codes while less frequent letters are represented by longer codes. Later, when mainframe computers began to take hold in the middle of the 20th century, Claude Shannon and Robert Fano invented Shannon-Fano coding [40, 105]. The rough idea is again that symbols with high probabilities are represented by short codes and symbols with lower probabilities are represented by longer codes. This approach was optimized by David Huffman, who constructed a similar yet more efficient encoding; the Huffman code [62]. In fact, the Huffman code is an optimal prefix-free binary encoding of symbols based on given probabilities (or relative frequencies) and it is part of practical compression methods such as, for example, bzip2. In 1977, Abraham Lempel and Jacob Ziv published their groundbreaking compression algorithm LZ77 [112], the first *dictionary-based*

compression algorithm. In contrast to the codes mentioned so far, dictionary-based compression does not rely exclusively on the frequencies of the occurring symbols but exploits the repetitiveness that most of the fastest-growing datasets exhibit. LZ77 is a greedy left-to-right parse of the input text into maximal factors such that each factor is either a symbol or already occurs to the left; factorizations with this property are known as unidirectional factorizations. The LZ77 algorithm computes an optimal unidirectional factorization, where the size is measured in the number of factors. One year after LZ77, Lempel and Ziv introduced a second dictionary-based compression algorithm known as LZ78 [113]. The LZ78 algorithm also performs a greedy left-to-right parse of the input into maximal factors, but here each factor is either a symbol or a concatenation of a previous factor followed by a symbol. This brings us back to grammar-based compression already, because LZ78 can be seen as the first example of this compression technique (even if it was not introduced this way).

Grammar-based string compression. The goal of grammar-based compression is to represent a given word or text w by a small context-free grammar that produces only w. Consider the grammar with the following rules:

$$S \to bXY\ Z\ bZY, \quad X \to an, \quad Y \to Xas, \quad Z \to Xd$$

Here, S is the start variable (or nonterminal) of the grammar. Applying the above rules as rewrite rules yields

$$S \Rightarrow bXY\ Z\ bZY \Rightarrow^* banXas\ Xd\ bXdXas \Rightarrow^* bananas\ and\ bandanas$$

as the text represented by the grammar. Such context-free grammars, where exactly one word is produced, are called *straight-line programs* (SLPs for short). This formalism was introduced independently in different contexts [13, 37, 100] and under different names. For instance, in [13, 37] the term *word chains* was used since SLPs generalize addition chains from numbers to words. An algorithm that computes an SLP for a given word w is called a *grammar-based compressor*. There are various grammar-based compressors that can be found in many places in the literature. Besides LZ78, well known examples are BiSection [74], RePair [77], Greedy [7, 8] and LongestMatch [72], just to mention the algorithms which will be the subject of further investigation in this work.

In the best case, one gets an SLP of size $\Theta(\log n)$ for a word of length n, where the size of an SLP is defined as the sum of the lengths of all right-hand sides of the rules (the SLP above has size 16, including the two blanks). Consider for example the SLP with rules $A_k \to ab$ and $A_{i-1} \to A_i A_i$ for $1 \leq i \leq k$, where A_0 is the start nonterminal. The produced word is $(ab)^{2^k}$ and the SLP has size $2k+2$. On the negative side, it is shown in [29, 102] that an SLP of size m can be transformed into a unidirectional factorization of size at most m, which implies that the LZ77 factorization of any word is at most as big as a smallest SLP for this word. Further, it is known from [29, 108] that in general it is not even possible to compute a smallest SLP for a given word in polynomial time unless

$\mathsf{P} = \mathsf{NP}$. An unfamiliar reader might wonder at this point why grammar-based compression is worth further investigation. A significant advantage of SLPs over other compression methods such as LZ77 is that a lot of computational tasks can be efficiently performed on the compressed representation of the data without decompressing it. This is a beneficial property, because in a lot of applications the compressed data is not only stored, but also processed on demand, and decompressing the data for every access defeats the purpose of having a succinct representation in the first place. The basic example here is accessing random positions of a compressed text. If a text is compressed via LZ77, then random access essentially requires to decompress the text from the beginning. In contrast, for a given SLP of size m producing a text of size n, one can construct in time $\mathcal{O}(m)$ a data structure of size $\mathcal{O}(m)$ (in words of bit length $\log n$) such that the access time is $\mathcal{O}(\log n)$ [17, 48], or alternatively, for any $\varepsilon > 0$ one can construct in time $\mathcal{O}(m \log^{\varepsilon} n)$ a data structure of size $\mathcal{O}(m \log^{\varepsilon} n)$ (again in words of bit length $\log n$) such that the access time is $\mathcal{O}(\log n / \log\log n)$ [12, 48][1]. A more detailed comparison between grammar-based compression and LZ77 (and a third compression technique known as compressed suffix arrays) for the indexing problem can be found in [92]. Other basic problems where given SLPs admit efficient algorithms are for example pattern matching [51, 63, 67, 80, 91], equality checking [57, 89, 96] and testing membership in a regular language given by a finite automaton [87, 97]. The reader can find a more detailed overview for algorithmics on SLP-compressed inputs in [81]. An interesting application in this context is that in some situations one can even speed up computations by first compressing the (uncompressed) inputs using SLPs and then applying the solution strategy to the SLPs in order to efficiently solve the problem. This approach is known as acceleration by compression and the reader can find an example in [35, 56], where the edit-distance of two given strings is computed in sub-quadratic time. It is worth mentioning here that the acceleration is based on the fact that there are linear time grammar-based compressors such as LZ78 where it is guaranteed that the produced SLP has size $\mathcal{O}(n/\log_{\sigma} n)$ for any word of length n over an alphabet of size σ. This worst-case result is well known for SLPs (see e.g. [13, 34, 74]) and will play a crucial role later when we apply grammar-based compression to trees.

A second point in favor of grammar-based compression is that the gap between the size of a smallest SLP and the size of the LZ77 factorization is bounded by a factor that depends only logarithmically on the word length. To be more accurate, one can efficiently construct an SLP of size $\mathcal{O}(m \cdot \log(n/m))$ for an input of length n, where m is the size of the LZ77 factorization [29, 102]. It is also shown in [29] that this construction is almost optimal, i.e, there is an infinite family of words such that the smallest SLP has size $\Omega(m \cdot (\log n / \log\log n))$, where n is again the word length and m is the size of the LZ77 factorization. As mentioned above, the size of the LZ77 factorization is a lower bound for the size of a smallest SLP, so the grammar-based compressors presented in [29, 102] approximate a

[1] In [12] this result was only shown for balanced SLPs, whereas in [48] it is shown that one can balance a given SLP such that the size only increases by a constant multiplicative factor.

smallest grammar for a given word of length n by the same multiplicative factor $\mathcal{O}(\log n)$. Other grammar-based compressors that achieve the same upper bound are presented in [64, 65, 103]. The study of grammar-based compressors as approximation algorithms for a smallest SLP was initiated in the seminal work of Charikar et al. [29] and will be the main subject of the first part of this work. Formally, for a grammar-based compressor $\mathcal{C}$ that computes an SLP $\mathcal{C}(w)$ for a word w, one defines the approximation ratio of $\mathcal{C}$ on w as the quotient of the size of $\mathcal{C}(w)$ and the size $g(w)$ of a smallest SLP for w. The approximation ratio $\alpha_{\mathcal{C}}(n)$ is the maximal approximation ratio of $\mathcal{C}$ among all words of length n.

Results for grammar-based string compression (Chapter 3). In [29] the authors provide lower and upper bounds for the approximation ratios of several grammar-based compressors, among them are the compressors LZ78, BiSection, RePair and Greedy, but for none of the compressors the lower and upper bounds match. These compression algorithms are among the most popular grammar-based compressors. Before we state our results, let us briefly justify the interest that those compressors arouse. As mentioned before, LZ78 is a classical algorithm and it is the basis of several widely used text compressors such as LZW (Lempel-Ziv-Welch). RePair and Greedy belong to the same family of grammar-based compressors known as *global algorithms*. Global algorithms show excellent practical compression results in many applications. In [8] the authors observe that compression based on Greedy outperforms gzip (based on LZ77) and bzip2 (based on the Burrows-Wheeler Transform [25] and Huffman coding) on DNA sequences. RePair, probably the best known global algorithm, achieves the best compression results among the tested grammar-based compressors in [16, 44] and found applications, among others, in web graph compression [31], searching compressed texts [69], suffix array compression [54] and (in a slightly modified form in) XML compression [82]. Some variants and improvements of RePair can be found in [16, 43, 44, 49, 88]. BiSection was first studied in the context of universal lossless compression [74] (called MPM there). On binary strings of length 2^n, BiSection basically produces the ordered binary decision diagram (OBDD) of the Boolean function represented by the bit string (each bit represents the output of the function for some input from $\{0,1\}^n$); see also [71]. OBDDs are widely used as a data structure in the area of hardware verification.

In this work, we present improved lower bounds on the approximation ratios of LZ78, BiSection, RePair and Greedy. For LZ78 and BiSection, these improvements yield the first known matching bounds for grammar-based compressors. To be more precise, we show that the approximation ratio of LZ78 is $\Theta((n/\log n)^{2/3})$ and the approximation ratio of BiSection is $\Theta(\sqrt{n/\log n})$. For the global algorithms, we improve the lower bound for RePair from $\Omega(\sqrt{\log n})$ to $\Omega(\log / \log\log n)$ and for Greedy from approximately 1.138 to approximately 1.348. The best known upper bound in both cases is $\mathcal{O}((n/\log n)^{2/3})$ [29]. The obtained results are summarized in Table 1.1. Additionally, we investigate the approximation ratios of those algorithms and another global algorithm known as LongestMatch in the setting where inputs are restricted to be unary. In this

Algorithm	Approximation ratio	
	Lower bound	Upper bound
BiSection	$\Theta(\sqrt{n/\log n})$	
LZ78	$\Theta((n/\log n)^{2/3})$	
RePair	$\Omega(\log n/\log\log n)$	$\mathcal{O}((n/\log n)^{2/3})$
Greedy	$1.34847194\ldots$	$\mathcal{O}((n/\log n)^{2/3})$

Table 1.1: The best known bounds on the approximation ratios of the grammar-based compressors studied in this work. The red colored bounds are presented in Chapter 3. For BiSection and LZ78 we provide the matching lower bounds, the corresponding upper bounds as well as all non-colored bounds are shown in [29].

setting, grammar-based compression is strongly related to the field of addition chains. The reader can find a more detailed summary of those latter achievements at the beginning of Chapter 3.

Another result presented in this work connects the hardness of grammar-based compression over binary alphabets to arbitrary alphabets. We show that if there is a polynomial time grammar-based compressor with approximation ratio c (a constant) on binary words, then there is a polynomial time grammar-based compressor with approximation ratio $6c$ on arbitrary words. This improves upon a result of Arpe and Reischuk [10], where a quite technical block encoding of arbitrary alphabets is used. Our approach uses a very simple construction, where the i-th symbol of the arbitrary alphabet is encoded by a^ib (the binary alphabet is $\{a, b\}$).

Finally, we revisit two previously known results for grammar-based string compression: We prove that the worst-case size of SLPs produced by BiSection is $\mathcal{O}(n/\log_\sigma n)$ [74] for inputs of size n over an alphabet of size σ, and we show how grammar-based compression is used for universal source coding in [72]. Both topics play an important role in the second part of this work, where we achieve similar results for *grammar-based tree compression*.

Grammar-based tree compression. In [26], grammar-based compression was extended from strings to trees. Trees are ubiquitous in computer science and they appear in various data structures such as, for example, suffix trees, binary search trees, Cartesian trees, wavelet trees, red-black trees and AVL-trees; see [20] for details. More generally, a lot of data has a more complex structure than a one-dimensional text. In particular, quite often data features a hierarchical structure, which can be modeled as a tree. The basic example here is XML, the universal format for structured documents and data on the world wide web. While it is possible to flatten trees and compress the result using string compression, this disregards the structure of the data and might make processing the compressed representation less efficient [47]. This motivates the goal of compressing trees without losing too much of the structure. The trees we

aim to compress in this work are *rooted ordered trees* over a ranked alphabet, i.e., every node has an order among its children and is labeled by a ranked symbol, where the rank of this symbol is equal to the number of children of the node.

The basic formalism that we use is the tree counterpart of context-free string grammars: Linear context-free tree grammars (see [32] for details about tree grammars). A linear context-free tree grammar that produces only a single tree is called a *tree straight-line program* (TSLP) or *straight-line context-free tree grammar* (SLCF tree grammar) and an algorithm which computes a TSLP for a given tree is called a *grammar-based tree compressor*. It is a nice property of TSLPs that they can be seen as a generalization of *directed acyclic graphs* (DAGs), which are widely used as a compact tree representation. Whereas DAGs only allow to share repeated subtrees of a given tree, TSLPs can also share repeated internal tree patterns. DAGs found applications in numerous areas such as compiler construction [3, Chapter 6.1 and 8.5], unification [95], XML querying [24, 42], and symbolic model-checking (binary decision diagrams) [22].

Similar to SLPs, it is known that TSLPs efficiently support various computational tasks. Let us first mention the problem of navigating in a tree represented by a given TSLP. Tree navigation plays an important role, for example, when it comes to querying or transforming XML documents as it is done by widely used languages such as XQuery or XSLT. In [84], the authors show that for a given TSLP of size m, one can precompute in time $\mathcal{O}(m)$ a data structure of size $\mathcal{O}(m)$ that allows to move from a node of the tree in constant time to its parent node or to its i-th child and to return in constant time the node label of the current node. Further, this data structure is extended in the same paper such that subtree equality checks can be carried out in constant time as well. This problem occurs in several contexts, see for instance [27]. Other basic examples where TSLPs admit polynomial time algorithms are equality checking [104] and evaluating tree automata [85]. The reader can find more details in the survey [81].

While it is not possible to compute a smallest TSLP in polynomial time unless $\mathsf{P} = \mathsf{NP}$ (this follows directly from the corresponding result for SLPs), several grammar-based tree compressors are published [5, 19, 26, 66, 82][2]. The algorithms from [53, 66] achieve an approximation ratio of $\mathcal{O}(\log n)$ compared to a smallest TSLP (for a constant set of node labels), which matches the best known bound for SLPs as described above. It is worth mentioning that in [53], the authors use a familiar approach: First, they extended the LZ77 factorization from strings to trees. Then, for a given tree with n nodes, the LZ77 factorization of size m is transformed into a TSLP of size $\mathcal{O}(m \cdot \log(n/m))$. Similar to strings, the LZ77 factorization of a tree is a lower bound on the size of a smallest TSLP, which yields the claimed approximation ratio. On the other hand, for none of the mentioned compressors it is known whether for every input tree with n nodes the size of the TSLP is bounded by $\mathcal{O}(n/\log n)$ as it is the case for grammar-based string compression.

[2]The tree compressor presented in [5] is based on a different type of tree grammars, so-called elementary ordered tree grammars.

Results for grammar-based tree compression (Chapter 4). Our main result for grammar-based tree compression addresses the worst-case size of TSLPs. We present two grammar-based compressors, TreeBiSection and BU-Shrink, which compute TSLPs of size $\mathcal{O}(n/\log_\sigma n)$ for a given tree with n nodes and σ many different node labels. It is important that this result is based on the assumption that the maximal number of children of a node in the tree is bounded by a constant. The first compressor TreeBiSection is basically an extension of the BiSection algorithm [74] from strings to trees. An important part of this algorithm is that one can hierarchically decompose a given tree in a top-down way into pieces of roughly equal size. This decomposition is based on a well known technique from [78]. In a second step, we use DAG compression to identify identical pieces produced by this decomposition and form a TSLP based on this information. The TSLP obtained by TreeBiSection is balanced, i.e., the depth of the corresponding derivation tree is bounded by $\mathcal{O}(\log n)$. We prove that TreeBiSection can be implemented in logarithmic space and, using an alternative implementation, achieves the running time $\mathcal{O}(n \log n)$. We are not aware of a linear time implementation for this algorithm and therefore we present a second algorithm called BU-Shrink (for bottom-up shrink) that constructs a TSLP of size $\mathcal{O}(n/\log_\sigma n)$ in linear time. In a first step, BU-Shrink merges nodes of the input tree in a bottom-up way. Thereby it constructs a partition of the input tree into $\mathcal{O}(n/\log_\sigma n)$ many connected parts of small size. Every such connected part represents a pattern occurring in the input tree. We then apply DAG compression to the forest consisting of all those patterns in order to share identical patterns using the same nonterminal. The final TSLP is then produced by replacing the patterns in the shrinked version of the input tree by the obtained nonterminals. A more detailed summary of both algorithms can be found at the beginning of Chapter 4. While the TSLP produced by BU-Shrink is not balanced, we present a combination of this algorithm and TreeBiSection which yields a linear running time for constructing a TSLP of size $\mathcal{O}(n/\log_\sigma n)$ and depth $\mathcal{O}(\log n)$. It is worth mentioning that one could achieve the same result without using TreeBiSection by applying a balancing technique recently introduced in [48] to the TSLP obtained by BU-Shrink.

As mentioned above, it is important to note that our size bound $\mathcal{O}(n/\log_\sigma n)$ only holds for trees in which the maximal number of children of a node is bounded by a constant. In particular, it does not directly apply to unranked trees, which are the standard tree model for XML documents. To overcome this limitation, one can transform an unranked tree into its first-child-next-sibling encoding [76], which is a ranked tree of the same size as the original tree. Then, the first-child-next-sibling encoding can be transformed into a TSLP of the claimed size. A second possibility is to use a recently introduced generalization of TSLPs for unranked trees called *forest straight-line programs* (or FSLPs) [50]. It is shown in [50] that FSLPs for unranked trees and TSLPs for the first-child-next-sibling encoding of unranked trees are equally succinct up to constant multiplicative factors and that one can change between both representations in linear time. This directly yields an FSLP of size $\mathcal{O}(n/\log_\sigma n)$ for unranked trees of size n with σ many node labels based on the grammar-based compressors described

above. FSLPs are strongly related to another well studied compression technique called *top dags* [14, 15, 39, 58]. It is again shown in [50] that a top dag can be transformed in linear time into an equivalent FSLP with a constant multiplicative blow-up.

We present two applications of the worst-case size results mentioned above: A first application concerns arithmetical circuits. We refine a result of Brent [21], which states that an arithmetical formula of size n over a commutative (semi)ring can be transformed into an equivalent circuit of size $\mathcal{O}(n)$ and depth $\mathcal{O}(\log n)$. By applying our constructions, we transform a formula (which is a tree) into an equivalent circuit of size $\mathcal{O}((n \cdot \log m)/\log n)$ and depth $\mathcal{O}(\log n)$, where m is the number of different variables in the formula.

As a second application, we use grammar-based tree compression for universal coding. Universal source coding for strings over a finite alphabet Σ is a well-established topic of information theory. Its goal is to find prefix-free lossless codes that are universal (or optimal) for classes of information sources. In a series of papers, Cosman, Kieffer, Nelson, and Yang developed grammar-based codes that are universal for the class of finite state sources [72, 73, 74, 109]; the code presented in [72] is presented in Chapter 3. Over the last few years, we have seen increasing efforts aiming to extend universal source coding to structured data like trees [75, 86, 111] and graphs [30, 70]. In [111], Kieffer, Yang, and Zhang started to extend their work on grammar-based source coding from strings to unlabeled binary trees using DAG compression. The authors obtain an average-case universal code for certain tree sources. In this work, we extend the binary encodings presented in [72] (for SLPs) and [111] (for DAGs) in order to present a new encoding for unlabeled binary trees based on TSLPs. Based on the fact that the grammar-based tree compressors described above produce TSLPs of size $\mathcal{O}(n/\log n)$ for unlabeled binary trees of size n, we prove that our code is worst-case universal for certain tree sources.

A more detailed summary of those applications can again be found at the beginning of Chapter 4.

Chapter 2

Preliminaries

2.1 Numbers and functions

Numbers. By $\mathbb{N}$ we denote the natural numbers (including 0) and for $i, j \in \mathbb{N}$, let $[i, j] = \{i, i+1, \ldots, j\}$ for $i \leq j$ and $[i, j] = \emptyset$ otherwise. By $\mathbb{R}$ we denote the real numbers and we use $\mathbb{R}_{\geq 0}$ for the positive real numbers including 0 and $\mathbb{R}_{>0} = \mathbb{R}_{\geq 0} \setminus \{0\}$. Further, we use $\mathbb{R}_{[0,1]} = \{x \in \mathbb{R} \mid 0 \leq x \leq 1\}$ which occurs especially when probabilities are used. The set of integers is denoted by $\mathbb{Z}$ and we use $\mathbb{Z}_n = \{0, 1, \ldots, n-1\}$. For $m, n \in \mathbb{N}$, we denote by $m \operatorname{div} n$ the integer division of m and n. We denote by $m \bmod n$ the modulo of m and n, i.e., $m \bmod n \in [0, n-1]$ and

$$m = (m \operatorname{div} n) \cdot n + (m \bmod n).$$

If m/n or $\frac{m}{n}$ is used, then this refers to the standard division over $\mathbb{R}$. Note that $m \operatorname{div} n = \lfloor m/n \rfloor$ and $(m \operatorname{div} n) + (m \bmod n) \geq m/n$.

$\mathcal{O}$-Notation and functions. We use the $\mathcal{O}$-notation in the usual sense. Formally, for a function $f \colon \mathbb{N} \to \mathbb{R}_{\geq 0}$ we use the following notation:

- $\mathcal{O}(f) = \{g \colon \mathbb{N} \to \mathbb{R}_{\geq 0} \mid \exists c > 0\, \exists n_0 \in \mathbb{N}\, \forall n \geq n_0 : g(n) \leq c \cdot f(n)\}$
- $o(f) = \{g \colon \mathbb{N} \to \mathbb{R}_{\geq 0} \mid \forall c > 0\, \exists n_0 \in \mathbb{N}\, \forall n \geq n_0 : g(n) \leq c \cdot f(n)\}$
- $\Omega(f) = \{g \colon \mathbb{N} \to \mathbb{R}_{\geq 0} \mid \exists c > 0\, \exists n_0 \in \mathbb{N}\, \forall n \geq n_0 : g(n) \geq c \cdot f(n)\}$
- $\Theta(f) = \mathcal{O}(f) \cap \Omega(f)$

As usual, we write $g(n) \geq \Omega(f(n))$, $g(n) = \Theta(f(n))$ and $g(n) \leq \mathcal{O}(f(n))$ instead of $g(n) \in \Omega(f(n))$, $g(n) \in \Theta(f(n))$ and $g(n) \in \mathcal{O}(f(n))$ to emphasize the relation of f and g. For lower bounds we often use phrases like

"For infinitely many n, we have $g(n) \geq \Omega(f(n))$."

which formally means that there exists $c > 0$ such that $g(n) \geq c \cdot f(n)$ holds for infinitely many different $n \in \mathbb{N}$. In particular, for each $n' \in \mathbb{N}$ there exists $n \in \mathbb{N}$ such that $n > n'$ and $g(n) \geq c \cdot f(n)$.

If we use the logarithm log without a base, then this refers to $\log_2$. Note that $\log_b(x) = \Theta(\log_{b'}(x))$ for fixed constants b and b' due to the well known identity $\log_b(x) = \log_{b'}(x)/\log_{b'}(b)$.

Binary numbers. Occasionally, we use numeral systems with other bases than the decimal numeral system. A *base b numeral system* represents $n \in \mathbb{N}$ such that

$$n = \sum_{i=0}^{\lfloor \log_b n \rfloor} c_{n,i} \cdot b^i,$$

where $c_{n,i} \in [0, b-1]$ for $i \in [0, \lfloor \log_b n \rfloor]$. If base $b = 2$ is used, then we denote by $b_{n,i} \in \{0, 1\}$ the binary coefficient of 2^i, i.e.,

$$n = \sum_{i=0}^{\lfloor \log n \rfloor} b_{n,i} \cdot 2^i. \tag{2.1}$$

If n is clear from the context, then we just write b_i instead of $b_{n,i}$. The *binary representation* of n is the concatenation of the binary coefficients beginning with the most significant bit $b_{\lfloor \log n \rfloor}$ and ending with the least significant bit b_0. For example the binary representation of 6 is 110. We denote by

$$\nu(n) = \sum_{i=0}^{\lfloor \log n \rfloor} b_{n,i} \tag{2.2}$$

the number of 1's in the binary representation of n, for example we have $\nu(6) = 2$. Note that $\nu(n) \leq \log n + 1$.

2.2 Words, alphabets and languages

Words and alphabets. Let $w = a_1 \cdots a_n$ $(a_1, \ldots, a_n \in \Sigma)$ be a *word* or *string* over an *alphabet* Σ. The length $|w|$ of w is n and we denote by ε the word of length 0. For $i, j \in [1, n]$, we use $w[i:j] = a_i \cdots a_j$ if $i \leq j$ and $w[i:j] = \varepsilon$ otherwise. Further, we simply use $w[i] = w[i:i] = a_i$ for $i \in [1, n]$. As usual, Σ^* denotes the set of all words over Σ and we use $\Sigma^+ = \Sigma^* \setminus \{\varepsilon\}$ for the set of all nonempty words. Further, for $n \in \mathbb{N}$ we denote by $\Sigma^n = \{w \in \Sigma^* \mid |w| = n\}$ the set of all words of length n. For $u, v \in \Sigma^*$, we write $u \cdot v$ or simply uv to represent the concatenation of u and v. For $w \in \Sigma^+$, we call $v \in \Sigma^+$ a *factor* of w if there exist $x, y \in \Sigma^*$ such that $w = xvy$. If $x = \varepsilon$ (respectively $y = \varepsilon$) then we call v a *prefix* (respectively *suffix*) of w. If v is a factor of w, we also write that v occurs in w. A factorization of w is a decomposition $w = f_1 \cdots f_\ell$ into factors $f_1, \ldots, f_\ell$ for $\ell \geq 1$. For $n \in \mathbb{N}$ and $w \in \Sigma^*$, we use $w^n = w \cdots w$ for the word where w is repeated n times. For words $w_1, \ldots, w_n \in \Sigma^*$, we further denote by $\prod_{i=j}^{n} w_i$ the word $w_j w_{j+1} \cdots w_n$ if $j \leq n$ and ε otherwise.

Regular expressions. A *language* $L \subseteq \Sigma^*$ is a set of words. We use regular expressions over $*$, $+$ and concatenation to describe languages in the usual sense. Formally, we use the following inductive definition:

- The empty set $\emptyset$ and the empty word ε are regular expressions and we have $L(\emptyset) = \emptyset$ and $L(\varepsilon) = \{\varepsilon\}$.
- Every $a \in \Sigma$ is a regular expression and we have $L(a) = \{a\}$.
- If α and β are regular expressions, then $\alpha\beta$ is a regular expression and we have $L(\alpha\beta) = L(\alpha)L(\beta) = \{uv \mid u \in L(\alpha),\ v \in L(\beta)\}$.
- If α is a regular expression, then α^* is a regular expression and we have $L(\alpha^*) = \{w_1 \cdots w_n \mid n \geq 0,\ w_i \in L(\alpha) \text{ for all } i \in [1, n]\}$.
- If α is a regular expression, then α^+ is a regular expression and we have $L(\alpha^+) = \{w_1 \cdots w_n \mid n \geq 1,\ w_i \in L(\alpha) \text{ for all } i \in [1, n]\}$.

When we use a regular expression α in the following, we do not distinguish between α and $L(\alpha)$ due to better readability. For example, we simply use $w \in 0^*$ instead of $w \in L(0^*)$ to describe $w = 0^n$ for some $n \geq 0$.

Context-free languages. A second formalism which we use to describe certain languages are *context-free grammars*. A context-free grammar $\mathbb{G}$ is a tuple $\mathbb{G} = (N, \Sigma, P, S)$, where N is a finite set of nonterminals with $N \cap \Sigma = \emptyset$, $S \in N$ is the start nonterminal, and P is a finite set of productions (or rules) of the form $A \to w$ for $A \in N$, $w \in (N \cup \Sigma)^*$. A word $u \in (N \cup \Sigma)^*$ can be derived from a word $v \in (N \cup \Sigma)^*$, if and only if there exists a rule $(A \to w) \in P$ and $v = xAy$ and $u = xwy$ for $A \in N$ and $x, y \in (N \cup \Sigma)^*$. We write $u \Rightarrow_{\mathbb{G}} v$ in this case or just $u \Rightarrow v$ if $\mathbb{G}$ is clear from the context. The language $L(\mathbb{G})$ produced by a context-free grammar $\mathbb{G}$ is defined as $L(\mathbb{G}) = \{w \in \Sigma^* \mid S \Rightarrow_{\mathbb{G}}^* w\}$, where $\Rightarrow_{\mathbb{G}}^*$ is the reflexive, transitive closure of $\Rightarrow_{\mathbb{G}}$, i.e., $u \Rightarrow_{\mathbb{G}}^* v$ if and only if $u = v$ or there exists $n \geq 0$ and $u_1, \ldots, u_n$ such that $u \Rightarrow_{\mathbb{G}} u_1 \Rightarrow_{\mathbb{G}} \cdots \Rightarrow_{\mathbb{G}} u_n \Rightarrow_{\mathbb{G}} v$.

Context-free grammars which produce a language L such that $|L| = 1$ play a central role in Chapter 3 (see Section 3.1 for further information).

2.3 Graphs and trees

Graphs. A (directed) *graph* $G = (V, E)$ consists of a finite set V of nodes and a finite set $E \subseteq V \times V$ of edges. For an edge $e = (u, v) \in E$, we say that e starts in u and ends in v. The *in-degree* of v is the number of edges that end in v and the *out-degree* of v is the number of edges that start in v. There is a (directed) *path* of length n from $u \in V$ to $v \in V$ if and only if there are nodes $u = v_0, v_1, \ldots, v_n = v$ such that $(v_i, v_{i+1}) \in E$ for $i \in [0, n-1]$. Note that for each node $v \in V$ there is a path of length 0 from v to v. The digraph G is *acyclic* if and only if for all $v \in V$ there is no path of length $n \geq 1$ from v to v.

Trees. A graph $G = (V, E)$ is a *rooted tree* with *root node* $r \in V$ if and only if (i) for each $v \in V$ there is a path from r to v and (ii) the in-degree of r is zero and the in-degree of each node $v \in (V \setminus \{r\})$ is exactly one. Let $G = (V, E)$ be a rooted tree with root $r \in V$. A node v is a child of u if and only if $(u, v) \in E$. In this case, we also refer to u as the parent node of v. A node u is an ancestor of v if and only if there is path of length $n \geq 1$ from u to v. A node v is called a *leaf* if it has out-degree zero. The *depth* or *level* of a node $v \in V$ is the length of the path from r to v. The *depth* or *height* of a tree is the maximal depth of a node $v \in V$. We use the *depth-first left-to-right order* of nodes in V, where we assume an order of the children of each node: A node $u \in V$ comes before $v \in V$ with respect to the depth-first order if u is an ancestor of v or if there exists a node $w \in V$ and $i < j$ such that the i-th child of w is an ancestor of u and the j-th child of w is an ancestor of v.

In this work, we mainly deal with *labeled trees*, i.e., each node is labeled by some information. Trees occur in this work in two contexts: At multiple points throughout this work, we model a certain problem or information using a tree as it is widely spread in computer science and mathematics. In this context, we often put less emphasis on the formal definition of the tree and focus on the intuition behind it. The second context is strongly related to a main part of this work which is presented in Chapter 4, where we abstract away from the concrete purpose of a tree and consider it as a data structure which we want to compress. In this context, we consider so-called *ranked trees* which we define as terms over a *ranked alphabet*. We defer the formal definitions to Section 4.1 for the sake of better readability of this chapter.

2.4 Distributions and empirical entropy

A discrete *probability distribution* $p : A \to \mathbb{R}_{[0,1]}$ on a countably (not necessarily finite) set A satisfies $\sum_{a \in A} p(a) = 1$. If A is finite and $p(a) = \frac{1}{|A|}$ for all $a \in A$, then we call p the *uniform distribution* on A.

Let $\overline{a} = (a_1, a_2, \ldots, a_n)$ be a tuple of elements with $a_i \in A$ for $i \in [1, n]$. The *empirical distribution* $p_{\overline{a}} : \{a_1, a_2, \ldots, a_n\} \to \mathbb{R}_{[0,1]}$ of $\overline{a}$ is defined by

$$p_{\overline{a}}(a) = \frac{|\{i \mid 1 \leq i \leq n,\ a_i = a\}|}{n}.$$

Note that the empirical distribution is a probability distribution on A. We use this definition also for words over some alphabet by identifying a word $w = a_1 a_2 \cdots a_n$ with the tuple $(a_1, a_2, \ldots, a_n)$. The *unnormalized empirical entropy* of $\overline{a}$ is

$$H(\overline{a}) = -\sum_{i=1}^{n} \log p_{\overline{a}}(a_i).$$

A well known generalization of Shannon's inequality states that for all real

numbers $p_1, \ldots, p_k, q_1, \ldots, q_k > 0$, if $\sum_{i=1}^{k} p_i = 1 \geq \sum_{i=1}^{k} q_i$ then

$$\sum_{i=1}^{k} -p_i \log(p_i) \leq \sum_{i=1}^{k} -p_i \log(q_i);$$

see [1] for a proof. As a consequence, for a tuple $\overline{a} = (a_1, a_2, \ldots, a_n)$ with $a_1, \ldots, a_n \in A$ and real numbers $q(a) > 0$ $(a \in A)$ with $\sum_{i=1}^{n} q(a_i) \leq 1$ we have

$$\sum_{i=1}^{n} -\log(p_{\overline{a}}(a_i)) \leq \sum_{i=1}^{n} -\log(q(a_i)). \tag{2.3}$$

2.5 Computational models

We will consider time and space bounds for computational problems. For time bounds, we will use the standard RAM model. We make the assumption that for an input of size n, arithmetical operations on numbers with $\mathcal{O}(\log n)$ bits can be carried out in time $\mathcal{O}(1)$. We assume that the reader has some familiarity with logspace computations, see e.g. [9, Chapter 4.1] for more details. A function can be computed in logspace, if it can be computed on a Turing machine with three tapes: a read-only input tape, a write-only output tape, and a read-write working tape of length $\mathcal{O}(\log n)$, where n is the length of the input. It is an important fact that if functions f and g can be computed in logspace, then the composition of f and g can be computed in logspace as well. We will use this fact implicitly.

Chapter 3

Grammar-based string compression

In this chapter we start our investigation of grammar-based compression, where a word w is represented by a context-free grammar that produces exactly $\{w\}$. Such a grammar is called a straight-line program (SLP) for w and an algorithm which computes an SLP for a given input word is called a grammar-based compressor. The problem of computing a smallest SLP for a given word is known as the smallest grammar problem. Storer and Szymanski [108] and Charikar et al. [29] proved that the smallest grammar problem cannot be solved in polynomial time unless $\mathsf{P} = \mathsf{NP}$. Even worse, unless $\mathsf{P} = \mathsf{NP}$ one cannot compute in polynomial time for a given word w an SLP of size smaller than $(8569/8568) \cdot g(w)$ [29], where $g(w)$ is the size of a smallest SLP for w. The construction in [29] uses an alphabet of unbounded size, and it was unknown whether this lower bound holds also for words over a fixed alphabet. In [29] it is remarked that the construction in [108] shows that the smallest grammar problem for words over a ternary alphabet cannot be solved in polynomial time unless $\mathsf{P} = \mathsf{NP}$. But this is not clear at all as Casel et al. explain in [28]. In the same paper [28], the authors prove that the smallest grammar problem for an alphabet of size 24 cannot be solved in polynomial time unless $\mathsf{P} = \mathsf{NP}$ using a rather complex extension of the constructions in [29, 108].

In Section 3.2.1, we follow an approach from [10] which might yield a similar hardness result for binary alphabets in the future. We show that if there is a polynomial time grammar-based compressor with constant approximation ratio c on binary words, then there is a polynomial time grammar-based compressor with approximation ratio $6c$ on arbitrary words. The approximation ratio $\alpha_{\mathcal{C}}(n)$ of a grammar-based compressor $\mathcal{C}$ is defined as the worst-case quotient of the size of the SLP produced by $\mathcal{C}$ and the size of a smallest SLP over all words of length n. By the results above, lowering the constant 6 in our construction to a value smaller than 8569/8568 would imply that the smallest grammar problem over binary alphabets cannot be solved in polynomial time unless $\mathsf{P} = \mathsf{NP}$.

The biggest part of this chapter deals with the approximation ratios of the grammar-based compressors BiSection (Section 3.3), LZ78 (Section 3.4), RePair (Section 3.6), Greedy (Section 3.7) and LongestMatch (Section 3.8). We distinguish for every algorithm between the approximation ratio restricted to unary inputs, and the general approximation ratio where inputs over arbitrary alphabets are allowed. Grammar-based compression on unary words is strongly related to the field of addition chains, which has been studied for decades (see [76, Chapter 4.6.3] for a survey) and still is an active topic due to the strong connection to public key cryptosystems (see [93] for a review from that point of view). An addition chain for an integer n of size m is a sequence of integers $1 = k_1, k_2, \ldots, k_m = n$ such that for each $d \in [2, m]$, there exists i, j $(1 \leq i, j < d)$ such that $k_i + k_j = k_d$. It is straightforward to compute from an addition chain for an integer n of size m an SLP for a^n of size $2m - 2$. Vice versa, an SLP for a^n of size m yields an addition chain for n of size m. So when we study grammar-based compressors restricted to unary inputs, this research could also be interpreted as a study of those algorithms as addition chain solvers. More detailed information on the relation between grammar-based compression and addition chains can be found in Section 3.2.2.

For RePair and LongestMatch, it turns out that for all unary inputs the SLP produced by RePair has the same size as the SLP produced by LongestMatch. In fact, both algorithms are basically identical to the binary method that produces an addition chain for n by creating powers of two using repeated squaring, and then the integer n is represented as the sum of those powers of two that correspond to a one in the binary representation of n. Based on that information, we show that for any unary input w the produced SLPs of RePair and LongestMatch have size at most $\log(3) \cdot g(w)$, and we provide a matching lower bound. The BiSection algorithm follows the same strategy as RePair and LongestMatch for unary inputs, but the corresponding SLP has a slightly different form resulting in an extra factor $\frac{4}{3}$, i.e., for every unary input w the size of the produced SLP is at most $\frac{4}{3}\log(3) \cdot g(w)$, and we provide again a matching lower bound. LZ78 is the only algorithm studied in this work which is not able to produce SLPs of logarithmic size in the length of the input. For a unary word a^n, the LZ78 factors are $a, a^2, a^3, \ldots, a^m, a^\ell$, where $\ell \in [0, m]$ and $\ell + \sum_{i=1}^{m} i = n$. It follows that LZ78 produces an SLP of size $\Theta(m) = \Theta(\sqrt{n})$ for any word of length n, which yields the approximation ratio $\Theta(\sqrt{n}/\log n)$.

Unfortunately, when it comes to Greedy, it is hard to analyze the behavior of this algorithm even for unary inputs due to the discrete optimization problem in each round. The rather technical upper bound that we achieve on the size of the SLP produced by Greedy on input a^n is $\mathcal{O}((\log n)^9(\log\log n)^3)$, which yields the approximation ratio $\mathcal{O}((\log n)^8(\log\log n)^3)$ in this setting. On the positive side, we use unary inputs to provide a new lower bound of 1.348...[1] for the approximation ratio of Greedy that improves the best known lower bound for inputs over arbitrary alphabets. The key to achieve the new bound is the

[1] The table on page 2556 in [29] states the better lower bound of 1.37..., but the authors only show the lower bound 1.137..., see [29, Theorem 11].

Algorithm	Approximation ratio (unary)	
	Lower bound	Upper bound
BiSection	$(4/3)\log(3)$	
LZ78	$\Theta(\sqrt{n}/\log n)$	
RePair	$\log(3)$	
Greedy	$1.34847194\ldots$	$\mathcal{O}((\log n)^8(\log\log n)^3)$
LongestMatch	$\log(3)$	

Table 3.1: The bounds on the approximation ratio restricted to unary inputs of the grammar-based compressors studied in this work.

sequence $y_k = y_{k-1}^2 + 1$ with $y_0 = 2$, which has been studied in [4] (among other sequences), where it is shown that $y_k = \lfloor \gamma^{2^k} \rfloor$ for $\gamma = 2.258...$. In order to prove the lower bound, we show that the SLP produced by Greedy on input a^{y_k} has size $3 \cdot 2^k - 1$, while a smallest SLP for a^{y_k} has size $3 \cdot \log_3(\gamma) \cdot 2^k + o(2^k)$ (this follows from a construction used to prove the lower bound for Greedy in [29]). The results on the approximation ratios for unary inputs are shown in Table 3.1.

The best known lower and upper bounds in the general setting can be found in Table 1.1, where inputs over arbitrary alphabets are considered. We present improved lower bounds for LZ78, BiSection and RePair in this work (we also improve the result for Greedy in the general setting, but this is already discussed in the unary case). For LZ78 and BiSection those improvements yield matching bounds on the approximation ratios. In particular, the approximation ratio of LZ78 is $\Theta((n/\log n)^{2/3})$ and the approximation ratio of BiSection is $\Theta(\sqrt{n/\log n})$. For RePair we improve the lower bound from $\Omega(\sqrt{\log n})$ to $\Omega(\log / \log\log n)$. It is worth mentioning in this context that a grammar-based compressor with an approximation ratio of $o(\log n/\log\log n)$ would improve Yao's method for computing a smallest addition chain for a set of numbers [110], which is a long standing open problem. So our new lower bound for RePair excludes it as a candidate for improving Yao's method. Moreover, we present in Section 3.6.3 a modified version of RePair such that it matches the approximation ratio of $\mathcal{O}(\log n/\log\log n)$ for computing a smallest addition chain for a set of numbers as it is presented in [110]. More details about the relation of grammar-based compression and addition chains for a set of numbers can be found in Section 3.2.2.

Finally, we present two previously known results about grammar-based compression: We prove in Section 3.3.3 that the worst-case size of SLPs produced by BiSection is $\mathcal{O}(n/\log_\sigma n)$ [74] for inputs of size n over an alphabet of size σ. Later, we extend BiSection to tree compression and prove a similar result on the worst-case size of the obtained tree compressor. In Section 3.9, we present a second known result which applies grammar-based compression to universal coding in the information-theoretical sense. In [72], the authors present a binary encoding of words based on SLPs such that a universal code for so-called finite-state information sources is obtained. Again, we show a similar result for tree

compression later in this work. It is worth mentioning in this context that the recent paper [94] bounds the size of the encoding presented in [74] by the (unnormalized) k-th order empirical entropy of the encoded string plus some lower order terms in case the encoding is based on so-called irreducible SLPs (this includes SLPs produced by the compressors studied in this work).

Results of this chapter are published in [59, 61].

3.1 Straight-line programs

A *straight-line program* $\mathbb{A} = (N, \Sigma, P, S)$, briefly SLP, is a context-free grammar that produces a single word $w \in \Sigma^+$. Syntactically, this is a achieved if

(i) for every $A \in N$, there exists exactly one production $(A \to w) \in P$, and

(ii) the relation $\{(A, B) \in N \times N \mid (A \to w) \in P,\ B \text{ occurs in } w\}$ is acyclic.

It is straightforward to show that if (i) and (ii) are satisfied, then for any nonterminal $A \in N$ there exists a unique $w \in \Sigma^+$ such that $A \Rightarrow_{\mathbb{A}}^* w$. We use $\mathsf{val}_{\mathbb{A}}(A) = w$ to denote the unique $w \in \Sigma^+$ with $A \Rightarrow_{\mathbb{A}}^* w$. If $\mathbb{A}$ is clear from the context, we omit the subscript and just write $\mathsf{val}(A)$. The string defined by the SLP $\mathbb{A}$ is $\mathsf{val}(\mathbb{A}) = \mathsf{val}_{\mathbb{A}}(S)$, i.e., $L(\mathbb{A}) = \{\mathsf{val}(\mathbb{A})\}$. If $\mathsf{val}(\mathbb{A}) = w$, then we also write that $\mathbb{A}$ is an SLP for w or $\mathbb{A}$ produces w. The *size* of the SLP $\mathbb{A}$ is $|\mathbb{A}| = \sum_{(A \to w) \in P} |w|$. We denote by $g(w)$ the size of a smallest SLP producing the word $w \in \Sigma^+$. The *smallest grammar problem* is the problem of computing an SLP of size $g(w)$ for a given input w.

Example 3.1. *Consider the SLP* $\mathbb{A} = (\{S, X_1, X_2\}, \{a, b\}, P, S)$ *with rules* $P = \{S \to X_2X_1bX_1a,\ X_1 \to bbX_2aX_2,\ X_2 \to ab\}$. *We have* $|\mathbb{A}| = 12$ *and*

- $\mathsf{val}(X_2) = ab$,
- $\mathsf{val}(X_1) = bb \cdot \mathsf{val}(X_2) \cdot a \cdot \mathsf{val}(X_2) = bb\ ab\ a\ ab$,
- $\mathsf{val}(S) = \mathsf{val}(X_2) \cdot \mathsf{val}(X_1) \cdot b \cdot \mathsf{val}(X_1) \cdot a = ab\ bbabaab\ b\ bbabaab\ a$.

It follows that $\mathsf{val}(\mathbb{A}) = \mathsf{val}(S) = abbbabaabbbbabaaba$ *and* $|\mathsf{val}(\mathbb{A})| = 18$.

3.1.1 Basic results

In the following we present basic results about SLPs that we use several times throughout this chapter. It is easy to see that $g(w) \leq |w|$ since for each word w there is the trivial SLP for w that has a single rule $S \to w$. The following theorem gives a better bound on the worst-case size of an SLP for a word $w \in \Sigma^+$.

Theorem 3.1.1 (E.g. [13, 34, 74])**.** *For every word* $w \in \Sigma^+$ *of length* n*, we have*

$$g(w) \leq \mathcal{O}\left(\frac{n}{\log_{|\Sigma|} n}\right).$$

A proof of Theorem 3.1.1 is provided in Section 3.3.3, where we show that the BiSection algorithm constructs an SLP of the claimed size (a proof of this result in a more general framework is provided in [74]). In fact, any algorithm studied in this chapter achieves this upper bound. For LZ78 (Section 3.4), this result is well known (see e.g. [34, Lemma 12.10.1]). For any global algorithm (Section 3.5), this upper bound follows from [72, Lemma 2] and [29, Lemma 7]. In [72, Lemma 2] it is shown that any so-called irreducible SLP for a word of length n has size $\mathcal{O}(n/\log_{|\Sigma|} n)$ and in [29, Lemma 7] it is shown that SLPs produced by global algorithms are irreducible.

On the other hand, explicit examples of strings for which the smallest SLP has size $\Omega(n/\log n)$ were constructed in [6, 13, 99]. The following lemma from [29] provides a lower bound on the size of an SLP for a word of length n.

Lemma 3.1.2 ([29, Lemma 1])**.** *For every word $w \in \Sigma^+$ of length n, we have $g(w) \geq 3\log_3(n) - 3$.*

Proof. Let $\mathbb{A} = (N, \Sigma, P, S)$ be an SLP of size $|\mathbb{A}| = m$. For a nonterminal A, let $M_A = \{X \in N \mid (A \to w) \in P \text{ and } X \text{ occurs in } w\}$. We define a sequence of pairwise different nonterminals such that

(i) $X_1 = S$,

(ii) if $|M_{X_i}| > 0$, then $X_{i+1} \in M_{X_i}$ such that $|\mathsf{val}(X_{i+1})|$ is maximal, i.e., $|\mathsf{val}(X_{i+1})| = \max\{|\mathsf{val}(X)| \mid X \in M_{X_i}\}$

(iii) if otherwise $M_{X_i} = \emptyset$, then the sequence ends with X_i.

If according to point (ii) more than one nonterminal produces a word of maximal length, we choose an arbitrary nonterminal with this property. Let ℓ be the length of this sequence. For $i \in [1, \ell]$, we set $k_i = |w_i|$ where $(X_i \to w_i) \in P$. We have $|\mathsf{val}(X_i)| \leq k_i \cdot |\mathsf{val}(X_{i+1})|$ due to the fact that X_{i+1} evaluates to a longest word among all symbols that occur on the right-hand side of the rule of X_i. It follows that $|\mathsf{val}(X_1)| \leq \prod_{i=1}^{\ell} k_i$ by an inductive argument. On the other hand, $\sum_{i=1}^{\ell} k_i \leq m$ because k_i is the number of symbols on the right-hand side of the rule of X_i and the nonterminals in the sequence are pairwise different. It is well known that a set of positive integers with sum at most m has product at most $3^{\lceil m/3 \rceil}$. It follows that

$$n = |\mathsf{val}(\mathbb{A})| = |\mathsf{val}(\mathbb{X}_1)| \leq 3^{\lceil \frac{m}{3} \rceil} \leq 3^{\frac{m}{3}+1}.$$

This yields $m = |\mathbb{A}| \geq 3\log_3(n) - 3$ for every SLP $\mathbb{A}$ which produces a word of length n. □

For unary words, i.e., words of the form a^n for some symbol $a \in \Sigma$, one can always find an SLP that almost matches this lower bound as the following lemma from [29] shows.

Lemma 3.1.3 ([29, Proof of Theorem 11])**.** *For every unary word $w \in \Sigma^+$ of length n, we have $g(w) \leq 3\log_3(n) + o(\log n)$.*

Proof. We construct an SLP $\mathbb{A}$ for a^n of size $|\mathbb{A}| \leq 3\log_3(n) + o(\log n)$. Consider the representation of n as a numeral in base $b = 3^j$ for some $j \in \mathbb{N}$ that we will define later. Let $t = \lfloor \log_b(n) \rfloor$. We have

$$n = \sum_{i=0}^{t} c_i \cdot b^i,$$

where $c_i \in [0, b-1]$ for $i \in [0, t]$. First, we create rules $T_1 \to a$ and $T_i \to T_{i-1}a$ for $i \in [2, b-1]$, i.e., $\mathsf{val}(T_i) = a^i$ for $i \in [1, b-1]$. In the following, we create nonterminals U_k for $k \in [0, t]$ such that

$$|\mathsf{val}(U_t)| = c_t \text{ and } |\mathsf{val}(U_i)| = |\mathsf{val}(U_{i+1})| \cdot b + c_i \text{ for } i \in [0, t-1].$$

Note that this yields $|\mathsf{val}(U_0)| = \sum_{i=0}^{t} c_i \cdot b^i = n$ by an inductive argument. We start with U_t, where we only need to introduce the rule $U_t \to T_{c_t}$. For U_i with $|\mathsf{val}(U_i)| = |\mathsf{val}(U_{i+1})| \cdot 3^j + c_i$, we define rules

$$\begin{aligned}
&Z_{i,1} \to U_{i+1}U_{i+1}U_{i+1},\\
&Z_{i,k} \to Z_{i,k-1}Z_{i,k-1}Z_{i,k-1} \text{ for } k \in [2, j] \text{ and}\\
&U_i \to Z_j T_{c_i}.
\end{aligned}$$

The start nonterminal of $\mathbb{A}$ is U_0. Recall that $b = 3^j$ which yields

$$\begin{aligned}
|\mathbb{A}| = (3j+2) \cdot t + 2 \cdot 3^j &\leq (3j+2) \cdot \log_b n + 2 \cdot 3^j\\
&= (3j+2) \cdot \frac{\log_3 n}{j} + 2 \cdot 3^j\\
&= 3\log_3 n + \frac{2}{j}\log_3 n + 2 \cdot 3^j.
\end{aligned}$$

Setting $j = \lfloor (1/2)\log_3 \log_3 n \rfloor$ yields

$$|\mathbb{A}| \leq 3\log_3 n + 2\sqrt{\log_3 n} + \frac{4\log_3 n}{\log_3 \log_3 n},$$

which finishes the proof. □

The following lemma relates the size $g(w)$ of a smallest SLP to the number of distinct factors of a certain length occurring in the word w. It has been shown in [29], where it is called *mk* Lemma.

Lemma 3.1.4 ([29, Lemma 3])**.** *Let $\mathbb{A}$ be an SLP with $\mathsf{val}(\mathbb{A}) = w \in \Sigma^+$ and $|\mathbb{A}| = m$. Then w contains at most $m \cdot k$ distinct factors of length k.*

Proof. Let $\mathbb{A} = (N, \Sigma, P, S)$ with $|\mathbb{A}| = m$ and $\mathsf{val}(\mathbb{A}) = w$. For $A \in N$, we define $M_A = \{X \in N \mid (A \to u) \in P \text{ and } X \text{ occurs in } u\}$. For a rule $(A \to u) \in P$, we bound the number of factors of length k of $\mathsf{val}(A)$ which are not already a factor of $\mathsf{val}(B)$ for some $B \in M_A$. Each such factor either begins with a symbol $a \in \Sigma$ that occurs in u or else begins with a suffix of $\mathsf{val}(B)$ for some $B \in M_A$ of length between 1 and $k-1$. Hence there are at most $|u| \cdot k$ such length-k factors in $\mathsf{val}(A)$. Summing over all rules in P yields $m \cdot k$ as an upper bound on the factors of w of length k. □

When we apply Lemma 3.1.4, we often use it in the following way: If a word w has ℓ distinct factors of length k, then any SLP for w has size at least ℓ/k. Before we move on, we need one more lemma that summarizes some straightforward results about SLPs.

Lemma 3.1.5.

(i) For an SLP $\mathbb{A}$ and a natural number $n > 0$, there exists an SLP $\mathbb{B}$ of size $|\mathbb{A}| + \mathcal{O}(\log n)$ such that $\mathsf{val}(\mathbb{B}) = \mathsf{val}(\mathbb{A})^n$.

(ii) For SLPs $\mathbb{A}_1$ and $\mathbb{A}_2$ there exists an SLP $\mathbb{B}$ of size $|\mathbb{A}_1| + |\mathbb{A}_2|$ such that $\mathsf{val}(\mathbb{B}) = \mathsf{val}(\mathbb{A}_1)\mathsf{val}(\mathbb{A}_2)$.

(iii) For given words $w_1, \ldots, w_n \in \Sigma^*$, $u \in \Sigma^+$ and SLPs $\mathbb{A}_1, \mathbb{A}_2$ such that $\mathsf{val}(\mathbb{A}_1) = u$ and $\mathsf{val}(\mathbb{A}_2) = w_1 x w_2 x \cdots w_{n-1} x w_n$ for a symbol $x \notin \Sigma$, there exists an SLP $\mathbb{B}$ with $\mathsf{val}(\mathbb{B}) = w_1 u w_2 u \cdots w_{n-1} u w_n$ and $|B| = |\mathbb{A}_1| + |\mathbb{A}_2|$.

Proof. We start with point (i). We extend the SLP $\mathbb{A}$ to an SLP $\mathbb{B}$ such that $\mathsf{val}(\mathbb{B}) = \mathsf{val}(\mathbb{A})^n$. Let A_0 be the start nonterminal of $\mathbb{A}$ and $m = \lfloor \log n \rfloor$. Using fresh nonterminals $A_1, \ldots, A_m$, we introduce rules $A_i \to A_{i-1}A_{i-1}$ for $i \in [1, m]$. It follows that $\mathsf{val}(A_i) = \mathsf{val}(\mathbb{A})^{2^i}$. We now construct $\mathsf{val}(\mathbb{A})^n$ using the binary representation of $n = \sum_{i=0}^{m} b_i \cdot 2^i$, where $b_i \in \{0, 1\}$ for $i \in [0, m]$. The fresh start rule of the SLP $\mathbb{B}$ concatenates A_i's where $b_i = 1$, i.e., $S \to A_0^{b_0} A_1^{b_i} \cdots A_m^{b_m}$. We have $\mathsf{val}(S) = \mathsf{val}(\mathbb{A})^n$ and $|\mathbb{B}| \leq |\mathbb{A}| + 3 \log n$.

The SLP $\mathbb{B}$ in point (ii) is achieved by adding a rule $S \to A_1 A_2$ to the rules in $\mathbb{A}_1$ and $\mathbb{A}_2$, where A_i is the start nonterminal of $\mathbb{A}_i$ ($i \in [1, 2]$). Note that we assume that the sets of nonterminals of $\mathbb{A}_1$ and $\mathbb{A}_2$ are disjoint.

The proof of point (iii) is also straightforward: Simply replace in the SLP $\mathbb{A}_2$ every occurrence of the symbol x by the start nonterminal of $\mathbb{A}_1$ and add all rules in $\mathbb{A}_1$ to $\mathbb{A}_2$, where we assume again that the nonterminals of $\mathbb{A}_1$ and $\mathbb{A}_2$ are disjoint. □

As a first minor result, we show that there are words of length $2k^2 + 2k + 1$ over an alphabet of size k for which the size of a smallest SLP equals the word length. Additionally, we show that all longer words have strictly smaller SLPs.

Proposition 3.1.6. *Let Σ_k be an alphabet of size k and let $n_k = 2k^2 + 2k + 1$. Then for all $k > 0$, there exists $w_k \in \Sigma_k^*$ of length n_k such that $g(w_k) = |w_k| = n_k$. Further, for each word $w \in \Sigma_k^*$ of length $|w| > n_k$, we have $g(w) < |w|$.*

Proof. Let $\Sigma_k = \{a_1, \ldots, a_k\}$ and let $M_{n,\ell} \subseteq \Sigma_k^*$ be the set of all words $w \in \Sigma_k^*$ where a factor v of length ℓ occurs at least n times without overlap. It is easy to see that $g(w) < |w|$ if and only if $w \in M_{3,2} \cup M_{2,3}$. Hence, we have to show that every word $w \notin M_{3,2} \cup M_{2,3}$ has length at most $2k^2 + 2k + 1$. Moreover, we present words $w_k \in \Sigma_k^*$ of length $2k^2 + 2k + 1$ such that $w_k \notin M_{3,2} \cup M_{2,3}$.

Assume that $w \notin M_{3,2} \cup M_{2,3}$ and consider a factor $a_i a_j$ of length two, where $i, j \in [1, k]$. If $i \neq j$ then this factor can not overlap itself, and thus $a_i a_j$ occurs at most twice in w. Now consider $a_i a_i$. Then w contains at most

four (possibly overlapping) occurrence of $a_i a_i$, because (i) $a_i^5 \notin M_{3,2} \cup M_{2,3}$ has four (overlapping) occurrences of $a_i a_i$ and (ii) five occurrences of $a_i a_i$ would yield at least three non-overlapping occurrences of $a_i a_i$. It follows that w has at most $2(k^2 - k) + 4k$ positions where a factor of length 2 starts, which implies $|w| \leq 2k^2 + 2k + 1$.

Now we create a word $w_k \notin M_{3,2} \cup M_{2,3}$ which realizes the maximal occurrences of factors of length 2 as stated above:

$$w_k = \left(\prod_{i=1}^{k} a_{k-i+1}^5 \right) \prod_{i=1}^{k-1} \left(\prod_{j=i+2}^{k} (a_j a_i)^2 \right) a_{i+1} a_i a_{i+1}$$

For example we have $w_3 = a_3^5 a_2^5 a_1^5 (a_3 a_1)^2 a_2 a_1 a_2 a_3 a_2 a_3$. It is straightforward to verify $|w_k| = 2k^2 + 2k + 1$ and $w_k \notin M_{3,2} \cup M_{2,3}$. □

3.2 Approximation ratio

A grammar-based compressor $\mathcal{C}$ computes for a word w an SLP $\mathcal{C}(w)$ such that $\mathsf{val}(\mathcal{C}(w)) = w$. The *approximation ratio* $\alpha_{\mathcal{C}}(w)$ of $\mathcal{C}$ for an input w is defined as $|\mathcal{C}(w)|/g(w)$. The *worst-case approximation ratio* $\alpha_{\mathcal{C}}(k, n)$ of $\mathcal{C}$ is the maximal approximation ratio over all words of length n over an alphabet of size k:

$$\alpha_{\mathcal{C}}(k, n) = \max\{\alpha_{\mathcal{C}}(w) \mid w \in [1, k]^n\} = \max\left\{ \frac{|\mathcal{C}(w)|}{g(w)} \mid w \in [1, k]^n \right\}$$

If there is no restriction on the alphabet size, i.e., we allow alphabets of size $|w|$, then we write $\alpha_{\mathcal{C}}(n)$ instead of $\alpha_{\mathcal{C}}(n, n)$. Note that this is the definition of the worst-case approximation ratio in [29]. We study the approximation ratios of the grammar-based compressors BiSection [74], LZ78 [113], RePair [77], Greedy [7, 8] and LongestMatch[72] in this work. We will abbreviate the approximation ratio of BiSection by α_{BS} and the approximation ratio of LongestMatch by α_{LM}. For each algorithm, we distinguish between the approximation ratio $\alpha_{\mathcal{C}}(1, n)$ restricted to unary inputs and the general approximation ratio $\alpha_{\mathcal{C}}(k, n)$ for $k \geq 1$.

Before we start to analyze those algorithms, we discuss two aspects of the general difficulty of approximating a smallest grammar. First, we relate grammar-based compression over binary alphabets to grammar-based compression over unbounded alphabets. Afterwards, we describe the relations between grammar-based compression and the well-studied field of addition chains. The latter one is of special interest when unary inputs are considered.

3.2.1 Hardness for binary alphabets

In this section, we discuss the hardness of the smallest grammar problem for words over a binary alphabet. Recall that it is not possible to compute an SLP of size $(8569/8568) \cdot g(w)$ for a given input word w over an alphabet of size at least 24 unless P = NP [28]. It is far from clear whether this approach can be

adapted such that it works also for a binary alphabet. Another idea in order to obtain the hardness result for an alphabet of size 2 is to reduce the smallest grammar problem for arbitrary alphabets to the smallest grammar problem for a binary alphabet. This route was investigated in [10], where the following result was shown: If there is a polynomial time grammar-based compressor with approximation ratio c (a constant) on binary words, then there is a polynomial time grammar-based compressor with approximation ratio $24c + \varepsilon$ for every $\varepsilon > 0$ on arbitrary words. The construction in [10] uses a quite technical block encoding of arbitrary alphabets into a binary alphabet. Here, we present a very simple construction, which encodes the i-th alphabet symbol by $a^i b$ and yields the same result as [10] but with $24c + \varepsilon$ replaced by $6c$. Formally, the goal of this section is to prove the following result:

Theorem 3.2.1. *Let $c \geq 1$ be a constant. If there exists a polynomial time grammar-based compressor $\mathcal{C}$ with $\alpha_{\mathcal{C}}(2, n) \leq c$ then there exists a polynomial time grammar-based compressor $\mathcal{D}$ with $\alpha_{\mathcal{D}}(n) \leq 6c$.*

We split the proof of Theorem 3.2.1 into two lemmas that state translations between SLPs over arbitrary alphabets and SLPs over a binary alphabet. For the rest of this section fix the alphabets $\Sigma = \{c_0, \ldots, c_{k-1}\}$ and $\Sigma_2 = \{a, b\}$. To translate between these two alphabets, we define an injective homomorphism $\varphi\colon \Sigma^* \to \Sigma_2^*$ by

$$\varphi(c_i) = a^i b \quad (i \in [0, k-1]). \tag{3.1}$$

Lemma 3.2.2. *Let $w \in \Sigma^*$ be such that every symbol from Σ occurs in w. From an SLP $\mathbb{A}$ for w one can construct in polynomial time an SLP $\mathbb{B}$ for $\varphi(w)$ of size at most $3 \cdot |\mathbb{A}|$.*

Proof. To translate $\mathbb{A}$ into an SLP $\mathbb{B}$ for $\varphi(w)$, we first add the productions $A_0 \to b$ and $A_i \to aA_{i-1}$ for every $i \in [1, k-1]$. Finally, we replace in $\mathbb{A}$ every occurrence of $c_i \in \Sigma$ by A_i. This yields an SLP $\mathbb{B}$ for $\varphi(w)$ of size $|\mathbb{A}| + 2k - 1$. Because $k \leq |\mathbb{A}|$ (since every symbol from Σ occurs in w), we obtain $|\mathbb{B}| \leq 3 \cdot |\mathbb{A}|$. □

Lemma 3.2.3. *Let $w \in \Sigma^*$ such that every symbol from Σ occurs in w. From an SLP $\mathbb{B}$ for $\varphi(w)$ one can construct in polynomial time an SLP $\mathbb{A}$ for w of size at most $2 \cdot |\mathbb{B}|$.*

Proof. Let $\mathbb{B} = (N, \Sigma_2, P, S)$ be an SLP for $\varphi(w)$, where $w \in \Sigma^*$. We can assume that every right-hand side of $\mathbb{B}$ is a non-empty string. Consider a nonterminal $A \in N$ of $\mathbb{B}$. Since $\mathbb{B}$ produces $\varphi(w)$, A produces a factor of $\varphi(w)$, which is a word from $\{a, b\}^*$. We cannot directly translate $\mathsf{val}(A)$ back to a word over Σ^* because $\mathsf{val}(A)$ does not have to belong to the image of φ. But $\mathsf{val}(A)$ is a factor of a string from $\varphi(\Sigma^*)$. Note that a string over $\{a, b\}$ is a factor of a string from $\varphi(\Sigma^*)$ if and only if it does not contain a factor a^i with $i \geq k$. Let $\mathsf{val}(A) = a^{i_1} b \cdots a^{i_n} b a^{i_{n+1}}$ be such a string, where $n \geq 0$, and $0 \leq i_1, \ldots, i_{n+1} < k$. We factorize $\mathsf{val}(A)$ into three parts in the following way. If $n = 0$ (i.e., $\mathsf{val}(A) = a^{i_1}$) then we split $\mathsf{val}(A)$ into ε, ε, and a^{i_1}. If $n > 0$ then we split $\mathsf{val}(A)$ into $a^{i_1} b$, $a^{i_2} b \cdots a^{i_n} b$, and $a^{i_{n+1}}$.

Let us explain the intuition behind this factorization. We concentrate on the case $n > 0$; the case $n = 0$ is simpler. Note that irrespective of the context in which an occurrence of $\mathsf{val}(A)$ appears in $\mathsf{val}(\mathbb{B})$, we can translate the middle part $a^{i_2}b \cdots a^{i_n}b$ into $c_{i_2} \cdots c_{i_n}$. We will therefore introduce in the SLP $\mathbb{A}$ for w a variable A' that produces $c_{i_2} \cdots c_{i_n}$. For the left part $a^{i_1}b$ we can not directly produce c_{i_1} because an occurrence of $\mathsf{val}(A)$ could be preceded by an a-block a^{i_0}, yielding the symbol $c_{i_0+i_1}$. Therefore, the algorithm that produces $\mathbb{A}$ will only memorize the symbol c_{i_1} without writing it directly on a right-hand side of an $\mathbb{A}$-production. Similarly, the algorithm will memorize the length i_{n+1} of the final a-block of $\mathsf{val}(A)$.

Let us now come to the formal details of the proof. As usual, we write $\mathbb{Z}_k$ for $\{0, 1, \ldots, k-1\}$ and w.l.o.g. we assume that $\Sigma \cap \mathbb{Z}_k = \emptyset$. Consider a word $s = a^{i_1}b \cdots a^{i_n}ba^{i_{n+1}}$, where $n \geq 0$, and $0 \leq i_1, \ldots, i_{n+1} < k$. Motivated by the above discussion, we define $\ell(s) \in \Sigma \cup \{\varepsilon\}$, $m(s) \in \Sigma^*$ and $r(s) \in \mathbb{Z}_k$ as follows:

$$\ell(s) = \begin{cases} c_{i_1} & \text{if } n \geq 1, \\ \varepsilon & \text{if } n = 0, \end{cases}$$
$$m(s) = c_{i_2} \cdots c_{i_n},$$
$$r(s) = i_{n+1}.$$

Note that $\ell(s) = \varepsilon$ implies $m(s) = \varepsilon$. Finally, we define the word $\psi(s) \in \Sigma^*\mathbb{Z}_k$ as

$$\psi(s) = \ell(s)m(s)r(s).$$

For a nonterminal $A \in N$ we define $\ell(A) = \ell(\mathsf{val}(A))$, $m(A) = m(\mathsf{val}(A))$ and $r(A) = r(\mathsf{val}(A))$. We now define an SLP $\mathbb{A}'$ that contains for every nonterminal $A \in N$ a nonterminal A' such that $\mathsf{val}(A') = m(A)$. Moreover, the algorithm also computes $\ell(A) \in \Sigma \cup \{\varepsilon\}$ and $r(A) \in \mathbb{Z}_k$.

We define the productions of $\mathbb{A}'$ inductively over the structure of $\mathbb{B}$. Consider a production $(A \to \alpha) \in P$, where $\alpha = v_0A_1v_1A_2 \cdots v_{n-1}A_nv_n \neq \varepsilon$ with $n \geq 0$, $A_1, \ldots, A_n \in N$, and $v_0, v_1, \ldots, v_n \in \Sigma_2^*$. Let $\ell_i = \ell(A_i) \in \Sigma \cup \{\varepsilon\}$ and $r_i = r(A_i) \in \mathbb{Z}_k$, which have already been computed. The right-hand side for A' is obtained as follows. We start with the word

$$\psi(v_0)\,\ell_1\,A'_1\,r_1\,\psi(v_1)\,\ell_2\,A'_2\,r_2 \cdots \psi(v_{n-1})\,\ell_n\,A'_n\,r_n\,\psi(v_n). \tag{3.2}$$

Note that each of the factors $\ell_iA'_ir_i$ produces (by induction) $\psi(\mathsf{val}(A_i))$. Next we remove every A'_i that derives the empty word (which is equivalent to $m(A_i) = \varepsilon$). After this step, every occurrence of a symbol $i \in \mathbb{Z}_k$ in (3.2) is either the last symbol of the above word or it is followed by a symbol from $\mathbb{Z}_k \cup \Sigma$ (but not followed by a nonterminal A'_j). To see this, recall that $\ell_j = \varepsilon$ implies $m(A_j) = \varepsilon$, in which case A'_j is removed in (3.2).

The above fact allows us to eliminate all occurrences of symbols $i \in \mathbb{Z}_k$ in (3.2) except for the last one using the two reduction rules $i\,j \to i + j$ for $i, j \in \mathbb{Z}_k$ (which corresponds to $a^ia^j = a^{i+j}$) and $i\,c_j \to c_{i+j}$ (which corresponds to $a^ia^jb = a^{i+j}b$). If we perform these rules as long as possible (the order of

applications is not relevant since these rules form a confluent and terminating system), only a single occurrence of a symbol $i \in \mathbb{Z}_k$ at the end of the string will remain. The resulting string α' produces $\psi(A)$. Hence, we obtain the right-hand side for the nonterminal A' by removing the first symbol of α' if it is from Σ (this symbol is then $\ell(A)$) and the last symbol of α', which must be from $\mathbb{Z}_k$ (this symbol is $r(A)$). Note that if α' does not start with a symbol from Σ, then α' belongs to $\mathbb{Z}_k$, in which case we have $\ell(A) = \varepsilon$.

Note that $\psi(\varphi(w)) = w0$ for every $w \in \Sigma^*$, so for the start variable S of $\mathbb{B}$ we must have $r(S) = 0$, since $\mathsf{val}_{\mathbb{B}}(S) \in \varphi(\Sigma^*)$. Let $S' \to \sigma$ be the production for S' in $\mathbb{A}'$. We obtain the SLP $\mathbb{A}$ by replacing this production by $S' \to \ell(S)\sigma$. Since $\mathsf{val}_{\mathbb{A}'}(S') = m(S)$ and $\mathsf{val}_{\mathbb{B}}(S) = \varphi(w)$ we have $\mathsf{val}_{\mathbb{A}}(S') = \ell(S)m(S) = w$.

To bound the size of $\mathbb{A}$ consider the word in (3.2) from which the right-hand side of the nonterminal A' is computed. All occurrences of symbols from $\mathbb{Z}_k$ are eliminated when forming this right-hand side. This leaves a word of length at most $|\alpha| + n$ (where α is the original right-hand side of the nonterminal A). The additive term n comes from the symbols $\ell_1, \ldots, \ell_n$. Hence, $|\mathbb{A}'|$ is bounded by the size of $\mathbb{B}$ plus the total number of occurrences of nonterminals in right-hand sides of $\mathbb{B}$, which is at most $2|\mathbb{B}| - 1$ (there is at least one terminal occurrence in a right-hand side). Since $|\mathbb{A}| = |\mathbb{A}'| + 1$ we get $|\mathbb{A}| \leq 2|\mathbb{B}|$.

The algorithm's runtime for a production $A \to \alpha$ is linear in $|\alpha|$. This is because we start with the string (3.2) which can be computed in time $\mathcal{O}(|\alpha|)$. From this string, we remove all the A'_i that produce ε and we also apply the two rewriting rules. Both of these can be done in a single left-to-right sweep over the string. The number of operations needed is linear in $|\alpha|$, where each operation needs constant time, i.e. removing an A'_i takes constant time, and using one of the rewriting rules also takes constant time. Since the algorithm uses the structure of $\mathbb{B}$ to visit each of its productions once, we overall obtain a linear running time in the size of $\mathbb{B}$. □

Example 3.2. *Consider the production* $A \to a^3ba^5A_1a^3A_2a^2b^2A_3a^2$ *and assume that* $\mathsf{val}(A_1) = a^2$, $\mathsf{val}(A_2) = aba^3ba$ *and* $\mathsf{val}(A_3) = ba^2ba^3$. *Hence, when we produce the right-hand side for* A' *we have:* $\mathsf{val}(A'_1) = \varepsilon$, $\mathsf{val}(A'_2) = c_3$, $\mathsf{val}(A'_3) = c_2$, $\ell_1 = \varepsilon$, $r_1 = 2$, $\ell_2 = c_1$, $r_2 = 1$, $\ell_3 = c_0$, $r_3 = 3$. *We start with the word (every digit is a single symbol)*

$$c_3\,5\,A'_1\,2\,3\,c_1A'_2\,1\,c_2c_0\,0\,c_0A'_3\,3\,2.$$

Then we replace A'_1 *by* ε *and obtain* $c_3\,5\,2\,3\,c_1A'_2\,1\,c_2c_0\,0\,c_0A'_3\,3\,2$. *Applying the reduction rules finally yields* $c_3c_{11}A'_2c_3c_0c_0A'_3 5$. *Hence, we have* $\ell(A) = c_3$, $r(A) = 5$ *and the production for* A' *is* $A' \to c_{11}A'_2c_3c_0c_0A'_3$.

Proof of Theorem 3.2.1. Let $\mathcal{C}$ be an arbitrary grammar-based compressor working in polynomial time such that $\alpha_{\mathcal{C}}(2, n) \leq c$. The grammar-based compressor $\mathcal{D}$ works for an input word w over an arbitrary alphabet as follows: Let $\Sigma = \{c_0, \ldots, c_{k-1}\}$ be the set of symbols that occur in w and let φ be defined as in (3.1). Using $\mathcal{C}$, one first computes an SLP $\mathbb{B}$ for $\varphi(w)$ such that $|\mathbb{B}| \leq c \cdot g(\varphi(w))$. Then, using Lemma 3.2.3, one computes from $\mathbb{B}$ an SLP $\mathbb{A}$

for w such that $|\mathbb{A}| \leq 2c \cdot g(\varphi(w))$. Lemma 3.2.2 implies $g(\varphi(w)) \leq 3 \cdot g(w)$ and hence $|\mathbb{A}| \leq 6c \cdot g(w)$, which proves the theorem. □

3.2.2 Addition chains

When inputs are restricted to be unary, then grammar-based compression is strongly related to the field of addition chains as we will describe in the following. An *addition chain* of size m for an integer n is a sequence of integers $k_1, k_2, \ldots, k_m$ such that $k_1 = 1$, $k_m = n$ and for all $d \in [2, m]$ we have $k_d = k_i + k_j$ for some $i, j \in [1, d-1]$. The *addition chain problem* is to compute a smallest addition chain for a given integer n.

Example 3.3. *An addition chain for* $n = 22$ *is* $1, 2, 4, 8, 10, 11, 22$*, because* $1 + 1 = 2$*,* $2 + 2 = 4$*,* $4 + 4 = 8$*,* $2 + 8 = 10$*,* $1 + 10 = 11$*, and* $11 + 11 = 22$*.*

Let $\Sigma = \{a\}$ be unary alphabet, then an addition chain for n translates into an SLP for a^n and vice versa as the following propositions show.

Proposition 3.2.4. *If there is an addition chain for* n *of size* m*, then there is an SLP* $\mathbb{A}$ *for* a^n *of size at most* $2m - 2$*.*

Proof. Let $1 = k_1, k_2, \ldots, k_m = n$ be an addition chain for n. We introduce a nonterminal A_d for each k_d with $d \in [2, m]$. Let $h(1) = a$ and $h(d) = A_d$ for $d \in [2, m]$. We add for each $d \in [2, m]$ a rule $A_d \to h(i)h(j)$ where we choose $i, j \in [1, d-1]$ such that $k_d = k_i + k_j$. The start nonterminal is A_m. It follows that this SLP produces a^n since $k_1, k_2, \ldots, k_m$ is an addition chain for n. □

Proposition 3.2.5. *If there is an SLP* $\mathbb{A}$ *for* a^n *of size* m*, then there is an addition chain for* n *of size at most* m*.*

Proof. Let $\mathbb{A} = (N, \{a\}, P, S)$ be an SLP such that $\mathsf{val}(\mathbb{A}) = a^n$. We construct an addition chain $1 = k_1, \ldots, k_\ell = n$ such that $\ell \leq |\mathbb{A}|$. The first integer $k_1 = 1$ is always part of an addition chain. The remaining integers are created such that for each rule $(X \to w) \in P$, we add at most $|w|-1$ integers. Let $X_1 < \cdots < X_m = S$ be an order of the nonterminals in N such that the nonterminals $X_{i+1}, \ldots, X_m$ do not occur on the right-hand side of the rule $X_i \to w$ for all $i \in [1, m-1]$. Note that such an order exists since $\mathbb{A}$ is acyclic. We process the rules in this order, i.e., we start with the unique rule for X_1. We define $k_X \in \mathbb{N}$ for each $X \in N \cup \{a\}$ such that $k_a = 1$ and $k_X = |\mathsf{val}(X)|$ for $X \in N$. Now consider a rule $X \to w_1 \ldots w_t$, where $w_i \in N \cup \{a\}$ for $i \in [1, t]$. If $t = 1$, we have $k_X = k_{w_1}$ and we do nothing (k_{w_1} is added to the addition chain in a previous step). If otherwise $t \geq 2$, we add integers $k_{X,2}, \ldots, k_{X,t} = k_X$ to the addition chain, where $k_{X,2} = k_{w_1} + k_{w_2}$ and $k_{X,i} = k_{X,i-1} + k_{w_i}$ for $i \in [3, t]$. This means k_X is created by adding integers that correspond to symbols on the right-hand side and we do it symbol by symbol, see also Example 3.4. Note that in the base case (which occurs at least for X_1), we have a rule $X \to a^d$ and thus the integers $2, \ldots, d = k_X$ are added to the addition chain. At the end, we created an addition chain for $k_S = n$ inductively using $|w| - 1$ integers for each rule $(X \to w) \in P$ plus the integer $k_a = k_1 = 1$. This proves the proposition. □

Example 3.4. *Consider the SLP* $\mathbb{A} = (\{S, X_1, X_2\}, \{a\}, P, S)$ *such that* P *contains the rules* $X_1 \to aaa$, $X_2 \to X_1 a X_1 a$ *and* $S \to X_2 X_1$. *We have* $\mathsf{val}(\mathbb{A}) = a^{11}$. *We transform* $\mathbb{A}$ *into an addition chain for* 11 *based on the proof of Proposition 3.2.5. Initially the addition chain only contains the integer* $k_a = 1$.

- *We start with the rule* $X_1 \to aaa$*: We add* $k_{X_1,2} = k_a + k_a = 2$ *and* $k_{X_1} = k_{X_1,3} = k_{X_1,2} + k_a = 3$.
- *We proceed with* $X_2 \to X_1 a X_1 a$*: We add* $k_{X_2,2} = k_{X_1} + k_a = 4$, $k_{X_2,3} = k_{X_2,2} + k_{X_1} = 7$ *and* $k_{X_2} = k_{X_2,4} = k_{X_2,3} + k_a = 8$.
- *Finally, we process* $S \to X_2 X_1$*: We add* $k_S = k_{S,2} = k_{X_1} + k_{X_2} = 11$.

The final addition chain is $1, 2, 3, 4, 7, 8, 11$.

As a consequence of those propositions, we can adapt the approximation ratio $\alpha_{\mathcal{C}}(1, n)$ of a grammar-based compressor $\mathcal{C}$ on unary inputs to the problem of approximating a smallest addition chain for n using the constants described in the propositions above.

In the remaining section we describe a second connection between grammar-based compression and addition chains. Addition chains have been generalized to a set of integers $n_1, n_2, \ldots, n_p$ as follows: An addition chain of size m for $n_1, n_2, \ldots, n_p$ is a sequence of integers $k_1, k_2, \ldots, k_m$ such that $k_1 = 1$, $\{n_1, n_2, \ldots, n_p\} \subseteq \{k_1, k_2, \ldots, k_m\}$ and for all $d \in [2, k]$ we have $k_d = k_i + k_j$ for some $i, j \in [1, d-1]$. The *general addition chain problem* is to compute a smallest addition chain for integers $n_1, n_2, \ldots, n_p$. For example, the addition chain in Example 3.3 is also an addition chain for $4, 10, 22$. The connection to grammar-based compression is described in the following proposition, where we use an alphabet $\Sigma = \{a, b_1, \ldots, b_{k-1}\}$ of size k. The proof idea is similar to the proofs for the propositions above and the reader can find a detailed proof in [29].

Proposition 3.2.6. *[29, Theorem 2] If there is an addition chain for the integers* $n_1, \ldots, n_k$ *of size* m*, then there is an SLP* $\mathbb{A}$ *for* $a^{n_1} b_1 a^{n_2} \cdots b_{k-1} a^{n_k} = (\prod_{i=1}^{k-1} a^{n_i} b_i) a^{n_k}$ *of size at most* $4m$*. If there is an SLP* $\mathbb{A}$ *for* $(\prod_{i=1}^{k-1} a^{n_i} b_i) a^{n_k}$ *of size* m*, then there is an addition chain of size at most* m *for* $n_1, \ldots, n_k$.

Even though addition chains have been studied for decades, there is no polynomial time algorithm that computes for integers $n_1, \ldots, n_k$ a general addition chain of size $m^* \cdot o(\log N / \log\log N)$, where $N = \max\{n_i \mid i \in [1, k]\}$ and m^* is the size of a smallest addition chain for $n_1, \ldots, n_k$. In particular, the best approximation ratio for the general addition chain problem is $\mathcal{O}(\log N / \log\log N)$ [110]. By Proposition 3.2.6, it follows that a polynomial time grammar-based compressor $\mathcal{C}$ with $\alpha_{\mathcal{C}}(n) \in o(\log n / \log\log n)$ would improve upon this more than 30 year old result. In Section 3.6.3, we present a general addition chain solver based on RePair [77] that matches the best known approximation ratio $\mathcal{O}(\log N / \log\log N)$.

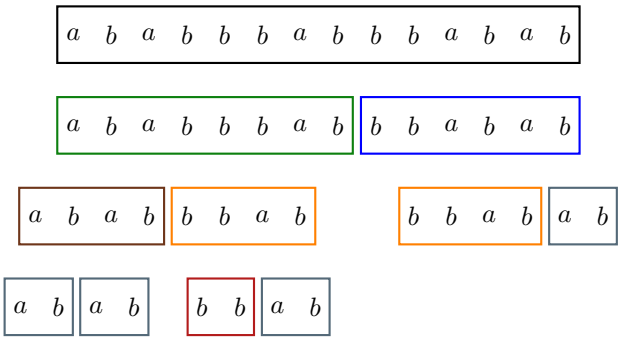

Figure 3.1: The splits of BiSection on input *ababbbaabbabab* are depicted. The top line shows the input. The second line illustrates the factors obtained by the first split. The third line shows the factors that occur when those factors are split again, and so on. A colored box refers to a factor obtained by a split and if a factor occurs multiple times, then the boxes share the same color. A factor is not split again if either it has length at most 2 or the factor occurs before.

3.3 BiSection

The BiSection algorithm [74] first splits an input word w with $|w| \geq 2$ as $w = w_1 w_2$ such that $|w_1| = 2^j$ for the unique number j with $2^j < |w| \leq 2^{j+1}$. This process is recursively repeated for the obtained factors w_1 and w_2 until we obtain words of length 1. During the process, we introduce a nonterminal for each distinct factor of length at least two and create a rule with two symbols on the right-hand side corresponding to the split. If a factor $a \in \Sigma$ of length 1 is obtained at some point, then we simply represent it by the symbol a in the corresponding rule.

Example 3.5. *We construct the SLP that is obtained by* BiSection *on input* $w = ababbbabbbabab$. *We use different colors for different factors obtained at some point of the algorithm and multiple occurrences of a factor are colored the same. Further, each nonterminal is colored the same as the corresponding factor obtained during a split of the algorithm. In Figure 3.1, the example is illustrated using the same coloring.*

(1) $w = w_1 w_2$ *with* $w_1 = ababbbab$, $w_2 = bbabab$
Introduced rule: $S \to W_1 W_2$

(2) $w_1 = x_1 x_2$ *with* $x_1 = abab$, $x_2 = bbab$, *and* $w_2 = x_2 x_3$ *with* $x_3 = ab$
Introduced rules: $W_1 \to X_1 X_2$, $W_2 \to X_2 X_3$, $X_3 \to ab$

(3) $x_1 = x_3 x_3$, $x_2 = y x_3$ *with* $y = bb$
Introduced rules: $X_1 \to X_3 X_3$, $X_2 \to Y X_3$, $Y \to bb$

3.3.1 Unary inputs

We start our analysis of BiSection by studying the approximation ratio $\alpha_{\mathsf{BS}}(1, n)$, i.e., we consider unary inputs first. To do so, recall that $\nu(n)$ denotes the number of 1's in the binary representation of n, see equation (2.2) for the formal definition.

Proposition 3.3.1. *For all $n \geq 3$, we have*

$$|\mathsf{BiSection}(a^n)| = 2(\lfloor \log n \rfloor + \nu(n) - 1).$$

Proof. Let n_1, n_2 be the integers representing the first split of BiSection on input a^n, i.e., $a^n = a^{n_1}a^{n_2}$. If $n = 2^k$ for some $k \geq 1$, then we have $n_1 = n_2 = 2^{k-1}$ and a rule $S \to XX$, otherwise we have $n_1 = 2^{\lfloor \log n \rfloor}$, $n_2 = n - n_1$. In both cases, n_1 is a power of two and thus the algorithm recursively splits a factor of the form a^{2^i} into $a^{2^{i-1}}$ twice for $i \in [2, \log n_1]$. The procedure stops when factors of length 2 are reached. This results in rules $X_i \to X_{i-1}X_{i-1}$ for $i \in [2, \log n_1]$ and $X_1 \to aa$, where $X = X_{\log n_1}$ represents the factor a^{n_1} in the start rule. If $n = 2^k$, then the algorithm stops at this point and we have $|\mathsf{BiSection}(a^n)| = 2k = 2(\lfloor \log n \rfloor + \nu(n) - 1)$ since $\nu(2^k) = 1$ for each $k \geq 0$. Otherwise the size of the rules obtained so far is $2\lfloor \log n \rfloor$ and the algorithm proceeds with the factor a^{n_2}. If $n_2 = 1$, then the algorithm stops as well and the start rule is $S \to Xa$. The produced SLP has size $2\lfloor \log n \rfloor + 2$, which matches the claimed result because $\nu(2^k + 1) = 2$. If otherwise $n_2 = n - n_1 \in [2, n_1 - 1]$, then the algorithm proceeds for a^{n_2} in the same manner as it is described above, but we represent each occurring factor of length 2^i (for some $i \in [1, \lfloor \log n \rfloor - 1]$) by the already existing nonterminal X_i. In particular, each left factor obtained by a split is represented by a nonterminal that already exists. So the remaining number of splits only depends on the right factors. Note that the binary representation of n_2 is achieved from the binary representation of n by inverting the most significant bit $b_{\lfloor \log n \rfloor}$ from 1 to 0 and then deleting the leading zeros. We have $\nu(n_2) = \nu(n) - 1$. If n_2 is not a power of two, then the length of the right factor obtained by splitting a^{n_2} is again achieved by inverting the most significant bit of n_2 and so on. This procedure is repeated until the binary representation of the length n' of a right factor satisfies $\nu(n') = 1$ and thus n' is a power of two. The latest possible point where this happens is when only the least significant bit remains after inverting all other 1's in the binary representation of n_2. Hence there are $\nu(n_2) = \nu(n) - 1$ splits for a^{n_2}, where each split adds a rule of length two. This gives the total size $|\mathsf{BiSection}(a^n)| = 2\lfloor \log n \rfloor + 2(\nu(n) - 1)$. □

Theorem 3.3.2. *For all n, we have*

$$\alpha_{\mathsf{BS}}(1, n) \leq \frac{4}{3} \log(3).$$

Proof. If $n = 1$ or $n = 2$, then BiSection produces on input a^n the trivial SLP containing the single rule $S \to a^n$, which is also optimal in this case. If otherwise $n \geq 3$, then it follows from Proposition 3.3.1 that

$$|\mathsf{BiSection}(a^n)| = 2(\lfloor \log n \rfloor + \nu(n) - 1) \leq 4 \log n$$

since $\nu(n) - 1 \leq \log n$. By Lemma 3.1.2, we have $g(a^n) \geq 3\log_3 n - 3$, which together with $(4\log n)/(3\log_3 n) = (4/3)\log(3)$ finishes the proof. □

Theorem 3.3.3. *For infinitely many n, we have*

$$\alpha_{\mathsf{BS}}(1, n) \geq \frac{4}{3}\log(3).$$

Proof. Let $s_k = a^{2^k-1}$ for $k \geq 2$. We have $\lfloor \log |s_k| \rfloor = k - 1$ and $\nu(|s_k|) = k$. By Proposition 3.3.1 it follows that

$$|\mathsf{BiSection}(s_k)| = 4k - 4.$$

By Lemma 3.1.3, we have

$$g(s_k) \leq 3\log_3(2^k - 1) + o(\log(2^k - 1)) \leq 3\log_3(2) \cdot k + o(k).$$

The equality $4/(3\log_3(2)) = (4/3)\log(3)$ finishes the proof. □

3.3.2 General case

The following upper bound on the approximation ratio of BiSection for arbitrary alphabets is shown in [29]:

Theorem 3.3.4 ([29, Theorem 6])**.** *For all n, we have*

$$\alpha_{\mathsf{BS}}(n) \leq \mathcal{O}\left(\sqrt{\frac{n}{\log n}}\right).$$

Further, it is shown in [29, Theorem 5] that $\alpha_{\mathsf{BS}}(2, n) \geq \Omega(\sqrt{n}/\log n)$ for infinitely many n. We improve this lower bound such that it matches the upper bound. Note that if $w = u_1 u_2 \cdots u_k$ with $|u_i| = 2^n$ for some n and for all $i \in [1, k]$, then the SLP produced by BiSection contains at least $|\{u_i \mid i \in [1, k]\}|$ many different nonterminals, because each u_i is the result of a split at some point of the algorithm.

Theorem 3.3.5. *For all $k \geq 2$ and infinitely many n, we have*

$$\alpha_{\mathsf{BS}}(k, n) \geq \Omega\left(\sqrt{\frac{n}{\log n}}\right).$$

Proof. We first show that $\alpha_{\mathsf{BS}}(3, n) \geq \Omega(\sqrt{n/\log n})$. In a second step, we encode a ternary alphabet into a binary alphabet while preserving the approximation ratio.

For every $k \geq 2$, we define a function $\mathrm{bin}_k : [0, k-1] \to \{0,1\}^{\lceil \log k \rceil}$ such that $\mathrm{bin}_k(j)$ is the binary representation of j padded with leading zeros for $j \in [0, k-1]$ (e.g. $\mathrm{bin}_9(3) = 0011$). We further define for every $k \geq 2$ the word

$$u_k = \left(\prod_{j=0}^{k-2} \mathrm{bin}_k(j) a^{m_k}\right) \mathrm{bin}_k(k-1),$$

where $m_k = 2^{k-\lceil \log k \rceil} - \lceil \log k \rceil$. For instance $k = 4$ leads to $m_k = 2$ and $u_4 = 00aa01aa10aa11$. We analyze the approximation ratio $\alpha_{\mathsf{BS}}(s_k)$ for the word

$$s_k = \left(u_k a^{m_k+1}\right)^{m_k} u_k.$$

Claim 1. $|\mathsf{BiSection}(s_k)| \geq \Omega(2^k)$

Proof. If s_k is factorized into consecutive, non-overlapping factors of length $m_k + \lceil \log k \rceil = 2^{k-\lceil \log k \rceil}$, then all resulting factors are pairwise different and the set F_k which contains all these factors is

$$F_k = \{a^i \mathrm{bin}_k(j) a^{m_k - i} \mid j \in [0, k-1],\ i \in [0, m_k]\}.$$

For example s_4 consecutively consists of the factors $00aa$, $01aa$, $10aa$, $11aa$, $a00a$, $a01a$, $a10a$, $a11a$, $aa00$, $aa01$, $aa10$ and $aa11$. We have $|F_k| = (m_k+1)\cdot k = \Theta(2^k)$, because $m_k = \Theta(2^k/k)$. It follows that the SLP produced by $\mathsf{BiSection}$ on input s_k has size $\Omega(2^k)$, because all factors in F_k have the same length $2^{k-\lceil \log k \rceil}$ and thus $\mathsf{BiSection}$ creates a nonterminal for each distinct factor in F_k.

Claim 2. $g(s_k) \leq \mathcal{O}(k)$

Proof. There is an SLP of size $\mathcal{O}(\log m_k) = \mathcal{O}(k)$ for the word a^{m_k} by Lemma 3.1.5, point (i). This yields an SLP for u_k of size $\mathcal{O}(k) + g(u'_k)$ by Lemma 3.1.5, point (iii), where $u'_k = (\prod_{i=0}^{k-2} \mathrm{bin}_k(i)x)\mathrm{bin}_k(k-1)$ is obtained from u_k by replacing all occurrences of a^{m_k} by a fresh symbol x. The word u'_k has length $\Theta(k \log k)$. Note that u'_k is a word over a ternary alphabet. Applying Theorem 3.1.1 yields

$$g(u'_k) \leq \mathcal{O}\left(\frac{k \log k}{\log(k \log k)}\right) = \mathcal{O}\left(\frac{k \log k}{\log k + \log\log k}\right) = \mathcal{O}(k).$$

Hence $g(u_k) \leq \mathcal{O}(k)$. Finally, the SLP of size $\mathcal{O}(k)$ for u_k yields an SLP of size $\mathcal{O}(k)$ for s_k again using Lemma 3.1.5, points (i) and (ii).

In conclusion: We showed that a smallest SLP for s_k has size $\mathcal{O}(k)$, while $|\mathsf{BiSection}(s_k)| \geq \Omega(2^k)$. It follows that $\alpha_{\mathsf{BS}}(s_k) \geq \Omega(2^k/k)$. Let $n = |s_k|$. Since s_k is the concatenation of $\Theta(2^k)$ factors of length $\Theta(2^k/k)$, we have $n = \Theta(2^{2k}/k)$ and thus $\sqrt{n} = \Theta(2^k/\sqrt{k})$. This yields $\alpha_{\mathsf{BS}}(s_k) \geq \Omega(\sqrt{n/k})$. Together with $k = \Theta(\log n)$ we obtain $\alpha_{\mathsf{BS}}(3, n) \geq \Omega(\sqrt{n/\log n})$.

Let us now encode words over $\{0, 1, a\}$ into words over $\{0, 1\}$. Consider the homomorphism $f : \{0, 1, a\}^* \to \{0, 1\}^*$ with $f(0) = 00$, $f(1) = 01$ and $f(a) = 10$. The same lower bound as above holds for $\alpha_{\mathsf{BS}}(f(s_k))$: The size of a smallest SLP for $f(s_k)$ is at most twice as large as the size of a smallest SLP for s_k, because an SLP for s_k can be transformed into an SLP for $f(s_k)$ by replacing every occurrence of a symbol $x \in \{0, 1, a\}$ by $f(x)$. Moreover, if we split $f(s_k)$ into non-overlapping factors of twice the length as we considered for s_k, then we obtain the factors from $f(F_k)$, whose length is again a power of two. Since f is injective, we have $|f(F_k)| = |F_k|$, which implies the same lower bound on $|\mathsf{BiSection}(f(s_k))|$. □

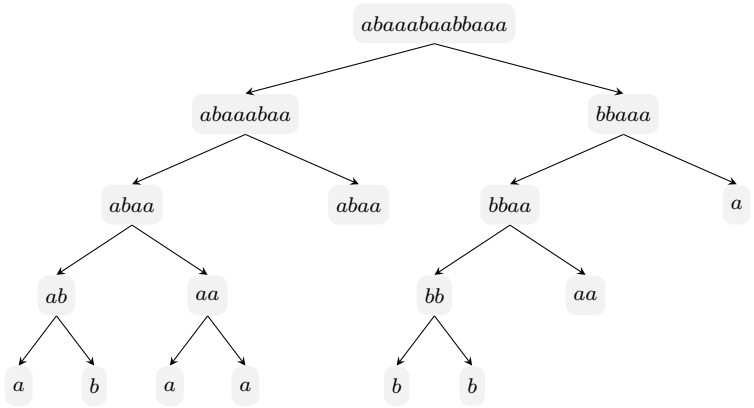

Figure 3.2: The tree described in the proof of Theorem 3.3.6 for the input $w = abaaabaabbaaa$.

3.3.3 Worst-case size

We already stated in Theorem 3.1.1 that for a word $w \in \Sigma^+$ of length n, there is always an SLP for w of size $\mathcal{O}(n/\log_{|\Sigma|} n)$. In the remaining section, we prove this theorem by showing that BiSection produces an SLP of the claimed size. The reader can find a proof of this result in a more general framework in [74]. We present a direct proof here because in Section 4.4 of the next chapter we generalize BiSection from strings to trees and show a similar result for the obtained grammar-based tree compressor. The high level proof idea is essentially the same for both algorithms and builds on a cut-point strategy that is used in the context of binary decision diagrams, see for instance [55, 79].

Theorem 3.3.6. *For all words $w \in \Sigma^+$ of length n, we have*

$$|\mathsf{BiSection}(w)| \leq \mathcal{O}\left(\frac{n}{\log_{|\Sigma|} n}\right).$$

Proof of Theorem 3.1.1 and Theorem 3.3.6. Assume an input word w of length n and consider a binary tree that is obtained as follows: The root node is labeled by the input w and if a factor x is split into x_1 and x_2 during the execution of BiSection, i.e., $x = x_1x_2$, then the corresponding node labeled by x gets a left child labeled by x_1 and a right child labeled by x_2. A node is a leaf if the label of the node is a factor of length one or if there is a node with the same label that occurs before. The order of the nodes is arbitrary, but we can assume depth-first left-to-right order (starting with the root node) here, i.e., if a factor x of length at least 2 occurs more than once as a node label, then the first node in this order that is labeled by x is split again and all other nodes with label x are leaves. In Figure 3.2 this tree is depicted for the input $w = abaaabaabbaaa$.

Let the level of a node be the length of the path from this node to the root. At level 0, there is only the root node that is labeled by the input string w of

length n. If $|w| > 1$, then at level 1 there are two nodes labeled by words w_1 and w_2 such that $w = w_1 w_2$ and $|w_1| = 2^j$ such that $2^j < n \leq 2^{j+1}$ and $|w_2| \leq |w_1|$ ($|w_1| = 2^{\lfloor \log n \rfloor}$). At level 2, there are at most 4 nodes, where each node is labeled by a word of length at most $2^{j-1} \leq n/2$. At level 3, there are at most 8 nodes, where each node is labeled by a word of length at most $2^{j-2} \leq n/4$, and so on. An inductive argument shows that each node that occurs at level ℓ is labeled by a word of length at most $n/2^{\ell-1}$.

Note that the SLP produced by BiSection on input w is twice as big as the number of inner nodes of the tree since each nonterminal produced by BiSection uniquely corresponds to a distinct factor of length at least 2 obtained by a split and thus each nonterminal corresponds to an inner node of the tree. The leaves of the tree are not represented by a nonterminal since BiSection only introduces nonterminals for factors of length at least two and multiple occurrences of a factor are represented by the same nonterminal. Additionally, each rule produced by BiSection has two symbols on the right-hand side.

Let k be an integer which will be defined later. In order to bound the number of inner nodes by $\mathcal{O}(n/\log_{|\Sigma|} n)$, we first bound the number of inner nodes that are labeled by a word of length at most k and afterwards the number of inner nodes that are labeled by a word of length more than k. There are $|\Sigma|^i$ different words of length i and

$$\sum_{i=2}^{k} |\Sigma|^i = \frac{|\Sigma|^{k+1} - |\Sigma|^2}{|\Sigma| - 1} \leq |\Sigma|^{k+1}$$

different words of length at least 2 and at most k. All inner nodes of the tree are labeled by different words since only the first occurrence of a factor of length at least 2 is split again and factors of length 1 are always leaf labels. It follows that $|\Sigma|^{k+1}$ is also an upper bound on the number of inner nodes that are labeled by words of length at most k. On the other side, if the level ℓ of a node is at least $\log n - \log k + 1$, then the node is labeled by a word of length at most k because each node at level ℓ is labeled by a word of length at most $n/2^{\ell-1}$ as argued above. It follows that only nodes at level at most $\log n - \log k$ are labeled by words of length more than k. There are at most 2^i many nodes at level i and at most

$$\sum_{i=0}^{\log n - \log k} 2^i = 2^{\log n - \log k + 1} - 1 = \frac{2n}{k} - 1$$

many nodes at level at most $\log n - \log k$. It follows that $|\Sigma|^{k+1} + 2n/k - 1$ is an upper bound on the number of inner nodes. For $k = \lfloor \log_{|\Sigma|} n - \log_{|\Sigma|} \log n \rfloor - 1$, this yields

$$\begin{aligned} |\Sigma|^{k+1} + \frac{2n}{k} - 1 &\leq |\Sigma|^{\log_{|\Sigma|} n - \log_{|\Sigma|} \log n} + \frac{2n}{\log_{|\Sigma|} n - \log_{|\Sigma|} \log n - 2} - 1 \\ &= \frac{n}{\log n} + \frac{2n}{\log_{|\Sigma|} n - \log_{|\Sigma|} \log n - 2} - 1. \end{aligned}$$

This proves the theorem. □

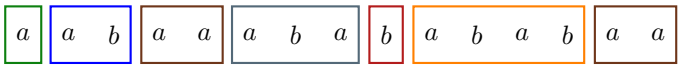

Figure 3.3: The LZ78-factorization of $w = aabaaababababaa$ is shown. Distinct factors have differently colored frames, where the colors refer to the nonterminals in the corresponding SLP presented in Examlpe 3.6.

3.4 LZ78

The LZ78 algorithm [113] on input $w \in \Sigma^+$ creates the so-called LZ78-*factorization* $w = f_1 \cdots f_\ell$ such that the following properties hold, where we set $f_0 = \varepsilon$:

- The factors are pairwise different, i.e., $f_i \neq f_j$ for all $i, j \in [0, \ell - 1]$ with $i \neq j$.
- For all $i \in [1, \ell - 1]$, there exist $j \in [0, i - 1]$, $a \in \Sigma$ such that $f_i = f_j a$.
- For the last factor f_ℓ, we have $f_\ell = f_i$ for some $i \in [0, \ell - 1]$.

Note that the LZ78-factorization is unique for each word w. To compute it, the LZ78 algorithm needs ℓ steps performed by a single left-to-right pass. In the k-th step ($k \in [1, \ell - 1]$) it chooses the factor f_k as the shortest prefix of the unprocessed suffix $f_k \cdots f_\ell$ such that $f_k \neq f_i$ for all $i < k$. If there is no such prefix, then the end of w is reached and the algorithm sets f_ℓ to the (possibly empty) unprocessed suffix of w.

The factorization $f_1, \ldots, f_\ell$ yields an SLP for w of size at most 3ℓ as described in the following example:

Example 3.6. *The* LZ78*-factorization of* $w = aabaaababababaa$ *is* $f_1 = a$, $f_2 = f_1 b = ab$, $f_3 = f_1 a = aa$, $f_4 = f_2 a = aba$, $f_5 = b$, $f_6 = f_4 b = abab$ *and* $f_7 = f_3 = aa$. *The factorization is depicted in Figure 3.3, where distinct factors are colored differently. We use the same coloring for the corresponding nonterminals in the following SLP that naturally arises from the factorization:*

- $S \to F_1 F_2 F_3 F_4 F_5 F_6 F_3$
- $F_1 \to a$, $F_2 \to F_1 b$, $F_3 \to F_1 a$, $F_4 \to F_2 a$, $F_5 \to b$, $F_6 \to F_4 b$

For each $i \in [1, 6]$, *we have a nonterminal* F_i *such that* $\mathsf{val}(F_i) = f_i$. *The last factor* $f_7 = f_3 = aa$ *is represented in the start rule by* F_3.

3.4.1 Unary inputs

The LZ78-factorization of a^n ($n > 0$) is $a^1, a^2, \ldots, a^m, a^k$, where $k \in [0, m]$ such that $n = k + \sum_{i=1}^{m} i$. Note that $m = \Theta(\sqrt{n})$ and thus

$$\alpha_{\mathsf{LZ78}}(1, n) = \Theta\left(\frac{\sqrt{n}}{\log n}\right).$$

3.4.2 General case

The following upper bound on the worst-case approximation ratio of LZ78 for arbitrary alphabets is provided in [29]:

Theorem 3.4.1 ([29, Theorem 4])**.** *For all n, we have*

$$\alpha_{\mathsf{LZ78}}(n) \leq \mathcal{O}\left(\left(\frac{n}{\log n}\right)^{2/3}\right).$$

Further, the lower bound $\alpha_{\mathsf{LZ78}}(2,n) \geq \Omega(n^{2/3}/\log n)$ for infinitely many n is shown in [29, Theorem 3]. In the following, we improve this lower bound such that it matches the upper bound.

Theorem 3.4.2. *For all $k \geq 3$ and infinitely many n, we have*

$$\alpha_{\mathsf{LZ78}}(k,n) \geq \Omega\left(\left(\frac{n}{\log n}\right)^{2/3}\right).$$

Proof. We show $\alpha_{\mathsf{LZ78}}(3,n) \geq \Omega((n/\log n)^{2/3})$. For $k \geq 2$ and $m \geq 1$, let

$$u_{m,k} = \left(a^k b^m c\right)^{k(m+2)-1} \quad \text{and} \quad v_{m,k} = \left(\prod_{i=1}^{m} b^i a^k\right)^{k^2}.$$

We now analyze the approximation ratio of LZ78 on the words

$$s_{m,k} = a^{k(k+1)/2}\, b^{m(m+1)/2}\, u_{m,k}\, v_{m,k}.$$

For example, we have $s_{2,4} = a^{10}\, b^3\, u_{2,4}\, v_{2,4}$, where $u_{2,4} = (a^4b^2c)^{15}$ and $v_{2,4} = (ba^4b^2a^4)^{16}$. The full LZ78 factorization of $s_{2,4}$ can be found in Example 3.7.

Claim 1. $|\mathsf{LZ78}(s_{m,k})| = \Theta(k^2 m)$

Proof. We consider the LZ78-factorization $f_1, \ldots, f_\ell$ of $s_{m,k}$. The prefix $a^{k(k+1)/2}$ produces the factors $f_i = a^i$ for $i \in [1,k]$ and $b^{m(m+1)/2}$ produces the factors $f_{k+i} = b^i$ for $i \in [1,m]$.

We next show that $u_{m,k}$ then produces (among other factors) all factors $a^i b^j$, where $i \in [1,k]$ and $j \in [1,m]$. All other factors produced by $u_{m,k}$ contain the symbol c and therefore do not affect the factorization of the final suffix $v_{m,k} \in \{a,b\}^*$. The first factors of $u_{m,k}$ in $s_{m,k}$ are $f_{k+m+1} = a^k b$ and $f_{k+m+2} = b^{m-1}c$, which together form the first occurrence of $a^k b^m c$. The next two factors are $a^k b^2$ and $b^{m-2}c$. This pattern continues and the prefix $(a^k b^m c)^m$ of $u_{m,k}$ yields $2m$ many factors, namely $f_{k+m+2i-1} = a^k b^i$ and $f_{k+m+2i} = b^{m-i}c$ for $i \in [1,m]$. The factorization of $u_{m,k}$ continues with $f_{k+3m+1} = a^k b^m c$ followed by $f_{k+3m+2} = a^k b^m ca$. Next, we have $f_{k+3m+3} = a^{k-1}b$ and $f_{k+3m+4} = b^{m-1}ca$, which is the beginning of a similar pattern as we discovered for $(a^k b^m c)^m$. Therefore, the next $2m$ factors are $f_{k+3m+2i+1} = a^{k-1}b^i$ and $f_{k+3m+2i+2} = b^{m-i}ca$ for $i \in [1,m]$. The next two factors are $f_{k+5m+3} = a^{k-1}b^m c$ followed by $f_{k+5m+4} = a^k b^m ca^2$. The iteration of these arguments yields k (consecutive) blocks of $2m+2$ factors (respectively $2m+1$ in the last block) for $u_{m,k}$:

$$
\begin{array}{lll}
1^{\text{st}} & \text{block:} & \prod_{i=1}^{m} \left(\begin{array}{ll} a^k b^i & b^{m-i} c \end{array} \right) \quad a^k b^m c \quad a^k b^m c a \\
2^{\text{nd}} & \text{block:} & \prod_{i=1}^{m} \left(\begin{array}{ll} a^{k-1} b^i & b^{m-i} c a \end{array} \right) \quad a^{k-1} b^m c \quad a^k b^m c a^2 \\
\cdots & & \\
(k-1)^{\text{th}} & \text{block:} & \prod_{i=1}^{m} \left(\begin{array}{ll} a^2 b^i & b^{m-i} c a^{k-2} \end{array} \right) \quad a^2 b^m c \quad a^k b^m c a^{k-1} \\
k^{\text{th}} & \text{block:} & \prod_{i=1}^{m} \left(\begin{array}{ll} a b^i & b^{m-i} c a^{k-1} \end{array} \right) \quad a b^m c
\end{array}
$$

We will show that the remaining suffix $v_{m,k}$ of $s_{m,k}$ produces then the set of factors

$$\left\{ a^i b^p a^j \mid i \in [0, k-1],\ j \in [1, k],\ p \in [1, m] \right\}.$$

Let $x = k + m + k(2m+2) - 1$ and note that this is the number of factors that we have produced so far. The factorization of $v_{m,k}$ in $s_{m,k}$ slightly differs whether m is even or is odd. We now assume that m is even and explain the difference to the other case afterwards. The first factor of $v_{m,k}$ in $s_{m,k}$ is $f_{x+1} = ba$. We have already produced the factors $a^{k-1}b^i$ for every $i \in [1, m]$ and hence $f_{x+i} = a^{k-1}b^i a$ for $i \in [2, m]$ and $f_{x+m+1} = a^{k-1}ba$. The next m factors are $f_{x+m+i} = a^{k-1}b^i a^2$ if $i \in [2, m]$ is even, $f_{x+m+i} = a^{k-2}b^i a$ if $i \in [2, m]$ is odd and $f_{x+2m+1} = a^{k-2}ba$. This pattern continues: The next m factors are $f_{x+2m+i} = a^{k-1}b^i a^3$ if $i \in [2, m]$ is even, $f_{x+2m+i} = a^{k-3}b^i a$ if $i \in [2, m]$ is odd and $f_{x+3m+1} = a^{k-3}ba$ and so on. Hence, we get the following sets of factors for $(\prod_{i=1}^{m} b^i a^k)^k$:

(i) $\{a^{k-i}b^p a \mid i \in [1, k],\ p \in [1, m] \text{ is odd}\}$ for $f_{x+1}, f_{x+3} \ldots, f_{x+km-1}$

(ii) $\{a^{k-1}b^p a^j \mid j \in [1, k],\ p \in [1, m] \text{ is even}\}$ for $f_{x+2}, f_{x+4}, \ldots, f_{x+km}$

The remaining factorization starts $f_{y+1} = ba^2$, where $y = x + km$. Now the former pattern can be adapted to the next k repetitions of $\prod_{i=1}^{m} b^i a^k$ which gives us the following factors:

(i) $\{a^{k-i}b^p a^2 \mid i \in [1, k],\ p \in [1, m] \text{ is odd}\}$ for $f_{y+1}, f_{y+3} \ldots, f_{y+km-1}$

(ii) $\{a^{k-2}b^p a^j \mid j \in [1, k],\ p \in [1, m] \text{ is even}\}$ for $f_{y+2}, f_{y+4}, \ldots, f_{y+km}$

The iteration of this process then reveals the whole pattern and thus yields the claimed factorization of $v_{m,k}$ in $s_{m,k}$ into factors $a^i b^p a^j$ for $i \in [0, k-1]$, $j \in [1, k]$ and $p \in [1, m]$. If m is odd then the patterns in (i) and (ii) switch after each occurrence of $\prod_{i=1}^{m} b^i a^k$, which does not affect the result but makes the pattern slightly more complicated. But the case that m is even suffices in order to derive the lower bound from the theorem.

We conclude that there are exactly $k + m + k(2m+2) - 1 + k^2 m$ factors (ignoring $f_\ell = \varepsilon$) and hence $|\mathsf{LZ78}(s_{m,k})| = \Theta(k^2 m)$.

Claim 2. $g(s_{m,k}) \leq \mathcal{O}(\log k + m)$

Proof. We will combine the points stated in Lemma 3.1.5 to prove this claim. Repeatedly applying the points (i) and (ii) yield an SLP of size $\mathcal{O}(\log k + \log m)$ for the prefix $a^{k(k+1)/2}\ b^{m(m+1)/2}\ u_{m,k}$. To bound the size of an SLP for $v_{m,k}$, note that there is an SLP of size $\mathcal{O}(\log k)$ producing a^k by applying point (i). By point (iii) and again point (i), it follows that there is an SLP of

size $\mathcal{O}(\log k) + g(v'_{m,k})$ for $v_{m,k}$, where $v'_{m,k} = \prod_{i=1}^{m} b^i x$ for some fresh symbol x. To get a small SLP for $v'_{m,k}$, we introduce m nonterminals $B_1, \ldots, B_m$ and rules $B_1 \to b$ and $B_{i+1} \to B_i b$ for $i \in [1, m-1]$, i.e., $\mathsf{val}(B_i) = b^i$ for $i \in [1, m]$. Replacing the maximal b-blocks in $v'_{m,k}$ by those nonterminals is enough to get an SLP of size $\mathcal{O}(m)$ for $v'_{m,k}$ and therefore an SLP of size $\mathcal{O}(\log k + m)$ for $v_{m,k}$. Together with our first observation and point (ii), this yields an SLP of size $\mathcal{O}(\log k + m)$ for $s_{m,k}$.

Claim 1 and 2 imply $\alpha_{\mathsf{LZ78}}(s_{m,k}) \geq \Omega(k^2 m/(\log k + m))$. Let us now fix $m = \lceil \log k \rceil$. We get $\alpha_{\mathsf{LZ78}}(s_{m,k}) \geq \Omega(k^2)$. Moreover, let $n = |s_{m,k}|$ be the length of $s_{m,k}$. We have $n = \Theta(k^3 m + k^2 m^2) = \Theta(k^3 \log k)$. It follows that $\alpha_{\mathsf{LZ78}}(s_{m,k}) \geq \Omega((n/\log k)^{2/3})$, which together with $\log n = \Theta(\log k)$ finishes the proof. □

Example 3.7. *Here is the complete* LZ78 *factorization of*

$$s_{2,4} = a^{10}b^3 \underbrace{(a^4b^2c)^{15}}_{u_{2,4}} \underbrace{(ba^4b^2a^4)^{16}}_{v_{2,4}}.$$

Factors of a^{10}*:* $a,\ a^2,\ a^3,\ a^4$
Factors of b^3*:* $b,\ b^2$
Factors of $u_{2,4}$*:*

a^4b	bc	a^4b^2	c	a^4b^2c	a^4b^2ca
a^3b	bca	a^3b^2	ca	a^3b^2c	$a^4b^2ca^2$
a^2b	bca^2	a^2b^2	ca^2	a^2b^2c	$a^4b^2ca^3$
ab	bca^3	ab^2	ca^3	ab^2c	

Factors of $v_{2,4}$*:*

ba	a^3b^2a
a^3ba	$a^3b^2a^2$
a^2ba	$a^3b^2a^3$
aba	$a^3b^2a^4$
ba^2	a^2b^2a
a^3ba^2	$a^2b^2a^2$
a^2ba^2	$a^2b^2a^3$
aba^2	$a^2b^2a^4$
ba^3	ab^2a
a^3ba^3	ab^2a^2
a^2ba^3	ab^2a^3
aba^3	ab^2a^4
ba^4	b^2a
a^3ba^4	b^2a^2
a^2ba^4	b^2a^3
aba^4	b^2a^4

In [11], our lower bound is extended to binary alphabets. More precisely, the words $u_{m,k}$ from the proof of Theorem 3.4.3 are slightly changed in order to eliminate occurrences of the symbol c while preserving the factorization of $s_{m,k}$. This yields the following theorem:

Theorem 3.4.3 ([11, Theorem 3.4]). *For all $k \geq 2$ and infinitely many n, we have*

$$\alpha_{\mathsf{LZ78}}(k,n) \geq \Omega\left(\left(\frac{n}{\log n}\right)^{2/3}\right).$$

3.5 Global algorithms

For a given SLP $\mathbb{A}$, a word γ is called a *maximal string* of $\mathbb{A}$ if

- $|\gamma| \geq 2$,
- γ appears at least twice without overlap as a factor of the right-hand sides,
- if m is the maximal number of non-overlapping occurrences of γ on right-hand sides of $\mathbb{A}$, then any word of length $> |\gamma|$ has at most $m-1$ non-overlapping occurrences on right-hand sides of $\mathbb{A}$.

Example 3.8. *Let $\mathbb{A} = (\{S, X, Y, Z\}, \{a, b\}, P, S)$ such that P contains*

- $S \to aXXXXbbYZYZ$,
- $X \to YbbYZYZa$,
- $Y \to bbbZZaZ$, *and*
- $Z \to abb$.

The maximal strings of $\mathbb{A}$ are bb, YZ and $bbYZYZ$. The factors bb and YZ occur four times on the right-hand sides without overlap and $bbYZYZ$ occurs twice without overlap. All other factors of the right-hand sides have length at most 5 and occur at most twice without overlap.

A *global grammar-based compressor* (or simply *global algorithm*) starts on input w with the SLP $\mathbb{A}_0 = (\{S\}, \Sigma, \{S \to w\}, S)$. In each round $i \geq 1$, the algorithm selects a maximal string γ of $\mathbb{A}_{i-1}$ and updates $\mathbb{A}_{i-1}$ to $\mathbb{A}_i$ by replacing a largest set of pairwise non-overlapping occurrences of γ in $\mathbb{A}_{i-1}$ by a fresh nonterminal X. Additionally, the algorithm introduces the rule $X \to \gamma$ in $\mathbb{A}_i$. The algorithm stops when no maximal string occurs. Note that the replacement is not unique, e.g. the word a^5 has a unique maximal string $\gamma = aa$, which yields SLPs with rules $S \to XXa, X \to aa$ or $S \to XaX, X \to aa$ or $S \to aXX, X \to aa$. We assume the first variant here, i.e., maximal strings are replaced from left to right.

The global grammar-based compressors studied in the following sections are RePair [77], Greedy [7, 8] and LongestMatch [72]. The best known upper bound on the approximation ratio of those algorithms for arbitrary alphabets is shown in [29], where the authors provide the following result for any global algorithm.

Theorem 3.5.1 ([29, Theorem 9]). *For any global grammar-based compressor $\mathcal{C}$ and for all n, we have*

$$\alpha_{\mathcal{C}}(n) \leq \mathcal{O}\left(\left(\frac{n}{\log n}\right)^{2/3}\right).$$

3.6 RePair

In this section we analyze the global grammar-based compressor RePair [77]. RePair selects in each round a most frequent maximal string. While in the original version RePair always selects a most frequent digram (a factor of length two), we follow the definition of [29] where RePair possibly selects a longer maximal string if this string occurs exactly as often as the most frequent digram. However, the bounds we present work in both settings.

Example 3.9. *Let $w = aaaaabbababbbaaabb$. We have*

$\mathbb{A}_0 : S \to aaaaabbababbbaaabb,$

$\mathbb{A}_1 : S \to aaaaXbXXbbaaXb,\ X \to ab,$

$\mathbb{A}_2 : S \to YYXbXXbbYXb,\ X \to ab,\ Y \to aa,$

$\mathbb{A}_3 : S \to YYZXZbYZ,\ X \to ab,\ Y \to aa,\ Z \to Xb,$

$\mathbb{A}_4 : S \to YAXZbA,\ X \to ab,\ Y \to aa,\ Z \to Xb,\ A \to YZ.$

Note that in round 2, instead of the maximal string aa the algorithm could also choose the maximal string Xb (that is chosen in round 3), because both factors occur three times without overlap.

3.6.1 Unary inputs

We first study the approximation ratio $\alpha_{\mathsf{RePair}}(1, n)$, i.e., we consider unary inputs. Recall the function $\nu(n)$ that denotes the number of 1's in the binary representation of n, see equation (2.2).

Proposition 3.6.1. *For all $n \geq 2$, we have*

$$|\mathsf{RePair}(a^n)| = 2\lfloor \log n \rfloor + \nu(n) - 1.$$

Proof. If $n \in [2, 3]$ then a^n has no maximal string and thus the final SLP has a single rule $S \to a^n$. We have $n = 2\lfloor \log n \rfloor + \nu(n) - 1$ for $n \in [2, 3]$.

Assume $n \geq 4$ in the following and let $m = \lfloor \log n \rfloor - 1$. On input a^n, RePair runs for exactly m rounds. Let X_i be the nonterminal introduced in round i, then RePair creates rules $X_1 \to aa$ and $X_i \to X_{i-1}X_{i-1}$ for $i \in [2, m]$, i.e., $\mathsf{val}(X_i) = a^{2^i}$. Those rules have total size $2m = 2\lfloor \log n \rfloor - 2$. The missing part of $\mathsf{RePair}(a^n)$ is the start rule. We have

$$S \to X_m X_m X_m^{b_m} X_{m-1}^{b_{m-1}} \cdots X_1^{b_1} a^{b_0},$$

where b_i is the coefficient of 2^i in the binary representation of n, see equation (2.1). In other words, the symbol a only occurs in the start rule if the least significant bit $b_0 = 1$, and the nonterminal X_i ($i \in [1, m-1]$) occurs in the start rule if and only if $b_i = 1$. Since RePair only replaces words with at least two occurrences, the most significant bit $b_{m+1} = 1$ is represented by $X_m X_m$. A third X_m occurs in the start rule if and only if $b_m = 1$. The size of the start rule is $2 + \nu(n) - 1$, which yields a total size of $2\lfloor \log n \rfloor + \nu(n) - 1$. □

Theorem 3.6.2. *For all n, we have*

$$\alpha_{\mathsf{RePair}}(1, n) \leq \log(3).$$

Proof. As a consequence of Proposition 3.6.1, RePair produces an SLP of size $2\lfloor \log n \rfloor + \nu(n) - 1 \leq 3 \log n$. By Lemma 3.1.2, we have $g(a^n) \geq 3 \log_3 n - 3$. The equality $\log n / \log_3 n = \log(3)$ finishes the proof. □

Theorem 3.6.3. *For infinitely many n, we have*

$$\alpha_{\mathsf{RePair}}(1, n) \geq \log(3).$$

Proof. Let $s_k = a^{2^k - 1}$. We have $2^k - 1 = \sum_{i=0}^{k-1} 2^i$ and thus $\nu(2^k - 1) = k$. By Proposition 3.6.1, we have $|\mathsf{RePair}(s_k)| = 3k - 3$. By Lemma 3.1.3, we have $g(s_k) \leq 3 \log_3(2^k - 1) + o(\log(2^k - 1)) \leq 3 \log_3(2) \cdot k + o(k)$. The equality $1 / \log_3(2) = \log(3)$ finishes the proof. □

3.6.2 General case

As mentioned in Section 3.5, the best known upper bound on the approximation ratio of RePair is $\alpha_{\mathsf{RePair}}(n) \leq \mathcal{O}((n / \log n)^{2/3})$ (Theorem 3.5.1). Further, $\alpha_{\mathsf{RePair}}(n) \geq \Omega\left(\sqrt{\log n}\right)$ is shown for infinitely many n in [29, Theorem 12]. The proof of this lower bound assumes an alphabet of unbounded size. To be more accurate, the authors construct for every k a word w_k of length $\Theta(\sqrt{k} 2^k)$ over an alphabet of size $\Theta(k)$ such that $g(w) \leq \mathcal{O}(k)$ and $|\mathsf{RePair}(w_k)| \geq \Omega(k^{3/2})$.

We will improve this lower bound using only a binary alphabet. To do so, recall how RePair compresses unary strings (Proposition 3.6.1). In particular, the start rule produced by RePair on a unary input a^n ($n \geq 4$) is

$$S \to X_m X_m X_m^{b_m} X_{m-1}^{b_{m-1}} \cdots X_1^{b_1} a^{b_0},$$

where $m = \lfloor \log n \rfloor$, b_i is the coefficient of 2^i in the binary representation of n for $i \in [0, m]$ and X_i is a nonterminal such that $\mathsf{val}(X_i) = a^{2^i}$ for $i \in [1, m]$.

We will use *De Bruijn sequences* [36] in the following theorem. A binary De Bruijn sequence of order k is a string $B_k \in \{0,1\}^*$ of length 2^k such that every string from $\{0,1\}^k$ is either a factor of B_k or a suffix of B_k concatenated with a prefix of B_k. Further, each word of length k occurs at most once as factor in B_k. As an example, the string 1100 is a De Bruijn sequence of order 2, since 11, 10 and 00 occur as factors and 01 occurs as a suffix concatenated with a prefix.

Note that De Bruijn sequences are not unique, for example the homomorphism $f(0) = 1$ and $f(1) = 0$ transforms a binary De Bruijn sequence w into a different binary De Bruijn sequence $f(w)$.

Theorem 3.6.4. *For all $k \geq 2$ and infinitely many n, we have*

$$\alpha_{\mathsf{RePair}}(k, n) \geq \Omega\left(\frac{\log n}{\log \log n}\right).$$

Proof. For $k \geq 2$, we construct binary words $s_k \in \{a, b\}^*$ such that the claimed lower bound holds for $\alpha_{\mathsf{RePair}}(s_k)$. To do so, we start with a binary De Bruijn sequence $B_{\lceil \log k \rceil} \in \{0, 1\}^*$ of length $2^{\lceil \log k \rceil}$. We have $k \leq |B_{\lceil \log k \rceil}| < 2k$. We assume that the De Bruijn sequence starts with 1. We define a homomorphism $h : \{0, 1\}^* \to \{0, 1\}^*$ by $h(0) = 01$ and $h(1) = 10$. The words w_k of length $2k$ are defined as

$$w_k = h(B_{\lceil \log k \rceil}[1 : k]).$$

For example $k = 4$ and $B_2 = 1100$ yield $w_4 = 10100101$. For $i \in [1, k]$, let n_i be the number such that $w_k[1 : k + i]$ is the binary representation of n_i, i.e.,

$$n_i = \sum_{j=0}^{k+i-1} w_k[k + i - j] \cdot 2^j.$$

We will analyze the approximation ratio of RePair for the binary words

$$s_k = a^{n_1} b a^{n_2} \cdots b a^{n_k} = \prod_{i=1}^{k-1} (a^{n_i} b)\, a^{n_k}.$$

For example we have $s_4 = a^{20}ba^{41}ba^{82}ba^{165}$. Since $B_{\lceil \log k \rceil}[1] = w_k[1] = 1$, we have $2^{k+i-1} \leq |a^{n_i}| \leq 2^{k+i} - 1$ for $i \in [1, k]$ and thus $|s_k| = \Theta\left(4^k\right)$.

Claim 1. $|\mathsf{RePair}(s_k)| \geq \Omega\left(\frac{k^2}{\log k}\right)$

Proof. The idea is that due to the properties of the De Bruijn sequence, RePair creates at some point of the algorithm a start rule which has $\Theta(k^2)$ distinct factors of length $\Theta(\log n)$. This leads to a lower bound on the size of the final SLP produced by RePair using Lemma 3.1.4.

On unary inputs of length n, the start rule produced by RePair is strongly related to the binary representation of n as described in Proposition 3.6.1. On input s_k, the algorithm begins to produce a start rule that is similarly related to the binary representations $w_k[1 : k + i]$ of n_i for $i \in [1, k]$. Consider the SLP $\mathbb{A}_{k-1}$ which is produced by RePair after $k - 1$ rounds on input s_k. We claim that up to this point RePair is not affected by the b's in s_k and therefore has introduced the rules $X_1 \to aa$ and $X_i \to X_{i-1}X_{i-1}$ for $i \in [2, k - 1]$. If this is true, then the first a-block is modified in the start rule after $k - 1$ rounds as follows:

$$S \to X_{k-1}X_{k-1}X_{k-1}^{w_k[2]}X_{k-2}^{w_k[3]} \cdots X_1^{w_k[k+2]}a^{w_k[k+1]}b \cdots$$

All other a-blocks are longer than the first one and hence each of the k factors of the start rule which corresponds to an a-block begins with $X_{k-1}X_{k-1}$.

Since the symbol b occurs only $k-1$ times in s_k, it follows that our assumption is correct and RePair is not affected by the b's in the first $k-1$ rounds on input s_k. Further, for each block a^{n_i} the $i-1$ least significant bits of $w_k[1:k+i]$ ($i \in [1,k]$) are represented in the corresponding factor of the start rule of $\mathbb{A}_{k-1}$, i.e., the start rule contains non-overlapping factors v_i such that

$$v_i = X_{k-2}^{w_k[2+i]} X_{k-3}^{w_k[3+i]} \cdots X_1^{w_k[k-1+i]} a^{w_k[k+i]}, \tag{3.3}$$

where $i \in [1,k]$. For example after 3 rounds on input $s_4 = a^{20}ba^{41}ba^{82}ba^{165}$, we have the start rule

$$S \to \underbrace{X_3X_3X_2}_{a^{20}}\, b\, \underbrace{X_3{}^5 a}_{a^{41}}\, b\, \underbrace{X_3{}^{10}X_1}_{a^{82}}\, b\, \underbrace{X_3{}^{20}X_2 a}_{a^{165}},$$

where $v_1 = X_2$, $v_2 = a$, $v_3 = X_1$ and $v_4 = X_2a$. The length of a factor $v_i \in \{a, X_1, \ldots, X_{k-2}\}^*$ from equation (3.3) is exactly the number of 1's in $w_k[i+2:k+i]$. Since w_k is constructed by the homomorphism h, it is easy to see that $|v_i| \geq (k-3)/2$. Note that no symbol occurs more than once in v_i, hence $g(v_i) = |v_i|$. Further, each factor of length $2\lceil \log k \rceil + 2$ occurs at most once in $v_1, \ldots, v_k$, because otherwise there would be a factor of length $\lceil \log k \rceil$ occurring more than once in $B_{\lceil \log k \rceil}$. It follows that there are at least

$$k \cdot \left(\left\lceil \frac{k-3}{2} \right\rceil - 2\lceil \log k \rceil - 1 \right) = \Theta(k^2)$$

different factors of length $2\lceil \log k \rceil + 2 = \Theta(\log k)$ on the right-hand side of the start rule of $\mathbb{A}_{k-1}$. By Lemma 3.1.4, it follows that a smallest SLP for the right-hand side of the start rule has size $\Omega(k^2/\log k)$ and thus $|\mathsf{RePair}(s_k)| \geq \Omega(k^2/\log k)$.

Claim 2. $g(s_k) \leq \mathcal{O}(k)$

Proof. There is an SLP $\mathbb{A}$ of size $\mathcal{O}(k)$ for a^{n_1} by Lemma 3.1.5, point (i) and $n_1 = \Theta(2^k)$. Let A be the start nonterminal of $\mathbb{A}$. Note that the binary representation of n_2 is obtained from the binary representation of n_1 by adding a least significant bit $w_k[k+2]$. Hence, we only need one additional rule for a^{n_2}: If $w_k[k+2] = 0$, then $n_2 = 2n_1$ and we can produce a^{n_2} by the fresh rule $B \to AA$. Otherwise, if $w_k[k+2] = 1$, then $n_2 = 2n_1 + 1$ and $B \to AAa$. By the same reasoning, we can produce the third a-block with one additional rule and so. The iteration of that process yields for $a^{n_2}, \ldots, a^{n_k}$ a single rule of size at most 3. If we replace the a-blocks in s_k by nonterminals as described, then the resulting word has size $2k+1$ and hence $g(s_k) \leq \mathcal{O}(k)$.

In conclusion: We showed that a smallest SLP for s_k has size $\mathcal{O}(k)$, while RePair produces on input s_k an SLP of size $\Omega(k^2/\log k)$. It follows that $\alpha_{\mathsf{RePair}}(s_k) \geq \Omega(k/\log k)$, which together with $n = |s_k|$ and $k = \Theta(\log n)$ finishes the proof. □

Note that in the proof, RePair chooses in the first $k-1$ rounds a factor of length two as the maximal string. Therefore, our lower bound also holds for the original RePair-variant where a digram is chosen in each round.

3.6.3 RePair extension for general addition chains

In the following, we provide evidence that based on our approach it is not possible to improve the lower bound even further. To do so, we present an algorithm that behaves like RePair on input $a^{n_1}b_1a^{n_2}\cdots b_{k-1}a^{n_k}$ as long as only rules $X_1 \to aa$ and $X_{i+1} \to X_iX_i$ for some i are introduced, and achieves approximation ratio $\mathcal{O}(\log N/\log\log N)$ on those inputs, where $N = \max\{n_1, n_2, \ldots, n_k\}$. Note first that in the proof of Theorem 3.6.4 we stopped RePair at a point where only rules of this form have been introduced and afterwards analyzed the obtained SLP, so the lower bound holds for the new algorithm as well. Further, the alphabet in Theorem 3.6.4 is binary, i.e., $b_i = b$ for all $i \in [1, k-1]$, but each SLP for $(\prod_{i=1}^{k-1} a^{n_i}b_i)a^{n_k}$ yields an SLP for $(\prod_{i=1}^{k-1} a^{n_i}b)a^{n_k}$ of equal size by replacing the b_i's by b. Finally, note that in Theorem 3.6.4 we have $\log N = \Theta(\log n)$ where n is the total length of the input string, which yields the matching upper bound for the new algorithm on those inputs.

We fix a set of integers $\{n_1, n_2, \ldots, n_k\}$ for the remainder of this section. Basically, we describe an algorithm that creates a general addition chain for $n_1, \ldots, n_k$ in terms of grammar-based compression, i.e., we create an SLP for $(\prod_{i=1}^{k-1} a^{n_i}b_i)a^{n_k}$ (see Section 3.2.2 for the relation between grammar-based compression and general addition chains). Let $t = \lfloor \log N \rfloor$, where we set $N = \max\{n_i \mid i \in [1, k]\}$ as above. We first create all powers of two up to 2^t by repeated squaring:

$$X_0 \to a, \quad X_{i+1} \to X_iX_i \quad (i \in [0, t-1])$$

Note that RePair does not introduce $X_0 \to a$, but the new algorithm does due to better readability in the following. We have $\mathsf{val}(X_i) = a^{2^i}$ for $i \in [0, t]$. Further, we start with a start rule $S \to (\prod_{i=1}^{k-1} a^{n_i}b_i)a^{n_k}$ and the algorithm successively replaces a maximal pairwise non-overlapping set of X_iX_i by X_{i+1} in the same manner as RePair (with the difference that a is represented by X_0). More formally, consider the binary representation of n_i for each $i \in [1, k]$. Let

$$n_i = \sum_{j=0}^{t} b_{i,j} \cdot 2^j,$$

where $b_{i,j} \in \{0, 1\}$ for all $i \in [1, k]$ and $j \in [0, t]$. Let

$$v_i = X_t^{b_{i,t}} \cdots X_1^{b_{i,1}} X_0^{b_{i,0}}.$$

It follows that $\mathsf{val}(v_i) = a^{n_i}$ for $i \in [1, k]$ and after the replacements, the start rule is $S \to v_1b_1v_2 \cdots b_{k-1}v_k$. The obtained SLP has (worst-case) size $2t + k(t+1)$. In a final step, we compress this start rule in order to achieve our final SLP. Let

$s < t$ be an integer which will be defined later and let $d = \lceil (t+1)/s \rceil$. The strategy is to divide each v_i into d many blocks such that each block contains at most s many nonterminals. We get a factorization $v_i = v_{i,d} \cdots v_{i,2} v_{i,1}$ for all $i \in [1,k]$, where

$$v_{i,j} = X_{js-1}^{b_{i,js-1}} \cdots X_{(j-1)s}^{b_{i,(j-1)s}}$$

for $j \in [1, d-1]$ and $v_{i,d} = X_t^{b_{i,t}} \cdots X_{(d-1)s}^{b_{i,(d-1)s}}$. Note that some of the $v_{i,j}$ might be empty (this is the case when all occurring exponents are 0). Now consider the set $V_j = \{v_{i,j} \mid i \in [1,k]\}$. We have $|V_j| \leq 2^s$ for each $j \in [1,d]$, because each word in V_j is uniquely defined by (at most) s exponents that are either 0 or 1. It follows that we can define nonterminals for all distinct $v_{i,j}$'s using at most $2^s \cdot d$ many rules, where each right-hand side of a rule has length at most s. Finally, we update the start rule such that each v_i is produced by a sequence of at most d of those nonterminals. The updated start rule has size $k \cdot (d+1)$ and the total grammar has size $2t + h(s)$, where $h(s) = k \cdot (d+1) + 2^s \cdot d \cdot s$. By applying $d = \lceil (t+1)/s \rceil$ and $t = \lfloor \log N \rfloor$, we get

$$\begin{aligned} h(s) &= k \cdot \left(\left\lceil \frac{\lfloor \log N \rfloor + 1}{s} \right\rceil + 1 \right) + 2^s \cdot \left\lceil \frac{\lfloor \log N \rfloor + 1}{s} \right\rceil \cdot s \\ &\leq k \cdot \left(\frac{\log N}{s} + 3 \right) + 2^s \cdot \left(\frac{\log N}{s} + 2 \right) \cdot s \\ &= k \cdot \left(\frac{\log N}{s} + 3 \right) + 2^s \cdot \log N + 2^{s+1} s \end{aligned}$$

Choosing $s = \lfloor \alpha \log\log N - \beta \log\log\log N \rfloor$ yields

$$\begin{aligned} h(s) \leq & k \cdot \left(\frac{\log N}{\alpha \log\ \log N - \beta \log\log\log N - 1} + 3 \right) + \frac{(\log N)^{1+\alpha}}{(\log\log N)^\beta} + \\ & + \frac{2(\log N)^\alpha (\alpha \log\ \log N - \beta \log\log\log N)}{(\log\log N)^\beta}. \end{aligned}$$

For example $\alpha = 1$ and $\beta = 2$ yield an SLP of size

$$k \cdot \left(\frac{\log N}{\log\ \log N - 2\log\log\log N - 1} + 3 \right) + \mathcal{O}\left(\frac{(\log N)^2}{(\log\log N)^2} \right). \tag{3.4}$$

Let $g = g((\prod_{i=1}^{k-1} a^{n_i} b_i) a^{n_k})$ be the size of a smallest SLP for $(\prod_{i=1}^{k-1} a^{n_i} b_i) a^{n_k}$. Then the algorithm described above produces an SLP of size

$$g \cdot \mathcal{O}\left(\frac{\log N}{\log\log N} \right),$$

because dividing (3.4) by g, where $g \geq k$ ($\prod_{i=1}^{k-1} a^{n_i} b_i) a^{n_k}$ has k different symbols) and $g \geq \Omega(\log N)$ (Lemma 3.1.2) bounds the approximation ratio by

$$\frac{\log N}{\log\ \log N - 2\log\log\log N - 1} + o\left(\frac{\log N}{\log\log N} \right).$$

In Figure 3.4 the pseudocode of the described algorithm is depicted.

There are two major implications. First of all, new approaches are necessary in order to improve our lower bound on the approximation ratio of RePair since from the point where we stopped RePair in Theorem 3.6.4, it is not possible to derive a better lower bound as the new algorithm shows. Second, it is worth mentioning that the approximation ratio $\mathcal{O}(\log N/\log\log N)$ that we achieved for the new algorithm on input $(\prod_{i=1}^{k-1} a^{n_i}b_i)a^{n_k}$ asymptotically matches the best known approximation ratio that a polynomial-time algorithm achieves for the general addition chain problem [110]. So the presented algorithm is of interest on its own as a general addition chain solver.

input : Integers $n_1, \ldots, n_k$
$N := \max\{n_1, \ldots, n_k\}$
$t := \lfloor \log N \rfloor$
$V := \{S\} \cup \{X_i \mid i \in [0, t]\}$ `(set of nonterminals)`
$P := \{X_0 \to a\} \cup \{X_{i+1} \to X_i X_i \mid i \in [0, t-1]\}$ `(set of rules)`
$s := \lfloor \log\log N - 2\log\log\log N \rfloor$
$d := \lceil (t+1)/s \rceil$

Let $b_{i,j} \in \{0,1\}$ be the coefficient of 2^j in $n_i = \sum_{j=0}^{t} b_{i,j} \cdot 2^j$ for $i \in [1,k]$.
Let $v_{i,j} = X_{js-1}^{b_{i,js-1}} \cdots X_{(j-1)s}^{b_{i,(j-1)s}}$ for $i \in [1,k]$ and $j \in [1, d-1]$
and $v_{i,d} = X_t^{b_{i,t}} \cdots X_{(d-1)s}^{b_{i,(d-1)s}}$ for $i \in [1,k]$.

$M := \{v_{i,j} \mid i \in [1,k],\ j \in [1,d]\}$ `(set of distinct` $v_{i,j}$`'s)`
foreach $w \in M$ **do**
  Let X be a fresh nonterminal.
  $V := V \cup \{X\}$
  $P := P \cup \{X \to w\}$ `(rules for distinct` $v_{i,j}$`'s)`
end
foreach $i = 1$ **to** k **do**
  Let $Y_j \in V$ be the nonterminal such that $(Y_j \to v_{i,j}) \in P$ for $j \in [i,d]$.
  $u_i := Y_1 \cdots Y_d$ `(`a^{n_i} `is represented by` d `nonterminals)`
end
$P := P \cup \{S \to u_1 b_1 u_2 \cdots b_{k-1} u_k\}$ `(start rule)`
while P *contains a rule of the form* $X \to \varepsilon$ **do**
  Remove all occurrences of X on right-hand sides of rules in P.
  $P := P \setminus \{X \to \varepsilon\}$ `(elimination of` ε`-rules)`
end
return SLP $\mathbb{A} = (V, \{a, b_1, \ldots, b_{k-1}\}, P, S)$

Figure 3.4: The algorithm that computes for given integers $n_1, \ldots, n_k$ an SLP for $(\prod_{i=1}^{k-1} a^{n_i}b_i)a^{n_k}$ as described in Section 3.6.3.

3.7 Greedy

The global grammar-based compressor Greedy [7, 8] selects in each round $i \geq 1$ a maximal string of $\mathbb{A}_{i-1}$ such that $\mathbb{A}_i$ has minimal size among all possible choices of maximal strings of $\mathbb{A}_{i-1}$.

Example 3.10. *Let* $w = aaaaabbababbbaaabb$. *We have*

$\mathbb{A}_0 : S \to aaaaabbababbbaaabb$,

$\mathbb{A}_1 : S \to aaaaXabXbaaX,\ X \to abb$,

$\mathbb{A}_2 : S \to YYXabXbYX,\ X \to abb,\ Y \to aa$,

$\mathbb{A}_3 : S \to YYXZXbYX,\ X \to Zb,\ Y \to aa,\ Z \to ab$,

$\mathbb{A}_4 : S \to YAZXbA,\ X \to Zb,\ Y \to aa,\ Z \to ab,\ A \to YX$.

Note that in the first round, instead of the maximal string abb *the algorithm could also choose the maximal string* $aaabb$, *because both choices yield SLPs of minimal size* 15. *In the second round, instead of* aa *the algorithm could also choose* aaX, *because both choices yield SLPs of size* 14. *Finally, the order of the choices* ab *(round* 3*) and* YX *(round* 4*) could be switched because both choices yield SLPs of unchanged size* 14.

3.7.1 Unary inputs

In contrast to the algorithms we discussed so far, we are not able to prove a matching upper and lower bound on the approximation ratio of Greedy when the inputs are unary. The challenge is the recursive character of the algorithm combined with the discrete optimization problem that has to be solved during each round. Basically, the optimal solution obtained in some round is the input of the optimization problem during the next round. The rather technical proof of the following upper bound on the size of $\mathsf{Greedy}(a^n)$ reflects this hardness.

Proposition 3.7.1. *For all* n, *we have*

$$|\mathsf{Greedy}(a^n)| \in \mathcal{O}((\log n)^9 \cdot (\log \log n)^3).$$

First, we need to prove several lemmas that are fulfilled for any global algorithm. When we apply specific arguments for Greedy at some point, we draw attention to it. For better readability, we will use $X_0 = a$ in the following, i.e., the input is X_0^n. Further, let $\mathbb{A}_i = (N_i, \{X_0\}, P_i, S)$ be the SLP obtained by the global algorithm on input X_0^n after i rounds. Note that until the algorithm stops, we have $|N_i \setminus \{S\}| = i$ since exactly one fresh nonterminal is introduced in each round. If we quantify over the rounds of the algorithm in the following, we always implicitly mean that the statements hold until the algorithm stops. If i is mentioned without a quantification, then the statement holds for any $\mathbb{A}_i$ constructed after some round i of the algorithm.

Lemma 3.7.2. *For every i, there is a fixed order $X_i > X_{i-1} > \cdots > X_1$ of the nonterminals in $N_i \setminus \{S\}$ such that every right-hand side of a rule $(X \to w) \in P_i$ satisfies*

$$w \in X_i^* X_{i-1}^* \cdots X_1^* X_0^*.$$

Proof. We prove this property by induction. Initially, the property holds for the SLP $\mathbb{A}_0$ since $N_0 \setminus \{S\} = \emptyset$ and the only rule $S \to X_0^n$ satisfies $X_0^n \in X_0^*$. Now assume the claim is true for $\mathbb{A}_i$, i.e., each right-hand side of a rule in P_i is a word from $X_i^* X_{i-1}^* \cdots X_1^* X_0^*$. Note that any nonempty factor of such a right-hand side is a word from $X_k^* \cdots X_{j+1}^* X_j^+$ for some $i \geq k > j \geq 0$. So, assume the global algorithm chooses a maximal string $\gamma \in X_k^* \cdots X_{j+1}^* X_j^+$ in round $i+1$ and $(X \to \gamma) \in P_{i+1}$ is the corresponding new rule. We show $w \in X_i^* X_{i-1}^* \cdots X^* X_j^* \cdots X_1^* X_0^*$ for all rules $(Y \to w) \in P_{i+1}$, i.e, the order of the nonterminals after round $i+1$ is obtained by inserting the fresh nonterminal X directly before X_j in the previous order. First of all, this is obviously true for the new rule $X \to \gamma$ as well as for all rules that have not been modified during round $i+1$. It remains to check rules $(Y \to w) \in P_{i+1}$ that are obtained from a rule $(Y \to w') \in P_i$ by replacing a largest set of pairwise non-overlapping occurrences of γ in w' by the fresh nonterminal X. If $\gamma = X_j^d$ $(d \geq 2)$ is unary and X_j^ℓ is the single maximal X_j-block that occurs in w', then replacing occurrences of γ from left to right yields $X^{\ell \operatorname{div} d} X_j^{\ell \bmod d}$ as the new maximal blocks of X and X_j in w. It follows that $w \in X_i^* X_{i-1}^* \cdots X^* X_j^* \cdots X_1^* X_0^*$. If otherwise γ is not a unary word, i.e., $\gamma \in X_k^+ \cdots X_{j+1}^* X_j^+$ for $i \geq k > j \geq 0$, then w' has exactly one occurrence of γ as a factor. It follows that $w \in X_i^* X_{i-1}^* \cdots X_k^* X X_j^* \cdots X_1^* X_0^*$ and thus w satisfies the claim. This finishes the induction. □

In other words, there is at most one maximal block for each symbol on each right-hand side and the order of those blocks is the same for all rules. Similar to the case distinction in the last steps of the proof of Lemma 3.7.2, we will distinguish two types of nonterminals in the following. Let $X \to \gamma$ be the introduced rule in some round of the algorithm. If γ is unary, then we call X a unary nonterminal. Otherwise we call X *non-unary*. We categorize X_0 as a unary nonterminal, although formally X_0 is not a nonterminal. Note that the type of a nonterminal is decided when it is introduced and does not change later, i.e., even if the right-hand side of a unary nonterminal becomes non-unary during the execution of the algorithm, the type of the nonterminal stays the same. Our strategy to prove Theorem 3.7.1 is to bound the total number of occurrences of unary nonterminals and non-unary nonterminals on right-hand sides independently. It follows from Lemma 3.7.2 that every factor that occurs more than once on the right-hand side of a single rule is unary. The following lemma is a direct consequence of that fact.

Lemma 3.7.3. *Every non-unary nonterminal occurs at most once on the right-hand side of each rule at any time of a global algorithm.*

Corollary 3.7.4. *If a unary nonterminal X is introduced and $X \to Z^d$ with $d \geq 2$ is the corresponding rule for X, then Z is a unary nonterminal.*

As a next step, we bound the number of rules that contain a unary nonterminal X on the right-hand side. To do so, we use the following definition for all nonterminals and X_0. Let

$$\#_i(X) = |\{(Y \to w) \in P_i \mid X \text{ occurs in } w\}|$$

be the number of rules of $\mathbb{A}_i$ where X occurs on the right-hand side. The next two lemmas describe how $\#_i$ evolves depending on the type of the introduced nonterminal.

Lemma 3.7.5. *If a non-unary nonterminal X is introduced in some round $i+1$, then for every $X' \neq X$ (including X_0), we have $\#_{i+1}(X') \leq \#_i(X')$.*

Proof. In order to prove this point, we use that all rules $(Y \to w) \in P_i$ satisfy $w \in X_i^* X_{i-1}^* \cdots X_1^* X_0^*$ (Lemma 3.7.2). Since X is non-unary, the chosen maximal string γ satisfies $\gamma \in X_k^+ \cdots X_{j+1}^* X_j^+$ for $i \geq k > j \geq 0$. If a nonterminal X' does not occur in γ, then $\#_{i+1}(X') = \#_i(X')$. So assume the nonterminal X' occurs in γ, i.e., $X' = X_t$ for $t \in [j,k]$. Note first that X_t occurs on the right-hand side of the fresh rule $(X \to \gamma) \in P_{i+1}$. It follows that in order to prove the claimed result, we must show that for at least one rule $(Y \to w) \in P_i$ such that X_t occurs in w, all occurrences of X_t must disappear, i.e., X_t does not occur in w' for $(Y \to w') \in P_{i+1}$. Let

$$M = \{Y \in N_i \mid (Y \to w) \in P_i \text{ and } \gamma \text{ is a factor of } w\} \subseteq N_i$$

be the set of nonterminals of $\mathbb{A}_i$ where the corresponding rule is modified in round $i+1$. Note that $|M| \geq 2$ since a maximal string occurs at least twice on all right-hand sides and γ occurs at most once as factor of each rule since X is non-unary (Lemma 3.7.3). If $k < t < j$ then for all $Y \in M$ and $(Y \to w') \in P_{i+1}$, the nonterminal X_t does not occur in w' anymore since the complete X_t-block (among other symbols) has been replaced. This means that $\#_{i+1}(X_t) < \#_i(X_t)$ since one new rule contains X_t on the right-hand side, while for at least two rules the occurrences of X_t has been removed in round $i+1$. If otherwise $t = k$ or $t = j$, then the same argument fails since for $Y \in M$ and $(Y \to w') \in P_{i+1}$, the right-hand side w' could still contain X_t since it is not necessarily true that the complete X_t-block has been replaced. But due to the properties of a maximal string, we show that X_t does not occur in w' for at least one $Y \in M$ and $(Y \to w') \in P_{i+1}$. Towards a contradiction, assume X_t occurs in w' for all $Y \in M$ and $(Y \to w') \in P_{i+1}$. That means that for all $Y \in M$ and $(Y \to w) \in P_i$, the length of the maximal X_t-block that occurs in w is strictly larger then the length of the maximal X_t-block that occurs in γ. If $t = k$, if follows that $X_k\gamma$ is a factor of w for all $Y \in M$ and $(Y \to w) \in P_i$, and symmetrically, if $t = j$ then γX_j is a factor of w for all $Y \in M$ and $(Y \to w) \in P_i$. This contradicts the property that no strictly longer string than γ occurs at least as often on the right-hand sides of the rules. It follows that in this case $\#_{i+1}(X_t) \leq \#_i(X_t)$, which finishes the proof. □

Lemma 3.7.6. *If a unary nonterminal X is introduced in some round $i+1$ and $X \to Z^d$ with $d \geq 2$ is the corresponding rule, then $\#_{i+1}(X) \leq \#_i(Z)$ and $\#_{i+1}(Z) \leq \#_i(Z) + 1$.*

Proof. Both points are straightforward: A rule $(Y \to w) \in P_{i+1}$ only contains X on the right-hand side if Z^d is a factor of w' for $(Y \to w') \in P_i$, which shows $\#_{i+1}(X) \leq \#_i(Z)$. For the second point, note that if $(Y \to w) \in P_i$ does not contain Z on the right-hand side, then the same is true for (the unchanged rule) $(Y \to w) \in P_{i+1}$. The only rule where Z occurs new is the fresh rule $X \to Z^d$, and thus $\#_{i+1}(Z) \leq \#_i(Z) + 1$. □

So far, we have showed that when a unary nonterminal X is introduced and $X \to Z^d$ with $d \geq 2$ is the corresponding rule, then Z is a unary nonterminal as well (Corollary 3.7.4). Further, we argued that introducing a non-unary nonterminal does not increase the number of rules where a unary nonterminal occurs on the right-hand side (Lemma 3.7.5). It follows that we can upper bound the number of unary nonterminals and the number of rules where those nonterminals occur on right-hand sides independently of the non-unary nonterminals.

To do so, we inductively define a binary tree T_i that describes how the unary nonterminals evolve until $\mathbb{A}_i$ is reached. All nodes in the tree are labeled by (X, k), where X is a unary nonterminal and k is an upper bound on $\#_j(X)$ for some j.

(i) Initially, the tree T_0 only contains a single node that is labeled by $(X_0, 1)$.

(ii) If a rule $X \to Z^d$ is introduced in round $i+1$ for some $d \geq 2$, then we update T_i to T_{i+1} by adding two children to the unique leaf that is labeled by (Z, k) for some k. The fresh left child is labeled by $(Z, k+1)$ and the fresh right child is labeled by (X, k) as depicted on the left of Figure 3.5.

(iii) If otherwise a non-unary nonterminal is introduced in round $i+1$, then $T_{i+1} = T_i$, i.e., non-unary nonterminals are ignored.

The initial tree T_0 reflects that the only unary nonterminal of $\mathbb{A}_0$ is X_0 and $\#_0(X_0) = 1$. If the tree is modified according to point (ii) of the definition, this refers to Lemma 3.7.6, where $\#_{i+1}(X) \leq \#_i(Z)$ and $\#_{i+1}(Z) \leq \#_i(Z) + 1$ is shown when a rule $X \to Z^d$ for $d \geq 2$ is introduced.

The *level* of a node is the length of the path from the node to the root. For a unary nonterminal $X \in N_i$, we denote by $\mathsf{level}_i(X)$ the level of the unique leaf of T_3 that is labeled by (X, k) for some k.

Example 3.11. *Assume that the first three rules introduced by a global algorithm are $X_2 \to X_0^{d_1}$ in the first round, $X_1 \to X_0^{d_2}$ in the second round and $X_3 \to X_2^{d_3}$ in the third round for $d_1, d_2, d_3 \geq 2$. The tree T_3 that corresponds to these introduced rules is depicted on the right of Figure 3.5. The indices for the introduced nonterminals are chosen such that the ordering of the nonterminals in $N_3 \setminus \{S\}$ (see Lemma 3.7.2) is $X_3 > X_2 > X_1$, i.e., all right-hand sides of rules are contained in $X_3^* X_2^* X_1^* X_0^*$. The corresponding SLPs $\mathbb{A}_0$, $\mathbb{A}_1$, $\mathbb{A}_2$ and $\mathbb{A}_3$ are*

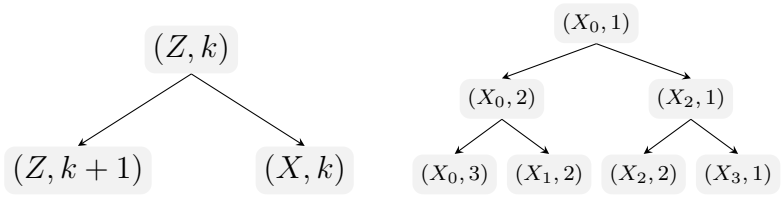

Figure 3.5: On the left, the general pattern that is applied during the construction of T_i is illustrated, where the split refers to a rule $X \to Z^d$ that has been introduced by the global algorithm. On the right, the tree T_3 that corresponds to the introduced rules of Example 3.11 is shown.

depicted in the following, where we simply use $*$ instead of the exact exponents of the symbols due to better readability.

$\mathbb{A}_0$: $S \to X_0^*$

$\mathbb{A}_1$: $S \to X_2^* X_0^*$, $X_2 \to X_0^*$

$\mathbb{A}_2$: $S \to X_2^* X_1^* X_0^*$, $X_2 \to X_1^* X_0^*$, $X_1 \to X_0^*$

$\mathbb{A}_3$: $S \to X_3^* X_2^* X_1^* X_0^*$, $X_2 \to X_1^* X_0^*$, $X_1 \to X_0^*$, $X_3 \to X_2^*$

Note that in this example, we have $\#_3(X_0) \leq 3$, $\#_3(X_2) \leq 2$, $\#_3(X_1) \leq 2$, $\#_3(X_3) \leq 1$ and this is exactly the information contained in the second components of the leaf-labels in T_3. Further, we have $\mathsf{level}_3(X_i) = 2$ for $i \in [0,3]$ in this example.

The following lemma is a direct consequence of the fact that the maximal k that occurs for some label (X,k) is incremented from one level to the next level (as described in Lemma 3.7.6).

Lemma 3.7.7. *For each node of T_i at level m that is labeled by (X,k) for some unary nonterminal $X \in N_i$, we have $k \leq m+1$. Let further $X \in N_i$ be a unary nonterminal, then we have $\#_i(X) \leq \mathsf{level}_i(X) + 1$.*

So far, we provided information about the number of rules where a unary nonterminal occurs. In the following, we move on to the total number of occurrences of a unary nonterminal on all right-hand sides. We denote by $t_i(X)$ the total number of occurrences of X on right-hand sides of rules in $\mathbb{A}_i$. We have $\#_i(X) \leq t_i(X)$ by the definition of both functions and for a non-unary nonterminal X, we have $\#_i(X) = t_i(X)$ due to Lemma 3.7.3.

Lemma 3.7.8. *Let $X \to \gamma$ be the rule that is introduced in some round $i+1$ and let $M = \{Y \in N_i \mid Y \text{ occurs in } \gamma\}$. We have*

(i) $t_{i+1}(Y) \leq t_i(Y)$ for all $Y \in N_i = N_{i+1} \setminus \{X\}$, and

(ii) $\sum_{Y \in M} t_{i+1}(Y) + t_{i+1}(X) \leq \sum_{Y \in M} t_i(Y)$.

Proof. Point (i) is straightforward: For $Y \in M$ let Y^ℓ be the maximal Y-block that occurs as a factor in γ for some $\ell \geq 1$. Replacing γ on right-hand sides

yields that at least two occurrences of Y^ℓ are eliminated while only Y^ℓ is added as a part of the fresh rule $X \to \gamma$. If otherwise $Y \notin M$, then $t_{i+1}(Y) = t_i(Y)$ because the occurrences of Y are not effected by the new rule.

Point (ii) is also based on a simple observation. Note that $\sum_{Y \in M} t_i(Y)$ describes the part of the SLP $\mathbb{A}_i$ that is effected by the replacement of γ in round $i+1$, and $\sum_{Y \in M} t_{i+1}(Y) + t_{i+1}(X)$ is the size of that part in $\mathbb{A}_{i+1}$ after the occurrences of γ are replaced by X plus the new occurrences in the introduced rule. All other parts of $\mathbb{A}_i$ are not effected by the fresh rule. Now the properties of a maximal string ensure that $|\mathbb{A}_{i+1}| \leq |\mathbb{A}_i|$. The extreme case where γ has length two and occurs only twice non-overlapping on the right-hand sides of rules in $\mathbb{A}_i$ satisfies $|\mathbb{A}_{i+1}| = |\mathbb{A}_i|$, all other cases even satisfy $|\mathbb{A}_{i+1}| < |\mathbb{A}_i|$. Point (ii) directly follows. □

Our next goal is to bound $t_i(X)$ in dependence on $\mathsf{level}_i(X)$ for a unary nonterminal X. To do so, we now apply specific arguments for Greedy. Recall that Greedy selects a maximal string that minimizes the size of the obtained SLP in each round.

Lemma 3.7.9. *Let $Z \in N_i \cup \{X_0\}$ be a unary nonterminal and assume a rule $X \to Z^d$ for some $d \geq 2$ is introduced by* Greedy *in round $i+1$. We have*

$$t_{i+1}(X) + t_{i+1}(Z) \leq 2\sqrt{t_i(Z)}\sqrt{\#_i(Z)+1} + 1.$$

Proof. If a unary nonterminal X and a rule $X \to Z^d$ with $d \geq 2$ are introduced in round $i+1$, then the choice of d only depends on the maximal Z-blocks occurring on all right-hand sides of rules in $\mathbb{A}_i$ since the remaining part of $\mathbb{A}_i$ does not change. Assume $\#_i(Z) = k$ and let $\ell_1, \ldots, \ell_k$ be the lengths of the maximal Z-blocks occurring on right-hand sides of $\mathbb{A}_i$, i.e., $\sum_{j=1}^{k} \ell_j = t_i(Z)$. Then Greedy minimizes $t_{i+1}(X) + t_{i+1}(Z) = d + \sum_{j=1}^{k} (\ell_j \operatorname{div} d) + (\ell_j \bmod d)$, where d is the size of the fresh rule $X \to Z^d$ and for each $j \in [1,k]$ a maximal block Z^{ℓ_j} on the right-hand side of a rule in $\mathbb{A}_i$ is transformed into $X^{\ell_j \operatorname{div} d} Z^{\ell_j \bmod d}$. Due to the greedy nature of the algorithm, the following equation holds for all $d \geq 1$:

$$\begin{aligned} t_{i+1}(X) + t_{i+1}(Z) &\leq d + \sum_{j=1}^{k} (\ell_j \operatorname{div} d) + (\ell_j \bmod d) \\ &\leq d + \sum_{j=1}^{k} \frac{\ell_j}{d} + k(d-1) \\ &= d + \frac{t_i(Z)}{d} + k(d-1). \end{aligned}$$

Note that the chosen maximal string has length at least 2, but the upper bound also holds for $d = 1$ since in this case we have $t_{i+1}(X) + t_{i+1}(Z) \leq t_i(Z)$ due to

Lemma 3.7.8 (point (ii)). If we apply $d = \left\lceil \sqrt{t_i(Z)}/\sqrt{k+1} \right\rceil$, we get

$$\begin{aligned} t_{i+1}(X) + t_{i+1}(Z) &\leq \left\lceil \frac{\sqrt{t_i(Z)}}{\sqrt{k+1}} \right\rceil + \frac{t_i(Z)}{\left\lceil \frac{\sqrt{t_i(Z)}}{\sqrt{k+1}} \right\rceil} + k \left(\left\lceil \frac{\sqrt{t_i(Z)}}{\sqrt{k+1}} \right\rceil - 1 \right) \\ &\leq \frac{\sqrt{t_i(Z)}}{\sqrt{k+1}} + 1 + \frac{t_i(Z)}{\frac{\sqrt{t_i(Z)}}{\sqrt{k+1}}} + k \left(\frac{\sqrt{t_i(Z)}}{\sqrt{k+1}} \right) \\ &= (k+1) \frac{\sqrt{t_i(Z)}}{\sqrt{k+1}} + \frac{t_i(Z) \cdot \sqrt{k+1}}{\sqrt{t_i(Z)}} + 1 \\ &= 2\sqrt{t_i(Z)}\sqrt{k+1} + 1. \end{aligned}$$

Together with $k = \#_i(Z)$ this proves the lemma. □

The following lemma is essential for the proof of Theorem 3.7.1 since we bound the total number of occurrences of a unary nonterminal in dependence on its level.

Lemma 3.7.10. *Let $X \in N_i \cup \{X_0\}$ be a unary nonterminal with $\mathsf{level}_i(X) = m$. We have*

$$t_i(X) \leq 2^{2-2^{1-m}} n^{2^{-m}} \prod_{j=1}^{m} (m+2-j)^{2^{-j}}. \tag{3.5}$$

Proof. We prove the lemma by induction on $m = \mathsf{level}_i(X)$ and we start with $m = 0$. The only SLP $\mathbb{A}_i$ that contains a unary nonterminal X such that $\mathsf{level}_i(X) = 0$ is the initial SLP $\mathbb{A}_0$ and the unary nonterminal is $X = X_0$. Note that the maximal string γ chosen by any global algorithm in the first round on input X_0^n trivially satisfies $\gamma \in X_0^*$ and thus the two unary nonterminals of $\mathbb{A}_1$ have level one. We have $t_0(X_0) = n$ and this is exactly what comes out when $m = 0$ is used on the right side of equation (3.5) (the empty product is considered to be 1).

Now assume any unary nonterminal that has level m satisfies the claimed bound and we consider a unary nonterminal X such that $\mathsf{level}_i(X) = m + 1 > 0$ for some i. It follows from the definition that there is a leaf node at level $m+1$ in T_i that is labeled by (X, k) for some k. There are two cases that need to be distinguished. Either this leaf is a left child or a right child of its parent node. Assume that (X, k) is the label of a right child and let $(Z, k+1)$ be the label of the left sibling of that node. In order to prove both cases simultaneously, we prove the upper bound for X and for Z, i.e., we use Z to cover the second case that the node is a left child. The parent node of $(Z, k+1)$ and (X, k) is labeled by (Z, k) (see Figure 3.5 on the left). Let $i' < i$ be the maximal i' such that $\mathsf{level}_{i'}(Z) = m$, i.e., $X \to Z^d$ for $d \geq 2$ is the introduced rule in round $i'+1$ and (Z, k) is the label of a leaf at level m in $T_{i'}$. By induction, we have $t_{i'}(Z) \leq 2^{2-2^{1-m}} n^{2^{-m}} \prod_{j=1}^{m} (m+2-j)^{2^{-j}}$. Now by Lemma 3.7.9, we have

$$t_{i'+1}(X) + t_{i'+1}(Z) \leq 2\sqrt{t_{i'}(Z)}\sqrt{\#_{i'}(Z)+1} + 1.$$

Together with $\#_{i'}(Z) \leq m+1$ (Lemma 3.7.7), this yields

$$\begin{aligned}
t_{i'+1}(X) + t_{i'+1}(Z) &\leq 2\left(2^{2-2^{1-m}} n^{2^{-m}} \prod_{j=1}^{m} (m+2-j)^{2^{-j}}\right)^{\frac{1}{2}} (m+2)^{\frac{1}{2}} + 1 \\
&= 2^{2-2^{-m}} n^{2^{-m-1}} \prod_{j=1}^{m} (m+2-j)^{2^{-j-1}} (m+2)^{2^{-1}} + 1 \\
&= 2^{2-2^{-m}} n^{2^{-m-1}} \prod_{j=2}^{m+1} (m+2-j+1)^{2^{-j}} (m+2)^{2^{-1}} + 1 \\
&= 2^{2-2^{-m}} n^{2^{-m-1}} \prod_{j=1}^{m+1} (m+2-j+1)^{2^{-j}} + 1.
\end{aligned}$$

Using the information $t_{i'+1}(X) \geq 2$ (there are at least two non-overlapping occurrences of a maximal string) and $t_{i'+1}(Z) \geq 2$ (the fresh rule contains Z at least twice) yield the claimed upper bound on $t_{i'+1}(X)$ and $t_{i'+1}(Z)$. Finally, this upper bound holds for $i \geq i'+1$ due to Lemma 3.7.8 (point (i)). □

Corollary 3.7.11. *Let $X \in N_i \cup \{X_0\}$ be a unary nonterminal with $\mathsf{level}_i(X) = m$. We have*

$$t_i(X) \leq 4n^{2^{-m}}(m+2).$$

Proof. Note that $2^{2-2^{-m}} \leq 4$ for all $m \geq 0$. We upper bound the right side of equation (3.5) (Lemma 3.7.10) as follows:

$$\begin{aligned}
2^{2-2^{1-m}} n^{2^{-m}} \prod_{j=1}^{m} (m+2-j)^{2^{-j}} &\leq 4n^{2^{-m}} \prod_{j=1}^{m} (m+2)^{2^{-j}} \\
&= 4n^{2^{-m}} (m+2)^{\sum_{j=1}^{m} 2^{-j}} \\
&\leq 4n^{2^{-m}} (m+2)
\end{aligned}$$

□

What we achieved so far is to bound the total size $t_i(X)$ that a unary nonterminal X contributes on right-hand sides of the rules in dependence on $\mathsf{level}_i(X)$. Next we bound the size that non-unary nonterminals contribute to $|\mathbb{A}_i|$ in dependence on the levels of all unary nonterminals. To do so, we need the following definitions. Let $R_i(X)$ be the number of distinct right neighbors of X (which are not equal to X) on right-hand sides plus the number of occurrences of X as the last symbol of a right-hand side in $\mathbb{A}_i$, i.e.,

$$\begin{aligned}
R_i(X) =& |\{A \in N_i \cup \{X_0\} \mid A \neq X,\ XA \text{ occurs on a right-hand side in } \mathbb{A}_i\}| \\
&+ |\{(Y \to vX) \in P_i \mid Y \in N_i,\ v \in (N_i \cup \{X_0\})^*\}|.
\end{aligned}$$

Let $L_i(X)$ be the number of distinct left neighbors of X (which are not equal to X) on right-hand sides plus the number of occurrences of X as the first symbol of a right-hand side in $\mathbb{A}_i$, i.e.,

$$\begin{aligned} L_i(X) =& |\{A \in N_i \cup \{X_0\} \mid A \neq X,\ AX \text{ occurs on a right-hand side in } \mathbb{A}_i\}| \\ &+ |\{(Y \to Xv) \in P_i \mid Y \in N_i,\ v \in (N_i \cup \{X_0\})^*\}|. \end{aligned}$$

Let further $f_i(X) = \#_i(X) - R_i(X)$ and $g_i(X) = \#_i(X) - L_i(X)$. Note that $R_i(X) \leq \#_i(X)$ and $L_i(X) \leq \#_i(X)$ since for each right-hand side of a rule there is at most one right (respectively, left) neighbor $A \neq X$ for some occurrence of X due to Lemma 3.7.2 and each right-hand side can contain X at most once as the last (respectively, first) symbol. Further, $\#_i(X) = R_i(X)$ means that all maximal X-blocks on right-hand sides are either at the end of the right-hand side or followed by a distinct symbol. Similarly, $\#_i(X) = L_i(X)$ means that all maximal X-blocks on right-hand sides are either at the beginning of the right-hand side or preceded by a distinct symbol. The following lemmas describe how the functions $f_i(X)$ and $g_i(X)$ evolve.

Lemma 3.7.12. *If $X \in N_i \cup \{X_0\}$ then $f_{i+1}(X) \leq f_i(X)$. If a non-unary maximal string $\gamma = Xw$ is selected in round $i+1$ for some $w \in (N_i \cup \{X_0\})^+$, then $f_{i+1}(X) < f_i(X)$.*

Proof. Let $Y \to \gamma$ be the introduced rule in round $i+1$. If X does not occur in γ, then it is straightforward to see that $f_{i+1}(X) \leq f_i(X)$ since $\#_{i+1}(X) = \#_i(X)$ and $R_{i+1}(X) \geq R_i(X)$. The fresh nonterminal Y could be a new right neighbor for some occurrences of X, but all occurrences of X which have this new right neighbor Y in $\mathbb{A}_{i+1}$ shared the same right neighbor in $\mathbb{A}_i$ (the first symbol of γ).

If otherwise X occurs in γ, then first assume $\gamma = X^d$ for some $d \geq 2$. Note that replacing an occurrence of γ on the right-hand side of a rule $(Z \to u) \in P_i$ either removes all occurrences of X on this right-hand side (in case u contains a maximal X-blocks of length $k \cdot d$ for some integer $k \geq 1$) or the right neighbor of the maximal X-block in u does not change in the modified rule $(Z \to u') \in P_{i+1}$ since occurrences of γ are replaced from left to right. It follows that the only way to obtain $R_{i+1}(X) < R_i(X)$ is to remove all occurrences of X on a right-hand side, but then $\#_i(X)$ decreases by the same value. Additionally, the fresh rule $Y \to X^d$ adds a new right-hand side to $\#_{i+1}(X)$, but since X is the last symbol on this right-hand side it follows that $R_{i+1}(X)$ is incremented as well. Together this yields $f_{i+1}(X) \leq f_i(X)$ in this case.

It remains the case where γ is non-unary and X occurs in γ. In this case, we have $\#_{i+1}(X) \leq \#_i(X)$ due to Lemma 3.7.5 and γ occurs at most once on each right-hand side due to Lemma 3.7.2. But again, the only way to reduce $R_{i+1}(X)$ compared to $R_i(X)$ is to remove all occurrences of X on a right-hand side, but then again $\#_{i+1}(X)$ decreases by the same value. This yields $f_{i+1}(X) \leq f_i(X)$.

Assume now a non-unary maximal string $\gamma = Xw$ for some $w \in (N_i \cup \{X_0\})^+$ is selected in round $i+1$. We show that $f_{i+1}(X) < f_i(X)$. If X only occurs in the fresh rule in $\mathbb{A}_{i+1}$ after occurrences of γ are replaced on all right-hand sides in $\mathbb{A}_i$, i.e., all modified rules do not contain X anymore, then $\#_{i+1}(X) < \#_i(X)$ because

at least two right-hand sides do not contain X as a factor anymore while only the fresh rule $Y \to \gamma$ adds a new right-hand side which contains X to $\#_{i+1}(X)$. Further, we have $R_{i+1}(X) = R_i(X)$ in this case and thus $f_{i+1}(X) < f_i(X)$ because all rules where the maximal X-block is removed shared the same right neighbor due to the fact that $\gamma = Xw$ is the chosen maximal string. But Xw still occurs in the fresh rule of $\mathbb{A}_{i+1}$ and thus $R_{i+1}(X) = R_i(X)$. If otherwise at least one of the modified rules still contains X on the right-hand side, then XY is a factor of this right-hand side in $\mathbb{A}_{i+1}$ after the replacement of γ. It follows that $R_{i+1}(X) > R_i(X)$ in this case and thus $f_{i+1}(X) < f_i(X)$ because each distinct right neighbor of X in $\mathbb{A}_i$ is still a right neighbor of X in $\mathbb{A}_{i+1}$ as argued above, but additionally XY is new since Y is a fresh nonterminal. □

The same result does not hold for $g_i(X)$. In particular, $g_{i+1}(X) > g_i(X)$ is possible when a rule $Y \to X^d$ is introduced in round $i+1$ for some $d \geq 2$ due to the assumption that global algorithms replace occurrences of the maximal string from left to right. For example, assume that AX^4, BX^7 and CX^{10} are the maximal X-blocks on right-hand sides of $\mathbb{A}_i$ including distinct left neighbors for each X-block (A, B and C). Therefore, we have $\#_i(X) = 3$, $L_i(X) = 3$ and thus $g_i(X) = 0$ in this example. If now a rule $Y \to X^3$ is introduced, then this yields AYX, BY^2X and CY^3X after replacing X^3. Hence we have $\#_{i+1}(X) = 4$, $L_{i+1}(X) = 2$ and thus $g_{i+1}(X) = 2$. We show in the following lemma that this is the only case where $g_{i+1}(X) > g_i(X)$ occurs.

Lemma 3.7.13. *Let $X \in N_i \cup \{X_0\}$. If a rule $Y \to \gamma$ is introduced in round $i+1$ such that $\gamma \notin X^+$, then $g_{i+1}(X) \leq g_i(X)$. If a non-unary maximal string $\gamma = Xw$ is selected in round $i+1$ for some $w \in (N_i \cup \{X_0\})^+$, then $g_{i+1}(X) < g_i(X)$*

Proof. The arguments are similar to the corresponding cases in Lemma 3.7.12. Let $Y \to \gamma$ be the introduced rule in round $i+1$. If X does not occur in γ, then $g_{i+1}(X) \leq g_i(X)$ since $\#_{i+1}(X) = \#_i(X)$ and $L_{i+1}(X) \geq L_i(X)$. The fresh nonterminal Y could be a new left neighbor for some occurrences of X in $\mathbb{A}_{i+1}$, but all of these occurrences of X shared the same left neighbor in $\mathbb{A}_i$ (the last symbol of γ).

If otherwise γ is non-unary and contains X, then we have $\#_{i+1}(X) \leq \#_i(X)$ due to Lemma 3.7.5 and γ occurs at most once on each right-hand side due to Lemma 3.7.2. The only way to obtain $L_{i+1}(X) < L_i(X)$ is again to remove all occurrences of X on a right-hand side, but then $\#_{i+1}(X)$ decreases by the same value. Note that due to the assumption that γ is non-unary, it is not possible to modify two (or more) rules such that the maximal X-blocks have different left neighbors in $\mathbb{A}_i$ and after the replacement those X-blocks share the same left neighbor in $\mathbb{A}_{i+1}$. This yields $g_{i+1}(X) \leq g_i(X)$.

Assume now a non-unary maximal string $\gamma = wX$ is selected in round $i+1$ for some word $w \in (N_i \cup \{X_0\})^+$. We show $g_{i+1}(X) < g_i(X)$. If X only occurs in the fresh rule in $\mathbb{A}_{i+1}$, i.e., X does not occur in the modified rules, then we have $\#_{i+1}(X) < \#_i(X)$ because at least two right-hand sides do not contain X as a factor anymore while only the fresh rule $Y \to \gamma$ adds a new right-hand

side which contains X to $\#_{i+1}(X)$. Moreover, we have $L_{i+1}(X) = L_i(X)$ in this case because all rules where the maximal X-block is removed shared the same left neighbor since $\gamma = wX$ is the selected non-unary maximal string. But wX still occurs on the right-hand side of the fresh rule of $\mathbb{A}_{i+1}$. It follows that $g_{i+1}(X) < g_i(X)$. If otherwise at least one of the modified rules still contains X on the right-hand side, then YX is a factor of this right-hand side in $\mathbb{A}_{i+1}$ after the replacement of γ. It follows $L_{i+1}(X) > L_i(X)$ and thus $g_{i+1}(X) < g_i(X)$ because each distinct left neighbor of X in $\mathbb{A}_i$ is still a left neighbor of X in $\mathbb{A}_{i+1}$ for some occurrence of X. Additionally, Y is a new left neighbor. □

Lemma 3.7.14. *Let $U_i = \{X \in N_i \cup \{X_0\} \mid X \text{ is a unary nonterminal}\}$ be the set of unary nonterminals and let $M_i = N_i \setminus U_i$ be the set of all non-unary nonterminals. We have*

$$\sum_{X \in M_i} t_i(X) \leq \sum_{X \in U_i} (\mathsf{level}_i(X) + 1) \cdot (\mathsf{level}_i(X) + 1 + (\mathsf{level}_i(X) + 1)^2)$$

Proof. Let $s(i) = \sum_{X \in M_i} t_i(X)$ be the total size that all non-unary nonterminals contribute to the size of $\mathbb{A}_i$. We first bound the number of rounds where the function s increases, i.e., we bound $|\{j \in [0, i-1] \mid s(j+1) > s(j)\}|$. If a unary nonterminal is introduced in some round $j+1$, then $s(j+1) = s(j)$, i.e., we can ignore those rules. So consider some round $j+1$ where a non-unary nonterminal X is introduced and let $X \to \gamma$ be the introduced rule. Let further $M = \{Z \in N_j \mid Z \text{ occurs in } \gamma\}$ be the set of nonterminals that occur at least once in γ. We first show that if $k = |M_j \cap M| \geq 2$, then $s(j+1) \leq s(j)$. In other words, if two non-unary nonterminals occur in γ, then $s(j+1) \leq s(j)$. Let r be the number of rules $(Z \to w) \in P_j$ such that γ is a factor of w. Recall that non-unary factors and nonterminals occur at most once on the right-hand side of a single rule (Lemma 3.7.3). We have $s(j+1) - s(j) = k + r - k \cdot r$ because the fresh non-unary nonterminal X occurs now on r right-hand sides, γ contains k non-unary nonterminals which occur exactly once in γ each, and the replacement of γ on right-hand sides deletes those k nonterminals on r right-hand sides. We have $r \geq 2$ (due to the properties of a maximal string) which together with $k \geq 2$ yields $s(j+1) - s(j) \leq 0$. Hence we can assume $k = |M_j \cap M| \leq 1$. The maximal string γ has length $|\gamma| \geq 2$ and is not unary, so the first and the last symbol of γ are different and at least one of both is unary due to our assumption that at most one non-unary nonterminal occurs in γ. Let Y be this unary nonterminal and assume Y is the first symbol, i.e, the non-unary maximal string is $\gamma = Yw$ for some (non-empty) w. Afterwards we discuss the case where Y is the last symbol, i.e., $\gamma = wY$.

We bound the number of rounds where a non-unary maximal string $\gamma = Yw$ is selected for some w. Let $j_0 \leq i$ be the round where the unary nonterminal Y has been introduced. We have

$$f_{j_0}(Y) \leq \#_{j_0}(Y) \leq \mathsf{level}_{j_0}(Y) + 1 \leq \mathsf{level}_i(Y) + 1$$

due to Lemma 3.7.7 and $\mathsf{level}_j(Y) \leq \mathsf{level}_i(Y)$ for all $j \leq i$. Further, we have $f_{j+1}(Y) \leq f_j(Y)$ for $j \in [j_0, i-1]$ by Lemma 3.7.12. Even more, if Yw is

the selected non-unary maximal string in round $j+1$ for some w, then we have $f_{j+1}(Y) < f_j(Y)$ again by Lemma 3.7.12. It follows that after at most $\mathsf{level}_i(Y)+1$ many rounds where the chosen maximal string is non-unary and has the form Yw for some (non-empty) w, we have $f_i(Y) = 0$. In this case all maximal Y blocks have distinct right neighbors or occur at the end of a right-hand side. Hence there is no possibility to select a non-unary maximal string Yw anymore.

Now we similarly bound the number of rounds such that a non-unary maximal string $\gamma = wY$ is selected for some w. But care has to be taken in this case, because it is possible that $g_{j+1}(Y) > g_j(Y)$ when a rule $X' \to Y^d$ for $d \geq 2$ is introduced in round $j+1$ as explained above. Fortunately, rules of this form (the selected maximal string is from Y^+) are introduced at most $\mathsf{level}_i(Y)$ times up to round i by the definition of $\mathsf{level}_i(Y)$. Let $j_0 \leq i$ be the round where the unary nonterminal Y has been introduced. We have

$$g_j(Y) \leq \#_j(Y) \leq \mathsf{level}_j(Y) + 1 \leq \mathsf{level}_i(Y) + 1$$

for each $j \in [j_0, i]$ due to Lemma 3.7.7 and $\mathsf{level}_j(Y) \leq \mathsf{level}_i(Y)$ for all $j \leq i$. Further, if the selected maximal string in round $j+1$ ($j \geq j_0$) is not from Y^+, we have $g_{j+1}(Y) \leq g_j(Y)$ due to Lemma 3.7.13. Moreover, if the maximal string γ is non-unary and $\gamma = wY$ for some (non-empty) w, then $g_{j+1}(Y) < g_j(Y)$. It follows that between two rounds where maximal strings from Y^+ are selected, there are at most $\mathsf{level}_i(Y)+1$ many rounds where a non-unary maximal string of the form wY is chosen because then $g_j(Y) = 0$ is reached (for some j) and thus all maximal Y blocks have distinct left neighbors or occur at the beginning of a right-hand side. Hence no non-unary string of the form wY for some w occurs twice on right-hand sides. Since maximal strings from Y^+ are chosen at most $\mathsf{level}_i(Y)$ times up to round i, it follows that the number of rounds where a non-unary maximal string of the form wY for some w is selected is at most $(\mathsf{level}_i(Y)+1)^2$.

Further, the maximal increase $\max\{s(j+1) - s(j) \mid j \in [0, i-1]\}$ in a single round is at most $\mathsf{level}_i(Y)+1$, because the fresh non-unary nonterminal occurs in $\mathbb{A}_{j+1}$ on at most $\#_j(Y) \leq \mathsf{level}_j(Y)+1 \leq \mathsf{level}_i(Y)+1$ many right-hand sides of rules for any $j \leq i$ and the total number of occurrences of all other (non-unary) nonterminals does not increase (Lemma 3.7.8, point (i)).

We conclude that for each unary nonterminal Y, at most

$$\mathsf{level}_i(Y) + 1 + (\mathsf{level}_i(Y)+1)^2$$

rules are introduced such that the non-unary maximal string γ satisfies $\gamma = Yw$ or $\gamma = wY$ for some w and each of those rules increases the total size that non-unary nonterminals contribute by at most $\mathsf{level}_i(Y)+1$. In all other cases, we showed that the size that non-unary nonterminals contribute does not increase. □

Now we are able the prove Proposition 3.7.1.

Proof of Proposition 3.7.1. Let $\mathbb{A}_f = \mathsf{Greedy}(X_0^n)$ be the final SLP obtained by Greedy, i.e., after f rounds the algorithm stops because $\mathbb{A}_f$ has no maximal

string. First, we want to bound the level of unary nonterminals occurring in $\mathbb{A}_f$. Assume there is a unary nonterminal X such that $\mathsf{level}_i(X) = \lceil \log \log n \rceil$ after some round $i \leq f$ of the algorithm. By Corollary 3.7.11, we have

$$t_i(X) \leq 4n^{2^{-\log \log n}}(\log \log n + 3) \leq 8(\log \log n + 3).$$

Consider the unique leaf node v_X in the tree T_i which has level $\lceil \log \log n \rceil$ and label (X, k) for some k. If in some round $j \in [i+1, f]$, two children with labels $(X, k+1)$ and (Y, k) are attached to v_X, i.e., the introduced rule in round j is $Y \to X^d$ for some $d \geq 2$, then we have $t_j(X) + t_j(Y) \leq t_{j-1}(X) \leq t_i(X)$ by Lemma 3.7.8. To be more specific, if the length of the chosen maximal string X^d is exactly $d = 2$ and this maximal string XX occurs exactly twice without overlap in $\mathbb{A}_{j-1}$, then we have $t_j(X) + t_j(Y) = t_{j-1}(X)$ (and $|\mathbb{A}_j| = |\mathbb{A}_{j-1}|$). Note that in this case, there does not exist a maximal string X^d or Y^d of $\mathbb{A}_k$ for all $k \in [j, f]$ (since Y occurs only twice and XX does not occur on right-hand sides of $\mathbb{A}_j$), i.e., the children of the node v_X in T_k are leaves for $k \in [j, f]$. Otherwise, if the maximal string has length $d \geq 3$ or occurs at least three times without overlap, then we have $t_j(X)+t_j(Y) < t_{j-1}(X)$ since $|\mathbb{A}_j| < |\mathbb{A}_{j-1}|$ holds in this setting. This means that when a new branch occurs in the tree T_j for some j, then the new children of the branching node are either leaves of the final tree T_f or the corresponding nonterminals contribute strictly less to the size of the current SLP than the nonterminal which corresponds to the parent node did before the branch. We can iterate this argument to the children of the children of v_X and so on, i.e., if we consider the subtree rooted at v_X in T_f, then from level to level the size that the nonterminals contribute decreases until only leaves occur at some level. Since $t_i(X) \leq 8(\log \log n + 3)$, it follows that the subtree of T_f rooted at v_X has depth at most $8(\log \log n + 3) + 1$ and thus the maximal level of any unary nonterminal in $\mathbb{A}_f$ is bounded by

$$\lceil \log \log n \rceil + 8(\log \log n + 3) + 1 \leq 9 \log \log n + 26.$$

As a consequence, the number of unary nonterminals (the number of leaves of T_f) is bounded by $\mathcal{O}((\log n)^9)$ since T_f is a binary tree of depth at most $9 \log \log n + 26$. Further, each unary nonterminal X in $\mathbb{A}_f$ satisfies $t_f(X) \leq \mathcal{O}(\log \log n)$. Either X fulfills that there is a round $i \leq f$ such that $\mathsf{level}_i(X) = \lceil \log \log n \rceil$ and thus $t_f(X) \leq t_i(X) \leq 8(\log \log n + 3)$ by Corollary 3.7.11 and Lemma 3.7.8, point (i). Or, if $\mathsf{level}_f(X) = m < \lceil \log \log n \rceil$, then $t_f(X) \leq m + 3 \leq \log \log n + 3$, because there is at most one non-overlapping occurrence of XX on right-hand sides of $\mathbb{A}_f$ (otherwise there would exist a maximal string of $\mathbb{A}_f$) and the number of rules where X occurs on the right-hand side is $\#_f(X) \leq m + 1$ by Lemma 3.7.7. To be more precise, a single right-hand side of $\mathbb{A}_f$ could have a maximal X-block of length 3 and all other right-hand sides must have at most one occurrence of X since two different right-hand sides where X-blocks of length 2 occur as well as one right-hand side where an X-block of length 4 occurs would contradict the fact that $\mathbb{A}_f$ has no maximal string. It follows that the size which unary nonterminals contribute to $\mathbb{A}_f$ is $\mathcal{O}((\log n)^9 \cdot \log \log n)$. By Lemma 3.7.14, we can bound the size that non-unary nonterminals contribute by $\mathcal{O}((\log n)^9 \cdot (\log \log n)^3)$ since

there are at most $\mathcal{O}((\log n)^9)$ unary nonterminals and each has level at most $\mathcal{O}(\log \log n)$ as argued above. It follows that $|\mathbb{A}_f| \leq \mathcal{O}((\log n)^9 \cdot (\log \log n)^3)$, which proves the proposition. □

The following theorem follows directly from Proposition 3.7.1 and Lemma 3.1.2, where $g(w) \geq \Omega(\log n)$ is shown for words w of length n.

Theorem 3.7.15. *For all n, we have*

$$\alpha_{\mathsf{Greedy}}(1, n) \leq \mathcal{O}((\log n)^8 \cdot (\log \log n)^3).$$

We proceed with the lower bound on the approximation ratio of Greedy. The best known lower bound [29, Theorem 11] so far is

$$\alpha_{\mathsf{Greedy}}(k, n) \geq \frac{5}{3 \log_3(5)} = 1.13767699...$$

for all $k \geq 1$ and infinitely many n. In particular, this bound is achieved using unary input strings. A key concept to prove a better lower bound is the sequence x_n described in the following lemma by [4]:

Lemma 3.7.16 ([4, Example 2.2])**.** *Let $x_{n+1} = x_n^2 + 1$ with $x_0 = 1$ and*

$$\beta = \exp\left(\sum_{i=1}^{\infty} \frac{1}{2^i} \log\left(1 + \frac{1}{x_i^2}\right)\right).$$

We have $x_n = \lfloor \beta^{2^n} \rfloor$.

In this work, we use the shifted sequence $y_n = x_{n+1}$, i.e., we start with $y_0 = 2$. It follows that $y_n = \lfloor \gamma^{2^n} \rfloor$, where $\gamma = \beta^2 = 2.25851845...$. Additionally, we need the following lemma:

Lemma 3.7.17. *Let $m \geq 1$ be an integer. Let $f_m : \mathbb{R}_{>0} \to \mathbb{R}$ with*

$$f_m(x) = x + \frac{m^2 + 1}{x}.$$

We have $f_m(x) > 2m$ for all $x > 0$.

Proof. The unique minimum of $f_m(x)$ is $2\sqrt{m^2 + 1}$ for $x = \sqrt{m^2 + 1}$. It follows that $f_m(x) \geq 2\sqrt{m^2 + 1} > 2\sqrt{m^2} = 2m$. □

Now we are able to prove the new lower bound for Greedy:

Theorem 3.7.18. *For all $k \geq 1$ and infinitely many n, we have*

$$\alpha_{\mathsf{Greedy}}(k, n) \geq \frac{1}{\log_3(\gamma)} = 1.34847194... \ .$$

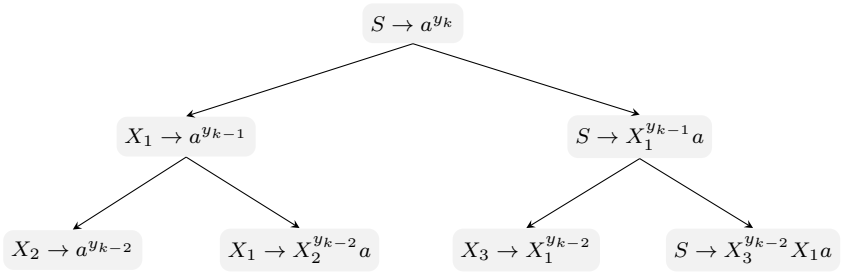

Figure 3.6: Three rounds of Greedy on input a^{y_k}.

Proof. Let $\Sigma = \{a\}$ be a unary alphabet. We define $w_k = a^{y_k}$. By Lemma 3.7.16, we have $|w_k| \leq \gamma^{2^k}$. Applying Lemma 3.1.2 yields

$$g(w_k) \leq 3 \cdot \log_3(\gamma) \cdot 2^k + o(2^k).$$

In the remaining proof we show that on input w_k, Greedy produces an SLP of size $3 \cdot 2^k - 1$, which directly implies $\alpha_{\mathsf{Greedy}}(1, n) \geq 3/(3 \log_3(\gamma))$. We start with the SLP $\mathbb{A}_0$ which has the single rule $S \to a^{y_k}$. Consider now the first round of the algorithm, i.e., we need to find a maximal string a^x of $\mathbb{A}_0$ such that the grammar $\mathbb{A}_1$ with rules

$$X_1 \to a^x, \;\; S \to X_1^{y_k \operatorname{div} x} a^{y_k \bmod x}$$

has minimal size. We have $|\mathbb{A}_1| = x + (y_k \operatorname{div} x) + (y_k \bmod x) \geq x + y_k/x$. By the definition of y_k we have $|\mathbb{A}_1| \geq x + (y_{k-1}^2 + 1)/x$. Applying Lemma 3.7.17 yields $|\mathbb{A}_1| \geq 2y_{k-1} + 1$. Note that for $x = y_{k-1}$ this minimum is achieved, i.e., we can assume that Greedy selects the maximal string $a^{y_{k-1}}$ and $\mathbb{A}_1$ is

$$X_1 \to a^{y_{k-1}}, \; S \to X_1^{y_{k-1}} a.$$

Each maximal string of $\mathbb{A}_1$ is either a unary word over X or a unary word over a, i.e., we can analyze the behavior of Greedy on both rules independently. The rule $X_1 \to a^{y_{k-1}}$ is obviously treated similarly as the initial SLP $\mathbb{A}_0$, so we continue with analyzing $S \to X_1^{y_{k-1}} a$. But again, the same arguments as above show that Greedy introduces a rule $X_3 \to X_1^{y_{k-2}}$ which yields $S \to X_3^{y_{k-2}} X_1 a$ as the new start rule. This process can be iterated using the same arguments for the leading unary strings of length y_i for some $i \in [1, k]$.

The reader might think of this process as a binary tree, where each node is labeled by a rule (the root is labeled by $S \to a^{y_k}$) and the children of a node are the two rules obtained by Greedy when the rule has been processed. We assume that the left child represents the rule for the chosen maximal string and the right child represents the parent rule where all occurrences of the maximal string are replaced by the fresh nonterminal. In Figure 3.6 this binary tree is depicted for the steps we discussed above. Note that when a rule is processed, the longest common factor of the two new rules has length 1 (the remainder).

More generally, after each round there is no word of length at least two that occurs as a factor in two different rules, since a possibly shared remainder has length 1 and otherwise only fresh nonterminals are introduced. It follows that we can iterate this process independently for each rule until no maximal string occurs. This is the case when each rule starts with a unary string of length $y_0 = 2$ or, in terms of the interpretation as a binary tree, when a full binary tree of height k is produced. Each right branch occurring in this tree adds a new remainder to those remainders that already occur in the parent rule and a left branch introduces a new (smaller) instance of the start problem. We show by induction that at level $i \in [0, k]$ of this full binary tree of height k, there is one rule of size $y_{k-i} + i$ and 2^{i-j-1} many rules of size $y_{k-i} + j$ for $j \in [0, i-1]$. At level 0, this is true since there is only a single rule of size $y_k + 0$. Assuming that our claim is true at level $i < k$, we derive from each rule at level i two new rules at level $i+1$: A right branch yields a rule that starts with a leading unary string of size y_{k-i-1} and adds a new remainder to the parent rule. A left branch yields a rule that contains only a unary string of size y_{k-i-1}. If we first consider the left branches, we derive that each of the 2^i many rules at level i adds a rule of size y_{k-i-1} at level $i+1$. For the right branches, the single rule of size $y_{k-i} + i$ at level i yields a rule of size $y_{k-i-1} + i + 1$ at level $i+1$. Further, each of the 2^{i-j-1} many rules of size $y_{k-i} + j$ ($j \in [0, i-1]$) yields a rule of size $y_{k-i-1} + j + 1$. When we put everything together, we get that at level $i+1$ there is a single rule of size $y_{k-i-1} + i + 1$ and 2^{i-j} many rules of size $y_{k-i-1} + j$ for $j \in [0, i]$. That finishes the induction. It follows that the final SLP (which consists of the rules at level k) has a single rule of size $y_0 + k = 2 + k$ and 2^{k-j-1} many rules of size $2 + j$ for $j = 0, \ldots, k-1$. This gives a total size of

$$\begin{aligned}
2 + k + \sum_{j=0}^{k-1} 2^{k-j-1}(2+j) &= 2 + k + 2^k \sum_{j=0}^{k-1} 2^{-j} + 2^k \sum_{j=0}^{k-1} 2^{-j-1} j \\
&= 2 + k + 2^k(2 - 2^{-k+1}) + 2^k(-2^{-k}k - 2^{-k} + 1) \\
&= 2 + k + 2^{k+1} - 2 - k - 1 + 2^k \\
&= 2^{k+1} + 2^k - 1 \\
&= 3 \cdot 2^k - 1.
\end{aligned}$$

□

3.7.2 General case

The general upper bound on the approximation ratio of any global algorithm provided in Theorem 3.5.1 is also the best known upper bound for Greedy, i.e., $\alpha_{\mathsf{Greedy}}(n) \leq \mathcal{O}((n/\log n)^{2/3})$. When it comes to lower bounds, the bound on the approximation ratio of Greedy for unary inputs presented in Theorem 3.7.18 is also the best known lower bound in the general setting.

3.8 LongestMatch

The global grammar-based compressor LongestMatch [72] selects a longest maximal string in each round.

Example 3.12. *Let* $w = aaaaabbababbbaaabb$. *We have*

$\mathbb{A}_0 : S \to aaaaabbababbbaaabb$,

$\mathbb{A}_1 : S \to aaXababbbX,\ X \to aaabb$,

$\mathbb{A}_2 : S \to aaXabYbX,\ X \to aaY,\ Y \to abb$,

$\mathbb{A}_3 : S \to ZXabYbX,\ X \to ZY,\ Y \to abb,\ Z \to aa$,

$\mathbb{A}_4 : S \to ZXAYbX,\ X \to ZY,\ Y \to Ab,\ Z \to aa,\ A \to ab$.

Note that the choices of the maximal strings aa *(round* 3*) and* ab *(round* 4*) could be switched, because both have length* 2.

3.8.1 Unary inputs

We start again by considering unary inputs. It turns out that the size of the SLP produced by LongestMatch on input a^n has the same size as we proved for RePair in Proposition 3.6.1.

Proposition 3.8.1. *For all* $n \geq 2$, *we have*

$$|\mathsf{LongestMatch}(a^n)| = 2\lfloor \log n \rfloor + \nu(n) - 1.$$

Proof. If $n \in [2,3]$ then a^n has no maximal string and thus the final SLP has a single rule $S \to a^n$. We have $n = 2\lfloor \log n \rfloor + \nu(n) - 1$ for $n \in [2,3]$.

Let b_i be the coefficient of 2^i in the binary representation of n (see equation (2.1)). In the first round, the chosen maximal string is $a^{\lfloor n/2 \rfloor}$, which yields rules $X_1 \to a^{\lfloor n/2 \rfloor}$ and $S \to X_1 X_1 a^{b_0}$, i.e., the symbol a occurs in the start rule if and only if n is odd and thus the least significant bit $b_0 = 1$. Assuming $n \geq 8$, this procedure is now repeated for the rule $X_1 \to a^{\lfloor n/2 \rfloor}$ (for $n < 8$ there is no maximal string and the algorithm stops after the first round). This yields $X_2 \to a^{\lfloor n/4 \rfloor}$, $X_1 \to X_2 X_2 a^{b_1}$ and $S \to X_1 X_1 a^{b_0}$ (note that $\lfloor (\lfloor n/2 \rfloor)/2 \rfloor = \lfloor n/4 \rfloor$). After $m = \lfloor \log n \rfloor - 1$ steps, the iteration of that process results in the final SLP with rules $S \to X_1 X_1 a^{b_0}$, $X_i \to X_{i+1} X_{i+1} a^{b_i}$ for $i \in [1, m-1]$ and $X_m \to aaa^{b_m}$. The size of this SLP is $2 \cdot (m+1) + \sum_{i=0}^{m} b_i$, which directly implies the claimed result for LongestMatch. □

Since LongestMatch and RePair produce equal size SLPs for unary inputs, we can take the results that we achieved for $\alpha_{\mathsf{RePair}}(1,n)$ (Theorems 3.6.2 and 3.6.3) and use these for $\alpha_{\mathsf{LM}}(1,n)$ as well.

Corollary 3.8.2. *For all* n, *we have* $\alpha_{\mathsf{LM}}(1,n) \leq \log(3)$ *and for infinitely many* n, *we have* $\alpha_{\mathsf{LM}}(1,n) \geq \log(3)$.

3.8.2 General case

Similar to RePair and Greedy, the upper bound in Theorem 3.5.1 is the best known upper bound for LongestMatch, i.e., $\alpha_{\mathsf{LM}}(n) \leq \mathcal{O}((n/\log n)^{2/3})$. On the other hand, the best known lower bound is shown in [29]:

Theorem 3.8.3 ([29, Theorem 10])**.** *For infinitely many n, we have*

$$\alpha_{\mathsf{LM}}(n) \geq \Omega(\log \log n).$$

The proof of this theorem works with an alphabet of unbounded size. In particular, the lower bound is shown for words σ_k of length $|\sigma_k| = \Theta(k2^k)$ where an alphabet of size $\Theta(k)$ is used.

3.9 Universal coding based on SLPs

In Section 4.9 of the next chapter, we show how grammar-based tree compression can be used to achieve universal lossless source coding. Since the techniques we apply there build on a similar work by Kieffer and Yang [72] for grammar-based string compression, we give a short survey of this work in the following. We focus on the binary encoding of SLPs used in [72] such that a universal code for so-called finite-state information sources is obtained. An *information source* $\mathcal{S}$ emits words over a fixed alphabet Σ with certain probabilities $p_{\mathcal{S}} : \Sigma^* \to \mathbb{R}_{[0,1]}$. It is required that $p_{\mathcal{S}}$ satisfies $p_{\mathcal{S}}(\varepsilon) = 1$ and $p_{\mathcal{S}}(w) = \sum_{a \in \Sigma} p_{\mathcal{S}}(wa)$ for all $w \in \Sigma^*$, i.e., $p_{\mathcal{S}}$ is a probability distribution on all words of length n for each $n \geq 0$. An information source is called a *finite-state information source* if it can be simulated by a probabilistic finite-state machine, see [72] for the precise definition. For a binary encoding $E : \Sigma^+ \to \{0,1\}^+$ of words over Σ and an information source $\mathcal{S}$, the so-called worst-case redundancy (or maximal pointwise redundancy) for strings of length n is defined as

$$R(E, \mathcal{S}, n) = \max_{w \in \Sigma^n, p_{\mathcal{S}}(w) > 0} \frac{1}{n} \cdot (|E(w)| + \log p_{\mathcal{S}}(w)). \tag{3.6}$$

Thus, the worst-case redundancy measures the maximal additive deviation of the code length from the self information with respect to $p_{\mathcal{S}}$, normalized by the length of the source string. A binary encoding E is called a (worst-case) *universal code* for an information source S if and only if the worst-case redundancy $R(E, \mathcal{S}, n)$ converges to zero for $n \to \infty$.

The universal code achieved in [72] for finite-state information sources is based on so-called *asymptotically compact* grammar-based compressors. A grammar-based compressor $\mathcal{C}$ is asymptotically compact if the following conditions are satisfied:

(i) Let N_w be the set of nonterminals of the SLP $\mathcal{C}(w)$ for $w \in \Sigma^+$. Then for all $w \in \Sigma^+$ it is required that different nonterminals in N_w produce different words, i.e., $\mathsf{val}_{\mathcal{C}(w)}(X) \neq \mathsf{val}_{\mathcal{C}(w)}(Y)$ for all $X, Y \in N_w$ with $X \neq Y$.

(ii) The function $r_{\mathcal{C}}(n) = \max_{w \in \Sigma^n}(\mathcal{C}(w)/n)$ converges to zero for $n \to \infty$.

Note that any grammar-based compressor studied in this work is asymptotically compact. For BiSection this is true since point (i) is fulfilled by the definition of the algorithm and $r_{\mathsf{BiSection}}(n) \leq \mathcal{O}(1/\log_{|\Sigma|} n)$ follows from the fact that BiSection produces an SLP of size $\mathcal{O}(n/\log_{|\Sigma|} n)$ for inputs of length n (Theorem 3.3.6). But as we argued in Section 3.1.1, any algorithm studied in this work achieves this upper bound and thus satisfies point (ii). For LZ78, point (i) follows directly from the definition of the algorithm and for any global algorithm point (i) is provided in [29, Lemma 7].

Based on a grammar-based compressor $\mathcal{C}$, we describe in the following the binary encoding $E_{\mathcal{C}} : \Sigma^+ \to \{0,1\}^+$ of words over Σ such that $E_{\mathcal{C}}$ encodes the SLP $\mathcal{C}(w)$. Before we do so, we need an assumption about the naming of the nonterminals. Let $\mathbb{A} = (N, \Sigma, P, S)$ be an SLP. We define the *derivation tree* $\mathcal{D}_{\mathbb{A}}$ of $\mathbb{A}$ as follows:

- The root node is labeled by the start nonterminal S.
- If $A \in N$ and $(A \to w_1 w_2 \cdots w_m) \in P$ with $w_i \in (\Sigma \cup N)$ for $i \in [1, m]$, then any node labeled by A has m (ordered) children and the i-th child is labeled by w_i for $i \in [1, m]$.
- A node is a leaf if and only if it is labeled by $a \in \Sigma$.

In the following, we assume that the set N of nonterminals of an SLP $\mathbb{A}$ with $|N| = k$ satisfies $N = \{A_0, A_1 \ldots, A_{k-1}\}$ such that $A_0, A_1, \ldots, A_{k-1}$ is obtained when the nonterminals are listed in order of their first appearance in $\mathcal{D}_{\mathbb{A}}$ using breadth-first left-to-right search. In particular, A_0 is the start nonterminal. In Example 3.13, we describe the renaming of the nonterminals of an SLP such that it fulfills the assumption.

Example 3.13. *Let $\mathbb{B} = (\{S, X, Y, Z\}, \{a, b\}, P, S)$ be an SLP such that P contains rules $S \to YXY$, $X \to ZZ$, $Y \to aXa$ and $Z \to bb$. The derivation tree $\mathcal{D}_{\mathbb{B}}$ is depicted in Figure 3.7. The breadth-first left-to-right traversal of $\mathcal{D}_{\mathbb{B}}$ is*

$$S\ Y\ X\ Y\ a\ X\ a\ Z\ Z\ a\ X\ a\ Z\ Z\ b\ b\ b\ b\ Z\ Z\ b\ b\ b\ b\ b\ b\ b\ b.$$

Hence we rename S into A_0, Y into A_1, X into A_2 and Z into A_3. The obtained SLP $\mathbb{A}$ is shown in Example 3.14.

Before we describe the encoding of an SLP $\mathbb{A} = (\{A_0, \ldots, A_{k-1}\}, \Sigma, P, A_0)$, we define words $\rho_{\mathbb{A}}$ and $\omega_{\mathbb{A}}$ over $\Sigma \cup \{A_1, \ldots, A_{k-1}\}$ which are the foundation of the encoding. Let $\rho_{\mathbb{A}} = w_0 w_1 \cdots w_{k-1}$ with $|\rho_{\mathbb{A}}| = |\mathbb{A}|$, where w_i is the right-hand side of $(A_i \to w_i) \in P$ for $i \in [0, k-1]$. Further, $\omega_{\mathbb{A}}$ is obtained from $\rho_{\mathbb{A}}$ by removing the first (left-to-right) occurrence of each nonterminal.

Example 3.14. *The SLP $\mathbb{A} = (\{A_0, A_1, A_2, A_3\}, \{a, b\}, P, S)$ is obtained by renaming the nonterminals from the SLP $\mathbb{B}$ as described in Example 3.13. The rules are $A_0 \to A_1 A_2 A_1$, $A_1 \to a A_2 a$, $A_2 \to A_3 A_3$ and $A_3 \to bb$. We have $\rho_{\mathbb{A}} = A_1 A_2 A_1 a A_2 a A_3 A_3 bb$ and $\omega_{\mathbb{A}} = A_1 a A_2 a A_3 bb$.*

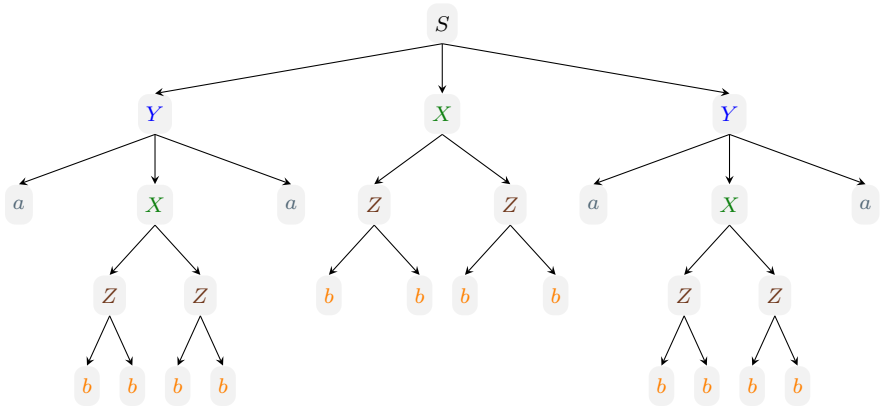

Figure 3.7: The derivation tree $\mathcal{D}_{\mathbb{B}}$ for the SLP $\mathbb{B}$ described in Example 3.13.

Let $\Sigma = \{\sigma_1, \ldots, \sigma_{|\Sigma|}\}$ such that $\sigma_1 < \sigma_2 < \cdots < \sigma_{|\Sigma|}$ is the lexicographic order of the symbols in Σ. The binary encoding $B(\mathbb{A}) \in \{0,1\}^+$ of an SLP $\mathbb{A} = (\{A_0, \ldots, A_{k-1}\}, \Sigma, P, A_0)$ is $B(\mathbb{A}) = B_1B_2B_3B_4B_5B_6$ such that

- $B_1 = 0^{k-1}1$ indicates the set of nonterminals.
- $B_2 = b_1b_2\cdots b_{|\Sigma|}$ where $b_i = 1$ if and only if $\sigma_i \in \Sigma$ occurs in $\rho_{\mathbb{A}}$ for $i \in [1, |\Sigma|]$.
- $B_3 = \prod_{a\in\Sigma'} 0^{k_a-1}1 \prod_{i=1}^{k-1} 0^{k_{A_i}-1}1$ where $\Sigma' \subseteq \Sigma$ contains the alphabet symbols which occur in the word $\rho_{\mathbb{A}}$ and k_x is the number of occurrences of $x \in \Sigma' \cup \{A_1, \ldots, A_{k-1}\}$ in $\rho_{\mathbb{A}}$. The factors in the product $\prod_{a\in\Sigma'} 0^{k_a-1}1$ are ordered with respect to the lexicographical order of the alphabet.
- $B_4 = \prod_{i=0}^{k-1} 0^{|w_i|-1}1$ where w_i is the right-hand side of $(A_i \to w_i) \in P$.
- $B_5 = b_1b_2\cdots b_{|\mathbb{A}|}$ where $b_i = 1$ if and only if a nonterminal occurs first (left-to-right) at position i in $\rho_{\mathbb{A}}$ for $i \in [1, |\mathbb{A}|]$.
- B_6 encodes the word $\omega_{\mathbb{A}}$ using enumerative encoding [33]. Let M be the set of all words which contain every symbol $x \in \Sigma \cup \{A_1, \ldots, A_{k-1}\}$ exactly as often as it occurs in $\omega_{\mathbb{A}}$. Let $v_0, v_1, \ldots v_{|M|-1}$ be the lexicographic enumeration of the words in M where $\sigma_1 < \cdots < \sigma_{|\Sigma|} < A_1 < \cdots < A_{k-1}$ is the order of the symbols. Then B_6 is the binary representation of the unique index i such that $v_i = \omega_{\mathbb{A}}$.

Note that for a given code $B(\mathbb{A})$ we can determine the SLP $\mathbb{A}$. The number of nonterminals is given in B_1 and alphabet symbols which occur on right-hand sides of rules in $\mathbb{A}$ are given in B_2. Using B_3 it is straightforward to get the number of occurrences of any symbol in $\omega_{\mathbb{A}}$, which together with B_6 allows the reconstruction of $\omega_{\mathbb{A}}$. Additionally, B_5 allows to reconstruct $\rho_{\mathbb{A}}$ from $\omega_{\mathbb{A}}$ since the nonterminals are numbered such that at the fist left-to-right position described

in B_5 we insert the nonterminal A_1, at the second position we insert A_2, and so on. Finally, B_4 allows to factorize $\rho_{\mathbb{A}}$ such that the rules in $\mathbb{A}$ are reconstructed.

Example 3.15. *Assume the SLP $\mathbb{A}$ from example 3.14. We describe the binary encoding $B(\mathbb{A}) = B_1B_2B_3B_4B_5B_6$. We have $B_1 = 0001$ because the set of nonterminals is $\{A_0, A_1, A_2, A_3\}$. We have $B_2 = 11$ because both alphabet symbols occur in $\rho_{\mathbb{A}} = A_1A_2A_1aA_2aA_3A_3bb$. Further, $B_3 = 0101010101$ because any symbol $x \in \{a, b, A_1, A_2, A_3\}$ occurs exactly twice in the word $\rho_{\mathbb{A}}$. We have $B_4 = 0010010101$ because $|A_1A_2A_1| = |aA_2a| = 3$ and $|A_3A_3| = |bb| = 2$. Moreover, $B_5 = 1100001000$ since the first left-to-right occurrence of A_1 in $\rho_{\mathbb{A}}$ is at position 1, the first occurrence of A_2 is at position 2 and the first occurrence of A_3 is at position 7. Concerning B_6, there are 1260 words such that the number of occurrences of each symbol is exactly the same as in $\omega_{\mathbb{A}} = A_1aA_2aA_3bb$ and if those words are ordered lexicographically, then $\omega_{\mathbb{A}}$ is the 758-th word. Therefore, B_6 is the binary representation of $758 - 1 = 757$ (we start the enumeration with zero), i.e., $B_6 = 1011110101$.*

For a grammar-based compressor $\mathcal{C}$ and for all words $w \in \Sigma^+$, we define the encoding $E_{\mathcal{C}}(w) = B(\mathcal{C}(w))$. Based on this encoding, the main result in [72] is the following:

Theorem 3.9.1 ([72, Theorem 7])**.** *Let $\mathcal{C}$ be an asymptotically compact grammar-based compressor and $\mathcal{S}$ be a finite-state information source. Then $E_{\mathcal{C}}$ is a universal code for $\mathcal{S}$. In particular, we have $R(E_{\mathcal{C}}, \mathcal{S}, n) \leq \mathcal{O}(\gamma(r_{\mathcal{C}}(n)))$, where $r_{\mathcal{C}}(n) = \max_{w \in \Sigma^n}(\mathcal{C}(w)/n)$ and $\gamma(x) = x \log(1/x)$.*

If the encoding $E_{\mathcal{C}}$ is based on a grammar-based compressor $\mathcal{C}$ such that $\mathcal{C}(w) \leq \mathcal{O}(n/\log n)$ for all $w \in \Sigma^n$, then Theorem 3.9.1 yields worst-case redundancy $R(E_{\mathcal{C}}, \mathcal{S}, n) \in \mathcal{O}(\log\log n/\log n)$. In [73], the authors improved upon this result and achieve $R(E, \mathcal{S}, n) \leq \mathcal{O}(1/\log n)$ for any finite-state information source $\mathcal{S}$, where E is a new grammar-based code. The rough idea in order to achieve convergence rate $\mathcal{O}(1/\log n)$ is to encode the structure of the SLP (the derivation tree) and the actual data independently.

3.10 Conclusion and open problems

Conclusion. We presented in this chapter various results about grammar-based string compression. We showed in Section 3.2.1 that any polynomial time grammar-based compressor that achieves constant approximation ratio c for binary inputs can be extended to a polynomial time grammar-based compressor which achieves approximation ratio $6c$ for inputs over arbitrary alphabets. This improves a rather technical construction presented in [10].

In the main part of this chapter we analyzed the approximation ratios of the grammar-based compressors BiSection (Section 3.3), LZ78 (Section 3.4), RePair (Section 3.6), Greedy (Section 3.7) and LongestMatch (Section 3.8). For each of those algorithms, we distinguished between unary inputs and inputs over arbitrary alphabets. The obtained bounds are depicted in Table 3.1 for the

unary case and Table 1.1 for the general case. The matching bounds in the general setting for LZ78 and BiSection deserve a special mentioning.

In Section 3.9 we revisited a paper of [72] where a universal code based on grammar-based compression is presented. A key ingredient in order to obtain universality for grammar-based codes is the fact that any word of length n (over an alphabet of constant size) is produced by an SLP of size $\mathcal{O}(n/\log n)$. This result is well known for grammar-based string compression and a proof can be found in Section 3.3.3. Both topics - $\mathcal{O}(n/\log n)$ worst-case size and universal coding - are the main components of the next chapter, where we extend the described techniques from grammar-based string compression to grammar-based tree compression.

Open problems. The first question concerns the connection between grammar-based compression over binary inputs and arbitrary inputs. Constant approximation ratio c for binary inputs yields approximation ratio $6c$ for inputs over arbitrary alphabets as described above. It remains open whether the constant 6 can be reduced even further. Concerning the hardness of grammar-based compression in general, recall that there is no polynomial time algorithm which computes a smallest SLP for inputs over an alphabet of size at least 24 unless P = NP [28]. It is one of the major open challenges in this context to prove a similar hardness result for smaller alphabets (in particular binary alphabets) or otherwise to come up with an optimal polynomial time algorithm for inputs over small alphabets.

The obvious questions that arise in this chapter concern the huge gaps between lower and upper bounds for the approximation ratios of the global grammar-based compressors RePair, Greedy and LongestMatch. The best known approximation ratio of a polynomial time grammar-based string compressor is $\mathcal{O}(\log(n/g))$ [29, 64, 65, 102, 103], where g is the size of a smallest SLP for the given input. Concerning the lower bounds for the global algorithms, it is still possible that any of those achieve approximation ratio $o(\log n)$. Moreover, the best known polynomial time general addition chain solver produces a general addition chain for $n_1, \ldots, n_k$ of size $m^* \cdot \mathcal{O}(\log N/\log\log N)$, where m^* is the size of a smallest general addition chain for $n_1, \ldots, n_k$ and $N = \max\{n_i \mid i \in [1, k]\}$. From what we know, it is still possible that Greedy and LongestMatch improve upon this more than 30 year old result (see Section 3.2.2 for the connection between grammar-based compression and general addition chains). In particular, the lower bound on the general approximation ratio of Greedy is unsatisfactory since (i) it is constant and (ii) it is achieved using unary inputs. Concerning the upper bounds, we only have a general upper bound for any global algorithms which already indicates space for improvement. When inputs are restricted to be unary, then Greedy is the only algorithm (considered in this work) where the lower and upper bound do not match. We conjecture that Greedy produces an SLP of size $\mathcal{O}(\log n)$ and thus have constant approximation ratio for unary inputs of size n.

Chapter 4

Grammar-based tree compression

In the following, we consider trees instead of strings as the data we want to compress. Grammar-based compression was extended from strings to trees in [26], where linear context-free tree grammars are used in order to represent a given tree. A linear context-free tree grammar that produces a single tree is called a tree straight-line program or TSLP for short. We call an algorithm which computes a TSLP for a given tree a grammar-based tree compressor.

The main result of this chapter are grammar-based tree compressors that produce for a given node-labeled tree of size n with σ different node labels (we always assume that $\sigma \geq 2$), a TSLP of size $\mathcal{O}(n/\log_\sigma n)$ and depth $\mathcal{O}(\log n)$, where the depth of a TSLP is the depth of the corresponding derivation tree. An important assumption here is that the maximal number of children of any node in the tree is bounded by a constant. In particular, for an unlabeled binary tree we get a TSLP of size $\mathcal{O}(n/\log n)$. After introducing the grammar-based compressors, we apply our result to arithmetical circuits and universal source coding. In the following, we present a summary of those results.

Our first compressor is an extension of the BiSection algorithm presented in [74] from strings to trees and is therefore called TreeBiSection. The algorithm is presented in Section 4.4 and we give an outline for binary trees in the following. In a first step, TreeBiSection hierarchically decomposes in a top-down way the input tree into pieces of roughly equal size. This is a well known technique that is also known as the $(1/3, 2/3)$-Lemma [78]. But care has to be taken in order to bound the ranks of the nonterminals of the resulting TSLP, which corresponds to the number of holes in the tree patterns occurring on the right-hand sides of the rules of the TSLP. Therefore, when the decomposition produces a tree pattern with three holes, we have to do an intermediate step that decomposes this pattern into two pieces having only two holes each. This may involve an unbalanced decomposition, but such steps are only necessary in every second step. The technique to bound the number of holes was used by Ruzzo [101] for

space-bounded alternating Turing machines. We identify the TSLP produced in this first step with its derivation tree, which is weakly balanced in the following sense: For each edge (u, v) in the derivation tree such that both u and v are internal nodes, the derivation tree is balanced at u or v. This follows directly from the decomposition described above. In a second step, TreeBiSection computes the minimal directed acyclic graph (minimal DAG) of the derivation tree, which is a widely used tree compression technique obtained by writing down repeated subtrees only once (see Section 4.3). We show in Section 4.3.1 that the minimal DAG of a weakly balanced tree has size at most $\mathcal{O}(n/\log_\sigma n)$, which is a result of independent interest. The nodes of the DAG of the derivation tree are the nonterminals of the final TSLP produced by TreeBiSection. We prove that the algorithm sketched above can be implemented so that it works in logarithmic space or running time $\mathcal{O}(n \log n)$, see Section 4.4.1.

In Section 4.5, we present BU-Shrink (for bottom-up shrink) that constructs a TSLP of size $\mathcal{O}(n/\log_\sigma n)$ in linear time. The following outline assumes again binary trees. In a first step, BU-Shrink merges nodes of the input tree in a bottom-up way. Thereby it constructs a partition of the input tree into $\mathcal{O}(n/\log_\sigma n)$ many connected parts of size at most $c \cdot \log_\sigma n$, where c is a suitably chosen constant. Every connected part is a pattern occurring in the input tree, where at most two subtrees were removed (i.e., a pattern with at most two holes). Using again DAG compression, we associate with each distinct pattern a nonterminal of rank at most two. We obtain a TSLP for the input tree consisting of a start rule $S \to s$, where s consists of $\mathcal{O}(n/\log_\sigma n)$ many nonterminals of rank at most two, each having a right-hand side consisting of $c \cdot \log_\sigma n$ many terminal symbols. By choosing the constant c suitably, we can bound the number of different subtrees of these right-hand sides by $\sqrt{n} \leq \mathcal{O}(n/\log_\sigma n)$, where we use the formula for the number of binary trees of size m given by the Catalan numbers. This allows to build up the right-hand sides for the non-start nonterminals using $\mathcal{O}(n/\log_\sigma n)$ many nonterminals.

Since BU-Shrink does not yield logarithmic depth of the produced TSLP, we present an algorithm which combines BU-Shrink and TreeBiSection — we call it BU-Shrink+TreeBiSection — such that a TSLP of size $\mathcal{O}(n/\log_\sigma n)$ and depth $\mathcal{O}(\log n)$ is constructed in linear time. This last step could be replaced by a balancing technique recently introduced in [48]. More precisely, it is shown in [48] that it is possible to balance a TSLP in linear time such that the size of the obtained TSLP increases only by a multiplicative constant.[1]

The case where the number of children of a node is not bounded by a constant (as it is the case for unranked trees) is discussed in Section 4.4.3. While most of what we explained so far fails in this setting, we give a simple workaround solution based on the first-child-next-sibling encoding [76], which encodes a given (unranked) tree by a ranked binary tree of the same size. Then, the first-child-next-sibling encoding can be transformed into a TSLP. This solution is strongly related to so-called *forest straight-line programs* [50].

[1] The authors in [48] show this balancing result for a more general formalism, which includes SLPs and TSLPs.

Transforming formulas into circuits. In Section 4.6, we apply our constructions of small TSLPs to the classical problem of transforming a given formula into a small circuit. Spira [106] has shown that for every Boolean formula of size n there exists an equivalent Boolean circuit of depth $\mathcal{O}(\log n)$ and size $\mathcal{O}(n)$, where the size of a circuit is the number of gates and the depth of a circuit is the length of a longest path from an input gate to the output gate. Brent [21] extended Spira's theorem to formulas over arbitrary semirings and moreover improved the constant in the $\mathcal{O}(\log n)$ bound for the depth. Subsequent improvements that mainly concern constant factors can be found in [18, 23]. We show that as a consequence of our constructions, we have for every (not necessarily commutative) semiring (or field), every formula of size n, in which only $m \leq n$ different variables occur, can be transformed into a circuit of depth $\mathcal{O}(\log n)$ and size $\mathcal{O}(n/\log_\sigma n)$. Hence, we refine the size bound from $\mathcal{O}(n)$ to $\mathcal{O}((n \cdot \log m)/\log n)$ (Theorem 4.6.3). The transformation can be achieved in logspace and, alternatively, in linear time. Another interesting point of our formula-to-circuit conversion is that most of the construction (namely the construction of a TSLP for the input formula) is purely syntactic. The remaining part (the transformation of the TSLP into a circuit) is straightforward. In contrast, the constructions from [18, 21, 23, 106] construct a log-depth circuit from a formula in one step.

Universal coding for unlabeled binary trees. In [111], Kieffer, Yang, and Zhang presented grammar-based source coding for unlabeled binary trees. For this, they first represent the input tree by its minimal DAG. In a second step, the minimal DAG is encoded by a binary string; this step is similar to the binary coding of SLPs from [72] presented in Section 3.9. Combining both steps yields a tree encoder $E_{\mathrm{dag}} : \mathcal{T} \to \{0,1\}^*$, where $\mathcal{T}$ denotes the set of all unlabeled binary trees. In order to define universality of such a tree encoder, the classical notion of an information source on finite sequences is replaced in [111] by the notion of a *structured tree source*. A tree source is a collection of probability distributions $(p_n)_{n\in\mathbb{N}}$, where every p_n is a distribution on a finite non-empty subset $\mathcal{P}_n \subseteq \mathcal{T}$, and these sets partition $\mathcal{T}$. The main cases considered in [111] as well as in this work are (i) *leaf-centric sources*, where $\mathcal{P}_n$ is the set of all binary trees with n leaves, and (ii) *depth-centric sources*, where $\mathcal{P}_n$ is the set of all binary trees of depth n. Then, the authors of [111] introduce two properties on binary tree sources: The domination property (see Section 4.8, where it is called the weak domination property) and the representation ratio negligibility property. The latter states that the average compression ratio achieved by the minimal DAG for trees of size n converges to zero for $n \to \infty$, where the size of a binary tree t is measured in the number of leaves of the tree (denoted by $\mathsf{leafsize}(t)$). The technical main result of [111] states that for every structured tree source $(p_n)_{n\in\mathbb{N}}$ satisfying the domination property and the representation ratio negligibility property the *average-case redundancy*

$$\sum_{t\in\mathcal{P}_n, p_n(t)>0} \frac{1}{\mathsf{leafsize}(t)} \cdot (|E_{\mathrm{dag}}(t)| + \log p_n(t)) \cdot p_n(t) \tag{4.1}$$

converges to zero for $n \to \infty$. Finally, two classes of tree sources having the domination property and the representation ratio negligibility property are presented in [111]. One is a class of leaf-centric sources, the other one is a class of depth-centric sources. Both classes have the property that every tree with a non-zero probability is balanced in a certain sense, the precise definitions can be found in Section 4.8.1 and Section 4.8.2. As a first contribution, we show that for these sources not only the average-case redundancy but also the worst-case redundancy

$$\max_{t \in \mathcal{P}_n, p_n(t) > 0} \frac{1}{\mathsf{leafsize}(t)} \cdot (|E_{\mathrm{dag}}(t)| + \log p_n(t)) \tag{4.2}$$

converges to zero for $n \to \infty$. More precisely, we show that (4.2) is bounded by $\mathcal{O}(\log\log n / \log n)$ (respectively, $\mathcal{O}((\log\log n)^2 / \log n)$) for the presented class of leaf-centric tree sources (respectively, depth-centric tree sources). To prove this, we use that the minimal DAG of weakly balanced binary trees has size $\mathcal{O}(n/\log n)$ as described above. Further, we use a result presented in [58] according to which the size of the minimal DAG of a suitably balanced binary tree of size n is bounded by $\mathcal{O}(n \cdot \log\log n / \log n)$ (see also Section 4.3.1, Theorem 4.3.4).

Our main resut in the context of universal coding is the application of TSLPs. In Section 4.9.2, we define a binary encoding of TSLPs similar to the ones for SLPs [72] and DAGs [111]. We then consider the tree encoder $E_{\mathrm{tslp}} : \mathcal{T} \to \{0,1\}^*$ based on TSLPs and prove that its worst-case redundancy (that is defined as in (4.2) with E_{dag} replaced by E_{tslp}) is bounded by $\mathcal{O}(\log\log n / \log n)$ for every structured tree source that satisfies the *strong domination property* defined in Section 4.9.3. The strong domination property is a strengthening of the domination property from [111], and this is what we have to pay extra for our TSLP-based encoding in contrast to the DAG-based encoding from [111]. On the other hand, our approach has two main advantages over [111]: The representation ratio negligibility property from [111] is no longer needed and we get bounds on the worst-case redundancy instead of the average-case redundancy. Both advantages are based on the fact that the grammar-based compressor described above compute a TSLP of worst-case size $\mathcal{O}(n/\log n)$ for an unlabeled binary tree of size n. Finally, we present a class of leaf-centric sources (Section 4.8.1) as well as a class of depth-centric sources (Section 4.8.2) having the strong domination property. These classes are orthogonal to those considered in [111].

The results of this chapter appeared in [45, 46].

4.1 Trees and patterns

Ranked alphabets and ranked trees. A *ranked alphabet* $\mathcal{F}$ is a finite set of symbols where each symbol $f \in \mathcal{F}$ has an associated rank $i \in \mathbb{N}$. We denote by $\mathcal{F}_i \subseteq \mathcal{F}$ the set of symbols of rank i. We assume that $\mathcal{F}_0 \neq \emptyset$, i.e., there is at least one symbol of rank 0. The idea is that a tree consists of nodes which are labeled by symbols from $\mathcal{F}$ and if a node has a label $f \in \mathcal{F}_i$, then this node has exactly i ordered children.

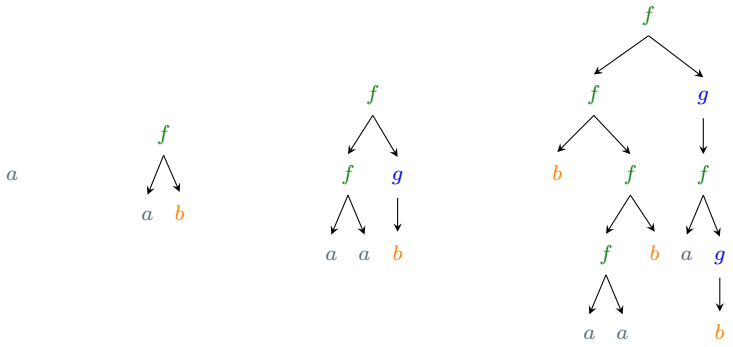

Figure 4.1: The trees t_1, t_2, t_3 and t_4 (from left to right) described in Example 4.1.

We consider *ranked, ordered, labeled trees* in this chapter (which we simply call trees). The trees are ranked and labeled because each node of a tree is labeled by a symbol from a ranked alphabet $\mathcal{F}$ and the rank of the label determines the number of children of the node. The trees are ordered because the children of each node are linearly ordered. Formally, we define trees as terms that are built from the symbols of $\mathcal{F}$ according to their rank. Let $\mathcal{T}(\mathcal{F})$ be the smallest set such that

- for all $a \in \mathcal{F}_0$ we have $a \in \mathcal{T}(\mathcal{F})$ and
- if $f \in \mathcal{F}_i$ for $i \geq 1$ and $t_1, \ldots, t_i \in \mathcal{T}(\mathcal{F})$, then $f(t_1, \ldots, t_i) \in \mathcal{T}(\mathcal{F})$.

A tree t over a ranked alphabet $\mathcal{F}$ is an element in the set $\mathcal{T}(\mathcal{F})$ and the set $\mathcal{T}(\mathcal{F})$ contains all trees over $\mathcal{F}$.

Example 4.1. *Let $\mathcal{F} = \{a, b, g, f\}$ such that a and b are symbols of rank 0, g is a symbol of rank 1 and f is a symbol of rank 2, i.e., $\mathcal{F}_0 = \{a, b\}$, $\mathcal{F}_1 = \{g\}$, $\mathcal{F}_2 = \{f\}$ and $\mathcal{F}_i = \emptyset$ for $i \in \mathbb{N} \setminus \{0, 1, 2\}$. For example $t_1 = a \in \mathcal{T}(\mathcal{F})$ denotes the tree that consists of a single (root) node labeled by a. Another example is $t_2 = f(a, b)$ which denotes the tree with root node labeled by f and two children, the left child is labeled by a and the right child is labeled by b. More complex examples are $t_3 = f(f(a, a), g(b))$ and $t_4 = f(f(b, f(f(a, a), b)), g(f(a, g(b))))$. The trees t_1, t_2, t_3 and t_4 are depicted in Figure 4.1.*

On multiple points throughout this chapter, we will consider a tree $t \in \mathcal{T}(\mathcal{F})$ as a graph with nodes and edges in the usual way (see Section 2.3). In this setting, we denote by $\mathsf{nodes}(t)$ the set of nodes of t and we use the function $\lambda_t : \mathsf{nodes}(t) \to \mathcal{F}$ such that $\lambda_t(v)$ is the label of node v. If t is clear from the context, we omit the subscript and just write $\lambda(v)$. Further, we denote by $\mathsf{labels}(t) = \{\lambda_t(v) \mid v \in \mathsf{nodes}(t)\}$ the set of symbols that occur as a label in the tree t. A node $v \in \mathsf{nodes}(t)$ is a *leaf* if and only $\lambda_t(v)$ has rank 0 and thus v has no children. The *size* $|t|$ of a tree t is the number of nodes of t, i.e., $|t| = |\mathsf{nodes}(t)|$. The *leaf size* $\mathsf{leafsize}(t)$ of t is the number of leaves, i.e., $\mathsf{leafsize}(t) = |\{v \in \mathsf{nodes}(t) \mid \lambda_t(v) \in \mathcal{F}_0\}|$.

A tree t is called a *binary tree* if every node has either zero or two children, i.e., $\mathsf{labels}(t) \subseteq \mathcal{F}_0 \cup \mathcal{F}_2$. Binary trees play a special role among trees in this work as well as in various applications. Note that for a binary tree t we have $|t| = 2 \cdot \mathsf{leafsize}(t) - 1$.

Example 4.2. *Let $a, b \in \mathcal{F}_0$ and $f \in \mathcal{F}_2$. Consider the binary tree $t = f(a, b)$ such that the root node of t is v_0, the left child node of v_0 is v_1 and the right child of v_0 is v_2 (this tree is called t_2 in Example 4.1 and is depicted at the second left-to-right position in Figure 4.1). We have $\mathsf{nodes}(t) = \{v_0, v_1, v_2\}$, $\lambda_t(v_0) = f$, $\lambda_t(v_1) = a$, $\lambda_t(v_2) = b$ and $\mathsf{labels}(t) = \{a, b, f\}$. The nodes v_1 and v_2 are leaves because they are labeled by symbols of rank 0. Further, we have $|t| = 3$ and $\mathsf{leafsize}(t) = 2$ because two of the three nodes of t are leaves.*

Occasionally, we consider *unlabeled trees*. We model unlabeled trees as follows: The set $\mathcal{T}(\mathcal{F})$ consists of unlabeled trees if and only if the ranked alphabet satisfies $|\mathcal{F}_i| \leq 1$ for all $i \geq 0$. This means that for unlabeled trees the number of children of a node uniquely determines the label and thus the label is basically irrelevant. In particular, we discuss universal coding for unlabeled binary trees in Section 4.7 and the following sections. We model this class as trees over the ranked alphabet $\mathcal{F} = \{a, f\}$ where $a \in \mathcal{F}_0$ and $f \in \mathcal{F}_2$.

The depth $d(t)$ of a tree t is recursively defined by $d(a) = 0$ for $a \in \mathcal{F}_0$ and $d(f(t_1, \ldots, t_n)) = \max\{d(t_i) \mid i \in [1, n]\} + 1$ for $f \in \mathcal{F}_n$. Note that the depth $d(t)$ is still the maximal length of a path from the root to a leaf as it is defined in Section 2.3. For a tree t and a node $v \in \mathsf{nodes}(t)$, we denote by $\mathsf{subtree}_t(v)$ the subtree of t rooted at v. Vice versa, a tree t' is a subtree of t if there is a node $v \in \mathsf{nodes}(t)$ such that $t' = \mathsf{subtree}_t(v)$. Occasionally, we use the leaf size $\mathsf{leafsize}_t(v)$ of a node v of t which refers to the leaf size of $\mathsf{subtree}_t(v)$, i.e., $\mathsf{leafsize}_t(v) = \mathsf{leafsize}(\mathsf{subtree}_t(v))$. If t is clear from the context, we again omit the subscript and just write $\mathsf{subtree}(v)$, respectively $\mathsf{leafsize}(v)$.

Example 4.3. *Let $a, b \in \mathcal{F}_0$, $g \in \mathcal{F}_1$, $f \in \mathcal{F}_2$ and $t = f(f(a, a), g(b))$ (this tree is called t_3 in Example 4.1 and is depicted at the third left-to-right position in Figure 4.1). The depth of t is $d(t) = 2$. Let v be the unique node in $\mathsf{nodes}(t)$ such that $\lambda_t(v) = g$. We have $\mathsf{subtree}_t(v) = g(b)$ and $\mathsf{leafsize}(v) = 1$ since $\mathsf{subtree}_t(v)$ has exactly one leaf node.*

The following well known counting lemma will be needed several times:

Lemma 4.1.1. *Let $|\mathcal{F}| \leq \sigma$. The number of trees $t \in \mathcal{T}(\mathcal{F})$ such that $|t| \in [1, n]$ is bounded by $\frac{4}{3}(4\sigma)^n$.*

Proof. The number of rooted ordered (but unranked trees) with k nodes is

$$\frac{1}{k+1}\binom{2k}{k} \leq 4^k$$

(the k-th Catalan number, see e.g. [41, 107]). Hence, the number of trees in the lemma can be bounded by

$$\sigma^n \cdot \sum_{k=1}^{n} 4^k \leq \sigma^n \cdot \frac{4^{n+1} - 1}{3} \leq \frac{4}{3}(4\sigma)^n.$$

□

Tree patterns and contexts. Fix a ranked alphabet $\mathcal{F}$ in the following. Let $\mathcal{X} = \{x_1, x_2, \dots\}$ be a countably infinite set of symbols of rank 0 which we call *parameters*. We assume $\mathcal{F} \cap \mathcal{X} = \emptyset$. We use the parameters as leaf labels in the same sense as we use symbols from $\mathcal{F}_0$. Intuitively, a *tree pattern* or simply *pattern* is an incomplete tree with holes which are represented by different parameters. Formally, let $\mathcal{T}(\mathcal{F} \cup \mathcal{X})$ be the smallest set such that

- $x_i \in \mathcal{T}(\mathcal{F} \cup \mathcal{X})$ for $i \geq 1$,
- for all $a \in \mathcal{F}_0$ we have $a \in \mathcal{T}(\mathcal{F} \cup \mathcal{X})$ and
- if $f \in \mathcal{F}_i$ for $i \geq 1$ and $t_1, \dots, t_i \in \mathcal{T}(\mathcal{F} \cup \mathcal{X})$, then $f(t_1, \dots, t_i) \in \mathcal{T}(\mathcal{F} \cup \mathcal{X})$.

An element $t \in \mathcal{T}(\mathcal{F} \cup \mathcal{X})$ is called a pattern if and only if there do not exist different nodes in t that are labeled by the same parameter, i.e., no parameter x_i occurs more than once in the term t for $i \geq 1$. For example, if $h \in \mathcal{F}_3$ then $h(x_1, x_{21}, x_{99})$ and $h(x_1, x_2, x_3)$ are patterns, whereas $h(x_1, x_1, x_3)$ is not a pattern since x_1 is used twice as a label. Nodes that are labeled by parameters are also called *parameter nodes*. Note that each tree $t \in \mathcal{T}(\mathcal{F})$ is a pattern. We naturally extend the definitions of $\mathsf{nodes}(t)$, λ_t and $\mathsf{labels}(t)$ from trees over a ranked alphabet $\mathcal{F}$ to patterns over $\mathcal{F} \cup \mathcal{X}$ as it can be seen in Example 4.4. The size of a pattern t is $|t| = |\{v \in \mathsf{nodes}(t) \mid \lambda_t(v) \notin \mathcal{X}\}|$, i.e., we do not count parameter nodes. Similarly, the leaf size $\mathsf{leafsize}(t)$ of t is the number of leaves of t which are not labeled by a parameter.

Example 4.4. *Let $a \in \mathcal{F}_0$ and $f \in \mathcal{F}_2$. Consider the pattern $t = f(a, x_1)$ such that v_0 is the root node of t, the left child of v_0 is v_1 and the right child of v_0 is v_2. We have $\mathsf{nodes}(t) = \{v_0, v_1, v_2\}$, $\lambda_t(v_0) = f$, $\lambda_t(v_1) = a$, $\lambda_t(v_2) = x_1$ and $\mathsf{labels}(t) = \{x_1, a, f\}$. We have $|t| = 2$ and $\mathsf{leafsize}(t) = 1$ because there are two nodes which are not parameter nodes and one of those is a leaf. The pattern t is depicted at the first left-to-right position in Figure 4.2.*

We use $<_t$ for the depth-first-order on $\mathsf{nodes}(t)$. Formally, $u <_t v$ if u is an ancestor of v or if there exists a node w and $i < j$ such that the i-th child of w is an ancestor of u and the j-th child of w is an ancestor of v. A pattern t is *valid* if (i) $\mathsf{labels}(t) \cap \mathcal{X} = \{x_1, \dots, x_n\}$ for some $n \geq 0$ and (ii) for all $u, v \in \mathsf{nodes}(t)$ with $\lambda(u) = x_i$, $\lambda(v) = x_j$ and $u <_t v$ we have $i < j$. In other words, a pattern t is valid if and only if the parameter nodes in t are consecutively numbered (starting with x_1) with respect to the depth-first-order. Note that in the term representation of a pattern t, the left-to-right order of the nodes in the term matches the depth-first-order of the nodes in the pattern. For example, if $h \in \mathcal{F}_3$ then the patterns $h(x_{21}, x_2, x_{99})$, $h(x_{10}, x_{11}, x_{99})$ and $h(x_4, x_5, x_6)$ are not valid, whereas $h(x_1, x_2, x_3)$ is valid. For a pattern t we define $\mathsf{valid}(t)$ as the unique valid pattern which is obtained from t by renaming the parameters. For instance, if $h \in \mathcal{F}_3$ then $\mathsf{valid}(h(x_{21}, x_2, x_{99})) = h(x_1, x_2, x_3)$. For a valid pattern t in

which the parameters $x_1, \ldots, x_n$ occur, we write $\mathsf{rank}(t) = n$ (the *rank* of the pattern t is n). Note that a valid pattern of rank 0 is just a tree.

We call a valid pattern of rank 1 a *context*, i.e., a context has exactly one leaf which is labeled by a parameter x_1. In Section 4.6 and Section 4.9 where contexts are mainly used, we label the single parameter node by x instead of x_1 since there is no other parameter occurring in a context.

We say that a valid pattern $p \in \mathcal{T}(\mathcal{F} \cup \mathcal{X})$ of rank n occurs in a tree $t \in \mathcal{T}(\mathcal{F})$ if there exist trees $t_1, \ldots, t_n \in \mathcal{T}(\mathcal{F})$ such that replacing the parameter x_i by t_i for $i \in [1, n]$ yields a subtree of t. We also write $p[t_1, \ldots, t_n]$ for the tree (or pattern) which is obtained from the pattern p and the trees (or patterns) $t_1, \ldots, t_n$ by replacing the parameter x_i by t_i for $i \in [1, n]$. Note that in general $p[t_1, \ldots, t_n]$ is not a (valid) pattern if $t_1, \ldots, t_n$ are patterns because parameters could occur more than once after the replacements, but whenever we use replacements of this form we make sure that the result is a (valid) pattern. For example, if $h \in \mathcal{F}_3$ and $s = t = h(x_1, x_2, x_3)$ then $s[x_1, t[x_2, x_3, x_4], x_5] = h(x_1, h(x_2, x_3, x_4), x_5)$ is a valid pattern.

In the case of two contexts s and t, the single parameter ensures that $s[t]$ and $t[s]$ are again contexts. The depth $d(t)$ of a pattern t is the depth of the tree $t[a, \ldots, a]$ for some $a \in \mathcal{F}_0$, i.e., the depth of a pattern is still the maximal length of a path from the root to a leaf (including leaves labeled by parameters).

Example 4.5. *Let $a, b \in \mathcal{F}_0$, $g \in \mathcal{F}_1$ and $f \in \mathcal{F}_2$. Consider the valid pattern $p = f(f(x_1, a), x_2))$ with $\mathsf{rank}(t) = 2$. The pattern p is depicted at the second left-to-right position in Figure 4.2. Further, consider the tree $t = f(f(b, f(f(a, a), b)), g(f(a, g(b))))$ (this tree is called t_4 in Example 4.1 and is depicted at the last left-to-right position in Figure 4.1). The pattern p occurs in t because $p[a, b] = f(f(a, a), b))$ is a subtree of t.*

We extend the definition of $\mathsf{subtree}_t(v)$ from trees to patterns as follows: For a valid pattern $t \in \mathcal{T}(\mathcal{F} \cup \mathcal{X})$ and a node $v \in \mathsf{nodes}(t)$ we denote by $\mathsf{subtree}_t(v)$ the valid pattern $\mathsf{valid}(s)$, where s is the subtree rooted at v in t. We further write $t \setminus v$ for the valid pattern $\mathsf{valid}(r)$, where r is obtained from t by replacing the subtree rooted at v by a new parameter.

Example 4.6. *Let $a \in \mathcal{F}_0$, $f \in \mathcal{F}_2$ and $h \in \mathcal{F}_3$. Consider the valid pattern $t = h(a, h(x_1, a, f(a, x_2)), x_3)$ and let v be the unique f-labeled node. We have $\mathsf{subtree}_t(v) = f(a, x_1)$ and $t \setminus v = h(a, h(x_1, a, x_2), x_3)$. The patterns t (third left-to-right position), $\mathsf{subtree}_t(v)$ (first left-to-right position) and $t \setminus v$ (last left-to-right position) are depicted in Figure 4.2.*

4.2 Tree straight-line programs

For the purpose of defining tree grammars, we need two types of ranked alphabets: *Terminals* and *nonterminals*. The ranked alphabet of terminals is mainly called $\mathcal{F}$ in the following and we denote by $\mathcal{N}$ the ranked alphabet for the nonterminals. As before, we use $\mathcal{F}_i \subseteq \mathcal{F}$ for the terminals of rank i and $\mathcal{N}_i \subseteq \mathcal{N}$ for the nonterminals

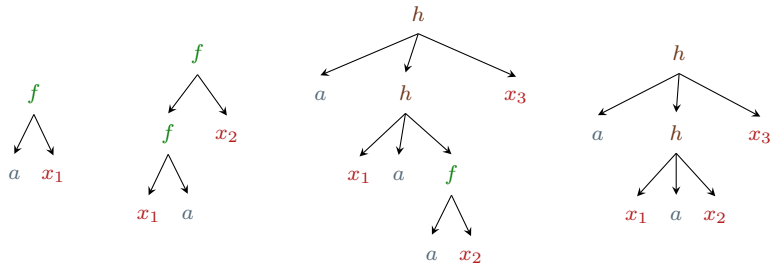

Figure 4.2: The patterns described in Example 4.4, Example 4.5 and Example 4.6.

of rank i. Further, we use $\mathcal{X} = \{x_1, x_2, \dots\}$ for the set of parameters as above. We assume that the three sets are pairwise disjoint, i.e., $\mathcal{N} \cap \mathcal{F} = \emptyset$, $\mathcal{N} \cap \mathcal{X} = \emptyset$ and $\mathcal{F} \cap \mathcal{X} = \emptyset$. A *tree straight-line program* (TSLP) $\mathcal{G}$ is a context-free tree grammar (see [32] for more details on context-free tree grammars) with the property that exactly one tree is derived. We use calligraphic capital letters (e.g. $\mathcal{G}$) for TSLPs in the following to distinguish from the notation we used for SLPs. The crucial point in order to extend SLPs (see Section 3.1) to TSLPs is the following: A rule of an SLP is of the form $A \to w$ where w is a string over the nonterminals and the alphabet symbols. A rule of a TSLP is of the form $A \to t$ where t is a valid pattern over $\mathcal{F} \cup \mathcal{N} \cup \mathcal{X}$.

Formally, a TSLP is a tuple $\mathcal{G} = (\mathcal{N}, \mathcal{F}, S, P)$, where $S \in \mathcal{N}_0$ is the start nonterminal, and P is a finite set of rules (or productions) of the form $A \to t$. If $n \geq 0$ and $A \in \mathcal{N}_n$, then it is required that $t \in \mathcal{T}(\mathcal{F} \cup \mathcal{N} \cup \mathcal{X})$ is a valid pattern such that $\mathsf{rank}(t) = n$. For instance, if $A \in \mathcal{N}_0$ then $t \in \mathcal{T}(\mathcal{F} \cup \mathcal{N})$ is a tree over the ranked alphabet $\mathcal{F} \cup \mathcal{N}$. As for SLPs, we have the following conditions in order to obtain the property that exactly one tree is derived from $\mathcal{G}$:

- For every $A \in \mathcal{N}$ there is exactly one rule $(A \to t) \in P$.
- The relation $\{(A, B) \in \mathcal{N} \times \mathcal{N} \mid (A \to t) \in P, B \in \mathsf{labels}(t)\}$ is acyclic.

The above conditions ensure that from every nonterminal $A \in \mathcal{N}_n$ exactly one valid pattern $\mathsf{val}_\mathcal{G}(A) \in \mathcal{T}(\mathcal{F} \cup \mathcal{X})$ with $\mathsf{rank}(\mathsf{val}_\mathcal{G}(A)) = n$ is derived by using the rules as rewrite rules in the usual sense until no nonterminal occurs.

Formally, for valid patterns $t, t' \in \mathcal{T}(\mathcal{F} \cup \mathcal{N} \cup \mathcal{X})$ we say t' is derived from t by $\mathcal{G}$, briefly $t \Rightarrow_\mathcal{G} t'$, if and only if there is $A \in \mathcal{N}_n$ (for some $n \geq 0$) and a rule $(A \to s) \in P$ such that t contains a node $v \in \mathsf{nodes}(t)$ with $\lambda_t(v) = A$ and replacing the subtree $A(t_1, \dots, t_n)$ rooted at v in t by $s[t_1, \dots, t_n]$ yields the valid pattern t'. Let $\Rightarrow^*_\mathcal{G}$ be the reflexive, transitive closure of $\Rightarrow_\mathcal{G}$. Then we have $\mathsf{val}_\mathcal{G}(A) \in \mathcal{T}(\mathcal{F} \cup \mathcal{X})$ with $A \in \mathcal{N}_n$ is the uniqe pattern such that $A(x_1, \dots, x_n) \Rightarrow^*_\mathcal{G} \mathsf{val}_\mathcal{G}(A)$.

We omit the subscript $\mathcal{G}$ and simply use $\mathsf{val}(A)$ and $\Rightarrow$ if the TSLP $\mathcal{G}$ is clear from the context. The tree defined by $\mathcal{G}$ is $\mathsf{val}(\mathcal{G}) = \mathsf{val}_\mathcal{G}(S)$. The size $|\mathcal{G}|$ of a TSLP $\mathcal{G} = (\mathcal{N}, \mathcal{F}, S, P)$ is the total size of all patterns on the right-hand sides of rules in P, i.e., $|\mathcal{G}| = \sum_{(A \to t) \in P} |t|$.

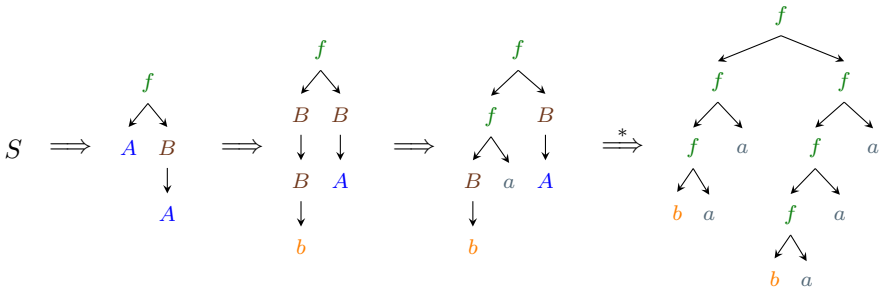

Figure 4.3: A derivation of the TSLP from Example 4.7.

Example 4.7. *Let* $\mathcal{G} = (\{S, A, B\}, \{a, b, f\}, S, P)$ *with* $S, A \in \mathcal{N}_0, B \in \mathcal{N}_1$, $a, b \in \mathcal{F}_0$, $f \in \mathcal{F}_2$ *and rules*

$$P = \{S \to f(A, B(A)),\ A \to B(B(b)),\ B \to f(x_1, a)\}.$$

A possible derivation of $\mathsf{val}(\mathcal{G})$ *from* S *is depicted in Figure 4.3. We have* $|\mathcal{G}| = 9$.

Note that for the size of a TSLP we do not count nodes of right-hand sides that are labeled by a parameter. To justify this, we use the following internal representation of valid patterns (which is also used in [66]): For every non-parameter node v of a tree, with children $v_1, \ldots, v_n$ we store in a list all pairs (i, v_i) such that v_i is a non-parameter node. Moreover, we store for every symbol (node label) its rank. This allows to reconstruct the valid pattern, since we know the positions where parameters have to be inserted.

Chomsky normal form and monadic TSLPs. A TSLP is in *Chomsky normal form* (see e.g. [85]) if for every rule $A \to t$ with $A \in \mathcal{N}_n$ for some $n \geq 0$ one of the following two cases holds:

$$t = B(x_1, \ldots, x_{i-1}, C(x_i, \ldots, x_k), x_{k+1}, \ldots, x_n) \text{ for } B, C \in \mathcal{N} \tag{4.3}$$

$$t = f(x_1, \ldots, x_n) \text{ for } f \in \mathcal{F}_n. \tag{4.4}$$

If the tree t in the corresponding rule $A \to t$ is of type (4.3), we write $\mathsf{index}(A) = i$. If otherwise t is of type (4.4), we write $\mathsf{index}(A) = 0$. One can transform every TSLP efficiently into an equivalent TSLP in Chomsky normal form with a small size increase. More precisely, it is shown in [85, Theorem 5] that from a TSLP $\mathcal{G}$, where all terminals and nonterminals have rank at most r, one can construct in time $\mathcal{O}(r \cdot |\mathcal{G}|)$ a TSLP $\mathcal{G}'$ in Chomsky normal form of size $|\mathcal{G}'| \leq \mathcal{O}(\mathcal{G})$ such that $\mathsf{val}(\mathcal{G}') = \mathsf{val}(\mathcal{G})$.

We define the rooted, ordered *derivation tree* $\mathcal{D}_\mathcal{G}$ of a TSLP $\mathcal{G} = (\mathcal{N}, \mathcal{F}, S, P)$ in Chomsky normal form: The inner nodes of the derivation tree are labeled by nonterminals and the leaves are labeled by terminal symbols. Formally, we start with the root node of $\mathcal{D}_\mathcal{G}$ and label it by the start nonterminal S. Then, for

every nonterminal A whose corresponding right-hand side is of type (4.3) and every node in $\mathcal{D}_\mathcal{G}$ labeled by A, we attach a left child labeled by B and a right child labeled by C. If the right-hand side of the rule for A is of type (4.4), we attach a single child labeled by f to A. Note that these nodes are the leaves of $\mathcal{D}_\mathcal{G}$ and they represent the nodes of the tree $\mathsf{val}(\mathcal{G})$. We denote by $\mathrm{depth}(\mathcal{G})$ the depth of the derivation tree $\mathcal{D}_\mathcal{G}$.

A TSLP is called *monadic* if every nonterminal has rank at most one and thus each right-hand side is a pattern of rank at most one (a tree or a context). The following result was shown in [85]:

Theorem 4.2.1 ([85, Theorem 10])**.** *From a given TSLP $\mathcal{G}$ in Chomsky normal form such that every nonterminal has rank at most k and every terminal symbol has rank at most r, one can compute in time $\mathcal{O}(|\mathcal{G}| \cdot k \cdot r)$ a monadic TSLP $\mathcal{H}$ with the following properties:*

- $\mathsf{val}(\mathcal{G}) = \mathsf{val}(\mathcal{H})$,
- $|\mathcal{H}| \leq \mathcal{O}(|\mathcal{G}| \cdot r)$,
- $\mathrm{depth}(\mathcal{H}) \leq \mathcal{O}(\mathrm{depth}(\mathcal{G}))$.

Moreover, one can assume that every rule of $\mathcal{H} = (\mathcal{N}, \mathcal{F}, S, P)$ has one of the four forms

- $A \to B(C)$ $\qquad (A, C \in \mathcal{N}_0,\ B \in \mathcal{N}_1)$
- $A \to f(A_1, \ldots, A_n)$ $\qquad (A, A_1, \ldots, A_n \in \mathcal{N}_0)$
- $A \to B(C(x_1))$ $\qquad (A, B, C \in \mathcal{N}_1)$
- $A \to f(A_1, \ldots, A_{i-1}, x_1, A_i, \ldots, A_{n-1})$ $\qquad (A_1, \ldots, A_{n-1} \in \mathcal{N}_0,\ A \in \mathcal{N}_1)$

where $f \in \mathcal{F}_n$ is a terminal symbol.

When we use monadic TSLPs later we will simply label the parameter node by x instead of x_1 (since there is at most one parameter occurring on each right-hand side).

Grammar-based tree compressors. A *grammar-based tree compressor* ψ is an algorithm which constructs from a given tree $t \in \mathcal{T}(\mathcal{F})$ a TSLP $\psi(t)$ which produces the tree t. Key results of this chapter are the constructions of two grammar-based tree compressors: TreeBiSection (Section 4.4) and BU-Shrink (Section 4.5). Both algorithms use a well known tree compression technique where a tree is represented by a directed acyclic graph. This concept is explained in the next section. The main achievement of both grammar-based tree compressors is that the produced TSLP has size $\mathcal{O}(n/\log_\sigma n)$, where n is the size of the input tree t and $\sigma = |\mathsf{labels}(t)|$ is the number of labels occurring in t. It is important that this result is based on the assumption that the maximal number of children of a node in t is bounded by a constant. In Section 4.4.3 the reader finds a discussion as well as a simple workaround solution for the case that the number of children of a node is unrestricted.

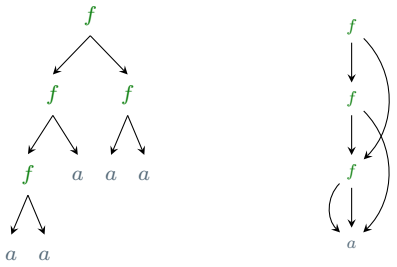

Figure 4.4: A tree (left) and its minimal DAG (right).

4.3 Directed acyclic graphs

A commonly used tree compression scheme is obtained by writing down repeated subtrees only once. In that case all occurrences except for the first are replaced by a pointer to the first one. This leads to a node-labeled minimal *directed acyclic graph* or briefly minimal DAG for a tree. It is well known that every tree t has a unique minimal DAG which we call simply the DAG of t or $\mathsf{DAG}(t)$ in the following. The size $|\mathsf{DAG}(t)|$ is the number of nodes of the DAG of t, i.e., $|\mathsf{DAG}(t)|$ is the number of different (pairwise non-isomorphic) subtrees of the input tree t. An example can be found in Figure 4.4 where the minimal DAG of size 4 is shown for the input tree $t = f(f(f(a,a),a),f(a,a))$.

The DAG of a tree can be seen as a TSLP in which every nonterminal has rank zero: Each node v of the DAG corresponds to a nonterminal $A_v \in \mathcal{N}_0$. A node v with label $f \in \mathcal{F}_n$ and n children $v_1, \ldots, v_n$ corresponds to the rule $A_v \to f(A_{v_1}, \ldots, A_{v_n})$. The root node of the DAG corresponds to the start nonterminal. In this form, the DAG can be seen as grammar-based tree compressor where the produced TSLP has size at most $r \cdot |\mathsf{DAG}(t)|$, where r is the maximal rank of a symbol in the input tree t. The TSLP which corresponds to the DAG depicted in Figure 4.4 is shown in Example 4.8.

Example 4.8. *Let $a \in \mathcal{F}_0$, $f \in \mathcal{F}_2$ and $t = f(f(f(a,a),a),f(a,a))$. The tree t is depicted on the left of Figure 4.4. The minimal DAG of t is depicted on the right of Figure 4.4. If this DAG is represented by a TSLP, we obtain $(\{A_0, A_1, A_2, A_3\}, \{f, a\}, A_0, P)$, where*

$$P = \{A_0 \to f(A_1, A_2),\ A_1 \to f(A_2, A_3),\ A_2 \to f(A_3, A_3),\ A_3 \to a\}.$$

Vice versa, it is straightforward to transform a TSLP in which every nonterminal has rank zero into an equivalent (not necessarily minimal) directed acyclic graph. It is shown in [38] that the DAG of a tree t can be constructed in time $\mathcal{O}(|t|)$. The following lemma shows that the DAG of a tree can be also constructed in logspace.

Lemma 4.3.1. *The DAG of a given tree t can be computed in logspace.*

Proof. Assume that the node set of the input tree t is $\mathsf{nodes}(t) = \{1, \dots, n\}$. Given two nodes i, j of t one can verify in logspace whether the subtrees $\mathsf{subtree}_t(i)$ and $\mathsf{subtree}_t(j)$ are isomorphic (we write $\mathsf{subtree}_t(i) \cong \mathsf{subtree}_t(j)$ in this case) by performing a depth-first left-to-right traversal over both trees and thereby comparing the two trees symbol by symbol.

The nodes and edges of the DAG of t can be enumerated in logspace as follows. A node $i \in [1, n]$ of t is a node of the DAG if there is no $j < i$ with $\mathsf{subtree}_t(i) \cong \mathsf{subtree}_t(j)$. By the above remark, this can be checked in logspace. Let i be a node of the DAG and let j be the k-th child of i in t. Then j' is the k-th child of i in the DAG where j' is the smallest number such that $\mathsf{subtree}_t(j') \cong \mathsf{subtree}_t(j)$. Again by the above remark this j' can be found in logspace. □

The following example first shows that DAG compression is not enough to obtain a TSLP of size $o(n)$ for an input tree t of size n even if t is an unlabeled binary tree. Further, it shows the advantage of TSLPs over DAGs when nonterminals of rank greater than zero are allowed:

Example 4.9. *Consider the binary tree* $t_n = f(f(f(\cdots f(a, a), \cdots a), a), a)$, *where* $f \in \mathcal{F}_2$ *occurs* n *times and* $a \in \mathcal{F}_0$ *occurs* $n + 1$ *times, i.e.,* $|t_n| = 2n + 1$ *and* $\mathsf{leafsize}(t_n) = n + 1$. *All subtrees of* t_n *which are rooted at an* f*-labeled node have different size and thus are pairwise non-isomorphic. Adding the subtree with a single node labeled by* a *yields* $|\mathsf{DAG}(t)| = n + 1$.

Consider the TSLP $\mathcal{G}_k = (\{A_0, \dots, A_k\}, \{a, f\}, A_0, P)$, *where* P *contains*

$$A_k \to f(x_1, a), \quad A_i \to A_{i+1}(A_{i+1}(x_1)) \text{ for } i \in [1, k-1], \quad A_0 \to A_1(A_1(a)).$$

We have $\mathsf{val}(\mathcal{G}_k) = t_{2^k}$ *and* $|\mathcal{G}_k| = 2k + 3$.

4.3.1 DAG compression of weakly balanced binary trees

We provide a key result of this chapter in the following. We prove in Theorem 4.3.2 that the DAG of certain weakly balanced binary trees has size $\mathcal{O}(n/\log_\sigma n)$, where σ is the number of different labels occurring in the input tree. We will use this result in Section 4.4.2, where we analyze the size of the TSLP produced by the grammar-based tree compressor TreeBiSection. Further, we apply this result in Section 4.8, where DAG compression is used for universal source coding.

Let $t \in \mathcal{T}(\mathcal{F})$ be a binary tree, i.e., all nodes of t have either zero or two children and thus $\mathsf{labels}(t) \subseteq \mathcal{F}_0 \cup \mathcal{F}_2$. Let $\beta \in \mathbb{R}$ such that $0 < \beta \leq 1$. Recall that the leaf size $\mathsf{leafsize}_t(v)$ of a node $v \in \mathsf{nodes}(t)$ is the number of leaves of the subtree rooted at v, i.e., $\mathsf{leafsize}_t(v) = \mathsf{leafsize}(\mathsf{subtree}_t(v))$. We say that an inner node v with children v_1 and v_2 is *β-balanced* if the following holds: If $\mathsf{leafsize}_t(v_i) = n_i$ for $i \in [1, 2]$, then $n_1 \geq \beta n_2$ and $n_2 \geq \beta n_1$. We say that t is *β-balanced* if the following holds: For all inner nodes u and v such that v is a child of u, we have that u is β-balanced or v is β-balanced.

Theorem 4.3.2. *If t is a β-balanced binary tree such that $|\mathsf{labels}(t)| = \sigma$ and $\mathsf{leafsize}(t) = n$, then $|\mathsf{DAG}(t)| \leq \mathcal{O}\big(\frac{\alpha \cdot n}{\log_\sigma n}\big)$, where $\alpha = 1 + \log_{1+\beta}(\beta^{-1})$ only depends on β.*[2]

Proof. Let us fix a binary tree t with $\mathsf{leafsize}(t) = n$ and $|\mathsf{labels}(t)| = \sigma$ as in the theorem. First, note that $|t| = 2n - 1$ and $\sigma \leq 2n - 1$. Let us fix a number k that will be defined later. We first bound the number of different subtrees with at most k leaves in t. Afterwards we will estimate the size of the remaining *top tree*. The same strategy is used for instance in [55, 79] to derive a worst-case upper bound on the size of binary decision diagrams.

Claim 1. *The number of different subtrees of t with at most k leaves is bounded by d^k with $d = 16\sigma^2$.*

Proof. A subtree of t with at most k leaves has at most $2k - 1$ nodes, each of which is labeled by one of σ many labels. Hence, by Lemma 4.1.1 we can bound the number of different subtrees of t with at most k leaves by

$$\frac{4}{3}(4\sigma)^{2k-1} = \frac{1}{3\sigma}(4\sigma)^{2k} \leq (16\sigma^2)^k.$$

Let $\mathrm{top}(t, k)$ be the tree obtained from t by removing all nodes with leaf size at most k. Recall that $\alpha = 1 + \log_{1+\beta}(\beta^{-1})$.

Claim 2. *The number of nodes of $\mathrm{top}(t, k)$ is bounded by $4\alpha \cdot \frac{n}{k}$.*

Proof. The tree $\mathrm{top}(t, k)$ has at most n/k leaves since it is obtained from t by removing all nodes with leaf size at most k. Each node in $\mathrm{top}(t, k)$ has still at most two children but in contrast to the tree t there might be nodes with exactly one child in $\mathrm{top}(t, k)$ since it is possible that only one of the two children of a node in t has leaf size at most k. We show that the length of every *unary chain* in $\mathrm{top}(t, k)$ is bounded by 2α, where a unary chain is a sequence of (parent-child) nodes with exactly one child each. This implies that $\mathrm{top}(t, k)$ has at most $4\alpha \cdot n/k$ many nodes.

Let $v_1, \ldots, v_m$ be a unary chain in $\mathrm{top}(t, k)$ where v_i is the single child node of v_{i+1} for $i \in [1, m - 1]$. Moreover, let v'_i be the removed sibling of v_i in t, see Figure 4.5. Note that each node v'_i has leaf size at most k.

We claim that the leaf size of v_{2i+1} is larger than $(1 + \beta)^i k$ for all i with $2i + 1 \leq m$. For $i = 0$ note that v_1 has leaf size more than k since otherwise it would have been removed in $\mathrm{top}(t, k)$. For the induction step, assume that the leaf size of v_{2i-1} is larger than $(1 + \beta)^{i-1}k$. One of the nodes v_{2i} and v_{2i+1} must be β-balanced. Hence, v'_{2i-1} or v'_{2i} must have leaf size at least $\beta(1 + \beta)^{i-1}k$. Hence, v_{2i+1} has leaf size more than $(1 + \beta)^{i-1}k + \beta(1 + \beta)^{i-1}k = (1 + \beta)^i k$.

If $m \geq 2\alpha + 1$, then $v_{2\alpha-1}$ exists and has leaf size at least $(1 + \beta)^{\alpha-1}k = k/\beta$, which implies that the leaf size of $v'_{2\alpha-1}$ or $v'_{2\alpha}$ (both nodes exist) is more than k, which is a contradiction. Hence, we must have $m \leq 2\alpha$. Figure 4.5 shows an illustration of our argument.

[2] Since $0 < \beta \leq 1$, we have $\alpha \geq 1$.

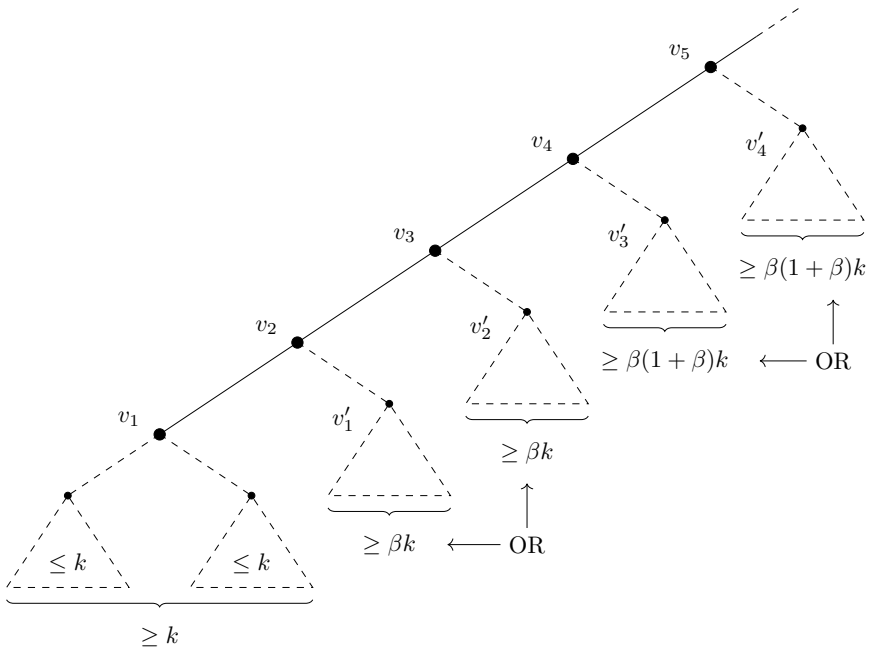

Figure 4.5: A chain within a top tree. The subtree rooted at v_1 has more than k leaves.

Using Claim 1 and 2 we can now prove the theorem: The number of nodes of the DAG of t is bounded by the number of different subtrees with at most k leaves (Claim 1) plus the size of the remaining tree $\mathrm{top}(t,k)$ (Claim 2). Let $k = \frac{1}{2}\log_d n$. Recall that $d = 16\sigma^2$ and hence $\log d = 4 + 2\log\sigma$, which implies that $\log_d n = \Theta(\log_\sigma n)$. With Claim 1 and 2 we get the following bound on the size of $\mathsf{DAG}(t)$:

$$d^k + 4\alpha \cdot \frac{n}{k} = d^{(\log_d n)/2} + \frac{8\alpha \cdot n}{\log_d n} = \sqrt{n} + \frac{8\alpha \cdot n}{\log_d n} \leq \mathcal{O}\left(\frac{\alpha \cdot n}{\log_d n}\right) = \mathcal{O}\left(\frac{\alpha \cdot n}{\log_\sigma n}\right)$$

This proves the theorem. □

Obviously, one could relax the definition of a β-balanced tree by only requiring that if $(v_1, v_2, \ldots, v_\delta)$ is a path down in the tree, where δ is a constant, then one of the nodes $v_1, v_2, \ldots, v_\delta$ must be β-balanced. Theorem 4.3.2 would still hold in this setting with α depending linearly on δ.

In the remaining section we present some more results on the size of DAGs that are related to Theorem 4.3.2. If β is a constant, then a β-balanced binary tree t has depth $\mathcal{O}(\log|t|)$. One might think that this logarithmic depth is responsible for the small size of the DAG in Theorem 4.3.2. But this intuition is wrong:

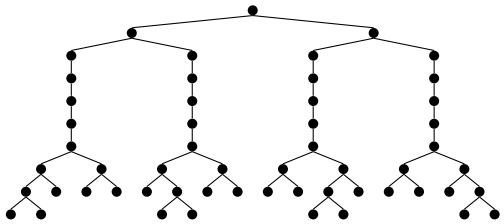

Figure 4.6: Tree t_{16} from the proof of Theorem 4.3.3. The labels (and the edge directions) are omitted due to better clarity of the figure. Formally, each leaf is labeled by a, each node with one child is labeled by b and each node with two children is labeled by c.

Theorem 4.3.3. *There is a family of trees $t_n \in \mathcal{T}(\{a, b, c\})$ with $a \in \mathcal{F}_0$, $b \in \mathcal{F}_1$, and $c \in \mathcal{F}_2$ $(n \geq 1)$ with the following properties:*

- *The depth of t_n is $\Theta(\log n)$.*
- $|t_n| = \Theta(n)$
- $|\mathsf{DAG}(t_n)| \geq n$

Proof. Let $k = n/\log n$ (we ignore rounding problems with $\log n$, which only affect multiplicative factors). Choose k different binary trees $s_1, \ldots, s_k \in \mathcal{T}(\{a, c\})$, each having $\log n$ many internal nodes. This is possible: By the asymptotic formula for the Catalan numbers (see e.g. [41, p. 38]) the number of different binary trees with $\log n$ many internal nodes is asymptotically equal to

$$\frac{4^{\log n}}{\sqrt{\pi \cdot \log^3 n}} = \frac{n^2}{\sqrt{\pi \cdot \log^3 n}} > n.$$

Then consider the trees $s'_i = b^{\log n}(s_i)$, i.e., s'_i starts with a unary chain of $\log n$ many nodes (all labeled by b) and at the end of this chain we attach the tree s_i. Each of these trees has size $\Theta(\log n)$ and depth $\Theta(\log n)$. Next, let $u_n \in \mathcal{T}(\{c\} \cup \mathcal{X})$ be a pattern of rank k (u_n contains the parameters $x_1, \ldots, x_k$ as leaf labels) such that u_n is a balanced binary valid pattern (all non-parameter nodes are labeled by c) of depth $\Theta(\log k) = \Theta(\log n)$ and size $\Theta(k) = \Theta(n/\log n)$. We finally take $t_n = u_n[s'_1, \ldots, s'_k]$. Figure 4.6 shows the tree t_{16}. We obtain

$$|t_n| = \Theta\left(\frac{n}{\log n}\right) + \Theta(k \cdot \log n) = \Theta(n).$$

The depth of t_n is $\Theta(\log n)$. Finally, in the DAG of t_n the unary b-labeled nodes cannot be shared. Basically, the pairwise different trees $s_1, \ldots, s_k$ work as different leaf labels that are attached to the b-chains. But the number of b-labeled nodes in t_n is $k \cdot \log n = n$. □

Note that the trees from Theorem 4.3.3 are not β-balanced for any constant $0 < \beta < 1$, and by Theorem 4.3.2 this is necessarily the case. Interestingly, if we assume that every subtree s of a binary tree t has depth at most $\mathcal{O}(\log |s|)$, then Hübschle-Schneider and Raman [58] have implicitly shown the bound $\mathcal{O}((n \cdot \log \log_\sigma n)/\log_\sigma n)$ for the size of the minimal DAG.

Theorem 4.3.4 ([58])**.** *Let α be a constant. Then there is a constant β that only depends on α such that the following holds: If t is a binary tree of size n with $|\mathsf{labels}(t)| = \sigma$ such that every subtree s of t has depth at most $\alpha \log |s| + \alpha$, then $|\mathsf{DAG}(t)| \leq \frac{\beta \cdot n \cdot \log \log_\sigma n}{\log_\sigma n} + \beta$.*

We can show that the bound in this result is (asymptotically) sharp:

Theorem 4.3.5. *There is a family of trees $t_n \in \mathcal{T}(\{a, b, c\})$ with $a \in \mathcal{F}_0$, $b \in \mathcal{F}_1$, and $c \in \mathcal{F}_2$ $(n \geq 1)$ with the following properties:*[3]

- $|t_n| = \Theta(n)$
- *Every subtree s of a tree t_n has depth $\mathcal{O}(\log |s|)$.*
- *The size of the minimal dag of t_n is $\Omega(\frac{n \cdot \log \log n}{\log n})$.*

Proof. The tree t_n is similar to the one from the proof of Theorem 4.3.3. Again, let $k = n/\log n$. Fix a balanced binary tree $v_n \in \mathcal{T}(\{a, c\})$ with $\log k = \Theta(\log n)$ many leaves. From v_n we construct k many different trees $s_1, \ldots, s_k \in \mathcal{T}(\{a, b, c\})$ by choosing in v_n an arbitrary subset of leaves (there are k such subsets) and replacing all leaves in that subset by $b(a)$. Note that $|s_i| = \Theta(\log n)$. Moreover, every subtree s of a tree s_i has depth $\mathcal{O}(\log |s|)$ (since v_n is balanced). Then consider the trees $s'_i = b^{\log \log n}(s_i)$ (so, in contrast to the proof of Theorem 4.3.3, the length of the unary chains is $\log \log n$). Clearly, $|s'_i| = \Theta(\log n)$. Moreover, we still have the property that every subtree s of a tree s'_i has depth $\mathcal{O}(\log |s|)$: If the subtree s is rooted in a node from s_i, then this is clear. Otherwise, the subtree s has the form $b^h(s_i)$ for some $h \leq \log \log n$. This tree has depth $h + \Theta(\log \log n) = \Theta(\log \log n)$ and size $\Theta(\log n)$. Finally we combine the trees $s'_1, \ldots, s'_k \in \mathcal{T}(\{a, b, c\})$ in a balanced way to a single tree using the binary valid pattern u_n of rank k from the proof of Theorem 4.3.3, i.e., we define $t_n = u_n[s'_1, \ldots, s'_k]$. Then, $|t_n| = \Theta(n)$. Moreover, by the same argument as in the proof of Theorem 4.3.3,we have

$$|\mathsf{DAG}(t_n)| \geq \Omega\left(\frac{n \cdot \log \log n}{\log n}\right)$$

since the nodes in the $k = n/\log n$ many unary chains of length $\log \log n$ cannot be shared with other nodes. It remains to show that every subtree s of t_n has depth $\mathcal{O}(\log |s|)$. For the case that s is a subtree of one of the trees s'_i, this has been already shown above. But the case that s is rooted

[3] Again, the unary node label b can replaced by the pattern $c(d, x)$, where $d \in \mathcal{F}_0 \setminus \{a\}$ to obtain a binary tree.

in a node from the pattern u_n is also clear: In that case, s is of the form $s = u'[s'_i, \ldots, s'_j]$, where $u' = \mathsf{subtree}_{u_n}(v)$ for some node $v \in \mathsf{nodes}(u_n)$ and $1 \leq i \leq j \leq k$. Assume that d is the depth of u'. Since u_n is a balanced, we have $|u'| \geq \Omega(2^d)$ and $j - i + 1 \geq \Omega(2^d)$. Hence, $s = u'[s'_i, \ldots, s'_j]$ has size $\Omega(2^d + 2^d \cdot \log n) = \Omega(2^d \cdot \log n)$ and depth $d + \Theta(\log \log n)$. This shows the desired property since $\log(2^d \cdot \log n) = d + \log \log n$. □

Hübschle-Schneider and Raman [58] used Theorem 4.3.4 to prove the upper bound $\mathcal{O}((n \cdot \log \log_\sigma n)/\log_\sigma n)$ for the size of the top dag of an unranked tree of size n with σ many node labels.

4.4 TreeBiSection

Let $t \in \mathcal{T}(\mathcal{F})$ be a tree of size n over a ranked alphabet $\mathcal{F}$ with $|\mathsf{labels}(t)| = \sigma$. In this section we present the grammar-based tree compressor TreeBiSection. The goal is to construct a TSLP for t of size $\mathcal{O}(n/\log_\sigma n)$ and depth $\mathcal{O}(\log n)$, assuming the maximal rank of symbols in $\mathsf{labels}(t)$ is bounded by a constant. TreeBiSection achieves this while only using logarithmic space, but needs time $\mathcal{O}(n \cdot \log n)$. In the next section, we introduce a second algorithm (BU-Shrink) which constructs a TSLP of size $\mathcal{O}(n/\log_\sigma n)$ in linear time, but the depth of the constructed TSLP is not necessarily logarithmic. We then combine BU-Shrink and TreeBiSection to obtain a linear time algorithm that achieves the claimed size as well as logarithmic depth[4].

TreeBiSection is a generalization of the grammar-based string compressor BiSection described in Section 3.3. A key aspect of TreeBiSection is the well known idea of splitting a tree recursively into smaller parts of roughly equal size, see e.g. [21, 106]. The following lemma is well known, at least for binary trees (see e.g. [78]).

Lemma 4.4.1. *Let $t \in \mathcal{T}(\mathcal{F} \cup \mathcal{X})$ be a pattern over some ranked alphabet $\mathcal{F}$ with $|t| \geq 2$ such that every node has at most r children. Then there is a node $v \in \mathsf{nodes}(t)$ such that*

$$\frac{1}{2(r+2)} \cdot |t| \;\leq\; |\mathsf{subtree}_t(v)| \;\leq\; \frac{r+1}{r+2} \cdot |t|.$$

Proof. We start a search at the root node, checking at each node v whether $|\mathsf{subtree}_t(v)| \leq \frac{d+1}{d+2} \cdot |t|$, where d is the number of children of v. If the property does not hold, we continue the search at a child that spawns a largest subtree (using an arbitrary deterministic tie-breaking rule that is fixed for the further discussion). Note that we eventually reach a node such that $|\mathsf{subtree}_t(v)| \leq \frac{d+1}{d+2} \cdot |t|$: If $|\mathsf{subtree}_t(v)| = 1$, then $|\mathsf{subtree}_t(v)| \leq \frac{1}{2}|t|$ since $|t| \geq 2$.

[4] By a recently introduced balancing technique presented in [48] one could achieve logarithmic depth for the TSLP obtained by BU-Shrink in linear time without using TreeBiSection.

So, let v be the first node with $|\mathsf{subtree}_t(v)| \leq \frac{d+1}{d+2} \cdot |t|$, where d is the number of children of v. We get

$$|\mathsf{subtree}_t(v)| \leq \frac{d+1}{d+2} \cdot |t| \leq \frac{r+1}{r+2} \cdot |t|.$$

Moreover v cannot be the root node. Let $u \in \mathsf{nodes}(t)$ be the parent node of v and let e be the number of children of u. Since v spans a largest subtree among the children of u, we get $e \cdot |\mathsf{subtree}_t(v)| + 1 \geq |\mathsf{subtree}_t(u)| \geq \frac{e+1}{e+2} \cdot |t|$, i.e.,

$$\begin{aligned}
|\mathsf{subtree}_t(v)| &\geq \frac{e+1}{e(e+2)} \cdot |t| - \frac{1}{e} \\
&= \left(\frac{e+1}{e(e+2)} - \frac{1}{|t| \cdot e} \right) \cdot |t| \\
&\geq \left(\frac{e+1}{e(e+2)} - \frac{1}{2e} \right) \cdot |t| \\
&= \frac{1}{2(e+2)} \cdot |t| \\
&\geq \frac{1}{2(r+2)} \cdot |t|.
\end{aligned}$$

This proves the lemma. □

For the remainder of this section we refer with $\mathsf{split}(t)$ to the unique node in a tree or pattern t which is obtained by the procedure from the proof above. Based on Lemma 4.4.1 we now construct a TSLP $\mathcal{G}_t = (\mathcal{N}, \mathcal{F}, S, P)$ with $\mathsf{val}(\mathcal{G}_t) = t$ for a given tree $t \in \mathcal{T}(\mathcal{F})$. It is *not* the final TSLP produced by TreeBiSection. For our later analysis, it is important to bound the number of parameters in the TSLP $\mathcal{G}_t$ by a constant. To achieve this, we use an idea from Ruzzo's paper [101].

We will first present the construction and analysis of $\mathcal{G}_t$ only for trees in which every node has at most two children, i.e., we consider trees over a ranked alphabet $\mathcal{F}$ such that $\mathcal{F}_i = \emptyset$ for $i \geq 3$. Let us fix a ranked alphabet $\mathcal{F}_{\leq 2}$ with this property in the following (the naming emphasizes the bound on the rank). In Section 4.4.3, we will consider trees of larger branching degree. For the case that $r = 2$, Lemma 4.4.1 yields for every pattern $s \in \mathcal{T}(\mathcal{F}_{\leq 2} \cup \mathcal{X})$ with $|s| \geq 2$ a node $v = \mathsf{split}(s)$ such that

$$\frac{1}{8} \cdot |s| \;\leq\; |\mathsf{subtree}_s(v)| \;\leq\; \frac{3}{4} \cdot |s|. \tag{4.5}$$

Consider an input tree $t \in \mathcal{T}(\mathcal{F}_{\leq 2})$ (we assume that $|t| \geq 2$). The TSLP $\mathcal{G}_t$ we are going to construct has the property that every nonterminal has rank at most three. Our algorithm stores two sets of rules, P_{temp} and P_{final}. The set P_{final} will contain the rules in the TSLP $\mathcal{G}_t$ and the rules from P_{temp} ensure that the TSLP with rules $P_{\text{temp}} \cup P_{\text{final}}$ produces t at any given time of the procedure. We start with the initial setting $P_{\text{temp}} = \{S \to t\}$ and $P_{\text{final}} = \emptyset$, where S is the start nonterminal. While P_{temp} is non-empty we proceed for each rule $(A \to s) \in P_{\text{temp}}$ as follows:

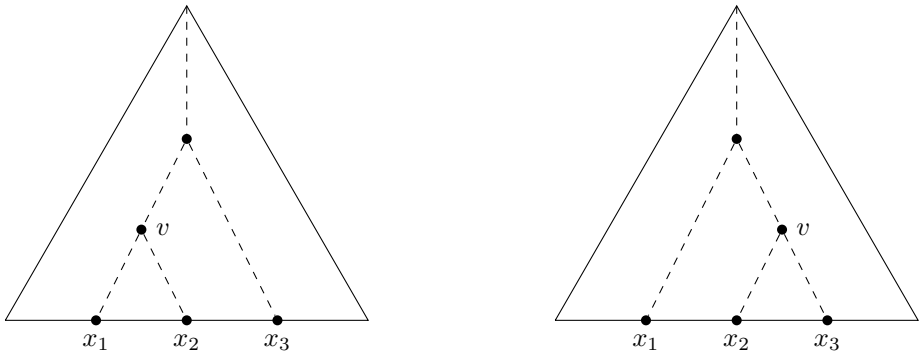

Figure 4.7: Splitting a tree with three parameters

Remove $A \to s$ from P_{temp}. Let $A \in \mathcal{N}_r$. If $r \leq 2$ we determine the node $v = \mathsf{split}(s)$ in s. Then we split the pattern s into $\mathsf{subtree}_s(v)$ and $s \setminus v$. Let $r_1 = \mathsf{rank}(\mathsf{subtree}_s(v))$ and $r_2 = \mathsf{rank}(s \setminus v)$. We add two fresh nonterminals to $\mathcal{N}$ such that $A_1 \in \mathcal{N}_{r_1}$ and $A_2 \in \mathcal{N}_{r_2}$. Note that $r = r_1 + r_2 - 1$. If $|\mathsf{subtree}_s(v)| > 1$ ($|s \setminus v| > 1$, respectively) then we add the rule $A_1 \to \mathsf{subtree}_s(v)$ ($A_2 \to s \setminus v$, respectively) to P_{temp}. Otherwise we add it to P_{final} as a final rule. Let k be the number of nodes of s that are labeled by a parameter and that are smaller (w.r.t. $<_s$) than v. To link the nonterminal A to the fresh nonterminals A_1 and A_2 we add the rule

$$A \to A_1(x_1, \ldots, x_k, A_2(x_{k+1}, \ldots, x_{k+r_2}), x_{k+r_2+1}, \ldots, x_r)$$

to the set of final rules P_{final}.

To bound the rank of the introduced nonterminals by three we handle rules $A \to s$ with $A \in \mathcal{N}_3$ as follows. Let $v_1, v_2, v_3 \in \mathsf{nodes}(s)$ be the parameter nodes of s, where $\lambda_s(v_i) = x_i$ for $i \in [1, 3]$. Instead of choosing the node v by $\mathsf{split}(s)$ we set v to the lowest common ancestor of (v_1, v_2) or (v_2, v_3), depending on which one has the greater distance from the root node (see Figure 4.7). This step ensures that the two trees $\mathsf{subtree}_s(v)$ and $s \setminus v$ have rank 2, so in the next step each of these two trees will be split in a balanced way according to (4.5). As a consequence, the resulting TSLP $\mathcal{G}_t$ has depth $\mathcal{O}(\log |t|)$ but size $\mathcal{O}(|t|)$. Also note that $\mathcal{G}_t$ is in Chomsky normal form.

Example 4.10. *Let t_n be the complete binary tree of height n, i.e. $t_0 = a$ and $t_{n+1} = f(t_n, t_n)$ where $f \in \mathcal{F}_2$ and $a \in \mathcal{F}_0$. Figure 4.8 illustrates how the tree t_3 is decomposed hierarchically during the algorithm. We only explain the first steps of the algorithm. Final rules are framed.*

1. *We start with the rule $S \to t_3$.*

2. *Possible split nodes of t_3 are the children of its root. We split $S \to t_3$ into $\boxed{S \to A(B)}$, $A \to f(x_1, t_2)$ and $\boxed{B \to t_2}$, where $A \in \mathcal{N}_1$ and $B \in \mathcal{N}_0$.*

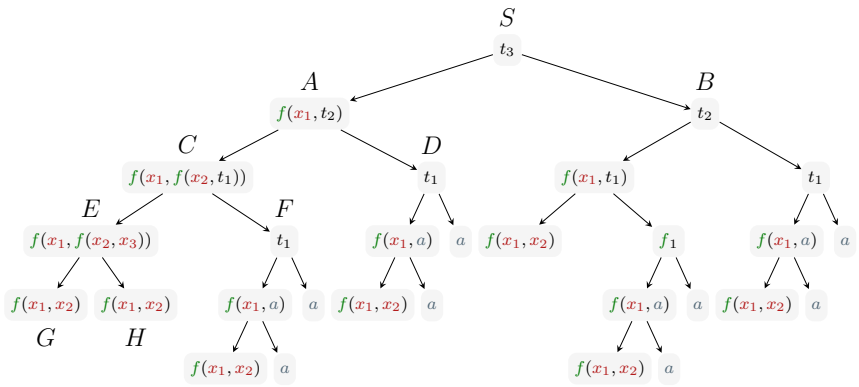

Figure 4.8: The hierarchical decomposition of a tree produced by TreeBiSection.

3. *We continue to split the rule* $A \to f(x_1, t_2)$. *After two more steps we obtain* $\boxed{A \to C(x_1, D)}$, $\boxed{D \to t_1}$, $\boxed{C \to E(x_1, x_2, F)}$, $E \to f(x_1, f(x_2, x_3))$ *and* $\boxed{F \to t_1}$, *where* $C \in \mathcal{N}_2$, $D, F \in \mathcal{N}_0$ *and* $E \in \mathcal{N}_3$.

4. *Since the nonterminal* E *has rank 3, we have to decompose* $f(x_1, f(x_2, x_3))$ *along the lowest common ancestor of two parameters as described above, in this case of* x_2 *and* x_3.[5] *Hence, the rule for the nonterminal* E *is split into the rules* $\boxed{E \to G(x_1, H(x_2, x_3))}$, $\boxed{G \to f(x_1, x_2)}$ *and* $\boxed{H \to f(x_1, x_2)}$, *where* $G, H \in \mathcal{N}_2$.

In the next step we want to compact the TSLP by considering the DAG of the derivation tree. For this we first build the derivation tree $\mathcal{D}_t := \mathcal{D}_{\mathcal{G}_t}$ from the TSLP $\mathcal{G}_t$ as described in Section 4.2.

We now want to identify nonterminals that produce the same pattern. Note that if we just omit the nonterminal labels from the derivation tree, then there might exist isomorphic subtrees of the derivation tree whose root nonterminals produce different patterns. This is due to the fact that we lost for an A-labeled node of the derivation tree with a left child labeled by a nonterminal B and a right child labeled by a nonterminal C the information at which argument position of B the nonterminal C is substituted. This information is exactly given by $\mathsf{index}(A) \in \{0, 1, 2, 3\}$, which was defined in Section 4.2. Moreover, we remove every leaf v and write its label into its parent node. We call the resulting tree the *modified derivation tree* and denote it by $\mathcal{D}_t^*$. Note that $\mathcal{D}_t^*$ is a binary tree (each node has two or zero children) with node labels from $\{1, 2, 3\} \cup \mathsf{labels}(t)$. The modified derivation tree for Example 4.10 is shown on the left of Figure 4.9.

As for trees and patterns, we denote by $\mathsf{subtree}_{\mathcal{D}_t^*}(v)$ the subtree of $\mathcal{D}_t^*$ rooted at node v. The following lemma shows that nonterminals which correspond to

[5] Here the lowest common ancestor of x_2 and x_3 happens to coincide with the node $\mathsf{split}(f(x_1, f(x_2, x_3)))$.

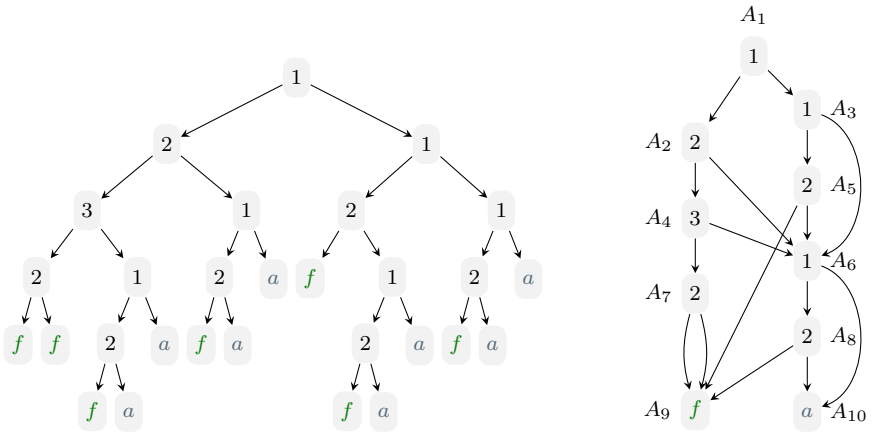

Figure 4.9: Modified derivation tree of the TSLP from Example 4.10 and its minimal DAG.

the same subtree in $\mathcal{D}_t^*$ produce the same pattern, i.e., we can compact the TSLP $\mathcal{G}_t$ by considering the DAG of $\mathcal{D}_t^*$.

Lemma 4.4.2. *Let u and v be nodes of $\mathcal{D}_t$ labeled by A and B, respectively. Moreover, let u' and v' be the corresponding nodes in $\mathcal{D}_t^*$. Then, the subtrees $\mathsf{subtree}_{\mathcal{D}_t^*}(u')$ and $\mathsf{subtree}_{\mathcal{D}_t^*}(v')$ are isomorphic (as labeled ordered trees) if and only if $\mathsf{val}_{\mathcal{G}_t}(A) = \mathsf{val}_{\mathcal{G}_t}(B)$.*

Proof. For the if-direction assume that $\mathsf{val}_{\mathcal{G}_t}(A) = \mathsf{val}_{\mathcal{G}_t}(B)$. Hence, when producing the TSLP $\mathcal{G}_t$, the intermediate productions $A \to \mathsf{val}_{\mathcal{G}_t}(A)$ and $B \to \mathsf{val}_{\mathcal{G}_t}(B)$ are split in exactly the same way (since the splitting process is deterministic). This implies that $\mathsf{subtree}_{\mathcal{D}_t^*}(u')$ and $\mathsf{subtree}_{\mathcal{D}_t^*}(v')$ are isomorphic.

We prove the other direction by induction over the size of the two subtrees $\mathsf{subtree}_{\mathcal{D}_t^*}(u')$ and $\mathsf{subtree}_{\mathcal{D}_t^*}(v')$. Consider u and v labeled by A and B, respectively. We have $\mathsf{index}(A) = \mathsf{index}(B) = i$. For the induction base assume that $i = 0$. Then u' and v' are both leaves labeled by the same terminal. Hence, $\mathsf{val}_{\mathcal{G}_t}(A) = \mathsf{val}_{\mathcal{G}_t}(B)$ holds. For the induction step assume that $i > 0$. Let A_1 be the label of the left child of u and let A_2 be the label of the right child of u. Further, let B_1 be the label of the left child of v and let B_2 be the label of the right child of v. By induction, we get $\mathsf{val}_{\mathcal{G}_t}(A_1) = \mathsf{val}_{\mathcal{G}_t}(B_1) = s$ for some pattern s of rank $m > 1$ (since s corresponds to a left child in D_t) and $\mathsf{val}_{\mathcal{G}_t}(A_2) = \mathsf{val}_{\mathcal{G}_t}(B_2) = s'$ for some pattern s' of rank m'. Therefore, $\mathsf{rank}(A) = \mathsf{rank}(B) = m + m' - 1$ and

$$\mathsf{val}_{\mathcal{G}_t}(A) = s[x_1, ..., x_{i-1}, s'[x_i, ..., x_{i+m'-1}], x_{i+m'}, ..., x_{m+m'-1}] = \mathsf{val}_{\mathcal{G}_t}(B).$$

This proves the lemma. □

By Lemma 4.4.2, if two subtrees of $\mathcal{D}_t^*$ are isomorphic we can eliminate the nonterminal of the root node of one subtree. Hence, we can compact our TSLP

by constructing the minimal DAG of $\mathcal{D}_t^*$. The minimal DAG of the modified derivation tree that corresponds to the TSLP of Example 4.10 is shown on the right of Figure 4.9. The nodes of $\mathsf{DAG}(\mathcal{D}_t^*)$ are the nonterminals of the final TSLP produced by TreeBiSection. For a nonterminal that corresponds to an inner node of $\mathsf{DAG}(\mathcal{D}_t^*)$, we obtain a rule whose right-hand side has the form (4.3) and for a leaf of $\mathsf{DAG}(\mathcal{D}_t^*)$ the right-hand side has the form (4.4). Let n_1 be the number of inner nodes of $\mathsf{DAG}(\mathcal{D}_t^*)$ and n_2 be the number of leaves. Then the size of our final TSLP is $2n_1 + n_2$, which is bounded by twice the number of nodes of $\mathsf{DAG}(\mathcal{D}_t^*)$.

Example 4.11. *We continue Example 4.10 and obtain the final TSLP from the minimal DAG of $\mathcal{D}_t^*$ shown in Figure 4.9 on the right. We assign to each node of $\mathsf{DAG}(\mathcal{D}_t^*)$ a fresh nonterminal and define the rules according to the labels as follows. Here the start nonterminal is A_1.*

$$\begin{aligned}
A_1 &\to A_2(A_3) & A_2 &\to A_4(x_1, A_6)\\
A_3 &\to A_5(A_6) & A_4 &\to A_7(x_1, x_2, A_6)\\
A_5 &\to A_9(x_1, A_6) & A_6 &\to A_8(A_{10})\\
A_7 &\to A_9(x_1, A_9(x_2, x_3)) & A_8 &\to A_9(x_1, A_{10})\\
A_9 &\to f(x_1, x_2) & A_{10} &\to a
\end{aligned}$$

We have $A_1, A_3, A_6, A_{10} \in \mathcal{N}_0$, $A_2, A_5, A_8 \in \mathcal{N}_1$, $A_4, A_9 \in \mathcal{N}_2$ and $A_7 \in \mathcal{N}_3$.

Figure 4.10 shows the pseudocode of TreeBiSection. There, we denote for a valid pattern s of rank k by $\mathrm{lca}_s(x_i, x_j)$ the lowest common ancestor of the unique leaves that are labeled by x_i and x_j $(i, j \leq k)$.

In Section 4.4.1 we will analyze the running time of TreeBiSection, and we will present a logspace implementation. In Section 4.4.2 we will analyze the size of the produced TSLP using Theorem 4.3.2.

4.4.1 Running time and space consumption

In this section, we show that TreeBiSection can be implemented so that it works in logspace, and alternatively in time $\mathcal{O}(n \cdot \log n)$. Note that these are two different implementations.

Lemma 4.4.3. *Given a tree $t \in \mathcal{T}(\mathcal{F}_{\leq 2})$ of size n one can compute (i) in time $\mathcal{O}(n \log n)$ and (using a different algorithm) (ii) in logspace the TSLP produced by* TreeBiSection *on input t.*

Proof. Let t be the input tree of size n. The DAG of a tree can be computed in (i) linear time [38] and (ii) in logspace by Lemma 4.3.1. Hence, it suffices to show that the modified derivation tree $\mathcal{D}_t^*$ for t can be computed in time $\mathcal{O}(n \cdot \log n)$ as well as in logspace.

For the running time let us denote with $P_{\mathrm{temp},i}$ the set of productions P_{temp} after i iterations of the while loop. Moreover, let n_i be the sum of the sizes of

input : $t \in \mathcal{T}(\mathcal{F}_{\leq 2})$
$\mathcal{N} := \emptyset$
$P_{\text{temp}} := \{S \to t\}$
$P_{\text{final}} := \emptyset$
while $P_{\text{temp}} \neq \emptyset$ **do**
  foreach $(A \to s) \in P_{\text{temp}}$ **do**
    $P_{\text{temp}} := P_{\text{temp}} \setminus \{A \to s\}$
    if $\mathsf{rank}(s) = 3$ **then**
      $v :=$ the lower of nodes $\text{lca}_s(x_1, x_2), \text{lca}_s(x_2, x_3)$
    else
      $v := \mathsf{split}(s)$
    end
    $t_1 := \mathsf{subtree}_s(v)$; $t_2 := s \setminus v$
    $r_1 := \mathsf{rank}(t_1)$; $r_2 := \mathsf{rank}(t_2)$
    Let A_1 and A_2 be fresh nonterminals.
    $\mathcal{N} := \mathcal{N} \cup \{A_1, A_2\}$
    foreach $i = 1$ **to** 2 **do**
      if $|t_i| > 1$ **then**
        $P_{\text{temp}} := P_{\text{temp}} \cup \{A_i \to t_i\}$
      else
        $P_{\text{final}} := P_{\text{final}} \cup \{A_i \to t_i\}$
      end
    end
    $r := r_1 + r_2 - 1$
    Let k be the number of nodes in s labeled by parameters that are smaller than v w.r.t. $<_s$.
    $p := A \to A_1(x_1, \ldots, x_k, A_2(x_{k+1}, \ldots, x_{k+r_2}), x_{k+r_2+1}, \ldots, x_r)$
    $P_{\text{final}} := P_{\text{final}} \cup \{p\}$
  end
end
Let $\mathcal{G}$ be the TSLP $(\mathcal{N}, \mathcal{F}_{\leq 2}, S, P_{\text{final}})$.
Construct the modified derivation tree $\mathcal{D}_t^*$ of $\mathcal{G}$.
Compute $\mathsf{DAG}(\mathcal{D}_t^*)$ and let $\mathcal{H}$ be the corresponding TSLP.
return TSLP $\mathcal{H}$

Figure 4.10: TreeBiSection

all right-hand sides in $P_{\text{temp},i}$. Then, we have $n_{i+1} \leq n_i$: When a single rule $A \to s$ is replaced with $A_1 \to t_1$ and $A_2 \to t_2$ then each non-parameter node in t_1 or t_2 is one of the nodes of s. Hence, we have $|s| = |t_1| + |t_2|$ (recall that we do not count parameters for the size of a tree). We might have $n_{i+1} < n_i$ since rules with a single terminal symbol on the right-hand side are put into P_{final}. We obtain $n_i \leq n$ for all i. Hence, splitting all rules in $P_{\text{temp},i}$ takes time $\mathcal{O}(n)$, and a single iteration of the while loop takes time $\mathcal{O}(n)$ as well. On the other hand, since every second split reduces the size of the tree to which the split is applied by a constant factor (see (4.5)). Hence, the while loop is iterated at most $\mathcal{O}(\log n)$ times. This gives the time bound.

The inquisitive reader may wonder whether our convention of neglecting parameter nodes for the size of a tree affects the linear running time. This is not the case: Every right-hand side s in $P_{\text{temp},i}$ has at most three parameters, i.e., the total number of nodes in s is at most $|s| + 3 \leq 4|s|$. This implies that the split node can be computed in time $\mathcal{O}(|s|)$. Doing this for all right-hand sides in $P_{\text{temp},i}$ yields the time bound $\mathcal{O}(n)$ as above.

For the logspace version, we first describe how to represent a single valid pattern occurring in t in logspace and how to compute its split node. Let s be a valid pattern which occurs in t and has k parameters ($\mathsf{rank}(s) = k$) where $k \in [0, 3]$, i.e., $s[t_1, \ldots, t_k]$ is a subtree of t for some subtrees $t_1, \ldots, t_k$ of t. We represent the pattern s by the tuple $\mathsf{rep}(s) = (v_0, v_1 \ldots, v_k)$ where $v_0, v_1, \ldots, v_k$ are the nodes in t corresponding to the roots of $s, t_1, \ldots, t_k$, respectively. Note that $\mathsf{rep}(s)$ can be stored using $\mathcal{O}((k+1) \cdot \log(n))$ many bits. Given such a tuple $\mathsf{rep}(s) = (v_0, \ldots, v_k)$, we can compute in logspace the size $|s|$ by a depth-first left-to-right traversal of t, starting from v_0 and skipping subtrees rooted in the nodes $v_1, \ldots, v_k$. We can also compute in logspace the split node v of s: If s has at most two parameters, then $v = \mathsf{split}(s)$. Note that the procedure from Lemma 4.4.1 can be implemented in logspace since the size of a subtree of s can be computed as described before. If s has three parameters, then v is the lowest common ancestor of either v_1 and v_2, or of v_2 and v_3, depending on which node has the larger distance from v_0. The lowest common ancestor of two nodes can also be computed in logspace by traversing the paths from the two nodes to the root upwards. From $\mathsf{rep}(s) = (v_0, \ldots, v_k)$ and a split node v we can easily determine $\mathsf{rep}(\mathsf{subtree}_s(v))$ and $\mathsf{rep}(s \setminus v)$ in logspace.

Using the previous remarks we are ready to present the logspace algorithm in order to compute $\mathcal{D}_t^*$. Since $\mathcal{D}_t^*$ is a binary tree of depth $\mathcal{O}(\log n)$ we can identify a node of $\mathcal{D}_t^*$ with the string $u \in \{0,1\}^*$ of length at most $c \cdot \lfloor \log n \rfloor$ that stores the path from the root to the node, where $c > 0$ is a suitable constant. We denote by s_u the tree (with at most three parameters) described by a node u of $\mathcal{D}_t^*$ in the sense of Lemma 4.4.2. That is, if u' is the corresponding node of the derivation tree $\mathcal{D}_t$ and u' is labeled by the nonterminal A, then $s_u = \mathsf{val}_{\mathcal{G}_t}(A)$.

To compute $\mathcal{D}_t^*$, it suffices for each string $w \in \{0,1\}^*$ of length $c \cdot \lfloor \log n \rfloor$ to check in logspace whether it is a node of $\mathcal{D}_t^*$ and in case it is a node, to determine the label of w in $\mathcal{D}_t^*$. For this, we compute for each prefix u of w, starting with the empty word, the tuple $\mathsf{rep}(s_u)$ and the label of u in $\mathcal{D}_t^*$, or a bit indicating that u is not a node of $\mathcal{D}_t^*$ (in which case also w is not a node of $\mathcal{D}_t^*$). Thereby

we only store the current bit strings w, u and the value of $\mathsf{rep}(s_u)$, which fit into logspace. If $u = \varepsilon$, then $\mathsf{rep}(s_u)$ consists only of the root of t. Otherwise, we first compute in logspace the size $|s_u|$ from $\mathsf{rep}(s_u)$. If $|s_u| = 1$, then u is a leaf in $\mathcal{D}_t^*$ with label $\lambda(u)$ and no longer prefixes represent nodes in $\mathcal{D}_t^*$. If $|s_u| > 1$, then u is an inner node in $\mathcal{D}_t^*$ and, as described above, we can compute in logspace from $\mathsf{rep}(s_u)$ the tuples $\mathsf{rep}(s_{u0})$ and $\mathsf{rep}(s_{u1})$, from which we can easily read off the label of u from $\{1, 2, 3\}$. If $u = w$, then we stop, otherwise we continue with ui and $\mathsf{rep}(s_{ui})$, where $i \in \{0, 1\}$ is such that ui is a prefix of w. □

4.4.2 Size of the TSLP produced by TreeBiSection

In order to bound the size of the TSLP produced by TreeBiSection we have to bound the number of nodes in the DAG of the modified derivation tree. Let us fix the TSLP $\mathcal{G}_t$ for a tree $t \in \mathcal{T}(\mathcal{F}_{\leq 2})$ that has been produced by the first part of TreeBiSection. Let $n = |t|$ and $|\mathsf{labels}(t)| = \sigma$ be the number of the different node labels that appear in t. For the modified derivation tree $\mathcal{D}_t^*$ we have the following:

- $\mathcal{D}_t^*$ is a binary tree (every node has zero or two children) with n leaves and hence $2n - 1$ nodes.
- There are $\sigma + 3$ possible node labels, namely $\mathsf{labels}(t) \cup \{1, 2, 3\}$.
- $\mathcal{D}_t^*$ is $(1/7)$-balanced (see Section 4.3.1 for the definition of β-balanced binary trees). If we have two successive nodes in $\mathcal{D}_t^*$, then we split at one of the two nodes according to (4.5). Now, assume that we split at node v according to (4.5). Let v_1 and v_2 be the children of v, let n_i be the leaf size of v_i for $i \in [1, 2]$, and let $n = n_1 + n_2$ be the leaf size of v. We get $\frac{1}{8}n \leq n_1 \leq \frac{3}{4}n$ and $\frac{1}{4}n \leq n_2 \leq \frac{7}{8}n$ (or vice versa). Hence, $n_1 \geq \frac{1}{8}n \geq \frac{1}{7}n_2$ and $n_2 \geq \frac{1}{4}n \geq \frac{1}{3}n_1$.

Recall that the nodes of $\mathsf{DAG}(\mathcal{D}_t^*)$ are the nonterminals of the TSLP produced by TreeBiSection and that this TSLP is in Chomsky normal form. Moreover, recall that the depth of $\mathcal{D}_t^*$ is in $\mathcal{O}(\log n)$. Hence, with Theorem 4.3.2 and Lemma 4.4.3 we get:

Corollary 4.4.4. *Let $t \in \mathcal{T}(\mathcal{F}_{\leq 2})$ be a tree of size n with $|\mathsf{labels}(t)| = \sigma$. Then* TreeBiSection *produces a TSLP in Chomsky normal form of size $\mathcal{O}\big(\frac{n}{\log_\sigma n}\big)$ and depth $\mathcal{O}(\log n)$. Every nonterminal of the produced TSLP has rank at most 3, and the algorithm can be implemented in logspace and, alternatively, in time $\mathcal{O}(n \cdot \log n)$.*

In particular, if the size of the ranked alphabet is a fixed constant (e.g. in the case of unlabeled trees) we obtain TSLPs of size $\mathcal{O}(n/\log n)$ for input trees of size n.

4.4.3 Extension to trees of larger degree

If the input tree t has nodes with many children, then we cannot expect good compression by TSLPs. The extreme case is $t_n = f_n(a, \ldots, a)$ where f_n is a symbol of rank n. Hence, $|t_n| = n + 1$ and every TSLP for t_n has size at least $n + 1$. On the other hand, for trees in which the maximal rank is bounded by a constant $r \geq 1$, we can easily generalize TreeBiSection. Lemma 4.4.1 allows to find a splitting node v satisfying

$$\frac{1}{2(r+2)} \cdot |t| \;\leq\; |\mathsf{subtree}_t(v)| \;\leq\; \frac{r+1}{r+2} \cdot |t|. \tag{4.6}$$

The maximal arity of nodes also affects the arity of patterns: we allow patterns of rank up to r. Now assume that t is a valid pattern of rank $r+1$, where r is again the maximal number of children of a node. Then we find a subtree containing k parameters, where $2 \leq k \leq r$: Take a smallest subtree that contains at least two parameters. Since the root node of that subtree has at most r children, and every proper subtree contains at most one parameter (due to the minimality of the subtree), this subtree contains at most r parameters. By taking the root of that subtree as the splitting node, we obtain two valid patterns with at most r parameters each. Hence, we have to change TreeBiSection in the following way:

- As long as the number of parameters of the tree is at most r, we choose the splitting node according to Lemma 4.4.1.
- If the number of parameters is $r + 1$ (note that in each splitting step, the number of parameters increases by at most 1), then we choose the splitting node such that the two resulting fragments have rank at most r.

As before, this guarantees that in every second splitting step we split in a balanced way. But the balance factor β from Section 4.3.1 now depends on r. More precisely, if in the modified derivation tree $\mathcal{D}_t^*$ we have a node v with children v_1 and v_2 of leaf size n_1 and n_2, respectively, and this node corresponds to a splitting satisfying (4.6), then we get

$$\begin{aligned}
\frac{1}{2(r+2)} \cdot n \;&\leq\; n_1 \;\leq\; \frac{r+1}{r+2} \cdot n, \\
\frac{1}{r+2} \cdot n \;=\; \left(1 - \frac{r+1}{r+2}\right) \cdot n \;&\leq\; n_2 \;\leq\; \left(1 - \frac{1}{2(r+2)}\right) \cdot n \;=\; \frac{2r+3}{2(r+2)} \cdot n
\end{aligned}$$

or vice versa. This implies

$$\begin{aligned}
n_1 \;&\geq\; \frac{1}{2(r+2)} \cdot n \;\geq\; \frac{1}{2r+3} \cdot n_2, \\
n_2 \;&\geq\; \frac{1}{r+2} \cdot n \;\geq\; \frac{1}{r+1} \cdot n_1 \;\geq\; \frac{1}{2r+3} \cdot n_1.
\end{aligned}$$

Hence, the modified derivation tree becomes β-balanced for $\beta = 1/(2r+3)$. Moreover, the possible number of different labels in the modified derivation

tree now is at most $\sigma + r + 1$ (since the TSLP produced in the first step has nonterminals of rank at most $r + 1$). Theorem 4.3.2 yields the following bound on size of the DAG of the modified derivation tree and hence the size of the final TSLP:

$$\mathcal{O}\left(\frac{n}{\log_{\sigma+r}(n)} \cdot \log_{1+\frac{1}{2r+3}}(2r+3)\right) = \mathcal{O}\left(\frac{n}{\log_{\sigma+r}(n)} \cdot \frac{\log(2r+3)}{\log(1+\frac{1}{2r+3})}\right)$$

Note that $\log(1+x) \geq x$ for $0 \leq x \leq 1$. Hence, we can simplify the bound to

$$\mathcal{O}\left(\frac{n \cdot \log(\sigma+r) \cdot r \cdot \log r}{\log n}\right).$$

By Lemma 4.4.1, the depth of the produced TSLP can be bounded by $2 \cdot d$, where d is any number that satisfies

$$n \cdot \left(\frac{r+1}{r+2}\right)^d \leq 1.$$

Hence, we can bound the depth by

$$2 \cdot \left\lceil \frac{\log n}{\log(1+\frac{1}{r+1})} \right\rceil \leq 2 \cdot \lceil (r+1) \cdot \log n \rceil \leq \mathcal{O}(r \cdot \log n).$$

Theorem 4.4.5. *Let $t \in \mathcal{T}(\mathcal{F})$ be a tree of size n with $|\mathsf{labels}(t)| = \sigma$ such that each symbol in $\mathsf{labels}(t)$ has rank at most r. Then* TreeBiSection *produces a TSLP in Chomsky normal form of size $\mathcal{O}\big(\frac{n \cdot \log(\sigma+r) \cdot r \cdot \log r}{\log n}\big)$ and depth $\mathcal{O}(r \cdot \log n)$. Every nonterminal of that TSLP has rank at most $r + 1$.*

For the running time we obtain the following bound:

Theorem 4.4.6. TreeBiSection *can be implemented such that it works in time $\mathcal{O}(r \cdot n \cdot \log n)$ for a tree of size n in which each symbol has rank at most r.*

Proof. TreeBiSection makes $\mathcal{O}(r \cdot \log n)$ iterations of the while loop (this is the same bound as for the depth of the TSLP) and each iteration takes time $\mathcal{O}(n)$. To see the latter, our internal representation of valid patterns from Section 4.1 is important. Using this representation, we can still compute the split node in a right-hand side s from P_{temp} in time $\mathcal{O}(|s|)$: We first compute for every non-parameter node v of s (i) the size of the subtree rooted at v (as usual, excluding parameters) and (ii) the number of parameters below v. This is possible in time $\mathcal{O}(|s|)$ using a straightforward bottom-up computation. Using this information, we can compute the split node in s in time $\mathcal{O}(|s|)$ for both cases (the number of parameters in s is $r + 1$ or smaller than $r + 1$) by searching from the root downwards. □

In particular, if r is bounded by a constant, TreeBiSection computes a TSLP of size $\mathcal{O}(n/\log_\sigma n)$ and depth $\mathcal{O}(\log n)$ in time $\mathcal{O}(n \cdot \log n)$. Moreover, our logspace implementation of TreeBiSection (see Lemma 4.4.3) directly generalizes to the case of a constant rank.

Unranked trees. For *unranked trees* in which the number of children of a node is arbitrary and not determined by the node label (which is the standard tree model in XML) all this fails: TreeBiSection only yields TSLPs of size $\Theta(n)$ and this is unavoidable as shown by the example $f_n(a, \dots, a)$ from the beginning of this subsection. Moreover, the logspace implementation from Section 4.4.1 no longer works since nonterminals have rank at most $r + 1$ and we cannot store the pattern derived from a nonterminal in space $\mathcal{O}(\log n)$ anymore (we have to store $r + 1$ many nodes in the tree).

Fortunately, there is a simple workaround for all these problems: An unranked tree can be transformed into a binary tree of the same size using the well known first-child next-sibling encoding [19, 76]. Then, one can simply apply TreeBiSection to this encoding to get in logspace and time $\mathcal{O}(n \cdot \log n)$ a TSLP of size $\mathcal{O}(n/\log_\sigma n)$.

For the problem of traversing a compressed unranked tree t (which is addressed in [15] for *top dags*) another (equally well known) encoding is more favorable. Let $c(t)$ be a compressed representation (e.g., a TSLP or a top dag) of t. The goal is to represent t in space $\mathcal{O}(|c(t)|)$ such that one can efficiently navigate from a node to (i) its parent node, (ii) its first child, (iii) its next sibling, and (iv) its previous sibling (if they exist). For top dags [15], it was shown that a single navigation step can be done in time $\mathcal{O}(\log |t|)$. Using a suitable binary encoding, we can prove the same result for TSLPs: Let r be the maximal rank of a node of the unranked tree t. We define the binary encoding $\mathrm{bin}(t)$ by adding for every node v of rank $s \leq r$ a binary tree of depth $\lceil \log s \rceil$ with s many leaves, whose root is v and whose leaves are the children of v. This introduces at most $2s$ many new binary nodes, which are labeled by a new symbol. We get $|\mathrm{bin}(t)| \leq 3|t|$. In particular, we obtain a TSLP of size $\mathcal{O}(n/\log_\sigma n)$ for $\mathrm{bin}(t)$, where $n = |t|$ and σ is the number of different node labels. Note that a traversal step in the initial tree t (going to the parent node, first child, next sibling, or previous sibling) can be simulated by $\mathcal{O}(\log r)$ many traversal steps in $\mathrm{bin}(t)$ (going to the parent node, left child, or right child). But for a binary tree s, it was recently shown that a TSLP $\mathcal{G}$ for s can be represented in space $\mathcal{O}(|\mathcal{G}|)$ such that a single traversal step takes time $\mathcal{O}(1)$ [83].[6] Hence, we can navigate in t in time $\mathcal{O}(\log r) \leq \mathcal{O}(\log |t|)$.

Even simpler is the following approach for unranked trees: In [50] the authors introduced so-called *forest straight-line programs* (FSLPs) which can be seen as a proper generalization of TSLPs in order to compress unranked trees. It is shown there that FSLPs for unranked trees and TSLPs for the first-child-next-sibling encoding of unranked trees are equally succinct up to constant multiplicative factors and that one can change between both representations in linear time. A direct consequence is that based on our construction one obtains an FSLP of size $\mathcal{O}(n/\log_\sigma n)$ for an unranked tree t of size n. It seems likely that the traversing techniques presented in [84] for TSLPs can be extended to FSLPs.

[6] This generalizes a corresponding result for strings [52].

4.5 BU-Shrink

We showed that TreeBiSection can be implemented such that it works in time $\mathcal{O}(n \cdot \log n)$ for a tree of size n. In this section we present a linear time algorithm BU-Shrink (for bottom-up shrink) that also constructs a TSLP of size $\mathcal{O}(n/\log_\sigma n)$ for a given tree of size n with σ many node labels of constant rank. The basic idea of BU-Shrink is to merge in a bottom-up way nodes of the tree to patterns of size roughly k, where k is defined later. This is a bottom-up computation in the sense that we begin with individual nodes and gradually merge them into larger fragments (the term "bottom-up" should not be understood in the sense that the computation is done from the leaves of the tree towards the root). The DAG of the small trees represented by the patterns then yields the compression.

For a valid pattern p of rank d we define the weight of p as $|p| + d$. This is the total number of nodes in p including those nodes that are labeled by a parameter (which are not counted in the size $|p|$ of p). A *pattern tree* is a tree in which the labels of the tree are valid patterns. If a node v is labeled by the valid pattern p and $\mathsf{rank}(p) = d$, then we require that v has d children in the pattern tree. For convenience, BU-Shrink also stores in every node the weight of the corresponding pattern. For a node v, we denote by p_v its pattern and by $w(v)$ the weight of p_v.

Let us fix a number $k \geq 1$ that will be specified later. Given a tree t of size n such that all node labels in t are of rank at most r, BU-Shrink first creates a pattern tree t' by replacing every label $f \in \mathcal{F}_d$ by the valid pattern $f(x_1, \ldots, x_d)$ of rank d and weight $d+1$. Note that the parameters in these patterns correspond to the edges of the tree t. We will keep this invariant during the algorithm, which will shrink the pattern tree t'. Hence, the total number of all parameter occurrences in the patterns that appear as labels in the current pattern tree t' will be always the number of nodes of the current tree t' minus 1. This allows us to ignore the cost of handling parameters for the running time of the algorithm.

After generating the initial pattern tree t', BU-Shrink creates a queue Q that contains references to all nodes of t' having at most one child (excluding the root node) in an arbitrary order. During the run of the algorithm, the queue Q will only contain references to non-root nodes of the current tree t' that have at most one child (but Q may not contain references to all such nodes). For each node v of the queue we proceed as follows. Let v be the i-th child of its parent node u. If $w(v) > k$ or $w(u) > k$, we simply remove v from Q and proceed. Otherwise we merge the node v into the node u. More precisely, we delete the node v, and set the i-th child of u to the unique child of v if it exists (otherwise, u loses its i-th child). The pattern p_u is modified by replacing the parameter at the position of the i-th child by the pattern p_v and re-enumerating all parameters to get a valid pattern. We also set the weight $w(u)$ to $w(v) + w(u) - 1$ (which is the weight of the new pattern p_u). Note that in this way both the number of edges of t' and the total number of parameter occurrences in all patterns decreases by 1. For example, let u be a node with $p_u = f(x_1, x_2)$ and let v be its second child with $p_v = g(x_1)$. Then the merged pattern becomes $f(x_1, g(x_2))$, and its weight is 4. If the node u has at most one child after the merging and its weight is at most

k, then we add u to Q (if it is not already in the queue). We do this until the queue is empty. Note that every pattern appearing in the final pattern tree has rank at most r (the maximal rank of a symbol in the initial tree).

Now consider the forest (a disjoint union of trees or patterns) which consists of all patterns appearing in the resulting final pattern tree. We construct the DAG of this forest, which yields rules for all patterns with shared nonterminals. The DAG of a forest is constructed in the same way as for a single tree. This DAG has for every subtree appearing in the forest exactly one node. The parameters $x_1, x_2, \ldots, x_r$ that appear in the patterns are treated as ordinary leaf labels when constructing the DAG. As usual, the DAG can be viewed as a TSLP, where the nodes of the DAG correspond to the nonterminals. Here, we only introduce nonterminals for nodes which correspond to patterns of size at least two while patterns of size at most one are inserted directly into the rules where they occur. Note that in this way we omit rules of the form $A \to x_1$ and $A \to f(x_1, \ldots, x_d)$ for $f \in \mathcal{F}_d$ which only increase the size of the TSLP. We obtain a TSLP in which each pattern is derived by a nonterminal of the same rank as the pattern. Finally, we add to the TSLP the start rule $S \to s$, where s is obtained from the pattern tree by labelling each node v with the unique nonterminal A such that A derives the pattern p_v. Figure 4.11 shows the pseudocode for BU-Shrink and an example can be found in Example 4.12 and Figure 4.12.

Example 4.12. *Let $a \in \mathcal{F}_0$, $g \in \mathcal{F}_1$ and $f \in \mathcal{F}_2$. Consider the input tree $t = f(g(f(g(a), g(a))), f(g(a), f(g(a), g(a))))$ and the corresponding pattern tree depicted in Figure 4.12 (bottom right). Assuming no further mergings are done, the corresponding TSLP has rules*

$$S \to A(B(C), B(B(C))), \qquad A \to f(g(x_1), x_2)$$
$$B \to f(C, x_1), \qquad C \to g(a).$$

It is easy to see that BU-Shrink runs in time $\mathcal{O}(n)$ for a tree of size n. First of all, the number of mergings is bounded by n, since each merging reduces the number of nodes of the pattern tree by one. Moreover, if a node is removed from Q (because its weight or the weight of its parent node is larger than k) then it will never be added to Q again (since weights are never reduced). A single merging step needs only a constant number of pointer operations and a single addition (for the weights). For this, it is important that we do not copy patterns, when the new pattern (for the node u in the above description) is constructed. The forest, for which we construct the DAG, has size $\mathcal{O}(n)$: The number of non-parameter nodes is exactly n, and the number of parameters is at most $n - 1$: Initially the forest has $n - 1$ parameters (as there is a parameter for each node except the root) and during BU-Shrink we can only decrease the total amount of parameters.

Let us now analyze the size of the constructed TSLP. In the following, let t be the input tree of size n and let r be the maximal rank of a label in t. Further, let $|\mathsf{labels}(t)| = \sigma$.

Lemma 4.5.1. *Let t_p be the pattern tree resulting from* BU-Shrink. *Then* $|t_p| \leq \frac{4 \cdot r \cdot n}{k} + 2$.

input : tree $t \in \mathcal{T}(\mathcal{F})$, number $k \leq |t|$
$Q := \emptyset$
foreach $v \in \mathsf{nodes}(t)$ **do**
  let $f = \lambda_t(v) \in \mathcal{F}_d$ be the label of node v
  $w(v) := 1 + d$ `(the weight of node v)`
  $p_v := f(x_1, \ldots, x_d)$ `(the pattern stored in node v)`
  if $d \leq 1$ *and* v *is not the root* **then**
    $Q := Q \cup \{v\}$
  end
end
while $Q \neq \emptyset$ **do**
  choose arbitrary node $v \in Q$ and set $Q := Q \setminus \{v\}$
  let u be the parent node of v
  if $w(v) \leq k$ **and** $w(u) \leq k$ **then**
    $d := \mathsf{rank}(p_v)$; $e := \mathsf{rank}(p_u)$
    let v be the i-th child of u
    $w(u) := w(u) + w(v) - 1$
    $p_u := p_u(x_1, \ldots, x_{i-1}, p_v(x_i, \ldots, x_{i+d-1}), x_{i+d}, \ldots, x_{d+e-1})$
    if v *has a (necessarily unique) child* v' **then**
      set v' to the i-th child of u
    end
    delete node v
    if $d + e - 1 \leq 1$ *and* $w(u) \leq k$ **then**
      $Q := Q \cup \{u\}$
    end
  end
end
compute the minimal DAG $\mathcal{D}$ for the forest consisting of all patterns p_v
$P := \emptyset$
$\mathcal{N} := \{S\}$
foreach *node* v *of the* t **do**
  create a fresh nonterminal A_v of rank $d := \mathsf{rank}(p_v)$
  $\mathcal{N} := \mathcal{N} \cup A_v$
  create a rule in P for A_v with $\mathsf{val}(A_v) = p_v$ according to the DAG $\mathcal{D}$
  $\lambda_t(v) := A_v$ `(the new label of node v)`
end
return TSLP $(\mathcal{N}, \mathcal{F}, S, P \cup \{S \to t\})$

Figure 4.11: BU-Shrink

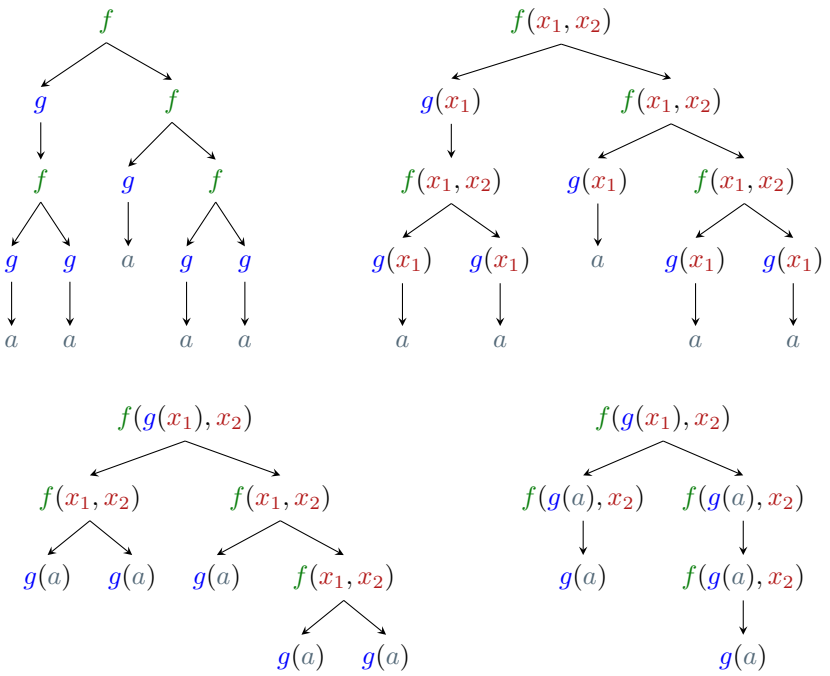

Figure 4.12: BU-Shrink first transforms the input tree on the top left into the pattern tree on the top right (the weights are omitted to improve readability). Then it starts to shrink this pattern tree. The two trees at the bottom depict possible intermediate trees during the shrinking process.

Proof. Let us first assume that $r \geq 2$. The number of non-root nodes in t_p of arity at most one is at least $|t_p|/2 - 1$. For each of those nodes, either the node itself or the parent node has weight at least k. We now map in t_p each non-root node of arity at most one to a node of weight at least k: Let u be a node of t_p (which is not the root) having arity at most one. If the weight of u is at least k, we map u to itself, otherwise we map u to its parent node, which then must be of weight at least k. Note that at most r nodes are mapped to a fixed node v: If v has arity at most one, then v and its child (if it exists) can be mapped to v; if v has arity greater than one then only its children can be mapped to v, and v has at most r children. Therefore there must exist at least $(|t_p| - 2)/(2r)$ many nodes of weight at least k. Because the sum of all weights in t_p is at most $2n$. This yields

$$\frac{|t_p| - 2}{2r} \cdot k \leq 2n,$$

which proves the lemma for the case $r \geq 2$. The case $r = 1$ can be proved in the same way: Clearly, the number of non-root nodes of arity at most one is $|t_p| - 1$ and at most 2 nodes are mapped to a fixed node of weight at least k (by the above argument). Hence, there exist at least $(|t_p| - 1)/2 = (|t_p| - 1)(2r)$ many

nodes of weight at least k. $\square$

Note that each node in the final pattern tree has weight at most $2k$ since BU-Shrink only merges nodes of weight at most k. By Lemma 4.1.1 the number of different patterns in $\mathcal{T}(\mathsf{labels}(t) \cup \{x_1, \ldots, x_r\})$ of weight at most $2k$ is bounded by $\frac{4}{3}(4(\sigma+r))^{2k} \leq d^k$ for $d = (6(\sigma+r))^2$. Hence, the size of the DAG constructed from the patterns is bounded by d^k. Adding the size of the start rule, i.e., the size of the resulting pattern tree (Lemma 4.5.1) we get the following bound for the constructed TSLP:

$$d^k + \frac{4 \cdot r \cdot n}{k} + 2.$$

Let us now set $k = \frac{1}{2} \log_d n$. We get the following bound for the constructed TSLP:

$$\begin{aligned} d^{\frac{1}{2} \cdot \log_d n} + \frac{8 \cdot n \cdot r}{\log_d n} + 2 = \sqrt{n} + \mathcal{O}\left(\frac{n \cdot r}{\log_d n}\right) = \mathcal{O}\left(\frac{n \cdot r}{\log_{\sigma+r} n}\right) \\ = \mathcal{O}\left(\frac{n \cdot \log(\sigma + r) \cdot r}{\log n}\right). \end{aligned}$$

Theorem 4.5.2. *Let $t \in \mathcal{T}(\mathcal{F})$ such that $|\mathsf{labels}(t)| = \sigma$ and the maximal rank of symbol in $\mathsf{labels}(t)$ is r. Then* BU-Shrink *computes in time $\mathcal{O}(n)$ a TSLP for t of size $\mathcal{O}\big(\frac{n \cdot \log(\sigma+r) \cdot r}{\log n}\big)$. Every nonterminal of that TSLP has rank at most r.*

Clearly, if r is bounded by a constant, we obtain the bound $\mathcal{O}(n/\log_\sigma n)$. On the other hand, already for $r \geq \Omega(\log n)$ the bound above exceeds $\mathcal{O}(n)$. But note that the size of the TSLP produced by BU-Shrink is at most n.

Combining TreeBiSection and BU-Shrink. In the remaining section, we aim to construct in linear time a TSLP of size $\mathcal{O}(n/\log_\sigma n)$ and depth $\mathcal{O}(\log n)$ for a given tree t of size n with $|\mathsf{labels}(t)| = \sigma$ (we assume here again that the maximal rank of a symbol in $\mathsf{labels}(t)$ is bounded by a constant). Fist of all, Ganardi, Jeż and Lohrey recently introduced a technique which allows to balance a given TSLP in linear time such that the size of the TSLP increases only by a multiplicative constant factor [48], i.e., we can apply the balancing algorithm presented in [48] to the TSLP produced by BU-Shrink in order to achieve the targeted properties. On the other side, it is a nice bonus of the two grammar-based tree compressors TreeBiSection and BU-Shrink that we achieve the same goal by a combination of those algorithms as we show in the following.

Recall that TreeBiSection produces a TSLP in Chomsky normal form of logarithmic depth, which will be important in the next section. Clearly, a TSLP produced by BU-Shrink is not in Chomsky normal form. To get a TSLP in Chomsky normal form we have to further partition the right-hand sides of the TSLP. As mentioned above, we assume in the following that the maximal rank of symbols appearing in the input tree is bounded by a constant. Hence, BU-Shrink produces for an input tree t of size n with σ many node labels a TSLP of size $\mathcal{O}(n/\log_\sigma n)$. The weight and hence also the depth of the patterns that appear in

the pattern tree t_p produced by BU-Shrink is $\mathcal{O}(\log_d n) = \mathcal{O}(\log_\sigma n) \leq \mathcal{O}(\log n)$. The productions that arise from the DAG of the forest of all patterns have the form $A \to f(A_1, \ldots, A_r)$ and $A \to x_i$ (for some $i \geq 1$) where f is a node label of the input tree and r is bounded by a constant (recall that in the DAG construction, we consider the parameters appearing in the patterns as ordinary leaf labels). Productions $A \to x_i$ are eliminated by replacing all occurrences of A in a right-hand side by the parameter x_i. All resulting productions (except the start rule $S \to s$) have the form $A \to f(\alpha_1, \ldots, \alpha_r)$ where r is a constant and every α_i is either a nonterminal or a parameter. These productions are then split such that all resulting productions (except the start rule $S \to s$) are in Chomsky normal form. This is straightforward. For instance the production $A \to f(A_1, A_2, A_3)$ is split into $A \to B(A_3)$, $B \to C(A_2, x_1)$, $C \to D(A_1, x_1, x_2)$ and $D \to f(x_1, x_2, x_3)$. Recall that we assume that the maximal rank of terminal symbols is bounded by a constant. Therefore, the above splitting increases the size and depth only by a constant.

Recall that for the start rule $S \to s$ returned by BU-Shrink, the tree s has size $\mathcal{O}(n/\log_\sigma n) = \mathcal{O}((n \cdot \log \sigma)/\log n)$. We want to apply TreeBiSection to balance the tree s. But we cannot use it directly because the resulting running time would not be linear if σ is not a constant: Since TreeBiSection needs time $\mathcal{O}(|s| \log |s|)$ on trees of constant rank (see Theorem 4.4.6), this yields the time

$$\begin{aligned}\mathcal{O}(|s| \log |s|) &= \mathcal{O}\left(\frac{n \cdot \log \sigma}{\log n} \cdot \log\left(\frac{n \cdot \log \sigma}{\log n}\right)\right) \\ &= \mathcal{O}\left(\frac{n \cdot \log \sigma}{\log n} \cdot (\log n + \log\log \sigma - \log\log n)\right) = \mathcal{O}(n \log \sigma).\end{aligned}$$

To eliminate the factor $\log \sigma$, we apply BU-Shrink again to the tree s with $k = \log \sigma \leq \log n$. Note that the maximal rank in s is still bounded by a constant (the same constant as for the input tree). By Lemma 4.5.1 this yields in time $\mathcal{O}(|s|) \leq \mathcal{O}(n)$ a tree s' of size $\mathcal{O}(n/\log n)$ on which we may now use TreeBiSection to get a TSLP for s' in Chomsky normal form of size $\mathcal{O}(|s'|) = \mathcal{O}(n/\log n)$ (note that every node of s' may be labeled by a different symbol, in which case TreeBiSection cannot achieve any compression for s', when we count the size in bits) and depth $\mathcal{O}(\log |s'|) = \mathcal{O}(\log n)$. Moreover, the running time of TreeBiSection on s' is

$$\begin{aligned}\mathcal{O}(|s'| \cdot \log |s'|) &= \mathcal{O}\left(\frac{n}{\log n} \log\left(\frac{n}{\log n}\right)\right) \\ &= \mathcal{O}\left(\frac{n}{\log n} \cdot (\log n - \log\log n)\right) = \mathcal{O}(n).\end{aligned}$$

Let us call this combined algorithm BU-Shrink+TreeBiSection.

Theorem 4.5.3. *Let t be a tree of size n with $|\mathsf{labels}(t)| = \sigma$ such that the maximal rank of a symbol in $\mathsf{labels}(t)$ is bounded by a constant. Then* BU-Shrink+TreeBiSection *computes for t in time $\mathcal{O}(n)$ a TSLP in Chomsky normal form of size $\mathcal{O}\big(\frac{n}{\log_\sigma n}\big)$ and depth $\mathcal{O}(\log n)$. The rank of every nonterminal of that TSLP is bounded by the maximal rank of a symbol in $\mathsf{labels}(t)$ (a constant).*

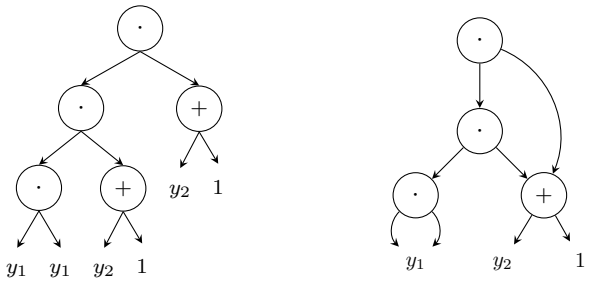

Figure 4.13: An artithmetical formula (left) and an arithmetical circuit (right). Both evaluate to the polynomial $(y_1^2 \cdot (y_2 + 1)) \cdot (y_2 + 1) = y_1^2 y_2^2 + 2y_1^2 y_2 + y_1^2$.

4.6 Arithmetical circuits

In this section, we present an application of Corollary 4.4.4 and Theorem 4.5.3. Let $\mathcal{S} = (S, +, \cdot)$ be a (not necessarily commutative) semiring. Thus, $(S, +)$ is a commutative monoid with identity element 0, $(S, \cdot)$ is a monoid with identity element 1, and $\cdot$ left and right distributes over $+$.

We use the standard notation of arithmetical formulas and circuits over $\mathcal{S}$: An *arithmetical formula* is just a labeled binary tree in which internal nodes are labeled by the semiring operations $+$ and $\cdot$, and leaf nodes are labeled by variables $y_1, y_2, \ldots$ or the constants 0 and 1. An *arithmetical circuit* is a (not necessarily minimal) directed, acyclic graph whose internal nodes are labeled by $+$ and $\cdot$ and whose leaf nodes are labeled by variables or the constants 0 and 1. The *depth* of a circuit is the length of a longest path from the root node to a leaf. An arithmetical circuit evaluates to a multivariate noncommutative polynomial $p(y_1, \ldots, y_n)$ over $\mathcal{S}$, where $y_1, \ldots, y_n$ are the variables occurring at the leaf nodes. An example is depicted in Figure 4.13. Two arithmetical circuits are equivalent if they evaluate to the same polynomial.

Brent [21] has shown that every arithmetical formula of size n over a commutative ring can be transformed into an equivalent circuit of depth $\mathcal{O}(\log n)$ and size $\mathcal{O}(n)$ (the proof easily generalizes to semirings). By first constructing a TSLP of size $\mathcal{O}((n \cdot \log m)/\log n)$, where m is the number of different variables in the formula, and then transforming this TSLP into a circuit, we will refine the size bound to $\mathcal{O}((n \cdot \log m)/\log n)$. Moreover, by Corollary 4.4.4 (Theorem 4.5.3, respectively) this conversion can be done in logspace (linear time, respectively).

In the following, we consider TSLPs over a ranked alphabet $\mathcal{F}_m$ which consists of the symbols $+$ and $\cdot$ of rank 2 and the symbols $0, 1, y_1, \ldots, y_m$ of rank 0 for some m. For our formula-to-circuit conversion, it will be important to work with monadic TSLPs, i.e., TSLPs in which every nonterminal has rank at most one (see Section 4.2). When we use this kind of TSLPs in the following, we label the single parameter node by x instead of x_1.

Lemma 4.6.1. *From a given tree $t \in \mathcal{T}(\mathcal{F}_m)$ of size n one can construct in logspace (linear time, respectively) a monadic TSLP $\mathcal{H}$ of size $\mathcal{O}\big(\frac{n \cdot \log m}{\log n}\big)$ and*

depth $\mathcal{O}(\log n)$ *with* $\mathsf{val}(\mathcal{H}) = t$ *such that all productions are of the following forms:*

- $A \to B(C)$ *for* $A, C \in \mathcal{N}_0$, $B \in \mathcal{N}_1$,
- $A \to B(C(x))$ *for* $A, B, C \in \mathcal{N}_1$,
- $A \to f(B, C)$ *for* $f \in \{+, \cdot\}$, $A, B, C \in \mathcal{N}_0$,
- $A \to f(x, B)$, $A(x) \to f(B, x)$ *for* $f \in \{+, \cdot\}$, $A \in \mathcal{N}_1$, $B \in \mathcal{N}_0$,
- $A \to a$ *for* $a \in \{0, 1, y_1, \ldots, y_m\}$, $A \in \mathcal{N}_0$,
- $A \to B(x)$ *for* $A, B \in \mathcal{N}_1$,
- $A \to x$ *for* $A \in \mathcal{N}_1$.

Proof. The linear time version is an immediate consequence of Theorem 4.2.1 and Theorem 4.5.3.[7] It remains to show the logspace version. We first apply TreeBiSection (Corollary 4.4.4) to get in logspace a TSLP $\mathcal{G}$ in Chomsky normal form of size $\mathcal{O}((n \cdot \log m)/\log n)$ and depth $\mathcal{O}(\log n)$ with $\mathsf{val}(\mathcal{G}) = t$. Note that every nonterminal of $\mathcal{G}$ has rank 3. Moreover, for every nonterminal A of rank $k \leq 3$, TreeBiSection computes $k+1$ nodes $v_0, v_1, \ldots, v_k$ of t that represent the pattern $\mathsf{val}_{\mathcal{G}}(A)$: v_0 is the root node of an occurrence of $\mathsf{val}_{\mathcal{G}}(A)$ in t and v_i $(1 \leq i \leq k)$ is the node of the occurrence to which the parameter x_i is mapped, see also the proof of Lemma 4.4.3. We can assume that for every nonterminal A of rank k this tuple s_A has been computed.

We basically show that the construction from [85], which makes a TSLP monadic, works in logspace if all nonterminals and terminals of the input TSLP have constant rank.[8] For a nonterminal A of rank 3 with $s_A = (v_0, v_1, v_2, v_3)$, the pattern $\mathsf{val}_{\mathcal{G}}(A)$ has two possible branching structures, which are the branching structures shown in Figure 4.7. By computing the paths from the three nodes v_1, v_2, v_3 up to v_0, we can compute in logspace, which of the two branching structures $\mathsf{val}_{\mathcal{G}}(A)$ has. Moreover, we can compute the two binary symbols $f_1, f_2 \in \{+, \cdot\}$ at which the three paths that go from v_1, v_2, and v_3, respectively, up to v_0 meet. We finally associate with each of the five dashed edges in Figure 4.7 a fresh unary nonterminal A_i $(0 \leq i \leq 4)$ of the TSLP $\mathcal{H}$. In this way we can built up in logspace what is called the *skeleton tree* for A. It is one of the following two valid patterns, depending on the branching structure of $\mathsf{val}_{\mathcal{G}}(A)$, see also Figure 4.14:

1. $A_0(f_1(A_1(f_2(A_2(x_1), A_3(x_1))), A_4(x_3)))$
2. $A_0(f_1(A_1(x_1), A_2(f_2(A_3(x_2), A_4(x_3)))))$

[7] Note that productions of the form $A \to B(x)$ and $A \to x$ do not appear in Theorem 4.2.1. We allow them in the lemma, since they make the logspace part of the lemma easier to show and do not pose a problem in the remaining part of this section.

[8] We only consider the case that nonterminals have rank at most three and terminals have rank zero or two, which is the case we need, but the general case, where all nonterminals and all terminals of the input TSLP have constant rank could be handled in a similar way in logspace.

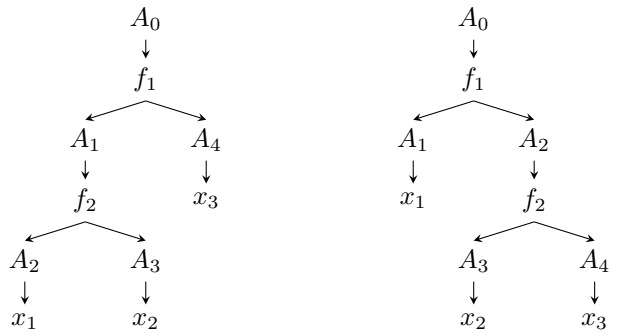

Figure 4.14: The two possible skeleton trees for a nonterminal A of rank three

For a nonterminal A of rank two there is only a single branching structure and hence a single skeleton tree $A_0(f(A_1(x_1), A_2(x_2)))$ for $f \in \{+, \cdot\}$. Finally, for a nonterminal A of rank at most one, the skeleton tree is A itself (this is in particular the case for the start nonterminal S, which will be also the start nonterminal of $\mathcal{H}$), and this nonterminal then belongs to $\mathcal{H}$ (nonterminals of $\mathcal{G}$ that have rank larger than one do not belong to $\mathcal{H}$). What remains is to construct in logspace productions for the nonterminals of $\mathcal{H}$ that allow to rewrite the skeleton tree of A to $\mathsf{val}_{\mathcal{G}}(A)$. For this, let us consider the productions of $\mathcal{G}$, whose right-hand sides have the form (4.3) and (4.4). A production $A \to f(x_1, \ldots, x_k)$ with $k \leq 1$ is copied to $\mathcal{H}$. On the other hand, if $k = 2$, then A does not belong to $\mathcal{H}$ and hence, we do not copy the production to $\mathcal{H}$. Instead, we introduce the productions $A_i \to x$ $(0 \leq i \leq 2)$ for the three nonterminals A_0, A_1, A_2 that appear in the skeleton tree of A. Now consider a production

$$A \to B(x_1, \ldots, x_{i-1}, C(x_i, \ldots, x_{i+l-1}), x_{i+l}, \ldots, x_k),$$

where $k, l, k-l+1 \leq 3$ (note that l is the rank of C and $k-l+1$ is the rank of B). We have constructed the skeleton trees t_A, t_B, t_C for A, B, and C, respectively. We now introduce the productions for the nonterminals that appear in t_A in such a way that t_A can be rewritten to the valid pattern

$$t_B[x_1, \ldots, x_{i-1}, t_C[x_i, \ldots, x_{i+l-1}], x_{i+l}, \ldots, x_k].$$

There are several cases depending on k, l, and i. Let us only consider two typical cases (all other cases can be dealt in a similar way): The patterns t_A and $t_B[x_1, t_C[x_2, x_3]]$ for a production $A \to B(x_1, C(x_2, x_3))$ are shown in Figure 4.15. Note that the skeleton tree t_A is the right tree from Figure 4.14. We add the following productions to $\mathcal{H}$:

$$A_0 \to B_0(x) \qquad A_1 \to B_1(x) \qquad A_2 \to B_2(C_0(x))$$
$$A_3 \to C_1(x) \qquad A_4 \to C_2(x)$$

Let us also consider the case $A \to B(x_1, x_2, C)$. The valid patterns t_A and $t_B[x_1, x_2, t_C]$ are shown in Figure 4.16 (we assume that the skeleton tree for B

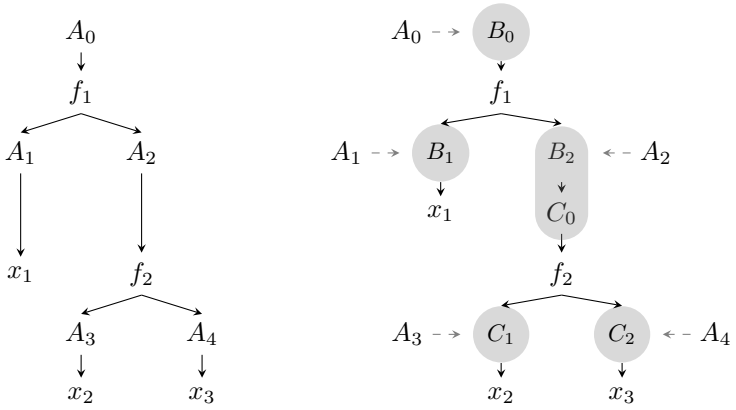

Figure 4.15: The skeleton tree t_A and the pattern $t_B[x_1, t_C[x_2, x_3]]$

is the left one from Figure 4.14). We add the following productions to $\mathcal{H}$:

$$A_0 \to B_0(f_1(B_1(x), B_4(C))), \quad A_1 \to B_2(x), \quad A_2 \to B_3(x) \tag{4.7}$$

Other cases can be dealt with similarly. In each case we write out a constant number of productions that clearly can be produced by a logspace machine using the shape of the skeleton trees. Correctness of the construction (i.e., $\mathsf{val}(\mathcal{G}) = \mathsf{val}(\mathcal{H})$) follows from $\mathsf{val}_{\mathcal{H}}(t_A) = \mathsf{val}_{\mathcal{G}}(A)$, which can be shown by a straightforward induction, see [85]. Clearly, the size and depth of $\mathcal{H}$ is linearly related to the size and depth, respectively, of $\mathcal{G}$. Finally, productions of the form $A \to B(f(C(x), D(E)))$ (or similar forms) as in (4.7) can be easily split in logspace into productions of the forms shown in the lemma. For instance, $A \to B(f(C(x), D(E)))$ is split into $A \to B(F(x))$, $F \to G(C(x))$, $G \to f(x, H)$, $H \to D(E)$. Again, the size and depth of the TSLP increases only by a linear factor. □

Going from a monadic TSLP to a circuit that evaluates over every semiring to the same noncommutative polynomial is easy:

Lemma 4.6.2. *From a given monadic TSLP $\mathcal{G}$ over the terminal alphabet $\mathcal{F}_m$ such that all productions are of the form shown in Lemma 4.6.1, one can construct in logspace (linear time, respectively) an arithmetical circuit C of depth $\mathcal{O}(\mathrm{depth}(\mathcal{G}))$ and size $\mathcal{O}(|\mathcal{G}|)$ such that over every semiring, C and $\mathsf{val}(\mathcal{G})$ evaluate to the same noncommutative polynomial in m variables.*

Proof. Fix an arbitrary semiring $\mathcal{S}$ and let $\mathcal{R}$ be the polynomial semiring $\mathcal{R} = \mathcal{S}[y_1, \ldots, y_m]$. Clearly, for a nonterminal A of rank 0, $\mathsf{val}_{\mathcal{G}}(A)$ is a tree without parameters that evaluates to an element p_A of the semiring $\mathcal{R}$. For a nonterminal A of rank 1, $\mathsf{val}_{\mathcal{G}}(A)$ is a context in which the only parameter x occurs exactly once. Such a context evaluates to a noncommutative polynomial $p_A(x) \in \mathcal{R}[x]$. Since the parameter x occurs exactly once in the tree $\mathsf{val}(A)$, it turns out that

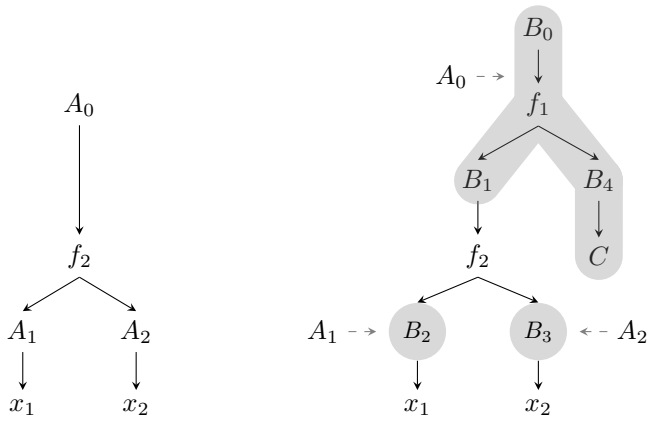

Figure 4.16: The skeleton tree t_A and the tree $t_B[x_1, x_2, t_C]$

$p_A(x)$ is linear and contains exactly one occurrence of x. More precisely, by induction on the structure of the TSLP $\mathcal{G}$ we show that for every nonterminal A of rank 1, the tree $\mathsf{val}_{\mathcal{G}}(A)$ evaluates in $\mathcal{R}[x]$ to a noncommutative polynomial of the form

$$p_A(x) = A_0 + A_1 x A_2,$$

where $A_0, A_1, A_2 \in \mathcal{R} = \mathcal{S}[y_1, \ldots, y_m]$. Using the same induction, one can build up a circuit of size $\mathcal{O}(|\mathcal{G}|)$ and depth $\mathcal{O}(\mathrm{depth}(\mathcal{G}))$ that contains gates evaluating to A_0, A_1, A_2. For a nonterminal A of rank zero, the circuit contains a gate that evaluates to the semiring element $p_A \in \mathcal{R}$, and we denote this gate with A as well.

The induction uses a straightforward case distinction on the rule for A. The cases that the unique rule for A has the form $A \to x$, $A \to B(x)$, $A \to f(x, B)$, $A \to f(B, x)$, $A \to f(B, C)$, or $A \to a$ is clear ($f \in \{+, \cdot\}$, $a \in \{0, 1, y_1, \ldots, y_m\}$). For instance, for a rule $A \to +(B, x)$, we have $p_A(x) = B + 1 \cdot x \cdot 1$, i.e., we set $A_0 := B$, $A_1 := 1$, $A_2 := 1$. Now consider a rule $A \to B(C(x))$ (for $A \to B(C)$ the argument is similar). We have already built up a circuit containing gates that evaluate to $B_0, B_1, B_2, C_0, C_1, C_2$, where

$$p_B(x) = B_0 + B_1 x_1 B_2, \qquad p_C(x) = C_0 + C_1 x C_2.$$

We get

$$\begin{aligned} p_A(x) &= p_B(p_C(x)) \\ &= B_0 + B_1(C_0 + C_1 x C_2)B_2 \\ &= (B_0 + B_1 C_0 B_2) + B_1 C_1 x_1 C_2 B_2 \end{aligned}$$

and therefore set

$$A_0 := B_0 + B_1 C_0 B_2,\ A_1 := B_1 C_1,\ A_2 := C_2 B_2.$$

So we can define the polynomials A_0, A_1, A_2 using the gates B_0, B_1, B_2, C_0, C_1, C_2 with only 5 additional gates. Note that also the depth only increases by a constant factor (in fact, 2).

The output gate of the circuit is the start nonterminal of the TSLP $\mathcal{G}$. The above construction can be carried out in linear time as well as in logspace. □

Now we can show the main result of this section:

Theorem 4.6.3. *A given arithmetical formula F of size n having m different variables can be transformed in logspace (linear time, respectively) into an arithmetical circuit C of depth $\mathcal{O}(\log n)$ and size $\mathcal{O}\big(\frac{n \cdot \log m}{\log n}\big)$ such that over every semiring, C and F evaluate to the same noncommutative polynomial in m variables.*

Proof. Let F be an arithmetical formula of size n and let $y_1, \ldots, y_m$ be the variables occurring in F. Fix an arbitrary semiring $\mathcal{S}$ and let $\mathcal{R}$ be the polynomial semiring $\mathcal{R} = \mathcal{S}[y_1, \ldots, y_m]$. Using Lemma 4.6.1 we can construct in logspace (linear time, respectively) a monadic TSLP $\mathcal{G}$ of size $\mathcal{O}((n \cdot \log m)/\log n)$ and depth $\mathcal{O}(\log n)$ such that $\mathsf{val}(\mathcal{G}) = F$. Finally, we apply Lemma 4.6.2 in order to transform $\mathcal{G}$ in logspace (linear time, respectively) into an equivalent circuit of size $\mathcal{O}((n \cdot \log m)/\log n)$ and depth $\mathcal{O}(\log n)$. □

Theorem 4.6.3 can also be shown for fields instead of semirings. In this case, the expression is built up using variables, the constants -1, 0, 1, and the field operations $+, \cdot$ and $/$. The proof is similar to the semiring case. Again, we start with a monadic TSLP of size $\mathcal{O}((n \cdot \log m)/\log n)$ and depth $\mathcal{O}(\log n)$ for the arithmetical expression. Again, one can assume that all rules have the form $A \to B(C(x))$, $A \to B(C)$, $A \to f(x, B)$, $A \to f(B, x)$, $A \to f(B, C)$, $A \to B(x)$, $A \to x$, or $A \to a$, where f is one of the binary field operations and a is either -1, 0, 1, or a variable. Using this particular rule format, one can show that every nonterminal A of rank 1 evaluates to a rational function $(A_0 + A_1 x)/(A_2 + A_3 x)$ for polynomials A_0, A_1, A_2, A_3 in the circuit variables, whereas a nonterminal of rank 0 evaluates to a fraction of two polynomials. Finally, these polynomials can be computed by a single circuit of size $\mathcal{O}((n \cdot \log m)/\log n)$ and depth $\mathcal{O}(\log n)$.

Lemma 4.6.2 has an interesting application to the problem of checking whether the polynomial represented by a TSLP over a ring is the zero polynomial. The question, whether the polynomial computed by a given circuit is the zero polynomial is known as *polynomial identity testing* (PIT). This is a famous problem in algebraic complexity theory. For the case that the underlying ring is $\mathbb{Z}$ or $\mathbb{Z}_n$ ($n \geq 2$) polynomial identity testing belongs to the complexity class **coRP** (the complement of randomized polynomial time), see [2]. PIT can be generalized to arithmetic expressions that are given by a TSLP. Using Lemma 4.6.2 and Theorem 4.2.1 we obtain:

Theorem 4.6.4. *Over any semiring, the question, whether the polynomial computed by a given TSLP is the zero polynomial, is equivalent with respect to polynomial time reductions to PIT. In particular, if the underlying semiring is $\mathbb{Z}$*

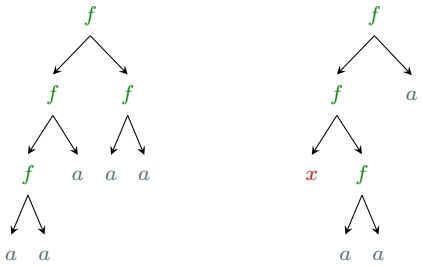

Figure 4.17: A tree from $\mathcal{T}$ (left) and a context from $\mathcal{C}$ (right)

or $\mathbb{Z}_n$, then the question, whether the polynomial computed by a given TSLP is the zero polynomial, belongs to **coRP**.

4.7 Source coding for unlabeled binary trees

In the following sections we aim to apply grammar-based tree compression to universal source coding for unlabeled binary trees. In particular, we extend the encoding presented in [72] from SLPs (see Section 3.9) to TSLPs (Section 4.9). The key to derive strong universality bounds for the new tree encoding is the fact that each unlabeled binary tree of size n is produced by a TSLP of size $\mathcal{O}(n/\log n)$ (Corollary 4.4.4 and Theorem 4.5.3). A similar attempt, but based on DAGs instead of general TSLPs, was made in [111]. As a second contribution, we will sharpen some universality bounds presented in [111] in the next section. To do so, we use that the minimal DAG has size $\mathcal{O}(n/\log n)$ for unlabeled β-balanced binary trees of size n as discussed in Section 4.3.1.

Unlabeled binary trees. We start with some notations. We fix a ranked alphabet $\mathcal{F} = \{a, f\}$ with $a \in \mathcal{F}_0$ and $f \in \mathcal{F}_2$ in the following and we simply denote by $\mathcal{T} = \mathcal{T}(\mathcal{F})$ the set of unlabeled binary trees, i.e., $\mathcal{T}$ is the smallest set of terms such that (i) $a \in \mathcal{T}$ and (ii) if $t_1, t_2 \in \mathcal{T}$ then also $f(t_1, t_2) \in \mathcal{T}$. Further, we deal with patterns of rank at most one in the remaining chapter and thus we are restricted to trees and contexts in the following. As a consequence, instead of numbering the parameters beginning with x_1, we just use the label x for the single leaf of a context which is labeled by a parameter. We denote by $\mathcal{C}$ the set of all contexts over $\mathcal{F} \cup \{x\}$, i.e., $\mathcal{C}$ is the smallest set such that (i) $x \in \mathcal{C}$ and (ii) if $s \in \mathcal{C}$ and $t \in \mathcal{T}$ then also $f(s,t), f(t,s) \in \mathcal{C}$. Examples for a tree in $\mathcal{T}$ and a context in $\mathcal{C}$ are depicted in Figure 4.17.

Since we exclusively deal with binary trees in the following, we mainly use the leaf size $\mathsf{leafsize}(t)$ of a tree or context $t \in \mathcal{T} \cup \mathcal{C}$ instead of the size $|t|$. Recall that if $t \in \mathcal{T}$ then $|t| = 2 \cdot \mathsf{leafsize}(t) - 1$ and if $t \in \mathcal{C}$ then $|t| = 2 \cdot \mathsf{leafsize}(t)$ since we do not count the leaf labeled by the parameter (neither in $\mathsf{leafsize}(t)$ nor in $|t|$). Let $\mathcal{T}_n = \{t \in \mathcal{T} \mid \mathsf{leafsize}(t) = n\}$ for $n \geq 1$ and $\mathcal{C}_n = \{t \in \mathcal{C} \mid \mathsf{leafsize}(t) = n\}$ for $n \geq 0$ (the context x satisfies $\mathsf{leafsize}(x) = 0$). It is well known that the

number of trees in $\mathcal{T}_n$ is exactly the $(n-1)$-th Catalan number. Let c_k be the k-th Catalan number. The following asymptotic is well known, see e.g. [41]:

$$c_k \sim \frac{4^k}{\sqrt{\pi} k^{\frac{3}{2}}}$$

In fact, we have $c_k \leq 4^k$ for all $k \geq 0$ and thus $|\mathcal{T}_n| \leq 4^{n-1}$ (this refines Lemma 4.1.1 for the case of unlabeled binary trees). We also consider the set of all trees of a certain depth in the following and we denote by $\mathcal{T}^d = \{t \in \mathcal{T} \mid d(t) = d\}$ the set of all trees of depth d for $d \geq 0$.

Tree sources. A *tree source* $\mathcal{S}$ is a pair $\mathcal{S} = ((\mathcal{P}_i)_{i\in\mathbb{N}}, p_{\mathcal{S}})$ such that

- $\mathcal{P}_i \subseteq \mathcal{T}$ is non-empty and finite for every $i \geq 0$,
- $\mathcal{P}_i \cap \mathcal{P}_j = \emptyset$ for $i \neq j$ and $\bigcup_{i\geq 0} \mathcal{P}_i = \mathcal{T}$, i.e., the sets $\mathcal{P}_i$ partition $\mathcal{T}$,
- $p_{\mathcal{S}} : \mathcal{T} \to \mathbb{R}_{[0,1]}$ such that $\sum_{t\in\mathcal{P}_i} p_{\mathcal{S}}(t) = 1$ for every $i \geq 0$, i.e., $p_{\mathcal{S}}$ is a probability distribution on $\mathcal{P}_i$ for each $i \geq 0$.

We consider two cases for the partition $(\mathcal{P}_i)_{i\in\mathbb{N}}$ in this work:

(i) $\mathcal{P}_i = \mathcal{T}_{i+1}$ for all $i \in \mathbb{N}$ (*leaf-centric*)

(ii) $\mathcal{P}_i = \mathcal{T}^i$ for all $i \in \mathbb{N}$ (*depth-centric*)

For the leaf-centric tree sources described in case (i), we start with $\mathcal{T}_1$ because there is no tree $t \in \mathcal{T}$ such that $\mathsf{leafsize}(t) = 0$. For the depth-centric tree sources described in case (ii) we have $d(t) = 0$ for the tree $t = a$ and thus we start with $\mathcal{T}^0$ in this case.

Tree encoders and redundancy. A *tree encoder* is an injective mapping $E : \mathcal{T} \to \{0,1\}^*$ such that the range $E(\mathcal{T})$ is prefix-free, i.e., there do not exist $t, t' \in \mathcal{T}$ with $t \neq t'$ such that $E(t)$ is a prefix of $E(t')$. We define the *worst-case redundancy* of E with respect to the fixed tree source $\mathcal{S} = ((\mathcal{P}_i)_{i\in\mathbb{N}}, p_{\mathcal{S}})$ as the mapping $n \mapsto R(E, \mathcal{S}, n)$ $(n \in \mathbb{N})$ with

$$R(E, \mathcal{S}, n) = \max_{t\in\mathcal{P}_n, p_{\mathcal{S}}(t)>0} \frac{1}{\mathsf{leafsize}(t)} \cdot (|E(t)| + \log p_{\mathcal{S}}(t))$$

(the maximum is taken over all trees $t \in \mathcal{P}_n$ such that $p_{\mathcal{S}}(t) > 0$). The worst-case redundancy is also known as the *maximal pointwise redundancy*. A tree encoder E is (worst-case) *universal* for a tree source $\mathcal{S}$ if $R(E, \mathcal{S}, n)$ converges to zero for $n \to \infty$. Moreover, we define the *average-case redundancy* of E with respect to the tree source $\mathcal{S} = ((\mathcal{P}_i)_{i\in\mathbb{N}}, p_{\mathcal{S}})$ as the mapping $n \mapsto R_{\varnothing}(E, \mathcal{S}, n)$ $(n \in \mathbb{N})$ with

$$R_{\varnothing}(E, \mathcal{S}, n) = \sum_{t\in\mathcal{P}_n, p_{\mathcal{S}}(t)>0} \frac{1}{\mathsf{leafsize}(t)} \cdot (|E(t)| + \log p_{\mathcal{S}}(t)) \cdot p_{\mathcal{S}}(t).$$

A tree encoder E is (average-case) universal for a tree source $\mathcal{S}$ if $R_{\varnothing}(E, \mathcal{S}, n)$ converges to zero for $n \to \infty$.

4.8 Universal coding based on DAGs

In this section we sharpen some of the results from [111], where universal source coding of unlabeled binary trees based on the minimal DAG is investigated. Recall that the DAG of a tree is a directed, acyclic graph such that each distinct subtree of t occurs only once (see Section 4.3). In [111], only bounds on the average-case redundancy for certain classes of tree sources were shown. Here we extend these bounds (for the same classes of tree sources) to the worst-case redundancy. To do so, we use two results presented in Section 4.3.1. First, we use the bound on the size of the DAG of β-balanced trees provided in Theorem 4.3.2. Second, we use the bound on the size of the DAG of trees where each subtree has logarithmic depth in dependence on its size (see Theorem 4.3.4; this result was implicitly shown in [58]).

The following condition on a tree source was introduced in [111], where it is called the domination property (later, we will introduce a strong domination property, so we call it weak domination property here): Let $\mathcal{S} = ((\mathcal{P}_i)_{i\in\mathbb{N}}, p_{\mathcal{S}})$ be a tree source. We say that $\mathcal{S}$ has the *weak domination property* if there exists a mapping $\lambda : \mathcal{T} \to \mathbb{R}_{>0}$ with the following properties:

(i) $\lambda(t) \geq p_{\mathcal{S}}(t)$ for every $t \in \mathcal{T}$,

(ii) $\lambda(f(s,t)) \leq \lambda(s) \cdot \lambda(t)$ for all $s, t \in \mathcal{T}$,

(iii) There are constants c_1, c_2 such that $\sum_{t\in\mathcal{T}_n} \lambda(t) \leq c_1 \cdot n^{c_2}$ for all $n \geq 1$.

In [111], the authors define a binary encoding $B(\mathsf{DAG}(t)) \in \{0,1\}^*$, such that $B(\mathsf{DAG}(t))$ is not a prefix of $B(\mathsf{DAG}(t'))$ for all $t, t' \in \mathcal{T}$ with $t \neq t'$. The precise definition of $B(\mathsf{DAG}(t))$ is not important for us; all we need is the following bound from [111], where $E_{\mathrm{dag}} : \mathcal{T} \to \{0,1\}^*$ is the tree encoder with $E_{\mathrm{dag}}(t) = B(\mathsf{DAG}(t))$.

Lemma 4.8.1 ([111, Theorem 2])**.** *Assume that $\mathcal{S} = ((\mathcal{P}_i)_{i\in\mathbb{N}}, p_{\mathcal{S}})$ has the weak domination property. Let $t \in \mathcal{T}_n$ with $n \geq 2$ and $p_{\mathcal{S}}(t) > 0$, then we have*

$$\frac{1}{n} \cdot (|E_{dag}(t)| + \log(p_{\mathcal{S}}(t))) \leq \mathcal{O}\left(\frac{|\mathsf{DAG}(t)|}{n}\right) + \mathcal{O}\left(\frac{|\mathsf{DAG}(t)|}{n} \cdot \log\left(\frac{n}{|\mathsf{DAG}(t)|}\right)\right).$$

Based on this lemma, the following bound on the average-case redundancy was derived:

Theorem 4.8.2 ([111, Theorem 2])**.** *Consider the mapping $g(x) = x \cdot \log(1/x)$ and assume that the tree source $\mathcal{S} = ((\mathcal{P}_i)_{i\in\mathbb{N}}, p_{\mathcal{S}})$ has the weak domination property. There exists a positive real number k, depending only on $\mathcal{S}$, such that for all $n \in \mathbb{N}$ we have*

$$R_{\varnothing}(E_{dag}, \mathcal{S}, n) \leq k \cdot g\left(\sum_{t\in\mathcal{P}_n} p_{\mathcal{S}}(t) \cdot \frac{|\mathsf{DAG}(t)|}{\mathsf{leafsize}(t)}\right).$$

This bound is used to show that for certain leaf-centric and depth-centric tree sources the encoding E_{dag} is universal in the sense that the average-case redundancy converges to zero. Here, we want to show that for the same tree sources already the worst-case redundancy converges to zero. Let us first define the specific classes of tree sources studied in [111].

4.8.1 Leaf-centric binary tree sources

We recall the definition of a natural class of leaf-centric tree sources from [111]: Let Σ_{leaf} be the set of all functions $\sigma : (\mathbb{N} \setminus \{0\}) \times (\mathbb{N} \setminus \{0\}) \to \mathbb{R}_{[0,1]}$ such that for all $n \geq 2$:

$$\sum_{i,j \geq 1,\ i+j=n} \sigma(i,j) = 1. \tag{4.8}$$

For $\sigma \in \Sigma_{\mathrm{leaf}}$ we define $p_\sigma : \mathcal{T} \to \mathbb{R}_{[0,1]}$ inductively by:

$$\begin{aligned} p_\sigma(a) &= 1, & (4.9)\\ p_\sigma(f(s,t)) &= \sigma(\mathsf{leafsize}(s), \mathsf{leafsize}(t)) \cdot p_\sigma(s) \cdot p_\sigma(t). & (4.10) \end{aligned}$$

We have $\sum_{t \in \mathcal{T}_n} p_\sigma(t) = 1$ and thus $((\mathcal{T}_i)_{i \geq 1}, p_\sigma)$ is a leaf-centric tree source.

Example 4.13. *Here are three examples of leaf-centric tree sources of the above form, which are also discussed in [75] with respect to their entropy rates:*

- *The* binary search tree model $((\mathcal{T}_i)_{i \geq 1}, p_{\sigma_{bst}})$*, where*

$$\sigma_{bst}(k, n-k) = \frac{1}{n-1}$$

  for all $k \in [1, n-1]$.

- *The* uniform model $((\mathcal{T}_i)_{i \geq 1}, p_{\sigma_{uni}})$*, where*

$$\sigma_{uni}(k, n-k) = \frac{c_{k-1} \cdot c_{n-k-1}}{c_{n-1}}$$

  for all $k \in [1, n-1]$*. Here,* $p_{\sigma_{uni}}$ *is the uniform distribution on* $\mathcal{T}_i$ *for* $i \geq 1$.

- *The* binomial random tree model $((\mathcal{T}_i)_{i \geq 1}, p_{\sigma_{bin}})$ *where*

$$\sigma_{bin}(k, n-k) = \binom{n-2}{k-1} \cdot \frac{1}{2^{n-2}}$$

  for all $k \in [1, n-1]$.

The following result is implicitly shown in [111] and can also be found in the proof of Theorem 4.9.9.

Lemma 4.8.3. *For every* $\sigma \in \Sigma_{leaf}$*, the leaf-centric tree source* $((\mathcal{T}_i)_{i \geq 1}, p_\sigma)$ *has the weak domination property.*

Worst-case redundancy

We say that a mapping $\sigma \in \Sigma_{\text{leaf}}$ is *leaf-balanced* if there exists a constant c such that for all $(i,j) \in (\mathbb{N} \setminus \{0\}) \times (\mathbb{N} \setminus \{0\})$ with $\sigma(i,j) > 0$ we have

$$\frac{i+j}{\min\{i,j\}} \leq c.$$

In [111] it is shown that for a leaf-balanced $\sigma \in \Sigma_{\text{leaf}}$, the tree source $((\mathcal{T}_i)_{i\geq 1}, p_\sigma)$ has the so-called representation ratio negligibility property. This property states that the average compression ratio achieved by the minimal DAG (formally, $\sum_{t\in\mathcal{T}_n} p_\sigma(t) \cdot |\mathsf{DAG}(t)|/n$) converges to zero for $n \to \infty$. Using our result from Section 4.3.1 (Theorem 4.3.2), we show the following stronger property.

Lemma 4.8.4. *For every leaf-balanced mapping $\sigma \in \Sigma_{leaf}$, there exists a constant α such that for each $t \in \mathcal{T}_n$ with $p_\sigma(t) > 0$ we have*

$$|\mathsf{DAG}(t)| \leq \alpha \cdot \frac{n}{\log n}.$$

Proof. Recall the notion of β-balanced trees presented in Section 4.3.1. By Theorem 4.3.2, we know that for every constant β there exists a constant α (depending only on β) such that the minimal DAG of a β-balanced tree $t \in \mathcal{T}_n$ satisfies $|\mathsf{DAG}(t)| \leq \alpha \cdot n/\log n$. It follows that we only need to show that a tree $t \in \mathcal{T}_n$ with $p_\sigma(t) > 0$ (where σ is leaf-balanced) is β-balanced for a constant β. Since the mapping σ is leaf-balanced, every subtree $f(t_1,t_2)$ of t with $n_i = \mathsf{leafsize}(t_i)$ for $i \in [1,2]$ satisfies $n_1 + n_2 \leq c \cdot \min\{n_1,n_2\}$, where $c \geq 1$ is a constant. Without loss of generality assume that $n_1 \leq n_2$. We get $n_1 \leq c \cdot n_2$ and

$$n_2 \leq n_1 + n_2 \leq c \cdot \min\{n_1,n_2\} = c \cdot n_1,$$

which shows that t is c-balanced. □

Corollary 4.8.5. *Let $\sigma \in \Sigma_{leaf}$ be leaf-balanced, and let $\mathcal{S} = ((\mathcal{T}_i)_{i\geq 1}, p_\sigma)$ be the corresponding leaf-centric tree source. Then, we have*

$$R(E_{dag}, \mathcal{S}, n) \leq \mathcal{O}\left(\frac{\log\log n}{\log n}\right).$$

Proof. Let α be the constant from Lemma 4.8.4. Let $t \in \mathcal{T}_n$ such that $p_\sigma(t) > 0$. Lemma 4.8.4 implies $|\mathsf{DAG}(t)| \leq \alpha \cdot n/\log n$. With Lemma 4.8.1 and 4.8.3 we get

$$\frac{1}{n} \cdot (|E_{\text{dag}}(t)| + \log(p_\sigma(t))) \leq \mathcal{O}\left(\frac{\log\log n}{\log n}\right).$$

This proves the corollary. □

4.8.2 Depth-centric binary tree sources

We recall the definition of a natural class of depth-centric tree sources from [111]: Let Σ_{depth} be the set of all mappings $\sigma : \mathbb{N} \times \mathbb{N} \to \mathbb{R}_{[0,1]}$ such that for all $n \geq 1$:

$$\sum_{i,j \geq 0,\ \max(i,j)=n-1} \sigma(i,j) = 1. \tag{4.11}$$

For $\sigma \in \Sigma_{\text{depth}}$, we define $p_\sigma : \mathcal{T} \to [0,1]$ by

$$p_\sigma(a) = 1, \tag{4.12}$$
$$p_\sigma(f(s,t)) = \sigma(d(s), d(t)) \cdot p_\sigma(s) \cdot p_\sigma(t). \tag{4.13}$$

We have $\sum_{t \in \mathcal{T}^n} p_\sigma(t) = 1$ and thus $((\mathcal{T}^i)_{i \geq 0}, p_\sigma)$ is a depth-centric tree source.

The following result is implicitly shown in [111] and can also be found as a part of the proof of Theorem 4.9.10.

Lemma 4.8.6. *For every $\sigma \in \Sigma_{depth}$, the depth-centric tree source $((\mathcal{T}^i)_{i \geq 0}, p_\sigma)$ has the weak domination property.*

We say the mapping $\sigma \in \Sigma_{\text{depth}}$ is *depth-balanced* if there exists a constant c such that for all $(i,j) \in \mathbb{N} \times \mathbb{N}$ with $\sigma(i,j) > 0$ we have $|i - j| \leq c$. In [111], the authors define a condition on σ that is slightly stronger than depth-balancedness, and show that for every such σ, the tree source $((\mathcal{T}^i)_{i \geq 0}, p_\sigma)$ has the representation ratio negligibility property. Similarly to Lemma 4.8.4, we will show an even stronger property. To do so, we introduce *β-depth-balanced* trees for $\beta \in \mathbb{N}$. A tree t is called *β-depth-balanced* if for each subtree $f(t_1, t_2)$ of t we have $|d(t_1) - d(t_2)| \leq \beta$. Note that for a depth-balanced mapping $\sigma \in \Sigma_{\text{depth}}$, there is a constant β such that every tree t with $p_\sigma(t) > 0$ is β-depth-balanced. We will use the following lemma:

Lemma 4.8.7. *Let $\beta \in \mathbb{N}$ and $c = 1 + \frac{1}{1+\beta}$ (thus, $1 < c \leq 2$). For every β-depth-balanced binary tree t, we have $\mathsf{leafsize}(t) \geq c^{d(t)}$.*

Proof. We prove the lemma by induction on $d(t)$. For the only tree $t = a$ of depth $d(t) = 0$, we have $\mathsf{leafsize}(t) = 1 = c^0$. Consider now a β-depth-balanced tree $t = f(t_1, t_2)$ of depth $d(t) > 0$. We assume $d(t_1) \geq d(t_2)$, the other case is symmetric. Since t is β-depth-balanced, it follows that $d(t_2) \geq d(t_1) - \beta$. Let $n = \mathsf{leafsize}(t)$ and $n_i = \mathsf{leafsize}(t_i)$ for $i \in [1,2]$. To estimate the size $n = n_1 + n_2$, we apply the induction hypothesis to t_1 and t_2, which yields

$$n = n_1 + n_2 \geq c^{d(t_1)} + c^{d(t_2)} \geq c^{d(t_1)} + c^{d(t_1)-\beta} = c^{d(t_1)} \cdot \left(1 + c^{-\beta}\right).$$

Since $d(t_1) + 1 = d(t)$, it only remains to show that $1 + c^{-\beta} \geq c$, which can be easily done by induction on $\beta \in \mathbb{N}$. □

Lemma 4.8.7 together with Theorem 4.3.4 (which was implicitly shown in [58]) implies:

Lemma 4.8.8. *For every depth-balanced mapping $\sigma \in \Sigma_{depth}$ there exists a constant α such that for every binary tree $t \in \mathcal{T}_n$ with $p_\sigma(t) > 0$ we have*

$$|\mathsf{DAG}(t)| \leq \alpha \cdot \frac{n \cdot \log\log n}{\log n}.$$

Proof. If $p_\sigma(t) > 0$, then there exists a constant β such that t and each of its subtrees is β-depth-balanced. By Lemma 4.8.7 this implies that every subtree t' has depth at most $c \cdot \log |t'|$ for a constant c that only depends on σ. Theorem 4.3.4 it follows that there exists a constant α (again, only dependent on σ) such that $|\mathsf{DAG}(t)| \leq \alpha \cdot (n \cdot \log\log n)/\log n$. □

Corollary 4.8.9. *Let $\sigma \in \Sigma_{depth}$ be depth-balanced and let $\mathcal{S} = ((\mathcal{T}^i)_{i\geq 0}, p_\sigma)$ be the corresponding depth-centric tree source. Then, we have*

$$R(E_{dag}, \mathcal{S}, n) \leq \mathcal{O}\left(\frac{(\log\log n)^2}{\log n}\right).$$

Proof. Let α be the constant from Lemma 4.8.8. Let $t \in \mathcal{T}^n$ such that $p_\sigma(t) > 0$ and let $\ell = \mathsf{leafsize}(t)$. Lemma 4.8.1 and 4.8.6 imply

$$\frac{1}{\ell} \cdot (|E_{\mathrm{dag}}(t)| + \log(p_\sigma(t))) \leq \mathcal{O}\left(\frac{|\mathsf{DAG}(t)|}{\ell}\right) + \mathcal{O}\left(\frac{|\mathsf{DAG}(t)|}{\ell} \cdot \log\left(\frac{\ell}{|\mathsf{DAG}(t)|}\right)\right).$$

Consider the mapping $g(x) = x \cdot \log(1/x)$. It is monotonically increasing for $0 \leq x \leq 1/e$. Note that for all $t \in \mathcal{T}^n$ we have $\mathsf{leafsize}(t) \geq n + 1$. Hence, if n is large enough, then Lemma 4.8.8 yields for all $t \in \mathcal{T}^n$ with $p_\sigma(t) > 0$ and $\ell = \mathsf{leafsize}(t)$ that

$$\frac{|\mathsf{DAG}(t)|}{\ell} \leq \alpha \cdot \frac{\log\log \ell}{\log \ell} \leq \frac{\log(\log(n+1))}{\log(n+1)} \leq 1/e.$$

We obtain

$$\frac{1}{\mathsf{leafsize}(t)} \cdot (|E_{\mathrm{dag}}(t)| + \log(p_\sigma(t))) \leq \mathcal{O}\left(\frac{(\log\log n)^2}{\log n}\right).$$

This proves the corollary. □

4.9 Universal coding based on TSLPs

In this section, we will use TSLPs in order to obtain a worst-case universal code for unlabeled binary trees. The limitations of DAGs for universal source coding can be best seen for a tree source $\mathcal{S} = ((\mathcal{P}_i)_{i\in\mathbb{N}}, p_\mathcal{S})$ such that $p_\mathcal{S}(t) > 0$ for all $t \in \mathcal{T}$. As it is shown in Example 4.9, the tree $t_n = f(f(f(\cdots f(a,a), \cdots a), a), a)$ (where f occurs n times) has leaf size $n + 1$ and $|\mathsf{DAG}(t_n)| = n + 1$. Together with $p_\mathcal{S}(t_n) > 0$ this implies that the bound stated in Lemma 4.8.1 cannot be used to show that the worst-case redundancy converges to zero. In the following we encode unlabeled binary trees using general TSLPs, where we can use the results achieved in Corollary 4.4.4 and Theorem 4.5.3. First, we present a normal form which we use to encode TSLPs.

4.9.1 TSLPs in normal form

In this section, we use TSLPs in a certain *normal form*. Basically, a normal form TSLP is a monadic TSLP combined with a specification on the naming of the nonterminals. Formally, a TSLP $\mathcal{G} = (\mathcal{N}, \mathcal{F}, A_0, P)$ with $\mathcal{F}_0 = \{a\}$, $\mathcal{F}_2 = \{f\}$ and $\mathcal{F} = \mathcal{F}_0 \cup \mathcal{F}_2$ is in normal form if the following conditions hold:

- $\mathcal{N} = \mathcal{N}_0 \cup \mathcal{N}_1 = \{A_0, A_1, \ldots, A_{n-1}\}$ for some $n \in \mathbb{N} \setminus \{0\}$.
- For every $A_i \in \mathcal{N}_0$, we have $(A_i \to A_j(\alpha)) \in P$ for some $A_j \in \mathcal{N}_1$ and $\alpha \in \mathcal{N}_0 \cup \{a\}$.
- For every $A_i \in \mathcal{N}_1$, we have $(A_i \to A_j(A_k(x))) \in P$, or $(A_i \to f(\alpha, x)) \in P$, or $(A_i \to f(x, \alpha)) \in P$, where $A_j, A_k \in \mathcal{N}_1$ and $\alpha \in \mathcal{N}_0 \cup \{a\}$.
- For every $A_i \in \mathcal{N}$ define the word $\rho(A_i) \in (\mathcal{N} \cup \{a\})^*$ as follows:

$$\rho(A_i) = \begin{cases} A_j\alpha & \text{if } (A_i \to A_j(\alpha)) \in P \\ A_jA_k & \text{if } (A_i \to A_j(A_k(x))) \in P \\ \alpha & \text{if } (A_i \to f(\alpha, x)) \in P \text{ or } (A_i \to f(x, \alpha)) \in P \end{cases}$$

  Let $\rho_{\mathcal{G}} = \rho(A_0)\rho(A_1)\cdots\rho(A_{n-1}) \in \{a, A_1, A_2, \ldots, A_{n-1}\}^*$. Then it is required that $\rho_{\mathcal{G}}$ is of the form $\rho_{\mathcal{G}} = A_1u_1A_2u_2\cdots A_{n-1}u_{n-1}$ such that $u_i \in \{a, A_1, A_2, \ldots, A_i\}^*$, i.e., the nonterminals are numbered according to their first left-to-right occurrence in $\rho_{\mathcal{G}}$.
- $\mathsf{val}_{\mathcal{G}}(A_i) \neq \mathsf{val}_{\mathcal{G}}(A_j)$ for $i \neq j$

A normal form TSLP as it is defined above can not produce the singleton tree a and thus we also allow the TSLP $\mathcal{G}_a = (\{A_0\}, \mathcal{F}, A_0, \{A_0 \to a\})$. In this case, we set $\rho_{\mathcal{G}_a} = \rho(A_0) = a$. Note that $|\rho_{\mathcal{G}}|$ is the total number of occurrences of symbols from $\mathcal{N} \cup \{a\}$ on all right-hand sides of $\mathcal{G}$. We have $|\rho_{\mathcal{G}}| \leq |\mathcal{G}| \leq 2 \cdot |\rho_{\mathcal{G}}|$ because the size of $\mathcal{G}$ is $|\rho_{\mathcal{G}}|$ plus the (missing) number of occurrences of f on right-hand sides and if the right-hand side of a rule contains the symbol f, then it contains a symbol from $\mathcal{N} \cup \{a\}$ as well due to the normal form.

It is not hard to show that a given TSLP $\mathcal{G}'$ can be transformed into a normal form TSLP $\mathcal{G}$ of size $|\mathcal{G}| \leq \mathcal{O}(|\mathcal{G}'|)$ that produces the same unlabeled binary tree. Basically, one transforms $\mathcal{G}'$ into a monadic TSLP as described in Theorem 4.2.1 (which is proven in [85, Theorem 10]), followed by some straightforward adjustments in order to obtain the required forms of the right-hand sides. For example, rules $A \to a$ are deleted and A is replaced by a on all right-hand sides, and a rule $A \to f(\alpha_1, \alpha_2)$ ($\alpha_i \in \mathcal{N}_0 \cup \{a\}$ for $i \in [1, 2]$) is modified to $A \to f(x, \alpha_2)$ and occurrences of A on right-hand sides are replaced by $A(\alpha_1)$. Finally, the nonterminals are renamed accordingly in order to satisfy the conditions described above. An example of such a transformation is shown in Example 4.14.

Let $\mathcal{G} = (\mathcal{N}, \mathcal{F}, A_0, P)$ be a normal form TSLP with $\mathcal{N} = \{A_0, A_1, \ldots, A_{n-1}\}$ for the further definitions. Let $\omega_{\mathcal{G}}$ be the word obtained from $\rho_{\mathcal{G}}$ by removing

for every $i \in [1, n-1]$ the first occurrence of A_i from $\rho_{\mathcal{G}}$. Thus, if $\rho_{\mathcal{G}} = A_1 u_1 A_2 u_2 \cdots A_{n-1} u_{n-1}$ with $u_i \in \{a, A_1, A_2, \ldots, A_i\}^*$, then $\omega_{\mathcal{G}} = u_1 u_2 \cdots u_{n-1}$. The *entropy* $H(\mathcal{G})$ of the normal form TSLP $\mathcal{G}$ is defined as the empirical unnormalized entropy (see Section 2.4) of the word $\omega_{\mathcal{G}}$:

$$H(\mathcal{G}) = H(\omega_{\mathcal{G}}).$$

Example 4.14. *Let $\mathcal{G}' = (\mathcal{N}', \mathcal{F}, S, P')$ be a TSLP with $\mathcal{F} = \{a, f\}$, $F_0 = \{a\}$, $\mathcal{F}_2 = \{f\}$, $\mathcal{N}' = \{S, A, B\}$, $\mathcal{N}'_0 = \{S, A\}$, $\mathcal{N}'_1 = \{B\}$ and*

$$P' = \{S \to f(A, B(a)),\ A \to B(B(a)),\ B \to f(x, a)\}.$$

We get $\mathsf{val}_{\mathcal{G}'}(B) = f(x, a)$, $\mathsf{val}_{\mathcal{G}'}(A) = f(f(a, a), a)$ and $\mathsf{val}(\mathcal{G}') = \mathsf{val}_{\mathcal{G}'}(S) = f(f(f(a, a), a), f(a, a))$. This tree is depicted on the left of Figure 4.17.

The corresponding normal form TSLP is $\mathcal{G} = (\mathcal{N}, \mathcal{F}, A_0, P)$ with $\mathcal{N} = \{A_0, A_1, A_2, A_3, A_4\}$, $\mathcal{N}_0 = \{A_0, A_2, A_3\}$, $\mathcal{N}_1 = \{A_1, A_4\}$, where P contains

$$A_0 \to A_1(A_2),\ A_1 \to f(x, A_3),\ A_2 \to A_4(A_3),$$
$$A_3 \to A_4(a),\ A_4 \to f(x, a).$$

We have $\mathsf{val}(\mathcal{G}) = \mathsf{val}(\mathcal{G}')$, $\rho_{\mathcal{G}} = A_1 A_2 A_3 A_4 A_3 A_4 aa$ ($u_1 = u_2 = u_3 = \varepsilon$, $u_4 = A_3 A_4 aa$), $|\rho_{\mathcal{G}}| = 8$, $|\mathcal{G}| = 10$ and $\omega_{\mathcal{G}} = A_3 A_4 aa$.

Recall the definition of the derivation tree of a TSLP in Chomsky normal form (see Section 4.2). Here, we use an adapted version of a derivation tree based on normal form TSLPs: The *normal form derivation tree* $T_{\mathcal{G}}$ of the normal form TSLP $\mathcal{G}$ is a rooted tree, where every node is labeled by a symbol from $\mathcal{N} \cup \{a\}$. We use $T_{\mathcal{G}}$ (instead of $\mathcal{D}_{\mathcal{G}}$) to distinguish the normal form derivation tree we use here from the derivation tree we defined for TSLPs in Chomsky normal form, but we simply call $T_{\mathcal{G}}$ the derivation tree of $\mathcal{G}$. Formally, the root of $T_{\mathcal{G}}$ is labeled by A_0 and nodes labeled by a are the leaves of T_G. A node v that is labeled by a nonterminal A_i has $|\rho(A_i)|$ many children. If $\rho(A_i) = \alpha \in \mathcal{N}_0 \cup \{a\}$ then the single child of v is labeled by α. If $\rho(A_i) = A_j \alpha$ with $\alpha \in \mathcal{N} \cup \{a\}$ then the left (resp., right) child of v is labeled by A_j (resp., α). We denote by $\mathsf{leafsize}(T_{\mathcal{G}})$ the number of leaves of $T_{\mathcal{G}}$ as usual. Note that $\mathsf{leafsize}(T_{\mathcal{G}}) = \mathsf{leafsize}(\mathsf{val}(\mathcal{G}))$.

For every node v of $\mathcal{T}_G$ we define the tree or context $t_v = \mathsf{val}_{\mathcal{G}}(A_i)$ if the label of v is A_i (for some $i \in [0, n-1]$) and $t_v = a$ if v is labeled by a. Note that if the label of v is $\alpha \in \mathcal{N}_0 \cup \{a\}$ then $t_v \in \mathcal{T}$ and if $\alpha \in \mathcal{N}_1$ then $t_v \in \mathcal{C}$. Recall that for a tree or context $t \in \mathcal{T} \cup \mathcal{C}$ and a context $s \in \mathcal{C}$, we denote by $s[t]$ the tree or context which results from s by replacing the parameter x by t. We have $\mathsf{leafsize}(s[t]) = \mathsf{leafsize}(s) + \mathsf{leafsize}(t)$ since the unique occurrence of the parameter x does not count to the leaf size of the context. The tree or context t_v can be inductively defined by the following rules:

- If v is labeled by a then $t_v = a$.

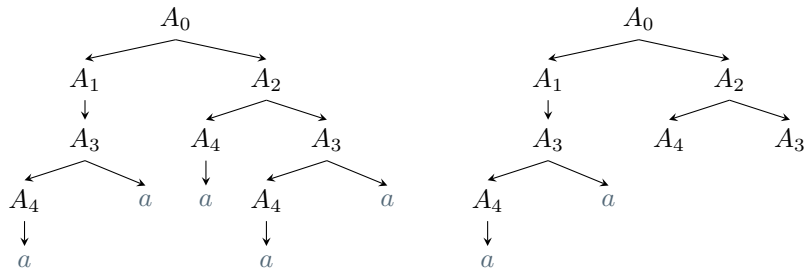

Figure 4.18: The derivation tree $T_{\mathcal{G}}$ of the TSLP from Example 4.14 (left) and an initial subtree T' of $T_{\mathcal{G}}$ (right).

- If v is labeled by $A_i \in \mathcal{N}_1$ and has a single child node u that is labeled by $\alpha \in \mathcal{N}_0 \cup \{a\}$, then $t_v = f(t_u, x)$ if $(A_i \to f(\alpha, x)) \in P$ and $t_v = f(x, t_u)$ if $(A_i \to f(x, \alpha)) \in P$ (note that the right hand side of the rule for A_i must be of the form $f(x, \alpha)$ or $f(\alpha, x)$ since v has exactly one child in $T_{\mathcal{G}}$).
- If v has the left child u_1 and the right child u_2, then $t_v = t_{u_1}[t_{u_2}]$ (note that if a node has two children in the derivation tree, then the left child u_1 is labeled by a nonterminal from $\mathcal{N}_1$ and thus t_{u_1} is a context from $\mathcal{C}$).

An *initial subtree* of the derivation tree $T_{\mathcal{G}}$ is a tree that can be obtained from $T_{\mathcal{G}}$ as follows: Take a subset U of the nodes of $T_{\mathcal{G}}$ and remove from $T_{\mathcal{G}}$ all proper descendants of nodes from U, i.e., all nodes that are located strictly below a node from U.

Example 4.15. *Let $\mathcal{G}$ be the normal form TSLP from Example 4.14. The derivation tree $T_{\mathcal{G}}$ is shown in Figure 4.18 on the left; an initial subtree T' of it is shown on the right.*

Lemma 4.9.1. *Let T' be an initial subtree of $T_{\mathcal{G}}$ and let $v_1, \ldots, v_\ell$ be the sequence of all leaves of T' (in left-to-right order). Then*

$$\mathsf{leafsize}(\mathsf{val}(\mathcal{G})) = \sum_{i=1}^{\ell} \mathsf{leafsize}(t_{v_i}).$$

Proof. Let u be a node of $T_{\mathcal{G}}$ and let T_u be the subtree of $T_{\mathcal{G}}$ rooted in u. Let us first show by induction that $\mathsf{leafsize}(t_u) = \mathsf{leafsize}(T_u)$ holds (where $\mathsf{leafsize}(T_u)$ denotes the number of leaves of T_u as usual). Recall that $\mathsf{leafsize}(t_u)$ is the number of a-labeled leaves of t_u, i.e., a leaf labeled by the parameter x is not counted. If u is a leaf of $T_{\mathcal{G}}$ then we have $t_u = a$ and therefore $\mathsf{leafsize}(t_u) = 1 = \mathsf{leafsize}(T_u)$. If u has two children u_1 and u_2 then $t_u = t_{u_1}[t_{u_2}]$ and we get

$$\mathsf{leafsize}(t_u) = \mathsf{leafsize}(t_{u_1}[t_{u_2}]) = \mathsf{leafsize}(t_{u_1}) + \mathsf{leafsize}(t_{u_2}).$$

Hence, we get by induction

$$\mathsf{leafsize}(t_u) = \mathsf{leafsize}(T_{u_1}) + \mathsf{leafsize}(T_{u_2}) = \mathsf{leafsize}(T_u).$$

Finally, if the node u has a single child v in $T_{\mathcal{G}}$ then either $t_u = f(x, t_v)$ or $t_u = f(t_v, x)$. In both cases we get by induction

$$\mathsf{leafsize}(t_u) = \mathsf{leafsize}(t_v) = \mathsf{leafsize}(T_v) = \mathsf{leafsize}(T_u).$$

This proves $\mathsf{leafsize}(t_u) = \mathsf{leafsize}(T_u)$.

To finish the proof of the lemma, recall that $\mathsf{leafsize}(\mathsf{val}(\mathcal{G})) = \mathsf{leafsize}(T_{\mathcal{G}})$ (by the definition of the derivation tree). Since T' is an initial subtree of $T_{\mathcal{G}}$ we get

$$\mathsf{leafsize}(T_{\mathcal{G}}) = \sum_{i=1}^{\ell} \mathsf{leafsize}(T_{v_i}) = \sum_{i=1}^{\ell} \mathsf{leafsize}(t_{v_i}).$$

□

For a grammar-based tree compressor ψ that produces for a given tree $t \in \mathcal{T}$ a TSLP $\mathcal{G}_t$ in normal form, we define the *compression ratio* of ψ as the mapping $n \mapsto \gamma_\psi(n)$ with

$$\gamma_\psi(n) = \max_{t \in \mathcal{T}_n} \frac{|\mathcal{G}_t|}{n}.$$

Based on Corollary 4.4.4 and Theorem 4.5.3, we know that there are grammar-based compressors which construct a TSLP of size $\mathcal{O}(n/\log n)$ for an unlabeled binary tree of (leaf) size n. Additionally, we explained above that a TSLP can be transformed into normal form such that the size of the TSLP increases only by a constant multiplicative factor. Together this yields the following theorem, which will be crucial in order to derive a universal tree encoder in the sense that the worst-case redundancy converges to zero.

Theorem 4.9.2. *There exists a grammar-based tree compressor ψ such that $\gamma_\psi(n) \leq \mathcal{O}\left(\frac{1}{\log n}\right)$.*

4.9.2 Binary coding of TSLPs in normal form

In this section we introduce a binary encoding for normal form TSLPs. This encoding is similar to the one for SLPs [72] and DAGs [111]. Let $\mathcal{G} = (\mathcal{N}, \mathcal{F}, A_0, P)$ be a TSLP in normal form with $n = |\mathcal{N}|$. Let further $m = |\rho_{\mathcal{G}}|$ and recall that $m = \Theta(|\mathcal{G}|)$. We define the type $\mathsf{type}(A_i) \in \{0, 1, 2, 3\}$ of a nonterminal $A_i \in \mathcal{N}$ as follows:

$$\mathsf{type}(A_i) = \begin{cases} 0 & \text{if } (A_i \to A_j(\alpha)) \in P \text{ for some } A_j \in \mathcal{N}_1 \text{ and } \alpha \in \mathcal{N}_0 \cup \{a\} \\ 1 & \text{if } (A_i \to A_j(A_k(x))) \in P \text{ for some } A_j, A_k \in \mathcal{N}_1 \\ 2 & \text{if } (A_i \to f(\alpha, x)) \in P \text{ for some } \alpha \in V_0 \cup \{a\} \\ 3 & \text{if } (A_i \to f(x, \alpha)) \in P \text{ for some } \alpha \in V_0 \cup \{a\} \end{cases}$$

We define the binary word $B(\mathcal{G}) = w_0 w_1 w_2 w_3 w_4$, where the words $w_i \in \{0,1\}^*$ for $i \in [0, 4]$ are defined as follows:

- $w_0 = 1^{n-1}0$,

- $w_1 = a_0b_0a_1b_1\cdots a_{n-1}b_{n-1}$, where a_jb_j is the 2-bit binary encoding of $\mathsf{type}(A_j)$ for $j \in [0, n-1]$,
- Let $\rho_{\mathcal{G}} = A_1u_1A_2u_2\cdots A_{n-1}u_{n-1}$ with $u_i \in \{a, A_1, A_2, \ldots, A_i\}^*$. Then $w_2 = 1^{|u_1|}01^{|u_2|}0\cdots 1^{|u_{n-1}|}0$, which is ε in case $n = 1$. Note that $|w_2| = m$.
- For $i \in [1, n-1]$ let $k_i \geq 1$ be the number of occurrences of the nonterminal A_i in the word $\rho_{\mathcal{G}}$. Then $w_3 = 1^{k_1-1}01^{k_2-1}0\cdots 1^{k_{n-1}-1}0$, which is ε in case $n = 1$. Note that $|w_3| \leq m$.
- The word w_4 encodes the word $\omega_{\mathcal{G}}$ using the well known enumerative encoding [33]. Every nonterminal A_i ($i \in [1, n-1]$), has $\eta(A_i) := k_i - 1$ occurrences in $\omega_{\mathcal{G}}$. The symbol a has $\eta(a) := m - (k_1 + \cdots + k_{n-1})$ many occurrences in $\omega_{\mathcal{G}}$. Let S be the set of words over the alphabet $\{a, A_1, \ldots, A_{n-1}\}$ with $\eta(a)$ occurrences of a and $\eta(A_i)$ occurrences of A_i for every $i \in [1, n-1]$. Hence,

$$|S| = \frac{(m-n+1)!}{\eta(a)! \cdot \prod_{i=1}^{n-1} \eta(A_i)!}. \tag{4.14}$$

  Let $v_0, v_1, \ldots, v_{|S|-1}$ be the lexicographic enumeration of the words from S with respect to the alphabet order $a, A_1, \ldots, A_{n-1}$. Then w_4 is the binary encoding of the unique index i such that $\omega_{\mathcal{G}} = v_i$, where $|w_4| = \lceil \log |S| \rceil$ (leading zeros are added to the binary encoding of i to obtain the length $\lceil \log |S| \rceil$).

Example 4.16. *Consider the normal form TSLP $\mathcal{G}$ from Example 4.14. We have $w_0 = 11110$, $w_1 = 0011000011$, $w_2 = 00011110$ and $w_3 = 001010$. To compute w_4, note first that there are $|S| = 12$ words with two occurrences of a and one occurrence of A_3 and A_4. It follows that $|w_4| = \lceil \log(12) \rceil = 4$. Further, since the order of the alphabet is a, A_3, A_4, there are only three words in S (A_4A_3aa, A_4aA_3a and A_4aaA_3), which are lexicographically larger than $\omega_{\mathcal{G}} = A_3A_4aa$. Hence, $\omega_{\mathcal{G}} = v_8$ and thus $w_4 = 1000$.*

Lemma 4.9.3. *The set of code words $B(\mathcal{G})$, where $\mathcal{G}$ ranges over all TSLPs in normal form, is a prefix code.*

Proof. Let $B(\mathcal{G}) = w_0w_1w_2w_3w_4$ as defined above. We show how to recover the TSLP $\mathcal{G}$. From $B(\mathcal{G})$ and the fact that $w_0 = 1^{n-1}0$ we can compute $n = |\mathcal{N}|$ and the suffix $w_1w_2w_3w_4$. Since $|w_1| = 2n$ we can then determine w_1 and the suffix $w_2w_3w_4$. The word w_1 encodes the type of every nonterminal. Since $w_2 = 1^{|u_1|}01^{|u_2|}0\cdots 1^{|u_{n-1}|}0$ and we know n we can compute from $w_2w_3w_4$ the word w_2 and the suffix w_3w_4. The word w_2 allows to compute the positions where we have deleted the nonterminals $A_1, A_2, \ldots, A_{n-1}$ (in that order) from $\rho_{\mathcal{G}}$ during the computation of $\omega_{\mathcal{G}}$. Hence, in order to compute $\rho_{\mathcal{G}}$, one only needs $\omega_{\mathcal{G}}$. Since $w_3 = 1^{k_1-1}01^{k_2-1}0\cdots 1^{k_{n-1}-1}0$ and we know n, we can compute w_3 and w_4. The word w_3 determines the frequencies $\eta(a), \eta(A_1), \ldots, \eta(A_{n-1})$ of the symbols in $\omega_{\mathcal{G}}$. Using these frequencies one computes the size $|S|$ from

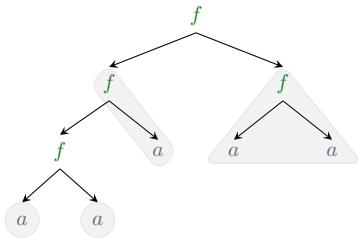

Figure 4.19: The tree $\mathsf{val}(\mathcal{G})$ of the TSLP from Example 4.14. The canonical occurrences of the trees/contexts in $(a, a, \mathsf{val}_{\mathcal{G}}(A_4), \mathsf{val}_{\mathcal{G}}(A_3)) = (a, a, f(x, a), f(a, a))$ used in the proof of Lemma 4.9.5 are highlighted.

(4.14) and hence the length $\lceil \log |S| \rceil$ of w_4. From w_4, one can then compute $\omega_{\mathcal{G}}$. Finally, $\rho_{\mathcal{G}}$ together with the types of the nonterminals (which are encoded by w_1) completely determines $\mathcal{G}$. The argument shows also that $B(\mathcal{G})$ cannot be a prefix of $B(\mathcal{G}')$ for different normal form TSLPs $\mathcal{G}$ and $\mathcal{G}'$. □

Note that $|B(\mathcal{G})| \leq \mathcal{O}(|\mathcal{G}|) + |w_4|$ because $|\rho_{\mathcal{G}}| \leq |G|$ and $|\mathcal{N}| \leq |\mathcal{G}|$. By using the well known bound on the enumerative encoding [34, Theorem 11.1.3], we get:

Lemma 4.9.4. *For the binary coding $B(\mathcal{G})$ we have $|B(\mathcal{G})| \leq \mathcal{O}(|\mathcal{G}|) + H(\mathcal{G})$.*

4.9.3 Universal coding based on TSLPs in normal form

Let $\mathcal{S} = ((\mathcal{P}_i)_{i \in \mathbb{N}}, p_{\mathcal{S}})$ be a tree source as defined in Section 4.7. We say that $\mathcal{S}$ has the *strong domination property* if there exists a mapping $\lambda : \mathcal{T} \cup \mathcal{C} \to \mathbb{R}_{>0}$ with the following properties:

(i) $\lambda(t) \geq p_{\mathcal{S}}(t)$ for every $t \in \mathcal{T}$.

(ii) $\lambda(f(s, t)) \leq \lambda(s) \cdot \lambda(t)$ for all $s, t \in \mathcal{T}$

(iii) $\lambda(s[t]) \leq \lambda(s) \cdot \lambda(t)$ for all $s \in \mathcal{C}$ and $t \in \mathcal{T}$

(iv) There are constants c_1, c_2 such that $\sum_{t \in \mathcal{T}_n \cup \mathcal{C}_n} \lambda(t) \leq c_1 \cdot n^{c_2}$ for all $n \geq 1$.

Recall the definition of the trees/contexts t_u for a node u of a derivation tree of a normal form TSLP from Section 4.9.1.

Lemma 4.9.5. *Let $\lambda : \mathcal{T} \cup \mathcal{C} \to \mathbb{R}_{>0}$ be a mapping that satisfies the properties (ii) and (iii) of the strong domination property. Let $\mathcal{G} = (\mathcal{N}, \mathcal{F}, A_0, P)$ be a TSLP in normal form with $\mathsf{val}(\mathcal{G}) = t$ and let T' be an initial subtree of the derivation tree $T_{\mathcal{G}}$. Let $v_1, \ldots, v_\ell$ be the sequence of leaves of T'. Then $\lambda(t) \leq \prod_{i=1}^{\ell} \lambda(t_{v_i})$.*

Proof. Consider a node u of the derivation tree $T_{\mathcal{G}}$. The tree/context t_u clearly occurs in t. One can define a canonical occurrence of t_u in t which is identified with a set $V_u \subseteq \mathsf{nodes}(t)$ of nodes of t. For the root r, $V_r = \mathsf{nodes}(t)$ is the set

of all nodes of t. Now consider a node u of $T_{\mathcal{G}}$ for which V_u has been defined. If u has two children u_1 and u_2 then $t_u = t_{u_1}[t_{u_2}]$. Then V_{u_1} and V_{u_2} can be uniquely defined by the following conditions: there is a node $y \in V_u$ such that V_{u_2} contains all nodes $z \in V_u$ that are descendants of y in the tree t (including y), $V_{u_1} = V_u \setminus V_{u_2}$ and the nodes in V_{u_i} induce an occurrence of t_i in t for $i \in [1, 2]$. If the node u has a single child v in $T_{\mathcal{G}}$ then either $t_u = f(x, t_v)$ or $t_u = f(t_v, x)$. Let $y \in V_u$ be the root node of V_u. If $t_u = f(t_v, x)$ then V_v contains the nodes in the subtree of t rooted in the left child of y and if $t_u = f(x, t_v)$ then V_v contains the nodes in the subtree of t rooted in the right child of y.

Now consider the nodes $v_1, \ldots, v_\ell$ from the lemma. Figure 4.19 shows the node sets $V_{v_1}, V_{v_2}, V_{v_3}, V_{v_4}$ for the four leaf nodes of the initial subtree T' from Figure 4.18. Since these nodes are pairwise incomparable with respect to the ancestor relation of $T_{\mathcal{G}}$, the node sets V_{v_i} are pairwise disjoint. This allows us to prove $\lambda(t) \leq \prod_{i=1}^{\ell} \lambda(t_{v_i})$ inductively. The case that $t = a$ is clear. Now assume that $t = f(t_1, t_2)$. First assume that the root node of t does not belong to some of the sets V_{v_i}. Then every set V_{v_i} is either contained in the left subtree t_1 or in the right subtree t_2. W.l.o.g. assume that $V_{v_1}, \ldots, V_{v_k}$ are contained in t_1 and $V_{v_{k+1}}, \ldots, V_{v_\ell}$ are contained in t_2. By induction and condition (ii) of the strong domination property we get

$$\lambda(t) \leq \lambda(t_1) \cdot \lambda(t_2) \leq \prod_{i=1}^{k} \lambda(t_{v_i}) \cdot \prod_{i=k+1}^{\ell} \lambda(t_{v_i}).$$

Now assume that the root node of t belongs to a set V_{v_i}. W.l.o.g. assume that $i = 1$. If t_{v_1} is a tree then we must have $t = t_{v_1}$ and $\ell = 1$ and the statement of the lemma holds. Otherwise, t_{v_1} is a context, $t = t_{v_1}[t']$ and all $V_{v_2}, \ldots, V_{v_\ell}$ are contained in t'. By induction and condition (iii) of the strong domination property we get

$$\lambda(t) \leq \lambda(t_{v_1}) \cdot \lambda(t') \leq \lambda(t_{v_1}) \cdot \prod_{i=2}^{\ell} \lambda(t_{v_i}).$$

This concludes the proof of the lemma. □

The proof of the following lemma combines ideas from [72] and [111].

Lemma 4.9.6. *Assume that $\mathcal{S} = ((\mathcal{P}_i)_{i \in \mathbb{N}}, p_{\mathcal{S}})$ has the strong domination property. Let $t \in \mathcal{T}_n$ with $n \geq 2$ and $p_{\mathcal{S}}(t) > 0$, and let $\mathcal{G} = (\mathcal{N}, \mathcal{F}, A_0, P)$ be a TSLP in normal form with $\mathsf{val}(\mathcal{G}) = t$. We have*

$$H(\mathcal{G}) \leq -\log p_{\mathcal{S}}(t) + \mathcal{O}(|\mathcal{G}|) + \mathcal{O}\left(|\mathcal{G}| \cdot \log\left(\frac{n}{|\mathcal{G}|}\right)\right).$$

Proof. Let $m = |\rho_{\mathcal{G}}| \leq |\mathcal{G}|$, $k = |\mathcal{N}|$, and $\ell := m + 1 - k \leq m$. We define an initial subtree T' of the derivation tree $T_{\mathcal{G}}$ as follows: We walk in depth-first left-to-right order over the tree $T_{\mathcal{G}}$. Every time we visit a non-leaf node v that is labeled by a nonterminal that has been seen before during the traversal, we

remove from $T_{\mathcal{G}}$ all proper descendants of v. Thus, for every $A_i \in V$ there is exactly one non-leaf node in T' that is labeled by A_i. For the TSLP from Example 4.14, the tree T' is shown in Figure 4.18 on the right.

Note that T' has exactly $m+1$ many nodes and k non-leaf nodes. Thus, T' has ℓ leaves. Let $v_1, v_2, \ldots, v_\ell$ be the sequence of all leaves of T' (w.l.o.g. in depth-first left-to-right order) and let $\alpha_i \in \{a, A_1, \ldots, A_{k-1}\}$ be the label of v_i. Let $\overline{\alpha} = (\alpha_1, \alpha_2, \ldots, \alpha_\ell)$. Then the number of occurrences of each symbol $\alpha \in \{a, A_1, \ldots, A_{k-1}\}$ in $\omega_{\mathcal{G}}$ is exactly the same as in $\overline{\alpha}$. Hence, $p_{\overline{\alpha}}$ and $p_{\omega_{\mathcal{G}}}$ describe the same empirical distribution (see Section 2.4). For the normal form TSLP from Example 4.14 we get $\overline{\alpha} = (a, a, A_4, A_3)$ (this is the sequence of labels of the leaf nodes for the tree in Figure 4.18 on the right). Let $t_i = \mathsf{val}_{\mathcal{G}}(\alpha_i) \in \mathcal{T} \cup \mathcal{C}$. Since $\mathsf{val}_{\mathcal{G}}(A_i) \neq \mathsf{val}_{\mathcal{G}}(A_j)$ for all $i \neq j$ and $\mathsf{val}_{\mathcal{G}}(A_i) \neq a$ for all i (this holds for every normal form TSLP that produces a tree of size at least two), the tuple $\overline{t} = (t_1, t_2, \ldots, t_\ell)$ satisfies $p_{\omega_{\mathcal{G}}}(\alpha_i) = p_{\overline{t}}(t_i)$ for all $i \in [1, \ell]$. For the TSLP from Example 4.14 we get $\overline{t} = (a, a, \mathsf{val}_{\mathcal{G}}(A_4), \mathsf{val}_{\mathcal{G}}(A_3)) = (a, a, f(x, a), f(a, a))$ (see Figure 4.19).

Let $n_i = \mathsf{leafsize}(t_i)$. Lemma 4.9.1 yields

$$\sum_{i=1}^{\ell} n_i = n \tag{4.15}$$

since $t \in \mathcal{T}_n$, whereas Lemma 4.9.5 yields

$$\lambda(t) \leq \prod_{i=1}^{\ell} \lambda(t_i). \tag{4.16}$$

For $j \in \mathbb{N}$ with $j \geq 1$ let

$$M_j = \sum_{u \in \mathcal{T}_j \cup \mathcal{C}_j} \lambda(u) \leq c_1 \cdot j^{c_2},$$

where c_1 and c_2 are the constants from condition (iv) of the strong domination property. Let $D := 6/\pi^2 \geq 1/2$ and define for every $u \in \mathcal{T}_j \cup \mathcal{C}_j$:

$$q(u) := \frac{D \cdot \lambda(u)}{M_j \cdot j^2} > 0. \tag{4.17}$$

We get

$$\sum_{j \geq 1} \sum_{u \in \mathcal{T}_j \cup \mathcal{C}_j} q(u) = D \cdot \sum_{j \geq 1} \frac{1}{j^2} = 1.$$

Hence, we have $q(t_1) + q(t_2) + \cdots + q(t_\ell) \leq 1$. Using Shannon's inequality (2.3) we get

$$H(\mathcal{G}) = H(\omega_{\mathcal{G}}) = \sum_{i=1}^{\ell} -\log p_{\omega_{\mathcal{G}}}(\alpha_i) = \sum_{i=1}^{\ell} -\log p_{\overline{t}}(t_i) \leq \sum_{i=1}^{\ell} -\log q(t_i).$$

Using (4.17) and $D \geq 1/2$ we obtain

$$\begin{aligned} H(\mathcal{G}) &\leq \sum_{i=1}^{\ell} -\log\left(\frac{D \cdot \lambda(t_i)}{M_{n_i} \cdot n_i^2}\right) \\ &= -\ell \cdot \log D - \sum_{i=1}^{\ell} \log \lambda(t_i) + \sum_{i=1}^{\ell} \log M_{n_i} + 2\sum_{i=1}^{\ell} \log n_i \\ &\leq \ell - \sum_{i=1}^{\ell} \log \lambda(t_i) + \sum_{i=1}^{\ell} (\log c_1 + c_2 \log n_i) + 2\sum_{i=1}^{\ell} \log n_i \\ &= (1 + \log c_1) \cdot \ell - \sum_{i=1}^{\ell} \log \lambda(t_i) + (2 + c_2) \cdot \sum_{i=1}^{\ell} \log n_i. \end{aligned}$$

From (4.16) and condition (i) of the strong domination property we get

$$\sum_{i=1}^{\ell} \log \lambda(t_i) \geq \log \lambda(t) \geq \log p_{\mathcal{S}}(t).$$

Moreover, Jensen's inequality and (4.15) gives

$$\sum_{i=1}^{\ell} \log n_i \leq \ell \cdot \log\left(\frac{1}{\ell} \cdot \sum_{i=1}^{\ell} n_i\right) = \ell \cdot \log(n/\ell).$$

With $\ell \leq m$ we obtain

$$\begin{aligned} H(\mathcal{G}) &\leq (1 + \log c_1) \cdot \ell - \log p_{\mathcal{S}}(t) + (2 + c_2) \cdot \ell \cdot \log(n/\ell) \\ &\leq -\log p_{\mathcal{S}}(t) + (1 + \log c_1) \cdot m + (2 + c_2) \cdot m \cdot \log(n/m). \end{aligned}$$

This shows the lemma. □

Let ψ be a grammar-based tree compressor which produces for an input tree $t \in \mathcal{T}$ the TSLP $\mathcal{G}_t$. We then consider the tree encoder $E_\psi : \mathcal{T} \to \{0,1\}^*$ defined by $E_\psi(t) = B(\mathcal{G}_t)$. Recall the definition of the worst-case redundancy $R(E_\psi, \mathcal{S}, n)$ from Section 4.7 and the compression ratio γ_ψ from Section 4.9.1.

Theorem 4.9.7. *Assume that $\mathcal{S} = ((\mathcal{P}_i)_{i\in\mathbb{N}}, p_{\mathcal{S}})$ has the strong domination property. Let ψ be a grammar-based compressor such that $\gamma_\psi(n) \leq \gamma(n)$ for a monotonically decreasing function $\gamma(n)$ with $\lim_{n\to\infty} \gamma(n) = 0$. Let further $m_i = \min\{\mathsf{leafsize}(t) \mid t \in \mathcal{P}_i\}$ and assume that $m_i < m_{i+1}$ for all $i \in \mathbb{N}$.[9] Then, we have*

$$R(E_\psi, \mathcal{S}, n) \leq \mathcal{O}\left(\gamma(m_n) \cdot \log\left(\frac{1}{\gamma(m_n)}\right)\right).$$

[9] This is the case for leaf-centric and depth-centric tree sources.

Proof. Let $t \in \mathcal{T}$ such that $\ell = \mathsf{leafsize}(t)$ and $\psi(t) = \mathcal{G}_t$. With Lemma 4.9.4 and 4.9.6 we get

$$\begin{aligned}\frac{1}{\ell} \cdot (|B(\mathcal{G}_t)| + \log p_{\mathcal{S}}(t)) &\leq \frac{1}{\ell} \cdot (H(\mathcal{G}_t) + \mathcal{O}(|\mathcal{G}_t|) + \log p_{\mathcal{S}}(t)) \\ &\leq \frac{1}{\ell} \cdot (\mathcal{O}(|\mathcal{G}_t|) + \mathcal{O}\left(|\mathcal{G}_t| \cdot \log\left(\frac{\ell}{|\mathcal{G}_t|}\right)\right) \\ &= \mathcal{O}\left(\frac{|\mathcal{G}_t|}{\ell}\right) + \mathcal{O}\left(\frac{|\mathcal{G}_t|}{\ell} \cdot \log\left(\frac{\ell}{|\mathcal{G}_t|}\right)\right).\end{aligned}$$

Consider the mapping g with $g(x) = x \cdot \log(1/x)$. It is monotonically increasing for $0 \leq x \leq 1/e$. If n is large enough, we have for all $t \in \mathcal{P}_n$ with $\ell = \mathsf{leafsize}(t)$ that

$$\frac{|\mathcal{G}_t|}{\ell} \leq \gamma_\psi(\ell) \leq \gamma(\ell) \leq \gamma(m_n) \leq \frac{1}{e}.$$

Hence, we get

$$\frac{|\mathcal{G}_t|}{\ell} \cdot \log\left(\frac{\ell}{|\mathcal{G}_t|}\right) = g\left(\frac{|\mathcal{G}_t|}{\ell}\right) \leq g(\gamma(m_n)) = \gamma(m_n) \cdot \log\left(\frac{1}{\gamma(m_n)}\right).$$

This implies

$$\begin{aligned}R(E_\psi, \mathcal{S}, i) &= \max_{t \in \mathcal{P}_n, p_{\mathcal{S}}(t) > 0} \frac{1}{\mathsf{leafsize}(t)} \cdot (|B(\mathcal{G}_t)| + \log p_{\mathcal{S}}(t)) \\ &\leq \mathcal{O}(\gamma(m_n)) + \mathcal{O}\left(\gamma(m_n) \cdot \log\left(\frac{1}{\gamma(m_n)}\right)\right) \\ &= \mathcal{O}\left(\gamma(m_n) \cdot \log\left(\frac{1}{\gamma(m_n)}\right)\right),\end{aligned}$$

which proves the theorem. □

Note that the minimal leaf size of a tree in $\mathcal{T}_{n+1}$ (resp. $\mathcal{T}^n$) is $n+1$. Hence, Theorem 4.9.2 and Theorem 4.9.7 yield:

Corollary 4.9.8. *There exists a grammar-based tree compressor ψ such that for every leaf-centric or depth-centric tree source $\mathcal{S}$ which satisfies the strong domination property, we have $R(E_\psi, \mathcal{S}, n) \leq \mathcal{O}\left(\frac{\log\log n}{\log n}\right)$.*

In the following, we will present classes of leaf-centric and depth-centric tree sources that have the strong domination property.

4.9.4 Leaf-centric binary tree sources

Recall the definition of the class of mappings Σ_{leaf} by equations (4.8), (4.9), and (4.10) in Section 4.8.1 and the corresponding class of leaf-centric tree sources. In this section, we state a condition on the mapping $\sigma \in \Sigma_{\text{leaf}}$ that enforces the strong domination property for the leaf-centric tree source $((\mathcal{T}_i)_{i \geq 1}, p_\sigma)$. This allows to apply Corollary 4.9.8.

Theorem 4.9.9. *If $\sigma \in \Sigma_{leaf}$ satisfies*

$$\sigma(i,j) \geq \sigma(i,j+1) \text{ and } \sigma(i,j) \geq \sigma(i+1,j) \tag{4.18}$$

for all $i,j \geq 1$, then $((\mathcal{T}_i)_{i\geq 1}, p_\sigma)$ has the strong domination property.

Proof. First, we naturally extend p_σ to a context $t \in \mathcal{C}$ using equations (4.9) and (4.10), where we set $\sigma(0,k) = \sigma(k,0) = 1$ for all $k \geq 1$ and $p_\sigma(x) = 1$. Note that $\mathsf{leafsize}(x) = 0$. Also note that p_σ is not a probability distribution on $\mathcal{C}_n$. For instance, we have $\sum_{t\in\mathcal{C}_1} p_\sigma(t) = p_\sigma(f(x,a)) + p_\sigma(f(a,x)) = 2$. Let $B_n = |\mathcal{T}_n| = c_{n-1}$ (the $(n-1)$-th Catalan number) for $n \geq 1$. Note that we have $B_{m+k} \geq B_m \cdot B_k$ for all $m,k \geq 0$ since this inequality is well known for the Catalan numbers. We set $B_0 = 1$ and define $\lambda : \mathcal{T} \cup \mathcal{C} \to \mathbb{R}_{>0}$ by

$$\lambda(t) = \max\left\{ \frac{1}{B_{\mathsf{leafsize}(t)}}, p_\sigma(t) \right\}.$$

In the following, we show the four points from the strong domination property for the mapping λ. The first point of the strong domination property, i.e., $\lambda(t) \geq p_\sigma(t)$ for all $t \in \mathcal{T}$, is obviously true. We now prove the second point, i.e., $\lambda(f(s,t)) \leq \lambda(s) \cdot \lambda(t)$ for all $s,t \in \mathcal{T}$. Let $\mathsf{leafsize}(s) = m$ and $\mathsf{leafsize}(t) = k$ and assume first that $\lambda(f(s,t)) = 1/B_{m+k}$. The inequality $B_{m+k} \geq B_m \cdot B_k$ yields

$$\frac{1}{B_{m+k}} \leq \frac{1}{B_m} \cdot \frac{1}{B_k} \leq \lambda(s) \cdot \lambda(t). \tag{4.19}$$

On the other hand, if $\lambda(f(s,t)) = p_\sigma(f(s,t))$, then we have

$$p_\sigma(f(s,t)) = \sigma(m,k) \cdot p_\sigma(s) \cdot p_\sigma(t) \leq p_\sigma(s) \cdot p_\sigma(t) \leq \lambda(s) \cdot \lambda(t),$$

since $0 \leq \sigma(i,j) \leq 1$ for all i,j.

We now consider the third point, i.e., $\lambda(s[t]) \leq \lambda(s) \cdot \lambda(t)$ for all $s \in \mathcal{C}$ and $t \in \mathcal{T}$. Note first that the case $\lambda(s[t]) = 1/B_{\mathsf{leafsize}(s[t])}$ follows again from equation (4.19), since $\mathsf{leafsize}(s[t]) = \mathsf{leafsize}(s) + \mathsf{leafsize}(t)$. So we assume that $\lambda(s[t]) = p_\sigma(s[t])$. Let ℓ be the depth of the unique x-labeled node in s, i.e., the length of the path (measured in the number of edges) from the root of s to the parameter node in s. We show $p_\sigma(s[t]) \leq p_\sigma(s) \cdot p_\sigma(t)$ by induction over $\ell \geq 0$. If $\ell = 0$ then $s = x$ and we get $p_\sigma(s[t]) = p_\sigma(t) = p_\sigma(x) \cdot p_\sigma(t) = p_\sigma(s) \cdot p_\sigma(t)$. Let us now assume that $\ell \geq 1$. Then, s must have the form $s = f(u,v)$. Without loss of generality assume that x occurs in u; the other case is of course symmetric. Therefore, we have $s[t] = f(u[t],v)$. The tree $u[t]$ fulfills the induction hypothesis and therefore $p_\sigma(u[t]) \leq p_\sigma(u) \cdot p_\sigma(t)$. Moreover, we have

$$\begin{aligned} \sigma(\mathsf{leafsize}(u[t]), \mathsf{leafsize}(v)) &= \sigma(\mathsf{leafsize}(u) + \mathsf{leafsize}(t), \mathsf{leafsize}(v)) \\ &\leq \sigma(\mathsf{leafsize}(u), \mathsf{leafsize}(v)) \end{aligned}$$

due to $\sigma(i,j) \geq \sigma(i+1,j)$ for all $i,j \geq 1$. We get

$$\begin{aligned} p_\sigma(s[t]) = p_\sigma(f(u[t],v)) &= p_\sigma(u[t]) \cdot \sigma(\mathsf{leafsize}(u[t]), \mathsf{leafsize}(v)) \cdot p_\sigma(v) \\ &\leq p_\sigma(t) \cdot p_\sigma(u) \cdot \sigma(\mathsf{leafsize}(u), \mathsf{leafsize}(v)) \cdot p_\sigma(v) \\ &= p_\sigma(t) \cdot p_\sigma(s). \end{aligned}$$

For the fourth property we show $\sum_{t\in\mathcal{T}_n\cup\mathcal{C}_n}\lambda(t)\le 8n-2$ for all $n\ge 1$. First, we have

$$\sum_{t\in\mathcal{T}_n}\lambda(t)\le\sum_{t\in\mathcal{T}_n}\left(B_n^{-1}+p_\sigma(t)\right)=2, \tag{4.20}$$

because p_σ as well as $p=1/B_n$ are probability distributions on $\mathcal{T}_n$ (the latter one is the uniform distribution). We show that $\sum_{t\in\mathcal{C}_n}\lambda(t)\le 8n-4$. Every tree $t\in\mathcal{T}_n$ has $2n-1$ nodes. Let v be a node of t and recall that $\mathsf{subtree}_t(v)$ is the subtree of t rooted at v. We obtain two contexts from t and v by replacing $\mathsf{subtree}_t(v)$ in t by either $f(x,\mathsf{subtree}_t(v))$ or $f(\mathsf{subtree}_t(v),x)$. Let us denote the resulting contexts by $t_{v,1}$ and $t_{v,2}$. We have $\lambda(t_{v,1})=\lambda(t_{v,2})=\lambda(t)$ for every node v of t since $\mathsf{leafsize}(x)=0$ and $\sigma(0,k)=\sigma(k,0)=1$ for all $k\ge 1$. Moreover, for every context $t\in\mathcal{C}_n$ there exists a tree $t'\in\mathcal{T}_n$ and a node $v\in\mathsf{nodes}(t')$ such that $t'_{v,1}=t$ or $t'_{v,2}=t$ (depending on whether x is the left or right child of its parent node in t). Since $|\mathsf{nodes}(t')|=2n-1$ for $t'\in\mathcal{T}_n$, we get

$$\sum_{t\in\mathcal{C}_n}\lambda(t)\le(4n-2)\cdot\sum_{t\in\mathcal{T}_n}\lambda(t). \tag{4.21}$$

Together with equation (4.20) we have $\sum_{t\in\mathcal{T}_n\cup\mathcal{C}_n}\lambda(t)\le 8n-2$. □

Example 4.17. *Let us come back to the three leaf-centric tree sources from Example 4.13. The mappings σ_{bst} and σ_{uni} satisfy the condition from (4.18). Hence, the grammar-based compressor E_ψ achieves a worst-case redundancy of $\mathcal{O}(\log\log n/\log n)$ for the binary search tree model and the uniform model by Corollary 4.9.8.*

4.9.5 Depth-centric binary tree sources

Recall the definition of the class of mappings Σ_{depth} by equations (4.11), (4.12), and (4.13) in Section 4.8.2, and the corresponding class of depth-centric tree sources. In this section, we state a condition on the mapping $\sigma\in\Sigma_{\mathrm{depth}}$ that enforces the strong domination property for the depth-centric tree source $((\mathcal{T}i)_{i\ge1},p_\sigma)$. This allows again to apply Corollary 4.9.8.

Theorem 4.9.10. *If $\sigma\in\Sigma_{depth}$ satisfies*

$$\sigma(i,j)\ge\sigma(i,j+1)\ \text{ and }\ \sigma(i,j)\ge\sigma(i+1,j)$$

for all $i,j\ge0$, then $((\mathcal{T}^i)_{i\ge0},p_\sigma)$ has the strong domination property.

Proof. Recall that the depth of a context $t\in\mathcal{C}$ is defined as the depth of the tree $t[a]$. Using this information, we extend p_σ to a context $t\in\mathcal{C}$ using equations (4.12) and (4.13), where we set $p_\sigma(x)=1$. Similarly to the proof of Theorem 4.9.9 for leaf-centric tree sources, we define

$$\lambda(t)=\max\left\{\frac{1}{B_{\mathsf{leafsize}(t)}},p_\sigma(t)\right\}.$$

The first point of the strong domination property, i.e., $\lambda(t) \geq p_\sigma(t)$ for all $t \in \mathcal{T}$, follows directly from the definition of λ. Now we prove the second point, i.e., $\lambda(f(s,t)) \leq \lambda(s) \cdot \lambda(t)$ for all $s, t \in \mathcal{T}$. Let $\mathsf{leafsize}(s) = m$ and $\mathsf{leafsize}(t) = k$ and assume first that $\lambda(f(s,t)) = 1/B_{m+k}$. This case is covered by equation (4.19). If otherwise $\lambda(f(s,t)) = p_\sigma(f(s,t))$, then

$$p_\sigma(f(s,t)) = \sigma(d(s), d(t)) \cdot p_\sigma(s) \cdot p_\sigma(t) \leq p_\sigma(s) \cdot p_\sigma(t) \leq \lambda(s) \cdot \lambda(t),$$

since $0 \leq \sigma(i,j) \leq 1$ for all i, j.

Consider now the third point, i.e., $\lambda(s[t]) \leq \lambda(s) \cdot \lambda(t)$ for all $s \in \mathcal{C}$ and $t \in \mathcal{T}$. Note that the case $\lambda(s[t]) = 1/B_{\mathsf{leafsize}(s[t])}$ follows again from equation (4.19). Hence, we can assume that $\lambda(s[t]) = p_\sigma(s[t])$. Similarly to the proof of Theorem 4.9.9, we prove $p_\sigma(s[t]) \leq p_\sigma(s) \cdot p_\sigma(t)$ by induction over the depth $\ell \geq 0$ of the unique x-labeled node in s. If $\ell = 0$ then $s = x$ and $p_\sigma(s) = 1$, which gives us $p_\sigma(s[t]) = p_\sigma(t) = p_\sigma(s) \cdot p_\sigma(t)$. We now assume $\ell \geq 1$ and $s = f(u,v)$. Without loss of generality assume that x occurs in u; the other case is symmetric. Therefore, we have $s[t] = f(u[t], v)$. We apply the induction hypothesis to the tree $u[t]$, which yields $p_\sigma(u[t]) \leq p_\sigma(u) \cdot p_\sigma(t)$. Moreover, since $d(u) \leq d(u[t])$ we have $\sigma(d(u[t]), d(v)) \leq \sigma(d(u), d(v))$. It follows that

$$\begin{aligned} p_\sigma(s[t]) = p_\sigma(f(u[t], v)) &= p_\sigma(u[t]) \cdot \sigma(d(u[t]), d(v)) \cdot p_\sigma(v) \\ &\leq p_\sigma(t) \cdot p_\sigma(u) \cdot \sigma(d(u), d(v)) \cdot p_\sigma(v) \\ &= p_\sigma(t) \cdot p_\sigma(s). \end{aligned}$$

For the fourth property we show $\sum_{t \in \mathcal{T}_n \cup \mathcal{C}_n} \lambda(t) \leq 4n^2 + 3n - 1$ for all $n \geq 1$. First, we have

$$\sum_{t \in \mathcal{T}_n} \lambda(t) \leq \sum_{t \in \mathcal{T}_n} \left(B_n^{-1} + p_\sigma(t)\right) = 1 + \sum_{t \in \mathcal{T}_n} p_\sigma(t). \tag{4.22}$$

Note that p_σ is not a probability distribution on $\mathcal{T}_n$ since this section deals with depth-centric tree sources. But for each $t \in \mathcal{T}_n$ we have $d(t) \in [\lceil \log(n) \rceil, n-1]$, which yields

$$\sum_{t \in \mathcal{T}_n} p_\sigma(t) \leq \sum_{i=\lceil \log(n) \rceil}^{n-1} \sum_{t \in \mathcal{T}^i} p_\sigma(t) = n - \lceil \log(n) \rceil \leq n.$$

Together with equation (4.22) we get $\sum_{t \in \mathcal{T}_n} \lambda(t) \leq n + 1$. The remaining part $\sum_{t \in \mathcal{C}_n} \lambda(t)$ can be estimated with help of equation (4.21) from the corresponding part in the proof of Theorem 4.9.9. In total, we have

$$\sum_{t \in \mathcal{T}_n \cup \mathcal{C}_n} \lambda(t) \leq (4n-1) \sum_{t \in \mathcal{T}_n} \lambda(t) \leq (4n-1)(n+1) = 4n^2 + 3n - 1.$$

□

4.10 Conclusion and open problems

Conclusion. The main result of this chapter is the construction of a TSLP of size $\mathcal{O}(n/\log_\sigma n)$ for an input tree t of size n with $|\mathsf{labels}(t)| = \sigma$, where we assume that the maximal rank of the symbols in $\mathsf{labels}(t)$ is bounded by a constant. The grammar-based tree compressor TreeBiSection (Section 4.4) produces a TSLP of the claimed size in time $\mathcal{O}(n \cdot \log n)$ and can be implemented in logspace. Additionally, the obtained TSLP has depth $\mathcal{O}(\log n)$. The grammar-based tree compressor BU-Shrink (Section 4.5) produces a TSLP of the claimed size in linear time, but the depth of the TSLP is not necessarily logarithmic. A TSLP of size $\mathcal{O}(n/\log_\sigma n)$ and depth $\mathcal{O}(\log n)$ can be obtained in linear time by using a recently introduced balancing technique [48] for the TSLP obtained by BU-Shrink. Alternatively, one can use a combination of BU-Shrink and TreeBiSection as described at the end of Section 4.5.

The construction and analysis of the grammar-based tree compressor TreeBiSection is strongly related to DAG compression. A key result of this chapter is that the DAG of certain weakly balanced binary trees has size $\mathcal{O}(n/\log_\sigma n)$ (Section 4.3.1), where again n is the size of the input tree and σ is the number of different labels occurring in the tree. This result is used (i) to prove the claimed size of the TSLP produced by TreeBiSection and (ii) to sharpen some of the results presented in [111], where universal source coding of unlabeled binary trees based on the minimal DAG is investigated (Section 4.8).

A first application of constructing a TSLP of size $\mathcal{O}(n/\log_\sigma n)$ and depth $\mathcal{O}(\log n)$ refines a contribution of Brent [21] in the area of arithmetical circuits, which states that every arithmetical formula of size n over a commutative ring can be transformed into an equivalent circuit of depth $\mathcal{O}(\log n)$ and size $\mathcal{O}(n)$. Based on our constructions, we reduce the circuit size to $\mathcal{O}((n \cdot \log m)/\log n)$, where m is the number of different variables in the formula (see Section 4.6). As a second application, we use grammar-based tree compression for universal source coding (Section 4.9). We present an encoding of unlabeled binary trees based on TSLPs such that a worst-case universal code is achieved for certain tree sources.

Open problems. It would be interesting to know the worst-case size of the TSLP produced by the grammar-based tree compressor from [66]. It works in linear time and produces a TSLP of size $\mathcal{O}(rg\log(n/rg))$ for a tree of size n and maximal rank r, where g is the size of a smallest TSLP. Since this function is monotonically increasing with g and $g \leq \mathcal{O}(n/\log_\sigma n)$ for trees of constant rank, the algorithm in [66] yields for a tree of constant rank a TSLP of size $\mathcal{O}((n\log\log_\sigma n)/\log_\sigma n)$. It remains open, whether the additional factor $\log\log_\sigma n$ is necessary. Vice versa, we plan to investigate the approximation ratio of the grammar-based tree compressors BU-Shrink and TreeBiSection, but it seems unlikely that those compressors construct TSLPs of size $\mathcal{O}(rg\log(n/rg))$ as the grammar-based tree compressor in [66]. For TreeBiSection, it seems possible that ideas used in Section 3.3 are applicable, where we proved that the grammar-based string compressor BiSection has approximation ratio $\Omega(\sqrt{n/\log n})$.

In [15] the authors proved that the so-called top dag of a given tree t of size n is at most by a factor $\log n$ larger than the minimal DAG of t. It is not clear, whether the TSLP constructed by TreeBiSection has this property. The construction of the top dag is done in a bottom-up way, and as a consequence identical subtrees are compressed in the same way. This property is crucial for the comparison with the DAG. TreeBiSection works in a top-down way. Hence, it is not guaranteed that identical subtrees are compressed in the same way. In contrast, BU-Shrink works bottom-up (although in a somehow different way as for the DAG) and thus it might be easier to compare the size of the TSLP produced by BU-Shrink with the size of the minimal DAG. Anyway, since a DAG can be seen as a TSLP, one could produce the TSLP of TreeBiSection (respectively, BU-shrink) and the TSLP which corresponds to the DAG in parallel and simply take the smaller one as the output. Note that in general this would not yield logarithmic depth of the produced TSLP but the balancing technique presented in [48] can be used to obtain depth $\mathcal{O}(\log n)$.

In the context of tree source coding, it would be nice to show the strong domination property also for other classes of tree sources. An interesting class are the tree sources derived from stochastic context-free grammars [90]. Another interesting question is, whether the convergence rate of $\mathcal{O}(\log\log n/\log n)$ in Corollary 4.9.8 can be improved to $\mathcal{O}(1/\log n)$. In the context of grammar-based string compression, such an improvement has been accomplished in [73]. Moreover, we would like to extend the encoding presented in Section 4.9.2 beyond unlabeled binary trees. A first step in this direction was taken in the recent paper [60], where our encoding is extended to labeled binary trees. To be more precise, the extended encoding is based on TSLPs for binary trees over an (unranked) alphabet of terminal symbols, where any node of the binary tree can be labeled by any symbol of the alphabet (inner nodes and leaf nodes share the same set of labels). Note that it is straightforward to use TSLPs for this tree model as well since the rank of the terminal symbols is not mandatory in order to define TSLPs. It is shown in [60] that the size of this extended encoding is bounded by the k-th order empirical entropy of the binary tree plus some low order terms. It would be interesting to extend this encoding even further by considering FSLPs [50] for unranked trees instead of TSLPs for binary trees. Using a small detour, one can use the encoding from [60] for unranked trees by first computing the well known first-child next-sibling encoding [19, 76] of the unranked tree and then encode the obtained binary tree. It remains open whether those codes for unranked trees achieve strong theoretical results in the context of entropy bounds and universal coding.

More generally, it was recently shown that many compression methods that exploit the repetitiveness of a text such as LZ77, grammar-based compression or the run-length Burrows-Wheeler transform can be generalized to a common formalism known as string attractors [68]. A string attractor for a string of length n is a subset of the positions $[1, n]$ such that every distinct substring of the string has an occurrence crossing one of the elements of the attractor. It is an open challenge to extend string attractors to trees.

Bibliography

[1] Janos Aczél. On Shannon's inequality, optimal coding, and characterizations of Shannon's and Renyi's entropies. Technical Report AA-73-05, University of Waterloo, 1973. https://cs.uwaterloo.ca/research/tr/1973/CS-73-05.pdf.

[2] Manindra Agrawal and Ramprasad Saptharishi. Classifying polynomials and identity testing. *Current Trends in Science – Platinum Jubilee Special*, pages 149–162, 2009.

[3] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley series in computer science / World student series edition. Addison-Wesley, 1986.

[4] Alfred V. Aho and Neil J. A. Sloane. Some doubly exponential sequences. *Fibonacci Quarterly*, 11(4):429–437, 1973.

[5] Tatsuya Akutsu. A bisection algorithm for grammar-based compression of ordered trees. *Information Processing Letters*, 110(18-19):815–820, 2010.

[6] Ingo Althöfer. Tight lower bounds on the length of word chains. *Information Processing Letters*, 34(5):275–276, 1990.

[7] Alberto Apostolico and Stefano Lonardi. Some theory and practice of greedy off-line textual substitution. In *Data Compression Conference, DCC 1998*, pages 119–128, 1998.

[8] Alberto Apostolico and Stefano Lonardi. Compression of biological sequences by greedy off-line textual substitution. In *Data Compression Conference, DCC 2000*, pages 143–152, 2000.

[9] Sanjeev Arora and Boaz Barak. *Computational Complexity - A Modern Approach*. Cambridge University Press, 2009.

[10] Jan Arpe and Rüdiger Reischuk. On the complexity of optimal grammar-based compression. In *Data Compression Conference, DCC 2006*, pages 173–182, 2006.

[11] Hideo Bannai, Momoko Hirayama, Danny Hucke, Shunsuke Inenaga, Artur Jeż, Markus Lohrey, and Carl P. Reh. The smallest grammar problem revisited. Technical report, arXiv.org, 2019. http://arxiv.org/abs/1908.06428.

[12] Djamal Belazzougui, Patrick H. Cording, Simon J. Puglisi, and Yasuo Tabei. Access, rank, and select in grammar-compressed strings. In *Algorithms - ESA 2015 - 23rd Annual European Symposium*, pages 142–154, 2015.

[13] Jean Berstel and Srecko Brlek. On the length of word chains. *Information Processing Letters*, 26(1):23–28, 1987.

[14] Philip Bille, Finn Fernstrøm, and Inge Li Gørtz. Tight bounds for top tree compression. In *International Symposium on String Processing and Information Retrieval, SPIRE 2017*, pages 97–102, 2017.

[15] Philip Bille, Inge Li Gørtz, Gad M. Landau, and Oren Weimann. Tree compression with top trees. *Information and Computation*, 243:166–177, 2015.

[16] Philip Bille, Inge Li Gørtz, and Nicola Prezza. Space-efficient Re-Pair compression. In *Data Compression Conference, DCC 2017*, pages 171–180, 2017.

[17] Philip Bille, Gad M. Landau, Rajeev Raman, Kunihiko Sadakane, Srinivasa Rao Satti, and Oren Weimann. Random access to grammar-compressed strings and trees. *SIAM Journal on Computing*, 44(3):513–539, 2015.

[18] Maria L. Bonet and Samuel R. Buss. Size-depth tradeoffs for boolean fomulae. *Information Processing Letters*, 49(3):151–155, 1994.

[19] Mireille Bousquet-Mélou, Markus Lohrey, Sebastian Maneth, and Eric Noeth. XML compression via DAGs. *Theory of Computing Systems*, 57(4):1322–1371, 2015.

[20] Peter Brass. *Advanced Data Structures*. Cambridge University Press, 2008.

[21] Richard P. Brent. The parallel evaluation of general arithmetic expressions. *Journal of the ACM*, 21(2):201–206, 1974.

[22] Randal E. Bryant. Symbolic boolean manipulation with ordered binary-decision diagrams. *ACM Computing Surveys*, 24(3):293–318, 1992.

[23] Nader H. Bshouty, Richard Cleve, and Wayne Eberly. Size-depth tradeoffs for algebraic formulas. *SIAM Journal on Computing*, 24(4):682–705, 1995.

[24] Peter Buneman, Martin Grohe, and Christoph Koch. Path queries on compressed XML. In *International Conference on Very Large Data Bases, VLDB 2003*, pages 141–152, 2003.

[25] Michael Burrows and David Wheeler. A block-sorting lossless data compression algorithm. Technical report, DIGITAL SRC Research Report, 1994. https://www.hpl.hp.com/techreports/Compaq-DEC/SRC-RR-124.pdf.

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

[27] Jiazhen Cai and Robert Paige. Using multiset discrimination to solve language processing problems without hashing. *Theoretical Computer Science*, 145(1-2):189–228, 1995.

[28] Katrin Casel, Henning Fernau, Serge Gaspers, Benjamin Gras, and Markus L. Schmid. On the complexity of grammar-based compression over fixed alphabets. In *International Colloquium on Automata, Languages, and Programming, ICALP 2016*, pages 122:1–122:14, 2016.

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

[30] Yongwook Choi and Wojciech Szpankowski. Compression of graphical structures: Fundamental limits, algorithms, and experiments. *IEEE Transactions on Information Theory*, 58(2):620–638, 2012.

[31] Francisco Claude and Gonzalo Navarro. Fast and compact web graph representations. *ACM Transactions on the Web*, 4(4):16:1–16:31, 2010.

[32] Hubert Comon, Max Dauchet, Remi Gilleron, Florent Jacquemard, Denis Lugiez, Christof Löding, Sophie Tison, and Marc Tommasi. Tree automata techniques and applications, 2007. http://tata.gforge.inria.fr/.

[33] Thomas M. Cover. Enumerative source encoding. *IEEE Transactions on Information Theory*, 19(1):73–77, 1973.

[34] Thomas M. Cover and Joy A. Thomas. *Elements of Information Theory*. Wiley, 2nd edition, 2006.

[35] Maxime Crochemore, Gad M. Landau, and Michal Ziv-Ukelson. A subquadratic sequence alignment algorithm for unrestricted scoring matrices. *SIAM Journal on Computing*, 32(6):1654–1673, 2003.

[36] Nicolaas G. de Bruijn. A combinatorial problem. *Koninklijke Nederlandse Akademie van Wetenschappen: Section of Sciences*, 49(7):758–764, 1946.

[37] Ajit A. Diwan. A new combinatorial complexity measure for languages. *Tata Institute, Bombay, India*, 1986.

[38] Peter J. Downey, Ravi Sethi, and Robert E. Tarjan. Variations on the common subexpression problem. *Journal of the ACM*, 27(4):758–771, 1980.

[39] Bartłomiej Dudek and Paweł Gawrychowski. Slowing down top trees for better worst-case compression. In *Symposium on Combinatorial Pattern Matching, CPM 2018*, pages 16:1–16:8, 2018.

[40] Robert M. Fano. The transmission of information. Technical Report 65, Research Laboratory of Electronics at MIT, 1949.

[41] Philippe Flajolet and Robert Sedgewick. *Analytic Combinatorics*. Cambridge University Press, 2009.

[42] Markus Frick, Martin Grohe, and Christoph Koch. Query evaluation on compressed trees (extended abstract). In *Annual IEEE Symposium on Logic in Computer Science, LICS 2003*, pages 188–197, 2003.

[43] Isamu Furuya, Takuya Takagi, Yuto Nakashima, Shunsuke Inenaga, Hideo Bannai, and Takuya Kida. MR-RePair: Grammar compression based on maximal repeats. In *Data Compression Conference, DCC 2019*, pages 508–517, 2019.

[44] Travis Gagie, Tomohiro I, Giovanni Manzini, Gonzalo Navarro, Hiroshi Sakamoto, and Yoshimasa Takabatake. Rpair: Rescaling RePair with Rsync. In *International Symposium on String Processing and Information Retrieval, SPIRE 2019*, 2019.

[45] Moses Ganardi, Danny Hucke, Artur Jeż, Markus Lohrey, and Eric Noeth. Constructing small tree grammars and small circuits for formulas. *Journal of Computer and System Sciences*, 86:136–158, 2017.

[46] Moses Ganardi, Danny Hucke, Markus Lohrey, and Louisa Seelbach Benkner. Universal tree source coding using grammar-based compression. *IEEE Transactions on Information Theory*, 65(10):6399–6413, 2019.

[47] Moses Ganardi, Danny Hucke, Markus Lohrey, and Eric Noeth. Tree compression using string grammars. *Algorithmica*, 80(3):885–917, 2018.

[48] Moses Ganardi, Artur Jeż, and Markus Lohrey. Balancing straight-line programs. In *Symposium on Foundations of Computer Science, FOCS 2019*, 2019.

[49] Michal Ganczorz and Artur Jeż. Improvements on Re-Pair grammar compressor. In *Data Compression Conference, DCC 2017*, pages 181–190, 2017.

[50] Adrià Gascón, Markus Lohrey, Sebastian Maneth, Carl P. Reh, and Kurt Sieber. Grammar-based compression of unranked trees. In *Computer Science - Theory and Applications - 13th International Computer Science Symposium in Russia, CSR 2018*, pages 118–131, 2018.

[51] Leszek Gasieniec, Marek Karpinski, Wojciech Plandowski, and Wojciech Rytter. Efficient algorithms for Lempel-Ziv encoding (extended abstract). In *Scandinavian Workshop on Algorithm Theory, SWAT 1996*, volume 1097, pages 392–403, 1996.

[52] Leszek Gasieniec, Roman M. Kolpakov, Igor Potapov, and Paul Sant. Real-time traversal in grammar-based compressed files. In *Data Compression Conference, DCC 2005*, page 458, 2005.

[53] Pawel Gawrychowski and Artur Jeż. LZ77 factorisation of trees. In *Foundations of Software Technology and Theoretical Computer Science, FSTTCS 2016*, pages 35:1–35:15, 2016.

[54] Rodrigo González and Gonzalo Navarro. Compressed text indexes with fast locate. In *Symposium on Combinatorial Pattern Matching, CPM 2007*, pages 216–227, 2007.

[55] Mark A. Heap and Melvin R. Mercer. Least upper bounds on OBDD sizes. *IEEE Transactions on Computers*, 43(6):764–767, 1994.

[56] Danny Hermelin, Gad M. Landau, Shir Landau, and Oren Weimann. Unified compression-based acceleration of edit-distance computation. *Algorithmica*, 65(2):339–353, 2013.

[57] Yoram Hirshfeld, Mark Jerrum, and Faron Moller. A polynomial algorithm for deciding bisimilarity of normed context-free processes. *Theoretical Computer Science*, 158(1-2):143–159, 1996.

[58] Lorenz Hübschle-Schneider and Rajeev Raman. Tree compression with top trees revisited. In *International Symposium on Experimental Algorithms, SEA 2015*, volume 9125, pages 15–27, 2015.

[59] Danny Hucke. Approximation ratios of RePair, LongestMatch and Greedy on unary strings. In *International Symposium on String Processing and Information Retrieval, SPIRE 2019*, 2019.

[60] Danny Hucke, Markus Lohrey, and Louisa Seelbach Benkner. Entropy bounds for grammar-based tree compressors. In *IEEE International Symposium on Information Theory, ISIT 2019*, 2019.

[61] Danny Hucke, Markus Lohrey, and Carl P. Reh. The smallest grammar problem revisited. In *International Symposium on String Processing and Information Retrieval, SPIRE 2016*, pages 35–49, 2016.

[62] David A. Huffman. A method for the construction of minimum-redundancy codes. *Proceedings of the Institute of Radio Engineers*, 40(9):1098–1101, 1952.

[63] Artur Jeż. Faster fully compressed pattern matching by recompression. In *International Colloquium on Automata, Languages, and Programming, ICALP 2012*, pages 533–544, 2012.

[64] Artur Jeż. Approximation of grammar-based compression via recompression. *Theoretical Computer Science*, 592:115–134, 2015.

[65] Artur Jeż. A really simple approximation of smallest grammar. *Theoretical Computer Science*, 616:141–150, 2016.

[66] Artur Jeż and Markus Lohrey. Approximation of smallest linear tree grammars. In *Symposium on Theoretical Aspects of Computer Science, STACS 2014*, pages 445–457, 2014.

[67] Marek Karpinski, Wojciech Rytter, and Ayumi Shinohara. Pattern-matching for strings with short descriptions. In *Symposium on Combinatorial Pattern Matching, CPM 95*, pages 205–214, 1995.

[68] Dominik Kempa and Nicola Prezza. At the roots of dictionary compression: string attractors. In *Symposium on Theory of Computing, STOC 2018*, pages 827–840, 2018.

[69] Takuya Kida, Tetsuya Matsumoto, Yusuke Shibata, Masayuki Takeda, Ayumi Shinohara, and Setsuo Arikawa. Collage system: A unifying framework for compressed pattern matching. *Theoretical Computer Science*, 298(1):253–272, 2003.

[70] John C. Kieffer. A survey of Bratteli information source theory. In *IEEE International Symposium on Information Theory, ISIT 2016*, pages 16–20, 2016.

[71] John C. Kieffer, Philippe Flajolet, and En-Hui Yang. Universal lossless data compression via binary decision diagrams. Technical report, arxiv.org, 2011. http://arxiv.org/abs/1111.1432.

[72] John C. Kieffer and En-Hui Yang. Grammar-based codes: A new class of universal lossless source codes. *IEEE Transactions on Information Theory*, 46(3):737–754, 2000.

[73] John C. Kieffer and En-Hui Yang. Structured grammar-based codes for universal lossless data compression. *Communications in Information and Systems*, 2(1):29–52, 2002.

[74] John C. Kieffer, En-Hui Yang, Gregory J. Nelson, and Pamela C. Cosman. Universal lossless compression via multilevel pattern matching. *IEEE Transactions on Information Theory*, 46(4):1227–1245, 2000.

[75] John C. Kieffer, En-Hui Yang, and Wojciech Szpankowski. Structural complexity of random binary trees. In *IEEE International Symposium on Information Theory, ISIT 2009*, pages 635–639, 2009.

[76] Donald E. Knuth. *The Art of Computer Programming, Volume II, 3rd Edition*. Addison-Wesley, 1998.

[77] N. Jesper Larsson and Alistair Moffat. Offline dictionary-based compression. In *Data Compression Conference, DCC 1999*, pages 296–305, 1999.

[78] Philip M. Lewis II, Richard E. Stearns, and Juris Hartmanis. Memory bounds for recognition of context-free and context-sensitive languages. In *IEEE Symposium on Switching Circuit Theory and Logic Design, SWCT 1965*, pages 191–202, 1965.

[79] Heh-Tyan Liaw and Chen-Shang Lin. On the OBDD-representation of general boolean functions. *IEEE Transactions on Computers*, 41(6):661–664, 1992.

[80] Yury Lifshits. Processing compressed texts: A tractability border. In *Symposium on Combinatorial Pattern Matching, CPM 2007*, pages 228–240, 2007.

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

[82] Markus Lohrey, Sebastian Maneth, and Roy Mennicke. XML tree structure compression using RePair. *Information Systems*, 38(8):1150–1167, 2013.

[83] Markus Lohrey, Sebastian Maneth, and Carl P. Reh. Traversing grammar-compressed trees with constant delay. In *Data Compression Conference, DCC 2016*, pages 546–555, 2016.

[84] Markus Lohrey, Sebastian Maneth, and Carl P. Reh. Constant-time tree traversal and subtree equality check for grammar-compressed trees. *Algorithmica*, 80(7):2082–2105, 2018.

[85] Markus Lohrey, Sebastian Maneth, and Manfred Schmidt-Schauß. Parameter reduction and automata evaluation for grammar-compressed trees. *Journal of Computer and System Sciences*, 78(5):1651–1669, 2012.

[86] Abram Magner, Krzysztof Turowski, and Wojciech Szpankowski. Lossless compression of binary trees with correlated vertex names. In *IEEE International Symposium on Information Theory, ISIT 2016*, pages 1217–1221, 2016.

[87] Nicolas Markey and Philippe Schnoebelen. A PTIME-complete matching problem for SLP-compressed words. *Information Processing Letters*, 90(1):3–6, 2004.

[88] Takuya Masaki and Takuya Kida. Online grammar transformation based on re-pair algorithm. In *Data Compression Conference, DCC 2016*, pages 349–358, 2016.

[89] Kurt Mehlhorn, Rajamani Sundar, and Christian Uhrig. Maintaining dynamic sequences under equality tests in polylogarithmic time. *Algorithmica*, 17(2):183–198, 1997.

[90] Michael I. Miller and Joseph A. O'Sullivan. Entropies and combinatorics of random branching processes and context-free languages. *IEEE Transactions Information Theory*, 38(4):1292–1310, 1992.

[91] Masamichi Miyazaki, Ayumi Shinohara, and Masayuki Takeda. An improved pattern matching algorithm for strings in terms of straight-line programs. In *Symposium on Combinatorial Pattern Matching, CPM 1997*, pages 1–11, 1997.

[92] Gonzalo Navarro. Indexing highly repetitive collections. In *International Workshop on Combinatorial Algorithms, IWOCA 2012*, pages 274–279, 2012.

[93] Adamu M. Noma, Abdullah Muhammed, Mohamad A. Mohamed, and Zuriati A. Zulkarnain. A review on heuristics for addition chain problem: Towards efficient public key cryptosystems. *Journal of Computer Science*, 13(8):275–289, 2017.

[94] Carlos Ochoa and Gonzalo Navarro. RePair and all irreducible grammars are upper bounded by high-order empirical entropy. *IEEE Transactions on Information Theory*, 65(5):3160–3164, 2019.

[95] Mike Paterson and Mark N. Wegman. Linear unification. *Journal of Computer and System Sciences*, 16(2):158–167, 1978.

[96] Wojciech Plandowski. Testing equivalence of morphisms on context-free languages. In *European Symposium on Algorithms, ESA 1994*, pages 460–470, 1994.

[97] Wojciech Plandowski and Wojciech Rytter. Complexity of language recognition problems for compressed words. In Juhani Karhumäki, Hermann A. Maurer, Gheorghe Paun, and Grzegorz Rozenberg, editors, *Jewels are Forever, Contributions on Theoretical Computer Science in Honor of Arto Salomaa*, pages 262–272. Springer, 1999.

[98] David Reinsel, John Gantz, and John Rydning. The digitization of the world from edge to core. Technical report, International Data Corporation (IDC) (sponsored by Seagate), 2018. https://www.seagate.com/files/www-content/our-story/trends/files/idc-seagate-dataage-whitepaper.pdf.

[99] Peter Roth. A note on word chains and regular languages. *Information Processing Letters*, 30(1):15–18, 1989.

[100] Frank Rubin. Experiments in text file compression. *Communications of the ACM*, 19(11):617–623, 1976.

[101] Walter L. Ruzzo. Tree–size bounded alternation. *Journal of Computer and System Sciences*, 21:218–235, 1980.

[102] Wojciech Rytter. Application of Lempel-Ziv factorization to the approximation of grammar-based compression. *Theoretical Computer Science*, 302(1–3):211–222, 2003.

[103] Hiroshi Sakamoto. A fully linear-time approximation algorithm for grammar-based compression. *Journal of Discrete Algorithms*, 3(2-4):416–430, 2005.

[104] Manfred Schmidt-Schauß. Polynomial equality testing for terms with shared substructures. Technical Report 21, Institut für Informatik, J. W. Goethe-Universität Frankfurt am Main, 2005.

[105] Claude E. Shannon. A mathematical theory of communication. *The Bell System Technical Journal*, 27(3):379–423, 7 1948.

[106] Philip M. Spira. On time-hardware complexity tradeoffs for boolean functions. In *Hawaii International Conference on System Sciences, HICSS 1971*, pages 525–527, 1971.

[107] Richard P. Stanley. *Catalan Numbers*. Cambridge University Press, 2015.

[108] James A. Storer and Thomas G. Szymanski. Data compression via textural substitution. *Journal of the ACM*, 29(4):928–951, 1982.

[109] En-Hui Yang and John C. Kieffer. Efficient universal lossless data compression algorithms based on a greedy sequential grammar transform - part one: Without context models. *IEEE Transactions on Information Theory*, 46(3):755–777, 2000.

[110] Andrew C. Yao. On the evaluation of powers. *SIAM Journal on Computing*, 5(1):100–103, 1976.

[111] Jie Zhang, En-Hui Yang, and John C. Kieffer. A universal grammar-based code for lossless compression of binary trees. *IEEE Transactions on Information Theory*, 60(3):1373–1386, 2014.

[112] Jacob Ziv and Abraham Lempel. A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory*, 23(3):337–343, 1977.

[113] Jacob Ziv and Abraham Lempel. Compression of individual sequences via variable-rate coding. *IEEE Transactions on Information Theory*, 24(5):530–536, 1978.