# Analysis of Grammar-Based Tree Compression

DISSERTATION
zur Erlangung des Grades eines Doktors
der Naturwissenschaften

genehmigte Dissertation von
Dipl. Math. Eric John Leo Nöth

Erstgutachter: Prof. Dr. Markus Lohrey
Zweitgutachter: Prof. Dr. Markus Nebel

Tag der Disputation: 8. Juni 2016

Eingereicht bei der Naturwissenschaftlich-Technischen
Fakultät
der Universität Siegen
Siegen 2016

Gedruckt auf alterungsbeständigem holz- und säurefreiem Papier.

# Contents

1 **Introduction** **5**

2 **Preliminaries** **11**
- 2.1 Trees . . . 11
  - 2.1.1 The FCNS-encoding . . . 12
- 2.2 The minimal directed acyclic graph . . . 13
- 2.3 Grammars . . . 17
  - 2.3.1 SLP grammars . . . 17
  - 2.3.2 TSLP grammars . . . 18

3 **The average-case size of the DAG** **23**
- 3.1 Generating functions and the Catalan numbers . . . 24
- 3.2 Exact counting . . . 25
- 3.3 Complex analysis and singularity analysis . . . 27
  - 3.3.1 Complex analysis . . . 29
  - 3.3.2 The standard function scale . . . 30
  - 3.3.3 Singularity transfer . . . 30
- 3.4 Proof of the main theorem . . . 31
- 3.5 Extensions . . . 39
  - 3.5.1 Exact counting . . . 40
  - 3.5.2 Asymptotic results . . . 42
- 3.6 On proving the main result by Mellin transform . . . 48
- 3.7 Open problems related to DAGs . . . 49
  - 3.7.1 Variance . . . 49
  - 3.7.2 Unordered trees . . . 50
  - 3.7.3 The average size of the DAG of regular tree languages . . . 50

4 **The hybrid DAG and worst-case sizes** **53**
- 4.1 The construction . . . 53
- 4.2 Comparison of the worst-case sizes of DAG, BDAG and HDAG . 55
  - 4.2.1 The number of nodes . . . 55
  - 4.2.2 The number of edges . . . 56
- 4.3 The average size of the HDAG . . . 60

5 **DAG compression using string grammars** **63**

**6 Tree compression using string grammars 69**
6.1 Trees as preorder traversals . . . 69
6.2 Checking whether an SLP produces a tree . . . 70
6.3 Comparison with other grammar-based tree representations . . . 71
6.3.1 Traversal SLPs versus TSLPs . . . 72
6.3.2 Traversal SLPs versus balanced parenthesis representation 75

**7 Experiments 79**
7.1 Corpora . . . 79
7.2 Experiment Setup . . . 80
7.3 Comparison . . . 80

**8 Algorithmic Problems 83**
8.1 Subtree equality check . . . 83
8.2 Tree Automata . . . 85
8.3 Tree Navigation . . . 86

**9 Accelerated computation by compression in term rewriting 89**
9.1 Term Rewriting . . . 90
9.1.1 Termination and matrix interpretations . . . 91
9.1.2 Cost of terms . . . 92
9.2 Term Compression with TreeRePair . . . 93
9.2.1 TreeRePair . . . 93
9.2.2 Minimizing the matrix multiplication cost . . . 97
9.3 Compression and the DP-transformation . . . 100
9.4 Experiments . . . 103
9.5 Discussion . . . 106

**10 Conclusion and final remarks 109**

# Chapter 1

# Introduction

Lossless data compression is a classic research area of computer science, both from a theoretic and from a practical point of view [Say06, Sal07]. The classic algorithms known as *LZ77* and *LZ78* by A. Lempel and J. Ziv [ZL77, ZL78], are probably the most famous lossless compression techniques. The purpose of data compression is manifold: While oftentimes the aim is to reduce the amount of space needed on the disk, data compression may also be used to find patterns (this is done for example in bioinformatics [ZHC$^+$15]) or to accelerate computations [FS87], see also Chapter 9.

When searching or manipulating the compressed data, it is advantageous to be able to execute the desired operations directly on the compressed data instead of performing a prior decompression and subsequent re-compression. A popular compression paradigm that enables this goal is so-called *grammar-based compression* [KY00].

The idea of grammar-based compression is to replace the data $d$ by a context-free grammar $\mathbb{A}$ that generates the data $d$. If the data is a string, these grammars are called *straight-line programs* (SLP). By repeated doubling, an SLP may generate a string which is exponentially larger than the grammar. Thus SLPs may be exponentially more succinct than their string counterpart.

Since computing a smallest grammar for a given string in polynomial time is not possible unless $\mathsf{P} = \mathsf{NP}$ [CLL$^+$05], the research focus on grammar-based compression lies on polynomial-time algorithms with a good *approximation ratio* (see page 17 below for a formal definition). There are several algorithms that achieve an approximation ratio of $\mathcal{O}(\log(n/g))$, where $n = |s|$ is the size of the input string $s$ and $g$ is the size of a smallest grammar for $s$ [Jez14, CLL$^+$05, Ryt03]. Other grammar-based string compressors that work well in practice are *Re-Pair* [LM99], *Sequitur* [NW97] and *BiSection* [KYNC00].

Grammar-based compression can be generalized from strings to tree-structured data. Trees are a central object in computer science which are used in many different contexts. We focus on *rooted, ordered, labelled* trees, i.e., every tree has a distinguished *root node*, there is an *ordering* among the children of a node and each node carries a label from a finite alphabet. If the label of a node determines the number of the node's children, we call the tree *ranked*, otherwise *unranked.* An important subclass of ranked trees are *full binary trees*, in which every node has zero or two successors.

In this thesis we treat different aspects on grammar-based tree compression.

Here one seeks to compress a tree $t$ by exploiting repeated tree patterns to construct a context-free tree grammar that produces only $t$ (see [CDG$^+$07] for a background in tree grammars). These grammars are called *tree straight-line programs* (TSLPs) (sometimes *SLCF tree grammar* [BLM08]). A *tree pattern* is any connected subgraph of a tree, and a *repeated tree pattern* is an identical (both in structure and in labels) occurrence of a tree pattern in the tree.

If one restricts the tree patterns to subtrees, i.e. to subgraphs consisting of a node and all its descendants, the tree grammars become regular and the resulting formalism is in strong correspondence with tree compression by so-called *directed acyclic graphs* (see below).

A different approach to grammar-based tree compression is to compress the depth-first left-to-right traversal of the node labels by an SLP. This is possible if the tree is ranked, since in that case this word determines the tree uniquely. This way non-connected tree patterns may also be compressed.

In this thesis we discuss different aspects of grammar-based tree compression. Since the subject is rather broad, we focus on three aspects therein, namely (i) *the minimal directed acyclic graph* of a tree, (ii) using straight-line programs in tree compression and (iii) algorithmics on compressed data. We now introduce these subjects in more detail.

**The minimal directed acyclic graph.** The *(minimal) directed acyclic graph* [Ers58] (or *DAG*) is a very widely used tree compression technique. The method is rather simple: Repeated occurrences of subtrees are saved only once and further occurrences of the tree are replaced by a link to the first one. This results in a unique minimal directed acyclic graph, which can be computed in linear time [DST80]. The DAG of a tree can very naturally be identified with a regular tree grammar, which means that DAGs can be seen as an important subclass of grammar-based tree compression.

Among many different uses, DAGs are used to compress the tree structure of XML documents [BGK03], to find common subexpressions in compiler construction [ASU86, Muc97], to optimize binary decision diagrams [MT98], or to reduce the average time and space complexity of automatic differentiation from $\mathcal{O}(n^{3/2})$ to $\mathcal{O}(n)$ [FS87].

While the size of the DAG of a tree of size $n$ may range from $\mathcal{O}(\log n)$ to $\mathcal{O}(n)$ (consider a perfect binary tree and a tree in which every node has at most one non-leaf successor, respectively), it is certainly interesting to know what the asymptotic average size of the DAG of a tree is. In [FSS90] it is stated that the average size of the DAG of an unlabelled full binary tree is

$$C_{\mathcal{F}} \frac{n}{\sqrt{\log n}} \left(1 + \mathcal{O}\left(\frac{1}{\log n}\right)\right),$$

with $C_{\mathcal{F}} = 2\sqrt{\ln 4/\pi}$. Regrettably [FSS90] lacks a full proof of the statement. We fill this gap in Chapter 3 of this work and provide a rigorous proof for this case. We then generalize the result to different classes of trees, namely unranked trees, binary trees (trees in which every node has zero successors, or a left child only, or a right child only, or two children) and to a labelled setting.

The proof is an example from the mathematical discipline of *Analytic Combinatorics*, the foundations of which have been set in the monumental book with the same title by P. Flajolet and R. Sedgewick [FS09]. Here, given a class $\mathcal{C}$ of objects such as trees, one is interested in the asymptotic growth order of the number of those objects of size $n$. The first step is then to use a certain set of constructors to define a combinatorial specification of the class investigated, which directly transfers to a *generating function* for the objects of interest. The (ordinary) generating function of a class $\mathcal{C}$ is defined as the formal power series $C(z) = \sum_{n\geq 0} |\mathcal{C}_n| z^n$, where $\mathcal{C}_n = \{c \in \mathcal{C} \mid |c| = n\}$. This step is called the *symbolic transfer* and is followed by an *analytic transfer*: The generating function $C(z)$ is investigated as a function of the complex plane. Knowing the position and the type of the function's *dominant singularity*[1] (in some cases, dominant singularities), one can oftentimes directly state the asymptotic growth order of the coefficients of the generating function. For some generating functions it might not be possible to derive the type of the singularity directly (this will be the case for the generating function investigated here). In this case one uses the methods described in [FO90]: If the investigated function can be approximated by a reference function with known coefficient asymptotics in a neighborhood of the dominant singularity, then the coefficient asymptotic of the reference function transfers to the coefficient asymptotics of the investigated function.

Apart from a rigorous proof of the asymptotic growth order of the average size of a tree's DAG, Chapter 3 also provides some theorems from analytic combinatoric that are necessary to understand the proof.

*The hybrid DAG and worst-case size differences between the DAG and the binary DAG.* Oftentimes unranked trees like the tree structures defined by XML-documents are represented via binary trees using the so-called *first-child/next-sibling* encoding [Koc03] (called *rotation correspondence* in [Knu68]). The *first-child/next-sibling* (Fcns) encoding of a tree replaces the tree's edge set by a new one in which we add from every node an edge to its first child and an edge to its next sibling (both in left-to-right order). Given an unranked tree $t$, we define the DAG of its FCNS-encoding as bdag($t$). It was observed in [BLM08] that the size of bdag($t$) and dag($t$) may differ in both directions for a tree $t$. In Chapter 4 we define a third structure, which we call *HDAG* for *Hybrid DAG*. The HDAG is generated from the DAG of a tree by first viewing the DAG as a regular tree grammar and then sharing the repeated sibling sequences that occur in the FCNS-encoding of the grammar's right-hand sides. We first prove that the HDAG of a tree has less edges than each of the DAG and the BDAG of a tree (Theorem 4.7). Using this, we prove that (Corollary 4.11)

$$\frac{1}{2}|\mathrm{bdag}(t)|_E \leq |\mathrm{dag}(t)|_E \leq \frac{1}{2}|\mathrm{bdag}(t)|_E^2,$$

where $|\cdot|_E$ denotes the number of edges of the structure. In Chapter 7 we perform experiments to evaluate the HDAG as a tree compressor on real data.

*Related work.* Famous tree related average-case analyses are the average height of unranked trees [dBKR72] (called *plane planted trees* by the authors) and the average height of binary trees [FO82]. The latter paper also spawns much of the transfer techniques described in [FO90] that are central to our proof.

As mentioned before, we assume that the input distribution is uniform, i.e. every tree is equally likely. The papers [Dev98, FGM97] treat the aver-

[1]Given a function $f$, a point $z_0$ is called *singularity* of $f$ if $f$ is not analytically continuable at $z_0$ (e.g. because $f$ or its derivative is infinite at $z_0$), see [FS09, p. 239 ff.]. A singularity is called *dominant* if it is a singularity of smallest modulus.

age size of the DAG in a non-uniform case, namely if the input tree distribution is induced by binary search trees. Here, one starts with the uniform distribution of permutations, builds for every permutation its binary search tree by inserting the nodes from left-to-right and deleting the node labels afterwards. The average size of the DAG turns out to be $\mathcal{O}(n/\log n)$ in that case.

**Combining string compression and tree compression.** In the Chapters 5 and 6 we discuss the use of SLPs for the compression of trees and their relation to TSLPs. In Chapter 5 we compress the child sequences of the DAG grammar (the grammar defined by the DAG of a tree) by SLPs, which we call *SLP-compressed DAGs*. The HDAG of Chapter 4 is a particular SLP-compressed DAG. We prove that SLP-compressed DAGs can be converted in linear time to TSLPs with only a moderate size increase (Theorem 5.5). Thus, while SLP-compressed DAGs are not tree grammars, they can easily be transformed into TSLPs. One advantage of this approach versus "normal" TSLP construction is that string compression is somewhat easier, as we do not need to build expensive pointer structures.

In Chapter 6 we investigate the compression of the depth-first left-to-right-traversal of a tree by SLPs, which we call *traversal SLPs*. We show that traversal SLPs may be exponentially more succinct than TSLPs (Theorem 6.6) and prove that they can be converted to TSLPs, where both the size increase and the runtime of the conversion depend on the maximal rank, the depth of the tree and the size of the grammar (Theorem 6.7). Note that for a given TSLP $\mathbb{A}$ for a tree $t$ one can efficiently construct a traversal SLPs of size $\mathcal{O}(|\mathbb{A}| \cdot r)$ for $t$, where $r$ is the maximal rank among nodes occurring in $t$ [BLM08]. Thus a TSLP cannot be exponentially more succinct than a traversal SLP. We also prove that the traversal SLP of an unlabelled tree may be exponentially more succinct than its *balanced parenthesis representation* (Theorem 6.9).

In Chapter 7 we evaluate the different introduced tree compression techniques experimentally. We use three different corpora of XML-documents and compare the sizes of their DAG, BDAG and HDAG. We also evaluate their sizes as SLP-compressed DAG or as traversal SLP, using Re-Pair as the string compressor.

**Related work.** Several grammar-based tree compressors have been developed. While the compressor *TreeRePair* [LMM13] performs well in practice (see e.g. [HR15]), there exists a family of (binary) trees $(t_n)_{n\geq 1}$ such that (i) $t_n$ has size $\mathcal{O}(n)$, (ii) there exists a grammar for $t_n$ of size $\mathcal{O}(\log n)$ and (iii) TreeRePair produces a grammar of size $\Omega(n)$ for $t_n$ [LMM13]. In contrast, *TTOG* by [JL14] produces for every tree $t$ with maximal rank $r$ a grammar of size $\mathcal{O}(r^2 g \log |t|)$, where $g$ is the size of a smallest grammar for $t$. The method from [HLN14] produces for every tree $t$ over a ranked alphabet with $\sigma$ different symbols a grammar of size $\mathcal{O}\left(|t|\frac{|t|}{\log_\sigma |t|}\right)$. Other grammar-based tree compressors are in [Aku10, BLM08], where the work from [Aku10] is based on another type of tree grammars (elementary ordered tree grammars).

Tree compression by *top DAGs* [BGLW15] is a tree compression scheme related to tree grammars. It is based on the hierarchical decomposition of the tree into clusters via top trees [AHdLT05]. The top DAG of a tree may be exponentially smaller than the DAG, but is never more than logarithmically larger. The worst-case size of the top DAG of a tree is $O\left(\frac{n \cdot \log\log_\sigma n}{\log_\sigma n}\right)$ [HR15], and navigation queries can be supported in logarithmic time.

Succinct data structures for trees are also used to represent trees in a compact way. Here, the goal is to represent a tree in a number of bits that asymptotically matches the information-theoretic lower bound, while at the same time providing efficient querying. For unlabelled (resp., node-labelled) unranked trees of size $N$ there exist representations with $2N+o(N)$ bits (resp., $(2+\log\sigma)\cdot N+o(N)$ bits, where $\sigma$ is the number of node labels) that support navigation and some other tree queries in time $\mathcal{O}(1)$ [BDM+05, FLMM09, JSS12, MR01].

**Algorithms on compressed trees and accelerated computation by compression.** As mentioned before, one reason to compress a tree by a tree grammar (as opposed to, say, treating the tree as a binary file) is that it allows tree operations *directly* on the compressed tree. In Chapter 8 we treat the primitive *subtree equality check*, which is very efficient for the DAG, BDAG and HDAG. We also discuss the *uniform membership problem* for compressed trees. Here, one is given a tree automaton $\mathcal{A}$ and a tree $t$ in compressed form (e.g. a grammar $\mathbb{A}$ that unfolds to $t$) and one wants to know whether $\mathcal{A}$ accepts $t$. We prove that the uniform membership problem is PSPACE-complete if the tree is given by an SLP for its preorder traversal (as in Chapter 6) and it is in P (polynomial time) for the other compressed tree representations we have treated. Last we discuss navigation on compressed trees.

In Chapter 9 we show an example on how compression may accelerate computation: A *term rewriting system* (TRS) consists of a set of rules on how terms may be rewritten and the question of interest is whether the TRS terminates for all possible inputs or not. While the general problem is undecidable, termination (and non-termination) may be proven in some instances. This is done as follows: From the rewrite rules of the TRS a boolean expression in conjunctive normal form is generated, which is then given to a SAT-solver. A satisfying assignment of that boolean expression leads to a proof that the TRS is terminating (if no such assignment exists, then the TRS may or may not be terminating). These boolean expressions are typically huge (typically about $10,000$ variables and $100,000$ clauses) and the SAT-solver is a lot more costly than the conversion of the TRS to a boolean expression. Thus one may try to compress the TRS before the conversion step in order to obtain a smaller boolean expression, which then may lead to a faster overall runtime.

We show how TreeRePair may be modified to achieve exactly that goal (it will become clear in Chapter 9 why the modification is necessary). We show in experiments that this approach indeed leads to smaller boolean expressions, which then lead to smaller runtimes for automatic termination proving.

**Related work.** In [BLR+15] the authors treat the *random access problem* for grammar-compressed strings and trees. The authors provide a data structure, that, after processing time linear in the grammar size, random access time logarithmic in the size of compressed string resp. tree. In [EWZ08] the authors introduce the so-called *matrix interpretations* of term rewriting systems that we use to generate the boolean expressions. Already in that paper TreeRePair is used to compress the terms prior, albeit in an unoptimized way.

**First Appearances of the contributions.** Most of the results from the Chapters 3, 4, 5 and 7 were first presented in [BLMN15]. Chapter 6 is from

[GHLN15] and Chapter 9 is taken from [BLNW13]. Chapter 8 carries material from [BLMN15] and [GHLN15].

**Acknowledgements.** I would like to thank my thesis supervisor Markus Lohrey for suggesting this subject and introducing me to the interesting field of trees and data compression. His great advice, support and ideas were essential for this work.

I wish to thank my coauthors Alexander Bau, Mireille Bousquet-Mélou, Moses Ganardi, Danny Hucke, Artur Jeż, Sebastian Maneth and Alexander Waldmann. Special thanks go to Mireille, who taught me the beauties of generatingfunctionology and Danny, with whom I spent countless hours commuting between Leipzig and Siegen. I would also like to thank Raphael Reitzig, with whom I tried to solve an open problem. Regrettably we were not successful, but the work was interesting and helpful.

I am grateful to my colleagues in Leipzig and in Siegen that were always open for discussions and helped me at many occasions: Thanks goes to Atefeh Babari, Claudia Carapelle, Doreen Götze, Hendrik Hochstetter, Alexander Kartzow, Enno Keßler, Daniel König and Philipp Reh.

Markus Nebel's comments and critique helped to improve this work and I am grateful for that.

Last but not least I would like to thank my loving family: My parents and my siblings, my wife Tiziana for her infinite support and my daughter Susie for many insightful discussions.

# Chapter 2

# Preliminaries

In this chapter we gather most of the necessary definitions for the remainder of this thesis. Prerequisites that only affect certain parts of the thesis are stated directly in the respective chapter. This is true especially for the mathematical background in combinatorics and analysis in Chapter 3, but also for the basics of term rewriting systems used in Chapter 9.

Let $\Sigma$ be a finite alphabet. For a string $w = a_1 \cdots a_n \in \Sigma^*$ we define $|w| = n$, $w[i] = a_i$ and $w[i : j] = a_i \cdots a_j$ where $w[i : j] = \varepsilon$, if $i > j$. Let $w[: i] = w[1 : i]$ and $w[i :] = w[i : |w|]$. $[1..n]$ denotes the set of numbers $\{1, 2, \dots, n\}$. For an integer $k$, $[k..]$ denotes the infinite set $\{k, k+1, ...\}$.

## 2.1 Trees

As mentioned in the introduction, we define trees in many different ways. In Chapter 3 we take a more combinatorial view, namely we define an *unranked tree* (or *plane tree*, as they oftentimes called in combinatorics) as a singleton node or a node with a sequence of trees attached and a *full binary tree* as a singleton node or a node with two trees attached. For Chapter 4 it will be useful to define trees as special acyclic graphs, since we want to discuss the minimal acyclic graph of a tree. In Chapter 6 we equate trees with their depth-first left-to-right traversal string. In Chapter 9 we define trees as a prefix-closed set.

Unless otherwise noted we stipulate that all trees are *rooted* and *ordered*, i.e. every tree has a distinguished *root* node with no predecessors, and that there is an ordering upon a node's child nodes.

Furthermore the trees we consider are *labelled*, meaning that every node in the tree carries a label from a finite alphabet $\Sigma$. The function $\lambda : V \to \Sigma$ denotes the labeling function. If the label $\lambda(v)$ of a node $v$ determines the number of $v$'s children uniquely, then we call the alphabet *ranked* and thus we speak of *ranked* trees, otherwise of *unranked* trees. Usually we denote unranked alphabets by $\Sigma$ and ranked alphabets by $\mathcal{F}$. The set of all trees over $\Sigma$ resp. $\mathcal{F}$ is denoted by $\mathcal{T}(\Sigma)$ resp. $\mathcal{T}(\mathcal{F})$. By $\mathcal{F}_i$ we denote the set of all symbols from $\mathcal{F}$ of rank $i$ (we sometimes use *arity* instead of rank). Nodes labelled by symbols of rank 0, 1 and 2 are called *leaves*, *unary nodes* and *binary nodes*, respectively. The function $\text{rank} : \mathcal{F} \to \mathbb{N}$ denotes the function that returns the rank of a symbol. We always assume that $\mathcal{F}_0 \neq \emptyset$. We denote trees by their usual term notation, e.g. $f(a,a)$ stands for a tree consisting of a root node labelled by $f$ that has two $a$-labelled children.

We gather the points above in the following definition.

**Definition 2.1.** Let $\mathcal{F}$ be a ranked alphabet and let $\Sigma$ be an unranked alphabet. The set $\mathcal{T}(\mathcal{F})$ of all *ranked trees* over $\mathcal{F}$ is defined inductively as follows: If $f \in \mathcal{F}_n$ with $n \geq 0$ and $t_1, \ldots, t_n \in \mathcal{T}(\mathcal{F})$, then $f(t_1, \ldots, t_n) \in \mathcal{T}(\mathcal{F})$.

The set $\mathcal{T}(\Sigma)$ of all *unranked trees* over $\Sigma$ is defined inductively as follows: If $f \in \Sigma$, $n \geq 0$ and $t_1, \ldots, t_n \in \mathcal{T}(\Sigma)$, then $f(t_1, \ldots, t_n) \in \mathcal{T}(\Sigma)$.

If a ranked alphabet $\mathcal{F}$ contains symbols of rank zero and two only, we speak of *full binary trees.*

By *binary trees* we denote trees which have leaf nodes, binary nodes and two different types of unary nodes: Those with a *left child* and those with a *right child.* These trees are sometimes called *pruned binary trees*, as they can be obtained by pruning the leaf nodes of a full binary tree. Edges to left resp. right children are called *left edges* resp. *right edges.*

Let $t$ be a (ranked or unranked) tree. By $|t|_E$ we denote the number of *edges* in $t$ and by $|t|_N$ we denote the number of *nodes* in $t$. We trivially have $|t|_N = |t|_E + 1$ for every tree $t$. Unless otherwise noted, when we speak of the *size* of a tree we mean the number of *edges* the tree contains.

A node $v \in t$ can be addressed in many different ways, and we will use two different possibilities. On the one hand, one may simply refer to its appearance in a preorder traversal of the tree, thus enumerating the $n$ nodes of $t$ by the integers from $[1..n]$. We will use this point of view when we discuss the string that is defined by the (preorder) traversal of a tree (Chapter 6). Note that this string defines the tree only uniquely if the alphabet is ranked.

A different possibility is to address nodes by words $w \in \{1, \ldots, k\}^*$, where $k \in \mathbb{N}$ is the maximal rank of a node in $t$. These words can be read as a navigation path through the tree: The first symbol in $w$ tells us which child of the root we go to (the root node itself is addressed by $\varepsilon$), thus a 1 means the first child etc., and the second symbol in $w$ tells us where to go to from there and so forth. For example the string "2 3" denotes the third child of the second child of the root. This notation also conveniently allows to define the *depth* of a node as the length of the word that addresses it and the *height* of a tree $t$ as the maximal depth among nodes in $t$.

Alternatively one may define the depth of a node $v$ or the height of a tree $t$ via the graph-theoretic interpretation of a tree, namely by the number of edges from the root node to $v$ respectively by the maximal depth of a node in the tree.

For a node $v$ in a tree $t$ we denote by $t/v$ the subtree of $t$ that is rooted at $v$.

### 2.1.1 The FCNS-encoding

For many tree-processing formalism like tree automata it is useful to deal with ranked trees, even if the input data itself is unranked, see [Sch07] for a discussion. Thus unranked trees are often represented by binary trees. A common encoding is the so-called *first child/next sibling encoding* (FCNS-encoding), which is called the *rotation correspondence* in Knuth's first book [Knu68] and is also known as *Rabin encoding.* The encoding fcns($t$) of an unranked tree $t$ is obtained by replacing all edges in $t$ by an edge set in which every node gets an edge pointing to its first child (if existent) and an edge which points to the node's next

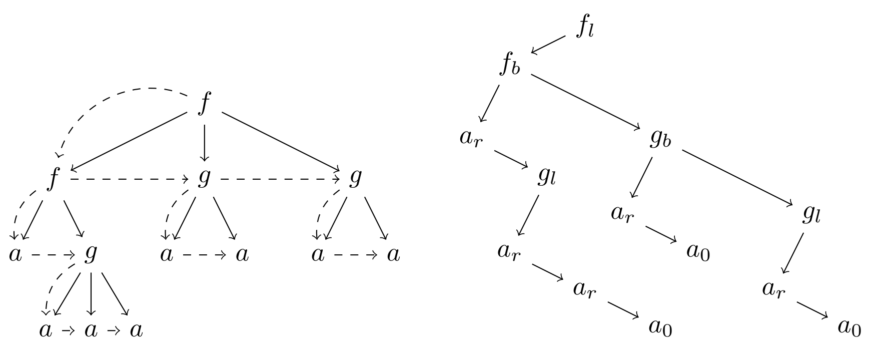


Figure 2.1: An unranked tree and its FCNS-encoding. The subscript $b$ stands for *binary*, $l$ for *left*, $r$ for *right* and 0 for *leaf*.

sibling (if existent). The prior are called *left edges*, while the latter are called *right edges*. This illustrates the name *rotation correspondence*: By redrawing the edges and rotating the tree by 45 degrees one obtains the desired FCNS-encoding. Formally:

**Definition 2.2.** Let $\Sigma$ be an unranked alphabet. We define a ranked alphabet $\mathcal{F}^\Sigma$ as follows: For each $f \in \Sigma$ we define four symbols, $f_0 \in \mathcal{F}_0^\Sigma$, $f_l, f_r \in \mathcal{F}_1^\Sigma$ and $f_b \in \mathcal{F}_2^\Sigma$. Finally we set $\mathcal{F}^\Sigma := \mathcal{F}_0^\Sigma \cup \mathcal{F}_1^\Sigma \cup \mathcal{F}_2^\Sigma$.

The mapping fcns : $\mathcal{T}(\Sigma)^* \to \mathcal{T}(\mathcal{F}^\Sigma)$ is defined as follows: We map the empty word to the empty tree. If $n \geq 1, t_1, \ldots, t_n \in \mathcal{T}(\Sigma)$ and $t_1 = f(u_1, \ldots, u_m)$ with $m \geq 0$, then

$$\text{fcns}(t_1 \ldots t_n) = \begin{cases} f_b(\text{fcns}(u_1 \ldots u_m), \text{fcns}(t_2 \ldots t_n)) & \text{if } m \geq 1 \\ f_r(\text{fcns}(t_2 \ldots t_n)) & \text{if } m = 0 \end{cases}$$

The encoding is bijective, hence the inverse $\text{fcns}^{-1} : \mathcal{T}(\mathcal{F}^\Sigma) \to \mathcal{T}(\Sigma)^*$ is defined. For clarity (e.g. if a node label already carries a subscript), we sometimes use $f(a, \Box)$ and $f(\Box, a)$ to denote a unary $f$-labelled node with an $a$-labelled left resp. right child.

## 2.2 The minimal directed acyclic graph

In this section we define the *minimal directed acyclic graph* of a tree in a general context, which will be useful for later chapters.

**Definition 2.3.** Let $\Sigma$ be a finite unranked alphabet and let $\mathcal{F}$ be a finite ranked alphabet. An *ordered $\Sigma$-labelled (resp. $\mathcal{F}$-labelled) multigraph* is a tuple $M = (V, \gamma, \lambda)$, where

- $V$ is a finite set of nodes
- $\gamma : V \to V^*$ (the *successor function*) assigns to each node a finite word over the set of nodes, and
- $\lambda : V \to \Sigma$ resp. $\lambda : V \to \mathcal{F}$ (the *labeling function*) assigns to each node a label from $\Sigma$ resp. $\mathcal{F}$.

- If the alphabet is ranked, we additionally assume that $|\gamma(v)| = \text{rank}(\lambda(v))$ for every node $v$.

In other words, an ordered labelled multigraph is a graph in which

- there is an *ordering* among the edges of a node,
- the nodes are *labelled* and
- *multiple* edges may point from one node to another.

The *underlying graph* is the directed graph $G_M = (V, E)$, where $(u, v) \in E$ iff $v$ occurs in $\gamma(u)$. The *node size* $|M|_N$ is defined as $|M|_N = |V|$ and the *edge size* is defined as $|M|_E = \sum_{v \in V} |\gamma(v)|$. When we simply refer to the *size* of $M$ we speak of the edge size; see below for a justification of this.

Two ordered $\Sigma$-labelled or $\mathcal{F}$-labelled multigraphs $M_1 = (V_1, \gamma_1, \lambda_1)$ and $M_2 = (V_2, \gamma_2, \lambda_2)$ are isomorphic if there exists a bijection $f : V_1 \to V_2$ such that for all $v \in V_1$: $\gamma_2(f(v)) = f(\gamma_1(v))$ and $\lambda_2(f(v)) = f(\lambda_1(v))$ (here we implicitly extend $f$ to a morphism $f : V_1^* \to V_2^*$). We do not distinguish between isomorphic multigraphs.

An *ordered $\Sigma$-labelled (resp. $\mathcal{F}$-labelled) DAG* is an ordered $\Sigma/\mathcal{F}$ labelled multigraph $d = (V, \gamma, \lambda)$ such that the underlying graph $G_d$ is acyclic. The nodes $r \in V$ for which there are no $v \in V$ such that $(v, r)$ is an edge in $G_d$ are called the *roots* of $d$. If the root is unique, we call the DAG a ordered $\Sigma$-labelled (resp. $\mathcal{F}$-labelled) *rooted* DAG. The nodes $l \in V$ such that $\gamma(l) = \varepsilon$ are called the *leaves* of $d$, nodes $m \in V$ such that $|\gamma(m)| = 1$ are called *monadic nodes* and nodes $m \in V$ such that $|\gamma(m)| = 2$ are called *binary nodes*.

This offers an alternate definition of unranked trees (recall that our trees are rooted and ordered): An *unranked tree* over $\Sigma$ is an ordered $\Sigma$-labelled ordered rooted DAG $t = (V, \gamma, \lambda)$ such that the underlying graph $G_d$ is a rooted tree, i.e., if, in the concatenation of all strings $\gamma(v)$ for $v \in V$, the root node does not occur and every other node occurs exactly once. Analogously ranked trees may also be defined this way.

Let $d = (V, \gamma, \lambda)$ be an ordered $\Sigma/\mathcal{F}$-labelled DAG. With every node $v \in V$ we associate a tree $\text{eval}_d(v) \in \mathcal{T}(\Sigma)$ resp. $\mathcal{T}(\mathcal{F})$ inductively as follows: If $\lambda(v) = f$ and $\gamma(v) = v_1 \dots v_n$, then

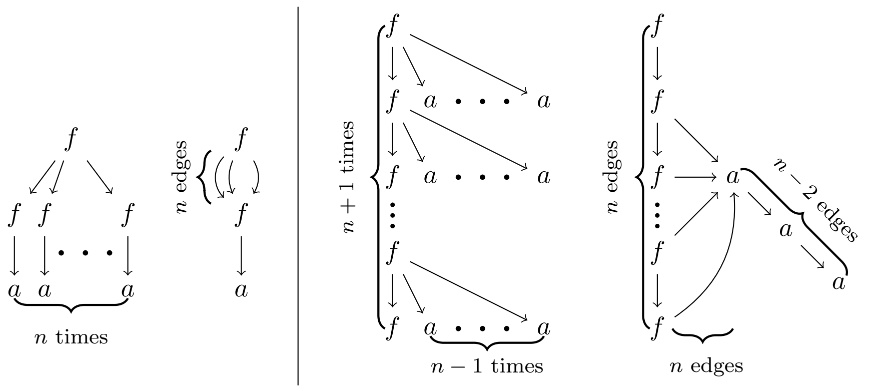
$$\text{eval}_d(v) = f(\text{eval}_d(v_1), \dots \text{eval}_d(v_n)).$$

Intuitively, $\text{eval}_d(v)$ is the tree obtained by unfolding $d$ starting at the node $v$. If $t = (V, \gamma, \lambda)$ is an unranked tree and $v \in V$, then $\text{eval}_t(v) = t/v$ is the subtree of $t$ rooted at $v$.

An ordered tree can be compacted by representing occurrences of repeated subtrees only once. Several edges then point to the same subtree, which we call a *repeated subtree*, thus making the tree an ordered DAG. It is known that the minimal DAG of a tree is unique and can be constructed in linear time [DST80]. Formally, we define the minimal DAG $\text{dag}(d)$ for every $\Sigma/\mathcal{F}$-labelled ordered DAG.

**Definition 2.4.** Let $d = (V, \gamma, \lambda)$ be a $\Sigma$ or $\mathcal{F}$-labelled ordered DAG. The *minimal DAG* of $d$, $\text{dag}(d)$ is defined as

$$\text{dag}(d) = (\{\text{eval}_d(v) \mid v \in V\}, \gamma', \lambda')$$



The tree family $(t_n)_{n\geq 0}$ and their DAGs

The tree family $(s_n)_{n\geq 0}$ and their BDAGs

Figure 2.2: A DAG and a BDAG-example

where $\lambda'(f(t_1,\ldots,f_n)) = f$ and $\gamma'(f(t_1,\ldots,t_n)) = t_1,\ldots,t_n$. Thus the nodes of dag($d$) are the different trees represented by the unfoldings of the nodes of $d$.

Unless otherwise specified, when we speak of *the DAG* of a tree we mean the *minimal DAG*.

The internal structure of the nodes of dag($d$) (which are trees in our definition) has no influence on the size of dag($d$), which is still defined to be the number of its edges. In general we cannot recover $d$ from dag($d$): For instance if $d$ is the disjoint union of two copies of the same tree $t$, then dag($d$) = dag($t$), but this will not be a problem. Indeed, we use DAGs only for the compression of forests consisting of different trees. Such a forest can be reconstructed from its minimal DAG. Note also that if $d$ is a rooted DAG, then dag($d$) is also rooted and we have eval(dag($d$)) = eval($d$).

As mentioned before, for many tree-processing formalisms it is useful to deal with ranked trees instead of unranked trees. Thus we define the *(minimal) binary DAG (BDAG)* of an unranked tree as follows:

**Definition 2.5.** Let $t = (V, \gamma, \lambda)$ be an unranked tree. The *(minimal) BDAG* of $t$, bdag($t$) is defined as

$$\mathrm{bdag}(t) = \mathrm{dag}(\mathrm{fcns}(t)).$$

Chapter 4 discusses the worst-case comparison between the sizes of the DAG and the BDAG of an unranked tree.

## On the definition of the size of the (B)DAG.

**Example 2.6.** Consider the tree family $t_n = f(a,\ldots,a)$ over the unranked alphabet $\Sigma = \{f, a\}$, where the $n^{\text{th}}$ tree has $n$ $a$-labelled children. Then the minimal DAG of $t_n$ has $n$ edges, while it only has two nodes (note that a DAG with $m$ edges has at most $m + 1$ nodes).

Thus there exists a tree family in which the difference between the number of nodes and the number of edges in the DAG becomes maximal (constant vs. linear in the input size), whereas the reverse is not possible. Hence we will be mainly interested in the number of *edges* a DAG contains, especially when dealing with worst-case sizes. On the other hand, as we will see in Chapter 3, the average number of edges and the average number of nodes in the DAG of a tree are only apart by a constant factor. Because the mathematical equations are somewhat simpler when counting nodes, we choose to deal with the number of nodes in a DAG when evaluating the average case.

**Example 2.7.** Consider the tree families $(t_n)_{n\geq 0}$ and $(s_n)_{n\geq 0}$ from Figure 2.2. The following table provides an overview over the tree size, the DAG size and the BDAG size of the families.

| | $(t_n)_{n\geq 0}$ | $(s_n)_{n\geq 0}$ |
|---|---|---|
| Tree size | $2n$ | $n^2$ |
| DAG size | $n+1$ | $n^2$ |
| BDAG size | $2n$ | $3n-2$ |

This example shows that (i) the size of the DAG can be half the size of the BDAG, and that (ii) the size of the BDAG can be quadratically smaller than the size of the DAG.

**DAGs share repeated trees, BDAGs repeated subtree sequences.** The minimal DAG of a tree $t$ contains every subtree exactly once. As a consequence, the number of nodes in $\text{dag}(t)$ equals the number of different subtrees $t/v$ of $t$.

On the other hand, BDAGs share *repeated subtree sequences*: For a node $v$ of a (ranked or unranked) tree $t \in \mathcal{T}(\Sigma)$ define $\text{sibseq}(v) \in \mathcal{T}(\Sigma)^*$ as the sequence of subtrees rooted at $v$ and all its right siblings. More formally, if $v$ is the root of $t$, then $\text{sibseq}(v) = t$. Otherwise, let $u$ be the parent node of $v$ and let $w_1 \dots w_m v v_1 \dots v_n$ be the sequence of child nodes of $u$. Then

$$\text{sibseq}(v) = (t/v)(t/v_1)\dots(t/v_n).$$

The next lemma follows directly from the definitions of the BDAG and sibseq:

**Lemma 2.8.** *The number of nodes of bdag($t$) is equal to the number of different sibling sequences sibseq($v$), for all $v \in V$.*

We treat the number of edges in $\text{dag}(t)$ and $\text{bdag}(t)$ in more detail in Section 3.5 and in Chapter 4.

**The reverse binary DAG.** Analogously to the first-child/next-sibling encoding of a tree $t$ one may also define the *last-child/previous-sibling* (LCPS) of $t$, in which we replace the edge set of $t$ by (right) edges pointing from each node to its last child and (left) edges pointing from each node to its previous sibling. Using this encoding, one may also define the *reverse binary DAG* (RBDAG) of $t$ as $\text{dag}(\text{lcps}(t))$. Though it is clear that in general the RBDAG shares the same worst-case and average-case size as the BDAG, the encoding may nevertheless be advantageous if the nodes share "sibling prefixes". Take for instance an XML-specification, in which every *book* is required to carry certain information (e.g. *author, title, year* etc.), but may also require additional information (e.g. *edition* or *run*). Typically those optional data will come last, in which case it is better to use the reverse binary DAG. This was actually true for test data used in [BLMN15]: The reverse binary DAGs were on average smaller than the BDAGs of the trees contained in the test data.

## 2.3 Grammars

Formal grammars and their generalization to trees are widely used in computer science. For a general introduction to tree grammars we refer to [CDG⁺07].

### 2.3.1 SLP grammars

A *straight-line program* (SLP) is a context-free string grammar that produces exactly one string. Formally, it is a tuple $\mathbb{A} = (N, \Sigma, P, S)$, where $N$ is a finite set of nonterminals, $\Sigma$ is a finite set of terminal symbols ($N \cap \Sigma = \emptyset$), $S \in N$ is the start nonterminal and $P$ is a finite set of productions of the form $A \to w$ with $A \in N$, $w \in (N \cup \Sigma)^*$ such that

- for every nonterminal $A$ there exists exactly one rule in $P$ with left-hand side $A$ and
- the binary relation $\{(A, B) \in N \times N \mid (A \to w) \in P,\ B \text{ occurs in } w\}$ is acyclic.

Every nonterminal $A \in N$ produces a unique string $\mathrm{val}_{\mathbb{A}}(A) \in \Sigma^*$. The string defined by $\mathbb{A}$ is $\mathrm{val}(\mathbb{A}) = \mathrm{val}_{\mathbb{A}}(S)$. We omit the subscript $\mathbb{A}$ when it is clear from the context. The *size* of the SLP $\mathbb{A}$ is $|\mathbb{A}| = \sum_{(A\to w)\in P} |w|$. An SLP is in *Chomsky normal form* if every production is either of the form $A \to a$ with $a \in \Sigma$ or $A \to BC$, where $B, C \in N$. The Chomsky normal form of a grammar may be computed in linear time, see e.g. [Loh14].

**Example 2.9.** Let $\mathbb{A}_1 = (N, \Sigma, P, S)$ with $N = \{S, A_1, \dots, A_9\}$, $\Sigma = \{a\}$ and $P = \{S \to A_9A_9,\ A_9 \to A_8A_8,\ \dots A_1 \to aa\}$. Then $\mathrm{val}(S) = a^{1024}$ and $|\mathbb{A}| = 20$.

**Example 2.10.** For $n \geq 3$, let $\mathbb{A}_n = (N, \Sigma, P, A_n)$ with $N = \{A_1, \dots, A_n\}$, $\Sigma = \{a, b\}$ and

$$P = \{A_n \to A_{n-1}A_{n-2},\ A_{n-1} \to A_{n-2}A_{n-3},\ \dots, A_2 \to a,\ A_1 \to b\}.$$

For example $\mathrm{val}(\mathbb{A}_6) = abaababa$. The words $a, b, \mathrm{val}(\mathbb{A}_3), \mathrm{val}(\mathbb{A}_4), \dots$ are known as *Fibonacci words*. While $|\mathbb{A}_n| = 2n - 2$, the lengths of the Fibonacci words are enumerated by the well-known *Fibonacci numbers* (series A000045 in the *Online Encyclopedia of Integer Sequences*, OEIS), which grow exponentially.

It is NP-complete to check for a given string $s$ and an integer $k$ whether there exists an SLP $\mathbb{A}$ of size at most $k$ such that $\mathrm{val}(\mathbb{A}) = s$ [CLL⁺05]. Thus, unless P = NP, there is no polynomial-time algorithm that, given a string $s$, outputs a smallest grammar $\mathbb{A}$ for $s$. Hence the focus of grammar-based compression algorithms lies on algorithms with good *approximation ratios*, where the approximation ratio of a grammar-based compressor $\mathbb{G}$ is defined as the function

$$\alpha_{\mathbb{G}}(n) = \max_s \frac{\text{size of SLP produced by } \mathbb{G} \text{ for } s}{\text{size of smallest SLP for } s}$$

where the maximum is taken over all strings of size $n$ over an arbitrary alphabet. The currently best-known approximation ratio of a polynomial-time compression algorithm is $\mathcal{O}(\log(|s|/g))$, where $g$ is the size of a smallest grammar for $s$. There are several linear-time compressors that achieve this approximation ratio, among them [Jez14, CLL$^+$05, Ryt03].

While not having the best approximation ratio, the compression algorithm *Re-Pair* [LM99] shows good heuristic results. We briefly discuss the algorithm, as it is the base for the tree compression algorithm *TreeRePair*. For a longer treatment we refer to the aforementioned paper or to [CLL$^+$05].

**Re-Pair.** Let $s \in \Sigma^*$ be a string over an alphabet $\Sigma$. Re-Pair recursively searches $s$ for a maximal set $M$ of non-overlapping occurrences of a factor (or *pair*) $\alpha\beta$ of length two (these could be from the terminal alphabet or nonterminals) in $s$ and replaces each occurrence of $\alpha\beta$ from $M$ in $s$ by a new nonterminal. This is done until no pair has a frequency greater than 1. While a naïve implementation naturally needs quadratic time, N. Larsson and A. Moffat show in [LM99] how Re-Pair can be implemented in linear time.

**Example 2.11.** Consider the string $s = a^{1024}$. Using Re-Pair to compress this string produces the grammar $\mathbb{A}$ from Example 2.9 of size 20.

### 2.3.2 TSLP grammars

Like its string counterpart, a *tree straight-line program* (TSLP) is a tree grammar that produces exactly one tree.

In the following, let $N$ and $\mathcal{F}$ be ranked alphabets, let $\Sigma$ be an unranked alphabet and let $\mathcal{X} = \{x_1, x_2, \dots\}$ be a countably infinite set, which we treat as symbols of rank 0. We assume that all alphabets are pairwise disjunct. Symbols from $N$ are called *nonterminals*, elements from $\mathcal{F}$ and $\Sigma$ are called *terminals* and elements from $\mathcal{X}$ are called *parameters*.

Depending on whether we treat ranked or unranked trees, we consider trees from $\mathcal{T}(N \cup \mathcal{F} \cup \mathcal{X})$ or $\mathcal{T}(N \cup \Sigma \cup \mathcal{X})$, respectively.

A *ranked tree straight-line program* is a tuple

$$\mathbb{A} = (N, \mathcal{F}, P, S),$$

where $N$ and $\mathcal{F}$ are as above, $S \in N_0$ is the start nonterminal and $P$ is a finite set of productions of the form $A(x_1, \dots, x_n) \to t$ (which is also briefly written as $A \to t$), where $n \geq 0$, $A \in N_n$ and $t \in \mathcal{T}(N \cup \mathcal{F} \cup \{x_1, \dots, x_n\})$ is a tree in which every parameter $x_i$ ($1 \leq i \leq n$) occurs at most once[1], such that

- For every $A \in N_n$ there exists exactly one production of the form

$$A(x_1, \dots, x_n) \to t$$

  and

- the binary relation $\{(A, B) \in N \times N \mid (A \to t) \in P, B \text{ is a label in } t\}$ is acyclic.

[1] If this restriction is dropped, one speaks of *nonlinear* TSLP. These can achieve doubly exponential compression, but have the disadvantage that many algorithmic problems become intractable, see e.g. [LM06]. We will not further consider nonlinear TSLPs.

We define *unranked tree straight-line programs* $\mathbb{A} = (N, \Sigma, P, S)$ accordingly.

As with SLPs, these conditions ensure that every nonterminal $A \in N_n$ derives to exactly one tree $\mathrm{val}_{\mathbb{A}}(A) = t \in \mathcal{T}(N \cup \mathcal{F} \cup \{x_1, \ldots, x_n\})$ by using the productions in $P$ as rewriting rules. Without loss of generality, we assume that the parameters $x_1, \ldots, x_n$ appear in depth-first left-to-right ordering, e.g. $f(x_1, x_2)$ rather than $f(x_2, x_1)$. We define $\mathrm{val}(\mathbb{A}) = \mathrm{val}_{\mathbb{A}}(S)$ as the tree that is derived by the grammar. Again we omit the subscript $\mathbb{A}$ when it is clear from the context.

We call a TSLP *monadic* if every nonterminal has rank at most one. A TSLP may be converted to a monadic one in polynomial time with a size (see below for the size of a tree grammar) increase of $r$ [LMS12], where $r$ is the maximal rank of a nonterminal appearing in the grammar.

We call a TSLP *regular* if every nonterminal has rank zero (since it represents a regular tree grammar). In such grammars nonterminals only appear as leaves in the right-hand sides of the productions.

**Example 2.12.** Let $\mathbb{A} = (N, \Sigma, P, S)$, where $N = \{S, A, B\}$, $\Sigma = \{f, a, b\}$, and $P$ contains the productions

$$\begin{aligned} S &\to f(B(A(b), B(B(a)))), \\ B(x) &\to A(A(x)), \\ A(x) &\to f(x, a). \end{aligned}$$

The tree $\mathrm{val}(\mathbb{A})$ is depicted on the right-hand side.

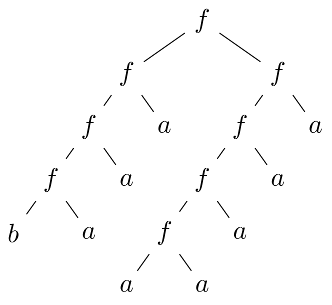


Figure 2.3: The tree $\mathrm{val}(\mathbb{A})$

Let $t$ be a tree and let $d$ be its minimal DAG. We can identify $d$ with a regular tree grammar $\mathbb{G}_d = (N, \mathcal{F}, P, S)$ as follows: Let $V = \{v_1, \ldots, v_n\}$ be the set of nodes in the DAG. For every node $v_i \in V$ we create a nonterminal $N_i$ of rank 0. Let $u_{i_1}, \ldots, u_{i_m}$ denote the successor nodes of $v_i$, let $R_{i_1}, \ldots, R_{i_m}$ be the nonterminals we created for the nodes $u_{i_1}, \ldots, u_{i_m}$ and let $f_i = \lambda(v_i)$ be the label of $v_i$. Then we add for each $v_i$ the production

$$N_i \to f_i(R_{i_1}, \ldots, R_{i_m})$$

to $P$. This defines a unique TSLP (up to renaming of the nonterminals).

**Example 2.13.** Consider the DAG $d_n$ of the tree $t_n$ from Example 2.7. Then a regular TSLP grammar for $d_n$ is

$$S \to f(\underbrace{A_1, \ldots, A_1}_{n \text{ many}}),\ A_1 \to f(A_2),\ A_2 \to a.$$

Note that the height of every right-hand side of a rule in $\mathbb{G}_d$ is either 0 or 1. As it is sometimes useful to consider rules of height 1 only, we may instead define the production for each $v_i$ as

$$N_i \to f_i(\alpha_{i1}, \ldots, \alpha_{im})$$

where

$$\alpha_{ij} = \begin{cases} R_{ij} & \text{if } u_{ij} \text{ has a successor node} \\ \lambda(u_{ij}) & \text{otherwise.} \end{cases}$$

Thus we eliminate all rules of the form $A \to a$ and all rules have height 1. We call such a grammar *reduced.* Note that a tree that consists of a root node only cannot have a reduced grammar; because such trees represent an uninteresting corner case we simply exclude them from our further discussion.

**Definition 2.14.** Let $t$ be a tree with at least two nodes. By the *DAG grammar* $\mathbb{G}_{\text{dag}}(t)$ of $t$ we refer to the reduced regular TSLP that corresponds to the minimal DAG of $t$.

As in the SLP case, a smallest TSLP for a given tree cannot be computed in polynomial time unless $\mathsf{P} = \mathsf{NP}$ (this is trivial since we can encode a string $w = w_1 \dots w_n$ by a monadic tree $t = w_1(\dots(w_n)))$. Thus the focus is once again on finding good approximation algorithms, either from a theoretical point of view (i.e. a good approximation ratio) or from a practical point of view. The algorithm *TTOG* from [JL14] is an example for the prior, having an approximation ratio of $\mathcal{O}(\log|t|)$ for a tree $t$, while *TreeRePair* [LMM13] is an example for the latter.

Next we briefly introduce *TreeRePair.* Though we will discuss the algorithm in more detail in Section 9.2, we include a brief description here to provide an example of a grammar-based tree compressor.

**TreeRePair.** Let $t \in \mathcal{T}(\mathcal{F})$ be a ranked tree[2] of size $n$. TreeRePair recursively replaces the most frequent *digram* in $t$ by a new symbol until every digram appears only once. Here, a digram is for trees what a pair is for strings: It is defined as the triple $[\alpha, i, \beta]$, where $i \in \mathbb{N}$ is an integer and $\alpha, \beta \in \mathcal{F} \cup N$ are either terminal symbols or nonterminals that have been introduced in a previous replacement step. Care must be taken for digrams of the form $[\alpha, i, \alpha]$, as they could be overlapping. An *occurrence* of a digram $d = [\alpha, i, \beta]$ is a node $v \in t$ such that $\lambda(v) = \alpha$ and $\lambda(v') = \beta$, where $v'$ is the $i^{\text{th}}$ child of $v$.

One replacement step is done as follows: Let $\text{rank}(\alpha) = n$ and $\text{rank}(\beta) = m$ and assume that $d = [\alpha, i, \beta]$ is a most frequent digram, i.e., the maximal set of non-overlapping occurrences of $d$ is maximal among the set of non-overlapping digram occurrence sets.

We introduce a new nonterminal $A$ of rank $n + m - 1$ together with the associated rule

$$A(x_1, \dots, x_{n+m-1}) \to \alpha(x_1, \dots, x_{i-1}, \beta(x_i, \dots, x_{m+i-1}), x_{m+i}, \dots, x_{n+m-1}).$$

Then every occurrence of $[\alpha, i, \beta]$ is replaced by $A$. E.g. if at one digram occurrence the children of $\alpha$ are labelled $g_1, \dots, g_{i-1}, \beta, g_{i+1}, \dots, g_n$ and the children of $\beta$ are labelled $h_1, \dots, h_m$, we now have the pattern
$A(g_1, \dots, g_{i-1}, h_1, \dots, h_m, g_{i+1}, \dots, g_n)$.

Again a naïve implementation takes quadratic time, but TreeRePair can be implemented in linear time [LMM13]. In Chapter 9 we use a modified version of TreeRePair to show a surprising application of compression, namely *accelerated computation by compression.* Here, we use TreeRePair (with a modified digram counter) to compress term rewriting systems before transforming it to a formula in propositional logic, which is then attempted to be solved by a SAT solver. With the help of this intermediate step the resulting Boolean formula is smaller, and thus a SAT solver can find a solution faster.

[2]For unranked trees, one can conveniently consider the FCNS-encoding of $t$ or introduce for every symbol $f \in \Sigma$ a new symbol for every different number of children $f$ has.

**On the size of the grammar.** Following the argumentation in [JL14], consider a rule of the form $A(X_1, \ldots, X_n) \to f(X_1, X_{i-1}, a, X_i, \ldots X_n)$. If this rule is part of a ranked TSLP, we may conveniently encode the right-hand side as $(f, (i, a))$. On the remaining positions we simply list the $n$ parameters (recall that we assume the ordering of the parameters to be generic, i.e. small to large). In general, we may encode each right-hand side of a production by specifying for each node the non-parameter children together with their positions. Since these are bounded, we may define the size $|\mathbb{A}|$ of a grammar as the total number of non-parameter nodes in all right-hand sides.

On the other hand, such a characterization is not possible when the rule is part of an unranked TSLP because a every terminal may appear with a different number of children. Thus, for unranked TSLPs we count the parameter nodes as well.

This bears a simple problem: Let $t$ be a unranked tree and $t' = \text{fcns}(t)$. Then a grammar for $t$ counts parameter nodes, whereas a grammar for $t'$ does not. Because this can be confusing, we will always specify whether a grammar size considers parameter nodes or not.

Note that the DAG and the BDAG grammar of a tree are regular, hence they both do not have any parameter nodes in the first place.

# Chapter 3

# The average-case size of the DAG

In this chapter we analyze the average sizes (both node and edge size) of the DAG of unranked trees and of full binary trees.

To ease the discussion, we first concentrate solely on the average node size of full binary DAGs over a unary alphabet. Later we show how to extend the result to larger alphabets, to unranked trees as well as to the average edge sizes. This has the advantage that the amount of notation stays slim and we can focus on the main difficulties of the proof.

The proof method we use is best described in the book *Analytic Combinatorics* [FS09]: We first derive the so-called *generating function* $N(z)$ that counts the total number of nodes of all DAGs of trees of size $n$. We then analyze the dominant singularity $z_0$ of $N(z)$ in the complex plane: We show that close to $z_0$, $N(z)$ can be approximated by a function with known coefficient asymptotics. We use this to prove the main theorem of this chapter:

**Theorem 3.1.** *The average number of nodes in the minimal DAG of a full binary tree of size $n$ satisfies*

$$\bar{N}_n = \frac{N_n}{B_n} = 2\kappa \frac{n}{\sqrt{\ln n}} \left(1 + \mathcal{O}\left(\frac{1}{\ln n}\right)\right) \quad \textit{with} \quad \kappa = \sqrt{\frac{\ln 4}{\pi}}.$$

The rest of this chapter is structured as follows. In Section 3.1 we explain background from the book *Analytic Combinatorics* [FS09] necessary to derive the generating function $N(z)$ that we are interested in, which we derive in Section 3.2. Then we switch again to a background section (Section 3.3), this time devoted to singular analysis and the central *transfer theorem.* After that we prove the main result (Section 3.4). Then we extend the results to larger alphabets, unranked trees and edge sizes (Section 3.5). Finally we discuss alternative proof strategies and open problems (Sections 3.6 and 3.7).

The results from this chapter are mostly an extension of [FSS90], where the authors sketched a proof for the case of full binary trees over a unary alphabet. We first provide a full proof of the result and then show extensions of it. These were first presented in [BLMN15].

## 3.1 Generating functions and the Catalan numbers

**Definition 3.2.** Let $\mathcal{C}$ be a countable set and let $|\cdot| : \mathcal{C} \to \mathbb{N}$ be a size function. If the number of objects of every given size is finite, we call $(\mathcal{C}, |\cdot|)$ a *combinatorial class*. The sequence $C_n := \{c \in \mathcal{C} \mid |c| = n\}$ is called the *counting sequence* of $\mathcal{C}$.

Let $(C_n)_{n\geq 0}$ be the counting sequence of a combinatorial class $\mathcal{C}$. Then we define its *(ordinary) generating function* $C(z)$ as

$$C(z) = \sum_{n\geq 0} C_n z^n.$$

We further denote $[z^n]C(z) := C_n$. Let $\uplus$ denote the *disjoint union operator*. For combinatorial classes $\mathcal{B}$ and $\mathcal{C}$ with their respective generating functions $B(z)$ and $C(z)$, we have the following constructors:

1. Disjoint Union: If $\mathcal{A} = \mathcal{B} \uplus \mathcal{C}$, then $A(z) = B(z) + C(z)$.
2. Cartesian Product: If $\mathcal{A} = \mathcal{B} \times \mathcal{C}$, then $A(z) = B(z) \cdot C(z)$.
3. Sequence Construction: If $B_0 = 0$ and $\mathcal{A} = \{\emptyset\} \uplus \mathcal{B} \uplus \mathcal{B} \times \mathcal{B} \uplus \ldots$, then $A(z) = \frac{1}{1-B(z)}$.

The proofs and other constructors can be found in [FS09].

**Binary trees and the Catalan numbers.** *Full binary trees*, i.e. trees in which every node has zero or two successors, are conveniently counted by the number of internal nodes. We denote by $\mathcal{F}$ the combinatorial class of such trees. By simply removing all leaf nodes and defining the size as the total number of nodes, one gets a bijection to binary trees (also called *pruned binary trees*), which we denote by $\mathcal{B}$. It follows that the DAG of a full binary tree has as many (internal) nodes as the DAG of its pruned version. This is not true for the number of edges; while the DAG of a full binary tree trivially has twice as many edges as internal nodes, it is not as simple for binary trees (we discuss the edge size of the latter in Section 3.5).

Because of this bijection, the counting sequences and the associated generating functions for $\mathcal{F}$ and $\mathcal{B}$ are the same.

Following the constructors from above, a full binary tree is a single node of size 0 or an internal node with two full binary trees attached. This yields the following equation for the generating function $B(z)$ of the class of full binary trees:

$$B(z) = 1 + zB^2(z)$$

Solving this equation for $B(z)$ yields the following lemma.

**Lemma 3.3.** *The generating function $B(z)$ of (full) binary trees is*

$$B(z) = \frac{1-\sqrt{1-4z}}{2z}. \tag{3.1}$$

*Equivalently, the number of full binary trees of size $p$ is*

$$B_p = \frac{1}{p+1}\binom{2p}{p}. \tag{3.2}$$

Equation (3.2) can be derived by Taylor analysis from Equation (3.1). The numbers $(B_p)_{p\geq 0}$ are the well-known *Catalan numbers* (series A000108 in the *Online Encyclopedia of Integer Sequences*, OEIS) that count many different combinatorial classes, e.g. Dyck paths, polygon triangulations or the number of well-formed bracket expressions. See [Sta99] for many other interpretations of the Catalan numbers.

## 3.2 Exact counting

The goal of this section is to obtain the generating function

$$N(z) = \sum_{n\geq 0} N_n z^n$$

where $N_n := \sum_{t\in\mathcal{B}_n} |\mathrm{dag}(t)|_N$ is the accumulated number of nodes of the DAGs of full binary trees. The expected size of the DAG of a tree of size $n$ is then obtained by dividing $N_n$ by $B_n$. We denote this quantity by

$$\bar{N}_n = \frac{N_n}{B_n}.$$

Because the minimal DAG of a tree contains every different subtree exactly once, the size of the DAG of a tree can equivalently be defined as the number of different subtrees it contains. Hence, instead of summing over the sizes of DAGs of trees of size $n$, we may also count how often trees $t \in \mathcal{B}$ occur in $\mathcal{B}_n$. Thus, with

$$\mathbb{1}_{u,t} := \begin{cases} 1 & \text{if } u \text{ is a subtree of } t \\ 0 & \text{otherwise} \end{cases}$$

we get

$$N_n = \sum_{t\in\mathcal{B}_n} |\mathrm{dag}(t)|_N = \sum_{t\in\mathcal{B}_n}\sum_{u\in\mathcal{B}} \mathbb{1}_{u,t}.$$

Define $C_{n,u}$ as the number of full binary trees of size $n$ that contain a given tree $u$. Then

$$N_n = \sum_{u\in\mathcal{B}} C_{n,u}.$$

Now let $u, v \in \mathcal{B}$ be two trees of equal size $p$. Then $C_{n,u} = C_{n,v}$ for every $n$, because there is a bijection between the two sets, which replaces each occurrence of $u$ by an occurrence of $v$, and vice versa. Hence we will also write $C_{n,p}$ instead of $C_{n,u}$ (with $p = |u|$). Thus we get for $N_n$:

$$N_n = \sum_{t\in\mathcal{B}_n} |\mathrm{dag}(t)|_N = \sum_{u\in\mathcal{B}} C_{n,u} = \sum_{p\geq 0} B_p C_{n,p} \tag{3.3}$$

Equation (3.3) can easily be generalized to other tree classes: For a tree class $\mathcal{T}$ with generating function $T(z)$, let $C_{n,p}^{\mathcal{T}}$ be the number of trees from $\mathcal{T}$ of size $n$ that contain a given tree $t \in \mathcal{T}$ of size $p$. Then

$$N_n = \sum_{p \geq 0} T_p C_{n,p}^{\mathcal{T}}. \tag{3.4}$$

**Lemma 3.4.** *The generating function $C_p(z)$ of full binary trees that contain a given tree $u \in \mathcal{B}$ of size $p$ is*

$$C_p(z) = \frac{1}{2z}\left(\sqrt{1-4z+4z^{p+1}} - \sqrt{1-4z}\right). \tag{3.5}$$

*Proof.* We first determine the generating function $A_p(z)$ counting full binary trees that do *not* contain (or *avoid*) a given tree $u$ of size $p$. Much like with the generating function $B(z)$, a full binary tree $t$ that avoids a given full binary tree $u$ of size $p$ is either a single node or a root node to which two $u$-avoiding full binary trees are attached. However, we must still exclude the tree $t = u$, which is included in the above recursive description. We thus get the following equation:

$$A_p(z) = 1 + zA_p^2(z) - z^p,$$

which yields

$$A_p(z) = \frac{1-\sqrt{1-4z+4z^{p+1}}}{2z}.$$

Using $C_p(z) = B(z) - A_p(z)$, this proves the lemma. □

**Theorem 3.5.** *The generating function of the accumulated number of nodes of minimal DAGs of full binary trees is*

$$\begin{aligned} N(z) &= \sum_{n \geq 0} N_n z^n \\ &= \frac{1}{2z}\sum_{p\geq 0} B_p \left(\sqrt{1-4z+4z^{p+1}} - \sqrt{1-4z}\right), \end{aligned}$$

*where the numbers $B_p$ are given by* (3.2).

*Proof.* As shown in equation (3.3), we have $N_n = \sum_{p\geq 0} B_p C_{n,p}$. Hence

$$N(z) = \sum_{n\geq 0} N_n z^n = \sum_{p \geq 0} B_p C_p(z).$$

□

*Remark.* Exact numbers can now be derived using a Taylor analysis of the power series $N(z)$. This way we know that

$$N(z) = 1 + z + 4z^2 + 14z^3 + 50z^4 + 180z^5 + 660z^6 + O(z^7).$$

Using e.g. $B_6 = 132$, this shows that the average DAG size of full binary trees of size 6 is 5. In [FSS90] the authors also provide a closed formula for $N_n$ via an Inclusion-Exclusion argument.

## 3.3 Complex analysis and singularity analysis

Now that we know the generating function $N(z)$, we want to use it to understand the asymptotic behavior of the sequence $(N_n)_{n\geq 0}$.

The crucial point here is to understand $N(z)$ not only as a formal power series, but as a function in the complex plane. If it has a radius of convergence greater than zero, we can treat it as an analytic function in its domain. It turns out that there is a general correspondence between

- the behavior of a power series close to its *dominant* singularities and
- the asymptotic growth of its coefficients.

We call a singularity *dominant* if it is a singularity with the smallest modulus among all singularities of the function. A generating function may have several dominant singularities on its radius of convergence; since this is not the case for the functions we are interested in, we restrict our discussion to generating functions with a unique dominant singularity. Furthermore, *Pringsheim's Theorem* [FS09, p. 240] states that a power series with non-negative coefficients and radius of convergence $\rho$ has a singularity at $z = \rho$. So, for the generating functions discussed here, if the generating function has a radius of convergence $\rho$, we may assume that $z = \rho$ is the (unique) dominant singularity.

Let $A(z) = \sum_{n\geq 0} A_n z^n$ be a generating function with radius of convergence $\rho$. In [FS09, p. 227] the correspondence from above is enunciated in two general principles:

**First principle of coefficient analysis.** *The* location *of a function's singularity dictates the* exponential growth *of its coefficients.*

In other words, by setting $A_n = K^n \Theta(n)$ with $\Theta(n)$ a subexponential function, we have

$$\limsup_{n\to\infty} |\Theta(n)|^{1/n} = 1.$$

The intuition behind this principle is that, if $A(z)$ has a singularity at $z = \rho$, then

$$A(\rho) = \sum_{n\geq 0} A_n \rho^n$$

does not converge. Hence $A_n \sim \rho^{-n}$.

Note that the rescaling rule

$$[z^n]A(z) = \rho^{-n}[z^n]A(\rho z)$$

allows us to restrict our discussion to functions that have a radius of convergence equal to 1.

**Second principle of coefficient analysis.** *The* nature *of a function's singularity determines the* subexponential factor $\Theta(n)$ *of its coefficients.*

To get an intuition for this principle, it is helpful to consider some functions singular at $z = 1$.

As is visible from Table 3.1 and Figure 3.1, faster growing functions correspond to faster growing coefficients and square roots or logarithms in the functions induce such terms in the coefficients.

| | Function | Coefficients (asympt.) |
|---|---|---|
| $f_1$ | $\frac{1}{1-z}\log\frac{1}{1-z}$ | $\log n$ |
| $f_2$ | $\frac{1}{1-z}$ | $1$ |
| $f_3$ | $1-\sqrt{1-z}$ | $\frac{1}{2\sqrt{\pi n^3}}$ |

Table 3.1: Some functions with a singularity at $z = 1$. Table adapted from [FS09, p. 377].

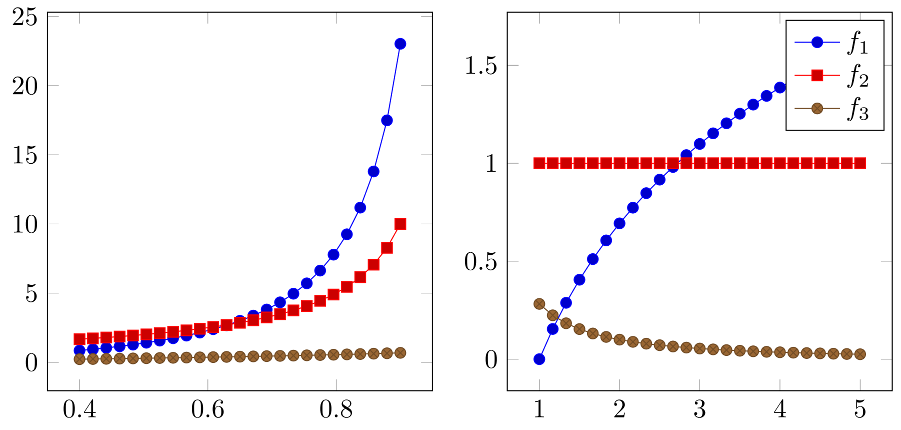


Figure 3.1: The left image plots the behavior of the functions $f_1$, $f_2$, $f_3$ close to their dominant singularity and the right image plots the growth of their coefficients.

For some functions, these two principles suffice for a complete asymptotic expansion. For example, by knowing the behavior of $f(z) = \sqrt{1-z}$ close to $\rho = 1$, we can expand the Catalan numbers as [FS09, p. 388]

$$B_n = \frac{4^n}{(n+1)\sqrt{\pi n}}\left(1 - \frac{1}{8n} + \frac{1}{128n^2} + \frac{5}{1024n^3} + \mathcal{O}\left(\frac{1}{n^4}\right)\right).$$

Yet for some functions the nature of the singularity might not be precisely known. This might be the case if the generating function is merely given by an infinite sum (as is the case for $N(z)$) or by an implicit equation. In those cases one can perform a so-called *singularity transfer*, in which one tries to transfer an approximation of a function near its dominant singularity into an asymptotic approximation of its coefficients:

Close to the singularity $z_0$, one splits the function $f(z)$ one wishes to analyze into two parts,

$$f(z) \underset{z\to z_0}{=} \sigma(z) + \mathcal{O}(\tau(z))$$

where $\sigma, \tau$ are functions with known asymptotic coefficient behavior (and $\tau \in o(\sigma)$), to obtain an estimate of the form

$$f_n \underset{n\to\infty}{=} \sigma_n + \mathcal{O}(\tau_n).$$

Theorem 3.12 below states under which circumstances such a split is permissible.

### 3.3.1 Complex analysis

A *region* in the complex plane $\mathbb{C}$ is defined as an open, connected subset of $\mathbb{C}$.

**Definition 3.6.** Let $G \subseteq \mathbb{C}$ be a region and let $f(z)$ be defined over $G$.

1. We say that $f(z)$ is *analytic* or *holomorphic* at a point $z_0 \in G$ if there exists an open disc $D \subseteq G$ centered at $z_0$ such that $f(z)$ can be written as
$$f(z) = \sum_{n \geq 0} f_n (z - z_0)^n$$
for all $z \in D$. We say that $f(z)$ is *analytic* or *holomorphic in* $G$ if it is analytic at every point of $G$.

2. We say that a function $g(z)$ is *meromorphic* at a point $z_0$, if, for $z$ in a neighborhood of $z_0$, $z \neq z_0$, $g(z)$ can be represented as the quotient of two holomorphic functions. In that case, $g(z)$ admits near $z_0$ an expansion of the form
$$g(z) = \sum_{n \geq -M} g_n (z - z_0)^n.$$
If $f_{-M} \neq 0$ and $M \geq 1$, then we say that $f(z)$ has a *pole* of order $M$ in $z_0$. The coefficient $f_{-1}$ is called *residuum* of $f$ at $z = z_0$ and is denoted by
$$\mathrm{Res}[f(z); z = z_0].$$
Again, if $g(z)$ is meromorphic for every point in $G$, we say that $g(z)$ is *meromorphic in* $G$.

A famous property of holomorphic functions is the so-called *null integral property.* It states that for holomorphic functions, integrals over *simple loops* in a region are zero (a *loop* is a curve that can continuously be rectified to a single point *within* the region; it is called *simple* if the path is non-crossing). The next theorem is known as *Cauchy's residue theorem.*

**Theorem 3.7** (Cauchy's residue theorem)**.** *Let $f(z)$ be meromorphic in $G$ and let $\gamma$ be a positively oriented simple loop in $G$, on which $f(z)$ is holomorphic. Then*
$$\frac{1}{2\pi i} \int_\gamma f(z)\, dz = \sum_s Res[f(z); z = s],$$
*where the sum is extended to all poles of $f(z)$ encircled by $\gamma$. Furthermore we have*
$$f_n \equiv [z^n] f(z) = \frac{1}{2\pi i} \int_\gamma f(z) \frac{dz}{z^{n+1}}. \tag{3.6}$$

Equation (3.6) is known as *Cauchy's coefficient formula.* Basically, setting $\gamma(t) = e^{it}$, $t \in [0, 2\pi]$ and $p \in \mathbb{Z}$ the proof of Theorem 3.7 is a vast generalization of
$$\oint_\gamma z^p\, dz = \int_0^{2\pi} e^{pit} i e^{it}\, dt = \begin{cases} 2\pi i & \text{if } p = -1 \\ 0 & \text{otherwise.} \end{cases}$$
One first shows the theorem for isolated singularities. The general case then follows because the region can be decomposed into cells, each containing one

singularity only. This integral also explains the importance of the coefficient $h_{-1}$ of a meromorphic function $h(z)$. Another property of holomorphic functions is that the number of zeros is invariant under small perturbations. This is formally stated by *Rouché's Theorem* (see [FS09, p. 270] or [FB93, p. 173]):

**Theorem 3.8** (Rouché's Theorem)**.** *Let $\gamma$ be a simple closed curve in a region $G \subseteq \mathbb{C}$ and let $f(z)$ and $g(z)$ be two functions holomorphic in $G$.*

*If $|g(z)| < |f(z)|$ on the curve $\gamma$, then $f(z)$ and $f(z) + g(z)$ have the same number of zeros inside the interior domain delimited by $\gamma$.*

As stated before, the aim of the transfer theorem is to show that the function to be analyzed is "similar" to a certain elementary function with known coefficient estimates. Now we introduce such a family of elementary functions.

## 3.3.2 The standard function scale

**Theorem 3.9** (Standard Function Scale)**.** *Let $\alpha \in \mathbb{C} \setminus \{\mathbb{Z}_{\leq 0}\}$ and $\beta \in \mathbb{C}$. The coefficient of $z^n$ in the function*

$$f(z) = (1-z)^{\alpha} \left( \frac{1}{z} \log \left( \frac{1}{1-z} \right) \right)^{\beta}$$

*admits for large $n$ an asymptotic expansion in descending powers of* $\log n$,

$$f_n = [z^n]f(z) \sim \frac{n^{\alpha-1}}{\Gamma(\alpha)} (\log n)^{\beta} \left( 1 + \mathcal{O}\left( \frac{1}{\log n} \right) \right).$$

The proof of Theorem 3.9 can be found, first for $\beta = 0$ and then in the general case, in [FS09, pp. 381-385]. One first expresses the coefficient of $[z^n]$ by means of Cauchy's coefficient formula (Equation (3.6)), starting with the integration contour $\{|z| = 1/2\}$. Then one deforms the integration contour into a so-called *Hankel contour* (depicted in Figure 3.2 below). This contour is then split into many parts. One evaluates the dominant ones and bounds the others. The proof of Theorem 3.1 closely follows these lines.

*Remark.* The coefficient $z^{-1}$ is introduced in front of the logarithm because $\log(1/(1-z)) = z + \mathcal{O}(z^2)$. This way $f(z)$ is a power series in $z$, even when $\beta$ is not an integer. This factor does not affect the asymptotic expansion in a logarithmic scale near $z = 1$.

## 3.3.3 Singularity transfer

**Definition 3.10** ($\Delta$-Region)**.** Let $\phi$, $R \in \mathbb{R}$ with $R > 1$ and $0 < \phi < \frac{\pi}{2}$. Define a $\Delta-$*domain* as follows:

$$\Delta(\phi, R) = \{z \mid |z| < R, z \neq 1, |\arg(z-1)| > \phi\}.$$

A function is $\Delta-$analytic if it is analytic for a $\Delta-$domain.

We start with a transfer theorem for the error terms.

**Theorem 3.11.** *Let $\alpha, \beta$ be real numbers, and let $f(z)$ be a function that is $\Delta-$analytic. Assume that $f(z)$ satisfies in the intersection of a neighborhood of 1 with its $\Delta-$domain the condition*

$$f(z) = \mathcal{O}\left((1-z)^{\alpha}\left(\log\frac{1}{1-z}\right)^{\beta}\right).$$

*Then one has*

$$[z^n]f(z) = \mathcal{O}(n^{\alpha-1}(\log n)^{\beta}).$$

**Theorem 3.12.** *Let $f(z)$ be a function analytic at 0 and with a singularity at $\zeta$, such that $f(z)$ can be continued to a domain of the form $\zeta \cdot \Delta_0$ for a $\Delta$-domain $\Delta_0$, where $\zeta \cdot \Delta_0$ is the image of $\Delta_0$ under the mapping $z \to \zeta z$. Assume that there exist two functions $\sigma(z)$ and $\tau(z)$ with $\tau(z) \in o(\sigma(z))$,*

$$\sigma(z) = (1-z)^{-\alpha_\sigma}\left(\frac{1}{z}\log\left(\frac{1}{1-z}\right)\right)^{\beta_\sigma}$$

*and*

$$\tau(z) = (1-z)^{-\alpha_\tau}\left(\frac{1}{z}\log\left(\frac{1}{1-z}\right)\right)^{\beta_\tau}$$

*such that*

$$f(z) = \sigma(z/\zeta) + \mathcal{O}(\tau(z/\zeta)) \quad \text{as } z \to \zeta \text{ in } \zeta \cdot \Delta_0.$$

*Then the coefficients of $f(z)$ satisfy the asymptotic estimate*

$$f_n = \zeta^{-n} n^{\alpha_\sigma - 1}(\log n)^{\beta_\sigma} + \mathcal{O}(\zeta^{-n} n^{\alpha_\tau - 1}(\log n)^{\beta_\tau}).$$

Theorem 3.11 can be found in [FS09, Theorem VI.3] and Theorem 3.12 has been adapted from [FS09, Theorem VI.4]. We briefly summarize the process of singularity analysis.

1. *Preparation.* First we locate the dominant singularity $\zeta$ of $f(z)$ and make sure that the function is analytic in an appropriate $\Delta$-domain.

2. *Expansion.* We find functions $\sigma$, $\tau$ with known coefficient asymptotics such that, as $z \to 1$

$$f(z) = \sigma(z/\zeta) + \mathcal{O}(\tau(z/\zeta)).$$

3. *Transfer.* With the known coefficient asymptotics of $\sigma$ and $\tau$ we conclude that

$$f_n = \zeta^{-n}\sigma_n + \mathcal{O}(\zeta^{-n}\tau_n)$$

   for $n \to \infty$.

## 3.4 Proof of the main theorem

We now prove Theorem 3.1, which we re-enunciate for better readability.

**Theorem 3.1.** *The average number of nodes in the minimal DAG of a full binary tree of size n satisfies*

$$\bar{N}_n = \frac{N_n}{B_n} = 2\kappa \frac{n}{\sqrt{\ln n}} \left(1 + \mathcal{O}\left(\frac{1}{\ln n}\right)\right) \quad \textit{with} \quad \kappa = \sqrt{\frac{\ln 4}{\pi}}.$$

The proof of this theorem follows the three basic steps stated above. We thus provide the singular expansion of $N(z)$ close to its dominant singularity $\zeta = 1/4$:

**Proposition 3.13.** *The generating function $N(z)$ is analytic in the domain $D$ defined by $|z| < \frac{1}{2}$ and $z \notin [\frac{1}{4}, \frac{1}{2}]$. As $z$ tends to $\frac{1}{4}$ in $D$, one has*

$$N(z) = \frac{2\,\kappa}{\sqrt{(1-4z)\ln((1-4z)^{-1})}} + \mathcal{O}\left(\frac{1}{\sqrt{(1-4z)\ln^3((1-4z)^{-1})}}\right),$$

*where $\kappa$ is defined as in Theorem 3.1.*

Using this proposition, we use estimates of the coefficients of the standard function scale (Theorem 3.9) and the transfer theorem (Theorem 3.12) to obtain the asymptotic behavior of the accumulated node size of minimal DAGs of full binary trees of size $n$:

$$N_n = [z^n]N(z) = \frac{2\kappa}{\sqrt{\pi}} \frac{4^n}{\sqrt{n \ln n}} \left(1 + \mathcal{O}\left(\frac{1}{\ln n}\right)\right).$$

Since the Catalan numbers $B_n$ satisfy

$$B_n = \frac{4^n}{\sqrt{\pi n^3}} \left(1 + \mathcal{O}\left(\frac{1}{n}\right)\right),$$

this proves Theorem 3.1.

*Proof.* We first show that the function $N(z)$ is analytic in $D$. Then we split $N(z)$ into three parts. The splitting depends on a threshold integer $n \equiv n(z)$. Using a suitable choice for $n(z)$ and

$$\sigma(z) = 2\kappa\,(1-z)^{-1/2} \left(\ln \frac{1}{1-z}\right)^{-1/2} \quad \text{and} \quad \tau(z) = (1-z)^{-1/2} \left(\ln \frac{1}{1-z}\right)^{-3/2}$$

we show that one part behaves like $\sigma(z/4)$ as $z \to 1/4$ and the other two parts behave like $\tau(z/4)$.

**Step 1: $N(z)$ is analytic in $D$.** The analyticity of $C_p(z)$ (Equation (3.5)) in $D$ is implied by the following lemma. Recall that

$$C_p(z) = \frac{1}{2z}\left(\sqrt{1-4z+4z^{p+1}} - \sqrt{1-4z}\right).$$

**Lemma 3.14.** *Let*

$$d_p(z) = 1 - 4z + 4z^{p+1}.$$

*For $p \geq 2$, this polynomial has exactly one root of modulus less than $1/2$. This root is real and larger than $1/4$. The polynomial $d_0(z) = 1 - 4z$ has a root at $z = 1/4$. The polynomial $d_1(z) = (1-2z)^2$ has a double root at $1/2$. Furthermore $d_p(z)$ does not vanish in $D$.*

*More generally, the rational function $u_p := 4z^{p+1}/(1-4z)$ does not take any value from $(-\infty, -1]$ for $z \in D$.*

*Proof.* To prove the existence of a unique root of modulus less than $1/2$, we apply Rouché's theorem (Theorem 3.8), using

$$g_p(z) = 4z^{p+1}, \quad f(z) = 1 - 4z \quad \text{and} \quad \gamma = \{|z| = 1/2\}.$$

Then $|g_p(z)| < 1 < |f(z)|$ on $\gamma$, hence $d_p(z) = f(z) + g(z)$ has exactly one root in $D$. Because the coefficients of $d_p$ are real, and $d_p(1/4) > 0$ and $d_p(1/2) < 0$, the intermediate value theorem yields that the root of $d_p(z)$ is in $[1/4, 1/2]$.

The proof of the final statement is similar, upon comparing the polynomials $c(1-4z) + 4z^{p+1}$ and $c(1-4z)$, for $c \geq 1$. □

In order to prove that $N(z)$ itself is analytic in the domain $D$, we rewrite it as follows, denoting $u_p = 4z^{p+1}/(1-4z)$ and

$$\begin{aligned}
N(z) &= \frac{\sqrt{1-4z}}{2z}\sum_{p\geq 0} B_p\left(\sqrt{1+u_p} - 1\right) \\
&= \frac{\sqrt{1-4z}}{2z}\sum_{p\geq 0} B_p\left(\frac{u_p}{2} + \sqrt{1+u_p} - 1 - \frac{u_p}{2}\right) \\
&= \frac{1}{\sqrt{1-4z}}\sum_{p\geq 0} B_p z^p + \frac{\sqrt{1-4z}}{2z}\sum_{p\geq 0} B_p\left(\sqrt{1+u_p} - 1 - \frac{u_p}{2}\right) \\
&= \frac{1}{\sqrt{1-4z}}B(z) + \frac{\sqrt{1-4z}}{2z}\sum_{p\geq 0} B_p\left(\sqrt{1+u_p} - 1 - \frac{u_p}{2}\right), \qquad (3.7)
\end{aligned}$$

Since $\sqrt{1-4z}$ and $B(z)$ are analytic in $D$, it suffices to study the convergence of the sum over $p$ in (3.7).

**Lemma 3.15.** *For all $u \in \mathbb{C} \setminus (-\infty, -1]$, we have*

$$\left|\sqrt{1+u} - 1 - \frac{u}{2}\right| \leq \frac{|u|^2}{2}.$$

*Proof.* Write $x = \sqrt{1+u}$, so that $\Re(x) > 0$. Then $|u| = |x^2 - 1| = |x-1||x+1|$, so that the above inequality reads

$$\frac{|x-1|^2}{2} \leq \frac{|x-1|^2|x+1|^2}{2},$$

or equivalently $1 \leq |x+1|$, which is true since $\Re(x) > 0$. □

Recall that all points of $D$ have modulus less than 1/2. Consider a disk included in $D$. For $z$ in this disk, the quantity

$$|u_p|^2 = \frac{16|z|^{2p+2}}{|1-4z|^2}$$

is uniformly bounded by $cr^p$, for constants $c$ and $r < 1/4$. Because $u_p$ does not take any value from $(-\infty, 1]$ in $D$ (Lemma 3.14), we can apply Lemma 3.15 to $u_p$. Hence

$$\sum_{p\geq 0} B_p \left|\sqrt{1+u_p} - 1 - \frac{u_p}{2}\right| \leq \frac{1}{2}\sum_{p\geq 0} B_p |u_p|^2 \leq \frac{c}{2}\sum_{p\geq 0} B_p r^p.$$

Since $B(z)$ is analytic in $D$, this proves that the series occurring in (3.7) converges uniformly in the disk, and that $N(z)$ is analytic in $D$. This finishes the first step of the proof: namely, that the generating function $N(z)$ is analytic in a $\Delta$-domain.

**Step 2: splitting $N(z)$.** We split the sum in $N(z)$ into three parts, namely one for a finite sum up to $n$, one for an infinite sum for $p > n$, and a third sum for the residual terms. The splitting depends on a threshold integer $n(z)$, to be defined later. We set

$$N(z) = N_1(n,z) + N_2(n,z) + N_3(n,z), \tag{3.8}$$

where

$$\begin{aligned} N_1(n,z) &= \frac{\sqrt{1-4z}}{2z}\sum_{p=0}^{n} B_p\left(\sqrt{1+u_p} - 1\right), \\ N_2(n,z) &= \frac{1}{\sqrt{1-4z}}\left(B(z) - \sum_{p=0}^{n} B_p z^p\right), \qquad (3.9) \\ N_3(n,z) &= \frac{\sqrt{1-4z}}{2z}\sum_{p>n} B_p\left(\sqrt{1+u_p} - 1 - \frac{u_p}{2}\right). \qquad (3.10) \end{aligned}$$

One readily checks that (3.8) indeed holds. Moreover, each $N_i(n,z)$ is analytic in $D$ for any $i$ and $n$.

**Step 3: an upper bound on $N_1$.**

**Lemma 3.16.** *For any $u \in \mathbb{C} \setminus (-\infty, -1]$, we have $|\sqrt{1+u} - 1| \leq \sqrt{|u|}$.*

*Proof.* The proof is similar to the proof of Lemma 3.15. Write $x = \sqrt{1+u}$, so that $\Re(x) > 0$. Then the inequality we want to prove reads $|x-1| \leq |x+1|$, which is clearly true. ☐

**Lemma 3.17.** *Let $z > 0$ and define*

$$a(n,z) = \sum_{p=0}^{n} B_p z^p.$$

*For any $c > \frac{1}{4}$, there exists a neighborhood of $c$ such that*

$$a(n, z) = \mathcal{O}\left((4z)^n n^{-3/2}\right)$$

*uniformly in $n$ and $z$ in the neighborhood of $c$.*

The proof of this lemma is inspired from Theorem VI.1 in [FS09] (which is also the base of the proof of Theorem 3.9), where the authors provide the asymptotic expansion of the coefficient of $z^n$ in $f(z) = (1-z)^{-\alpha}$ for any $\alpha \in \mathbb{C} \setminus \mathbb{Z}_{\leq 0}$.

*Proof.* We first form the generating function of the numbers $a(n, z)$ for a fixed $z$, then evaluate the $n^{\text{th}}$ coefficient via Cauchy's coefficient formula (Equation (3.6) in Theorem 3.7). We have

$$\begin{aligned}\sum_{n\geq 0} a(n,z)x^n &= \sum_{n\geq 0}\sum_{p=0}^{n} B_p z^p x^n = B(xz)x^0 + B(xz)x^1 + B(xz)x^2 + \dots \\ &= \frac{B(xz)}{1-x} = \frac{1-\sqrt{1-4xz}}{2xz(1-x)}.\end{aligned}$$

Thus

$$\begin{aligned}a(n,z) &= [x^n]\frac{1}{2xz(1-x)} - [x^n]\frac{\sqrt{1-4xz}}{2xz(1-x)} \\ &= \frac{1}{2z} - [x^n]\frac{\sqrt{1-4xz}}{2xz(1-x)} = \frac{1}{2z}\left(1 - I(n,z)\right),\end{aligned}$$

where

$$I(n,z) = [x^{n+1}]\frac{\sqrt{1-4xz}}{1-x}.$$

We now estimate $I(n, y)$ using Cauchy's coefficient formula:

$$I(n,z) = \frac{1}{2i\pi}\int_{\circlearrowleft} \frac{\sqrt{1-4xz}}{1-x}\frac{dx}{x^{n+2}},$$

where the integral is over a small circle around the origin. The dominant singularity of the integrand is at $x = 1/(4z)$. A second singularity occurs later at $x = 1$ (recall that $z$ is taken in a small neighborhood of $c > 1/4$). Writing $x = u/(4z)$ gives

$$I(n,z) = \frac{(4z)^{n+2}}{2i\pi}\int_{\circlearrowleft} \frac{\sqrt{1-u}}{4z-u}\frac{du}{u^{n+2}}, \tag{3.11}$$

the integral being again over a small circle around the origin. We now deform the integration contour into a *Hankel contour*, as illustrated in Figure 3.2:

$$\mathcal{H} = \mathcal{H}^-(n) \cup \mathcal{H}^+(n) \cup \mathcal{H}^\circ(n)$$

where

$$\begin{aligned}\mathcal{H}^-(n) &= \{z = w - i/n,\ w \geq 1\} \\ \mathcal{H}^+(n) &= \{z = w + i/n,\ w \geq 1\} \\ \mathcal{H}^\circ(n) &= \left\{z = 1 - \frac{e^{i\varphi}}{n},\ \varphi \in \left[-\frac{\pi}{2}, \frac{\pi}{2}\right]\right\}.\end{aligned}$$

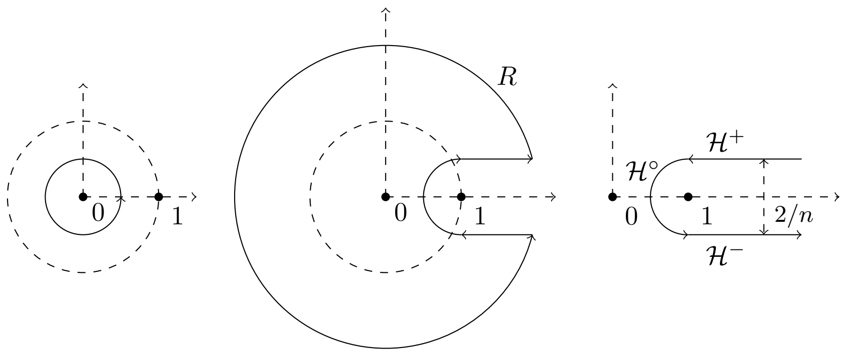


Figure 3.2: Changing the integration contour into a Hankel contour. We start with a positively oriented curve around 0, then let $R \to \infty$, while always keeping a distance of $1/n$ from the line $[1, \infty)$. The integral along the circle vanishes. Adapted from [FS09, p. 381].

A change of variable

$$u = 1 + \frac{t}{n}$$

in the integral (3.11) leads to

$$I(n, z) = \frac{(4z)^{n+2}}{2i\pi n^{3/2}} \int_{\mathcal{H}'} \frac{\sqrt{-t}}{4z - 1 - t/n} \left(1 + \frac{t}{n}\right)^{-n-2} dt.$$

Note that the new Hankel contour $\mathcal{H}'$ is now independent of $n$. It winds around 0, being at distance 1 from the positive real axis. The lemma will be proven if we can show that the remaining integral is bounded by a constant. As in [FS09, p. 382], we split the integral into two parts, depending on whether $\Re(t) \leq \ln^2 t$ or $\Re(t) \geq \ln^2 t$. On the first part, $4z - 1 - t/n$ remains uniformly away from 0. This means it remains to show that, setting $\mathcal{G} = \mathcal{H} \cap \{\Re(t) \leq \ln^2 n\}$, the integral

$$\int_{\mathcal{G}} \left| \sqrt{-t} \left(1 + \frac{t}{n}\right)^{-n-2} \right| dt$$

is bounded. Using the Taylor expansion of $\ln(1 + x)$ and $\exp(x)$, one can show that $(1 + t/n)^{-n-2} = e^{-(n+2)\ln(1+t/n)} = e^{-t}(1 + \mathcal{O}(n^{-1}))$. Because $|t| < \ln^2 n$, this proves that the integral over $\mathcal{G}$ is in $\mathcal{O}(1)$.

On the second part, $4z - 1 - t/n$ reaches its minimal value $1/n$ when $\Re(t) = n(4z - 1)$. Writing $t = s + i$, we can bound the modulus of the second part by

$$\begin{aligned} 2n \int_{\ln^2 n}^{\infty} \sqrt{2s} \left(1 + \frac{s}{n}\right)^{-n-2} ds &\leq 2n\, e^{-\ln^2(n)} \int_{\ln^2 n}^{\infty} \sqrt{2s} \left(1 + \frac{s}{n}\right)^{-2} ds \\ &\leq 2n^{5/2}\, e^{-\ln^2(n)} \int_0^{\infty} \sqrt{2u}(1 + u)^{-2} du \\ &= o(n^{-\alpha}) \end{aligned}$$

for any real $\alpha$. This completes the proof. □

Combining Lemmas 3.17 and 3.16 (with $c = 1/2$) gives, for $z$ in $D$ close enough to $1/4$:

$$\begin{aligned}
|N_1(n,z)| &\leq \frac{\sqrt{|1-4z|}}{2|z^2|}\sum_{p=0}^{n} B_p\sqrt{|u_p|} \\
&\leq \sum_{p=0}^{n} B_p|z|^{(p-2)/2} \\
&= \mathcal{O}\left(4^n|z|^{n/2}n^{-3/2}\right), \qquad (3.12)
\end{aligned}$$

uniformly in $n$ and $z$.

**Step 4: an upper bound on $N_3$.** We combine the estimate in Lemma 3.15 with the expression (3.10) of $N_3(n,z)$.

$$\begin{aligned}
N_3(n,z) &\leq \frac{4z}{|1-4z|^{3/2}}\sum_{p>n} B_p|z|^{2p} \\
&\leq \frac{4z}{|1-4z|^{3/2}}\sum_{p>n} B_n 4^{p-n}|z|^{2p} \qquad \text{(since } B_{p+1}\leq 4B_p\text{)} \\
&\leq \frac{16|z|^{2n+3}}{|1-4z|^{3/2}}B_n\sum_{p>n} 4^{p-n-1}|z|^{2(p-n-1)} \\
&= \mathcal{O}\left(\frac{|z|^{2n}}{|1-4z|^{3/2}}B_n\right) \qquad (3.13)
\end{aligned}$$

uniformly in $n$ and $z$, for $z$ in $D$ close enough to $1/4$.

**Step 5: an estimate of $N_2$.** We determine the generating function of the numbers $N_2(n,z)$ (given by (3.9)) for $z$ fixed. We find:

$$\begin{aligned}
\sum_{n\geq 0} N_2(n,z)x^n &= \frac{1}{\sqrt{1-4z}}\frac{B(z)-B(xz)}{1-x} \\
&= \frac{1}{2xz\sqrt{1-4z}}\left(\sqrt{1-4z}-1+\frac{4z}{\sqrt{1-4xz}+\sqrt{1-4z}}\right).
\end{aligned}$$

Thus

$$N_2(n,z) = \frac{2}{\sqrt{1-4z}}I(n,z)$$

where

$$I(n,z) = [x^{n+1}]\frac{1}{\sqrt{1-4xz}+\sqrt{1-4z}}.$$

Using the same principles as in the proof of Lemma 3.17, we now estimate $I(n,z)$ using Cauchy's coefficient formula (Equation (3.6)):

$$I(n,z) = \frac{1}{2i\pi}\int_{\circlearrowleft}\frac{1}{\sqrt{1-4xz}+\sqrt{1-4z}}\frac{dx}{x^{n+2}}$$

where the integral is over a small circle around the origin. The unique singularity of the integrand is at $x = 1/(4z)$. Substituting $x = u/(4z)$ gives

$$I(n,z) = \frac{(4z)^{n+1}}{2i\pi} \int_{\circlearrowleft} \frac{1}{\sqrt{1-u} + \sqrt{1-4z}} \frac{du}{u^{n+2}}.$$

Changing the integration contour into a Hankel contour and substituting $u = 1 + t/n$ gives, as in Lemma 3.17,

$$I(n,z) = \frac{(4z)^{n+1}}{2i\pi\sqrt{n}} \int_{\mathcal{H}} \frac{1}{\sqrt{-t} + \sqrt{n(1-4z)}} \left(1 + \frac{t}{n}\right)^{-n-2} dt.$$

Again, we split the integral into two parts, depending on whether $\Re(t) \leq \ln^2 n$ or $\Re(t) \geq \ln^2 n$. We assume moreover that

$$n \to \infty \quad \text{and} \quad n(1-4z) \to 0. \tag{3.14}$$

We define again $\mathcal{G} = \mathcal{H} \cap \{\Re(t) \leq \ln^2 n\}$. As shown in [FS09, p. 382ff], we have

$$\int_{\mathcal{G}} \frac{1}{\sqrt{-t} + \sqrt{n(1-4z)}} \left(1 + \frac{t}{n}\right)^{-n-2} dt = \frac{2\pi i}{\Gamma(1/2)}.$$

Thus the first part of $I(n,z)$ is then

$$\frac{(4z)^{n+1}}{\Gamma(1/2)\sqrt{n}} \left(1 + \mathcal{O}(\sqrt{n(1-4z)}) + \mathcal{O}(n^{-1})\right),$$

while the second part is found to be smaller than $(4z)^n n^{-\alpha}$, for any real $\alpha$. Hence

$$I(n,z) = \frac{(4z)^{n+1}}{\sqrt{\pi}\sqrt{n}} \left(1 + \mathcal{O}(\sqrt{n(1-4z)}) + \mathcal{O}(n^{-1})\right),$$

so that

$$N_2(n,z) = \frac{2(4z)^{n+1}}{\sqrt{\pi}\sqrt{1-4z}\sqrt{n}} \left(1 + \mathcal{O}(\sqrt{n(1-4z)}) + \mathcal{O}(n^{-1})\right). \tag{3.15}$$

**Step 6: the threshold function $n(z)$.** We finally want to correlate $n \equiv n(z)$ and $z$ so that, as $z$ tends to $1/4$ (in the domain $D$), the function $N(z)$ is dominated by $N_2(n,z)$. Given the bounds (3.12) and (3.13) on $N_1(n,z)$ and $N_3(n,z)$, the estimate $B_n \sim 4^n n^{-3/2}$ (up to a multiplicative constant), and the estimate (3.15) of $N_2$, we want

$$\frac{4^n |z|^{n/2}}{n^{3/2}} = o\left(\frac{(4z)^n}{\sqrt{1-4z}\sqrt{n}}\right)$$

and

$$\frac{4^n |z|^{2n}}{n^{3/2}|1-4z|^{3/2}} = o\left(\frac{(4z)^n}{\sqrt{1-4z}\sqrt{n}}\right).$$

We also want (3.14) to hold. These four conditions are met for

$$n = n(z) = \left\lfloor \frac{\ln|1-4z|}{\ln|z|} \right\rfloor.$$

Indeed, for this choice of $n$, we have

$$|z|^n = \mathcal{O}(1-4z) \quad \text{and} \quad (4z)^n = 1 + O\big(|1-4z| \ln |1-4z|\big),$$

so that

$$\begin{aligned} N_1(n,z) &= \mathcal{O}\left((1-4z)^{-1/2} \ln^{-3/2} \frac{1}{|1-4z|}\right), \\ N_3(n,z) &= \mathcal{O}\left((1-4z)^{-1/2} \ln^{-3/2} \frac{1}{|1-4z|}\right), \\ N_2(n,z) &= \frac{2\sqrt{\ln 4}}{\sqrt{\pi}\sqrt{1-4z}\sqrt{\ln \frac{1}{|1-4z|}}} \left(1 + \mathcal{O}\left(\ln^{-1} \frac{1}{|1-4z|}\right)\right). \end{aligned}$$

Finally, since

$$\ln |1-4z| = \ln(1-4z) \left(1 + \mathcal{O}\left(\ln^{-1} \frac{1}{|1-4z|}\right)\right),$$

and because $z$ is close to $\frac{1}{4}$, we have at last obtained

$$N(z) = \frac{2\sqrt{\ln 4}}{\sqrt{\pi}\sqrt{1-4z}\sqrt{\ln \frac{1}{1-4z}}} \left(1 + \mathcal{O}\left(\ln^{-1} \frac{1}{1-4z}\right)\right),$$

as stated in Proposition 3.13.

□

## 3.5 Extensions

In this section we extend Theorem 3.1 to rooted, ordered, labelled unranked trees, as are common e.g. in XML-processing. Unlike for full binary trees, the number of edges in the DAG of an unranked tree is not simply twice the number of nodes. This calls for a separate treatment of the number of edges in the DAG of an unranked tree.

Because unranked trees are often encoded by the FCNS-encoding, we also treat the average edge size of binary trees. As discussed in Section 3.1, the node size for binary trees is the same as the node size for full binary trees.

Most of terminology is completely analogous to the previous case. For convenience, we quickly gather all definitions. We use the terminology *m-labelled tree* for trees composed over an alphabet of size $m$. With $\mathcal{T}_m$ we denote the set $m$-labelled unranked trees and with $\mathcal{B}_m$ we denote the set of $m$-labelled binary trees. We set $T_{m,p} = |\{t \in \mathcal{T}_m \mid |t| = p\}|$ and $B_{m,p}$ accordingly. Let $\mathcal{U} \in \{\mathcal{B}, \mathcal{T}\}$. By $C^{\mathcal{U}_m}_{n,u}$ resp. $C^{\mathcal{U}_m}_{n,p}$ we define the number of trees from $\mathcal{U}_m$ that contain a given tree $u \in \mathcal{U}_m$ of size $p$. The according generating function is denoted by $C^{\mathcal{U}_m}_p(z)$.

**The number of edges.** For a tree $t$, we define by $|\mathrm{dag}(t)|_E$ the number of edges in the DAG of $t$. Similarly as for the numbers $N_n$ and their associated generating function $N(z)$, we want to obtain an expression for the accumulated number of edges $E^{\mathcal{U}_m}_n$ and their associated generating function $E^{\mathcal{U}_m}(z)$. Let

$\mathcal{U} \in \{\mathcal{B}, \mathcal{T}\}$. We denote by $U_{m,n}^{(d)}$ the number of trees from $\mathcal{U}_{m,n}$ that have root degree $d$ (i.e. the root has $d$ children) and by sub$(t)$ the set of subtrees of $t$. Then, in the same spirit as in Section 3.2, we get for the number of edges:

$$\begin{aligned} E_n^{\mathcal{U}_m} = \sum_{t \in \mathcal{U}_{m,n}} |\mathrm{dag}(t)|_E &= \sum_{t \in \mathcal{U}_{m,n}} \sum_{u \in \mathrm{sub}(t)} \deg(\mathrm{root}(u)) \\ &= \sum_{u \in \mathcal{U}_m} \deg(\mathrm{root}(u)) C_{n,u}^{\mathcal{U}_m} = \sum_{p,d \geq 0} d\, U_{m,p}^{(d)} C_{n,p}^{\mathcal{U}_m}. \end{aligned}$$

The associated generating function is

$$E^{\mathcal{U}_m}(z) = \sum_{n \geq 0} E_n^{\mathcal{U}_m} z^n = \sum_{p,d \geq 1} d\, U_{m,p}^{(d)} C_p^{\mathcal{U}_m}(z). \tag{3.16}$$

### 3.5.1 Exact counting

**Binary Trees.** First we generalize the Catalan numbers to carry labels over an $m$-ary alphabet. Binary trees are combinatorially equivalent to full binary trees, in which the leaf nodes don't carry a label. Extending Lemma 3.3, an $m$-labelled full binary tree is either a single node or an internal node with two full binary trees attached. This yields the equation $B_m(z) = 1 + mzB_m^2(z)$, which proves the following lemma.

**Lemma 3.18.** *The generating function $B_m(z)$ of $m$-labelled full binary trees is*

$$B_m(z) = \frac{1 - \sqrt{1 - 4mz}}{2mz}. \tag{3.17}$$

*Equivalently, the number of $m$-labelled full binary trees of size $p$ is*

$$B_{m,p} = \frac{1}{p+1} \binom{2p}{p} m^p. \tag{3.18}$$

Again Equation (3.18) follows by a Taylor expansion of (3.17). It can also easily be derived from Equation (3.2), since there are $m^p$ different ways to color a full binary tree with $m$ colors. Note that $B_m(z) = B(mz)$.

Analogous to Lemma 3.4, we get

**Lemma 3.19.** *The generating function $C_u^{\mathcal{B}_m}(z)$ of $m$-labelled full binary trees that contain a given tree $u \in \mathcal{B}_m$ of size $p$ is*

$$C_p^{\mathcal{B}_m}(z) = \frac{1}{2mz} \left( \sqrt{1 - 4mz + 4mz^{p+1}} - \sqrt{1 - 4mz} \right).$$

We now obtain expressions for the generating functions $N^{\mathcal{B}_m}(z)$ and $E^{\mathcal{B}_m}(z)$.

**Theorem 3.20.** *The generating function of the accumulated number of nodes of minimal DAGs of $m$-labelled (full) binary trees is*

$$N^{\mathcal{B}_m}(z) = \frac{1}{2mz} \sum_{p \geq 0} B_{m,p} \left( \sqrt{1 - 4mz + 4mz^{p+1}} - \sqrt{1 - 4mz} \right).$$

*The generating function of the accumulated number of edges of DAGs of $m$-labelled binary trees is*

$$E^{\mathcal{B}_m}(z) = \frac{3}{2mz}\sum_{p\geq 1}\frac{p-1}{2p-1}B_{m,p}\left(\sqrt{1-4mz+4mz^{p+1}}-\sqrt{1-4mz}\right).$$

*Proof.* The proof for $N^{\mathcal{B}_m}(z)$ follows from Lemma 3.19 and Equation (3.4). To express the series $E^{\mathcal{B}_m}(z)$, we first need to determine (according to Equation (3.16)) the number $B^{(d)}_{m,p}$ of $m$-labelled binary trees of size $p \geq 1$ with root degree $d$. Note that $d$ can only be 1 or 2. Clearly, $B^{(1)}_{m,p} = 2mB_{m,p-1}$, and thus $B^{(2)}_{m,p} = B_{m,p} - 2mB_{m,p-1}$. Hence, for $p \geq 1$,

$$\begin{aligned}\sum_{d\geq 1} d\cdot B^{(d)}_{m,p} &= 2mB_{m,p-1} + 2(B_{m,p} - 2mB_{m,p-1})\\ &= 2(B_{m,p} - mB_{m,p-1})\\ &= 2m^p\left(\frac{1}{p+1}\binom{2p}{p} - \frac{1}{p}\binom{2p-2}{p-1}\frac{2p(2p-1)}{p(p+1)}\frac{p+1}{2(2p-1)}\right)\\ &= \frac{3p-3}{2p-1}B_{m,p}.\end{aligned}$$

The expression for $E^{\mathcal{B}}_m(z)$ now follows, using (3.16) and Lemma 3.4. □

**Unranked trees.** An $m$-labelled unranked tree is a node to which a sequence of $m$-labelled unranked trees is attached. This yields the equation

$$T_m(z) = \frac{mz}{1-T_m(z)}$$

for the combinatorial class $\mathcal{T}_m$. Accordingly

$$T_m(z) = \frac{1-\sqrt{1-4mz}}{2}$$

and the numbers

$$T_{m,p} = B_{p-1}m^p \tag{3.19}$$

are given by shifted Catalan numbers multiplied by the number of nodes.

**Lemma 3.21.** *The generating function of $m$-labelled unranked trees that contain a given tree $u$ of size $p$ is*

$$C^{\mathcal{T}_m}_p(z) = \frac{z^p + \sqrt{1-4mz+2z^p+z^{2p}} - \sqrt{1-4mz}}{2}.$$

*Proof.* As in Lemma 3.4, we first determine the generating function $A^{\mathcal{T}_m}_p(z)$ of $m$-labelled unranked trees that *avoid* a given tree $u$ of size $p$. A $u$-avoiding tree is a root node to which a sequence of $u$-avoiding trees is attached. As before, we may not count $u$ itself and thus subtract $z^p$. This yields

$$A^{\mathcal{T}_m}_p(z) = \frac{zm}{1-A^{\mathcal{T}_m}_p(z)} - z^p,$$

which can be solved for $A^{\mathcal{T}_m}_p(z)$. Using $C^{\mathcal{T}_m}_p(z) = T_m(z) - A^{\mathcal{T}_m}_p(z)$, this proves the lemma. □

**Lemma 3.22.** *Let the numbers $T_{m,p}$ be given as in formula* (3.19)*. The generating function of the accumulated node size of minimal DAGs of $m$-labelled unranked trees is*

$$\begin{aligned} N^{\mathcal{T}_m}(z) &= \sum_{p\geq 0} T_{m,p} C_p^{\mathcal{T}_m}(z) \\ &= \frac{1}{2}\sum_{p\geq 0} T_{m,p}\left(z^p + \sqrt{1-4mz+2z^p+z^{2p}} - \sqrt{1-4mz}\right). \qquad (3.20) \end{aligned}$$

*The generating function of the accumulated edge size of minimal DAGs of $m$-labelled unranked trees is*

$$\begin{aligned} E^{\mathcal{T}_m}(z) &= \sum_{d,p\geq 0} d T_{m,p}^{(d)} C_p^{\mathcal{T}_m}(z) \\ &= \frac{3}{2}\sum_{p\geq 0} \frac{(p-1)T_{m,p}}{p+1}\left(z^p + \sqrt{1-4mz+2z^p+z^{2p}} - \sqrt{1-4mz}\right). \end{aligned}$$

*Proof.* As for binary trees, we have

$$N_n^{\mathcal{T}_m}(z) = \sum_{t\in\mathcal{T}_{m,n}} |\mathrm{dag}(t)| = \sum_{u\in\mathcal{T}_m} C_{n,u}^{\mathcal{T}_m} = \sum_{p\geq 0} T_{m,p} C_{n,p}^{\mathcal{T}_m},$$

from which the generating function $N^{\mathcal{T}_m}(z)$ follows.

To express the generating function $E^{\mathcal{T}_m}(z)$, we must first determine (according to (3.16)) the number of $m$-labelled unranked trees of size $p$ and root degree $d$, or, more precisely, the sum

$$\sum_{d\geq 1} d\cdot T_{m,p}^{(d)}$$

for any $p \geq 1$. This is done in [DZ80, Corollary 4.1] for the case $m = 1$ (the index needs to be shifted, since the authors count unranked trees by the number of edges). It suffices to multiply by $m^p$ to obtain the general case:

$$\sum_{d\geq 1} d\cdot T_{m,p}^{(d)} = \frac{3(p-1)T_{m,p}}{p+1}.$$

□

It may be worth noting that the numbers $T_{1,p}^{(d)}$ are known as *ballot numbers*, as they appear in certain combinatorial election problems. In [DZ80, Theorem 4] it is shown that $\mathcal{T}_{1,p}^{(d)} = \frac{d}{p-1}\binom{2p-3-d}{p-2}$ (with the index shifted appropriately). See also [FS09, p. 68].

### 3.5.2 Asymptotic results

We now provide asymptotic results for the average node and edge size of $m$-ary binary trees, as well as the corresponding results for unranked trees. The proofs of these results all work very much like the proof of Theorem 3.1, where we proved the asymptotic average node size for full binary trees over a unary alphabet. We hence only show the modifications that must be made.

**Binary Trees.**

**Theorem 3.23.** *The average number $\bar{N}_n^{\mathcal{B}_m}$ of nodes in the minimal DAG of an $m$-labelled binary tree of size $n$ satisfies*

$$\bar{N}_n^{\mathcal{B}_m} = 2\kappa_m \frac{n}{\sqrt{\ln n}}\left(1 + \mathcal{O}\left(\frac{1}{\ln n}\right)\right)$$

*with*

$$\kappa_m = \sqrt{\frac{\ln 4m}{\pi}}.$$

*Proof.* As in the proof of Theorem 3.1, we show that the Transfer Theorem can be applied to the series

$$N_m(z) := N^{\mathcal{B}_m}(z) = \frac{1}{2mz}\sum_{p\geq 0} B_{m,p}\left(\sqrt{1-4mz+4mz^{p+1}} - \sqrt{1-4mz}\right)$$

close to its dominant singularity $z = 1/(4m)$ with the approximation

$$N_m(z) = \frac{2\sqrt{\ln(4m)}}{\sqrt{\pi}\sqrt{1-4mz}\sqrt{\ln\frac{1}{1-4mz}}}\left(1 + \mathcal{O}\left(\ln^{-1}\frac{1}{1-4mz}\right)\right). \tag{3.21}$$

Thus we first prove that the series $N_m(z)$ is analytic in the domain defined by $|z| < 1/(2m)$ and $z \notin [1/(4m), 1/(2m)]$.

The counterpart of Lemma 3.14 states that $1 - 4mz + 4mz^{p+1}$ has exactly one root of modulus less than $1/(2m)$, and that this root is larger than $1/(4m)$. This now holds for any $p \geq 0$ (provided $m \geq 2$). Hence each series $C_p^{\mathcal{B}_m}(z)$ is analytic in $D$. We can also prove that $u_p := 4mz^{p+1}/(1-4mz)$ avoids the half-line $(-\infty, -1]$ for $z \in D$.

The proof that $N_m(z)$ is also analytic in $D$ transfers verbatim, once we have written

$$N_m(z) \quad = \quad \frac{1}{\sqrt{1-4mz}}B_m(z) + \frac{\sqrt{1-4mz}}{2mz}\sum_{p\geq 0} B_{m,p}\left(\sqrt{1+u_p} - 1 - \frac{u_p}{2}\right).$$

In Step 2, we split $N_m(z)$ into the three following parts:

$$\begin{aligned}
N_m^{(1)}(n,z) &= \frac{\sqrt{1-4mz}}{2mz}\sum_{p=0}^{n} B_{m,p}\left(\sqrt{1+u_p} - 1\right),\\
N_m^{(2)}(n,z) &= \frac{1}{\sqrt{1-4mz}}\left(B_m(z) - \sum_{p=0}^{n} B_{m,p}z^p\right),\\
N_m^{(3)}(n,z) &= \frac{\sqrt{1-4mz}}{2mz}\sum_{p>n} B_{m,p}\left(\sqrt{1+u_p} - 1 - \frac{u_p}{2}\right).
\end{aligned}$$

Now combining Lemmas 3.15 and 3.17 gives an upper bound for $N_m^{(1)}$:

$$N_m^{(1)}(n,z) = \mathcal{O}\left((4m)^n |z|^{n/2} n^{-3/2}\right),$$

uniformly in $n$ and $z$ taken in some neighborhood of $1/(4m)$.

The upper bound on $N_m^{(3)}$ is found to be

$$N_m^{(3)}(n,z) = \mathcal{O}\left(\frac{|z|^{2n}}{|1-4mz|^{3/2}} B_{m,n}\right).$$

Finally, since $N_m^{(2)}(n,z)$ is simply $N_2(n,mz)$ (with $N_2$ defined by (3.9)), we derive from (3.15) that

$$N_m^{(2)}(n,z) = \frac{2(4mz)^{n+1}}{\sqrt{\pi}\sqrt{1-4mz}\sqrt{n}}\left(1+\mathcal{O}(\sqrt{n(1-4mz)})+\mathcal{O}(n^{-1})\right). \tag{3.22}$$

The threshold function is now

$$n = n(z) = \left\lfloor \frac{\ln|1-4mz|}{\ln|z|} \right\rfloor,$$

and the rest of the proof follows verbatim, leading to Equation (3.21), which concludes the proof of Theorem 3.23. □

**Theorem 3.24.** *The average number of edges in the minimal DAG of an m-labelled binary tree of size n satisfies*

$$\bar{E}_n^{\mathcal{B}_m} = 3\kappa_m \frac{n}{\sqrt{\ln n}}\left(1+\mathcal{O}\left(\frac{1}{\ln n}\right)\right)$$

*with $\kappa_m$ as in Theorem 3.23.*

*Proof.* In the series $E^{\mathcal{B}_m}(z)$ given by Theorem 3.20, we replace the numbers $B_{m,p}$ by

$$\bar{B}_{m,p} := \frac{3(p-1)}{2p-1} B_{m,p}$$

with the generating function

$$\bar{B}_m(z) = \frac{1-3z-(1-z)\sqrt{1-4z}}{z}.$$

One can then adapt the proof of Theorem 3.1 without any difficulty. The only significant change is in the estimate (3.15) (and more generally (3.22)) of $N_m^{(2)}(n,z)$, which is multiplied by a factor 3/2. This leads to the factor 3 in Theorem 3.24.

□

**Unranked Trees.**

**Theorem 3.25.** *The average number of nodes in the minimal DAG of an m-labelled unranked tree of size n satisfies*

$$\bar{N}_n^{\mathcal{T}_m} = \kappa_m \frac{n}{\sqrt{\ln n}}\left(1+\mathcal{O}\left(\frac{1}{\ln n}\right)\right),$$

*with $\kappa_m$ as in Theorem 3.23.*

*Proof.* The proof is again a variation on the proof of Theorem 3.1. Our first objective is to obtain the following counterpart of Proposition 3.13: *The generating function $N^{\mathcal{T}_m}(z)$ is analytic in the domain $D$ defined by $|z| < \frac{1}{2m}$ and $z \notin [\frac{1}{4m}, \frac{1}{2m}]$. As $z$ tends to $\frac{1}{4m}$ in $D$, one has*

$$N^{\mathcal{T}_m}(z) = \frac{m\kappa_m}{\sqrt{(1-4mz)\ln\frac{1}{1-4mz}}} + \mathcal{O}\left(\frac{1}{\sqrt{(1-4mz)\ln^3\frac{1}{1-4mz}}}\right), \qquad (3.23)$$

*where $\kappa_m$ is defined as in Theorem 3.23.*

Using Theorem 3.12 we get

$$N_n^{\mathcal{T}_m} = [z^n]N^{\mathcal{T}_m}(z) = \frac{\kappa_m}{\sqrt{\pi}}\frac{4^n m^{n+1}}{\sqrt{n\ln n}}\left(1+\mathcal{O}(\ln^{-1} n)\right).$$

Since the number of $m$-labelled unranked trees of size $n$ is

$$T_{m,n} \sim \frac{4^n m^{n+1}}{\sqrt{\pi}n^{3/2}}\left(1+\mathcal{O}(n^{-1})\right),$$

this gives for the average number of nodes the expression of Theorem 3.25.

So let us focus on the proof of Equation (3.23), which will mimic the proof of Proposition 3.13. We start from Equation (3.20), given in Lemma 3.22:

$$N_m(z) := N^{\mathcal{T}_m}(z) = \sum_{p\geq 0} T_{m,p} C_p^{\mathcal{T}_m}(z)$$

where

$$C_p^{\mathcal{T}_m}(z) = \frac{z^p + \sqrt{1-4mz+2z^p+z^{2p}} - \sqrt{1-4mz}}{2}.$$

and $T_{m,p} = B_{p-1}m^p$.

**Step 1: $N_m(z)$ is analytic in $D$.** With Rouché's theorem (Theorem. 3.8) we can prove that, just as $C_p^{\mathcal{B}}(z)$, $C_p^{\mathcal{T}_m}(z)$ is analytic in the domain $D$, defined by $|z| < \frac{1}{2m}$ and $z \notin [\frac{1}{4m}, \frac{1}{2m}]$. We then define

$$u_p = \frac{z^p(2+z^p)}{1-4mz}$$

and prove that $u_p$ does not meet the half-line $(-\infty, -1]$ for $z \in D$. Writing

$$\begin{aligned} N_m(z) = \frac{T_m(z)}{2} + \frac{T_m(z)}{2\sqrt{1-4z}} + \frac{T_m(z^2)}{4\sqrt{1-4mz}} \\ + \frac{\sqrt{1-4mz}}{2}\sum_{p\geq 0} T_{m,p}\left(\sqrt{1+u_p} - 1 - \frac{u_p}{2}\right) \end{aligned}$$

we conclude with the arguments from Section 3.4 that $N_m(z)$ is also analytic in $D$.

**Step 2: splitting $N_m(z)$.** We fix an integer $n$ and write

$$N_m(z) = N_m^{(1)}(n,z) + N_m^{(2)}(n,z) + N_m^{(3)}(n,z) + N_m^{(4)}(n,z), \tag{3.24}$$

where

$$\begin{aligned}
N_m^{(1)}(n,z) &= \frac{\sqrt{1-4mz}}{2}\sum_{p=0}^{n} T_{m,p}\left(\sqrt{1+u_p}-1\right),\\
N_m^{(2)}(n,z) &= \frac{1}{2\sqrt{1-4mz}}\left(T_m(z)-\sum_{p=0}^{n} T_{m,p}z^p\right),\\
N_m^{(3)}(n,z) &= \frac{\sqrt{1-4mz}}{2}\sum_{p>n} T_{m,p}\left(\sqrt{1+u_p}-1-\frac{u_p}{2}\right),\\
N_m^{(4)}(n,z) &= \frac{1}{2}T_m(z) + \frac{1}{4\sqrt{1-4mz}}\left(T_m(z^2)-\sum_{p=0}^{n} T_{m,p}z^{2p}\right).
\end{aligned}$$

One readily checks that (3.24) indeed holds. Moreover, each $N_m^{(i)}(n,z)$ is analytic in $D$ for any $i$ and $n$.

**Step 3: an upper bound on $N_m^{(1)}$.** Using the same ingredients as before, we find that, as in the binary case,

$$N_m^{(1)}(n,z) = \mathcal{O}\left((4m)^n|z|^{n/2}n^{-3/2}\right)$$

uniformly in $n$ and $z$ taken in a neighborhood of $1/(4m)$.

**Step 4: an upper bound on $N_m^{(3)}$.** Again, the behavior remains the same as in the binary case:

$$N_m^{(3)}(n,z) = \mathcal{O}\left(\frac{m^n|z|^{2n}}{|1-4mz|^{3/2}}T_n\right),$$

uniformly in $n$ and $z$, for $z$ in $D$ close enough to $1/(4m)$.

**Step 5: an estimate of $N_m^{(2)}$.** With $B(z)$ and $B_{p-1}$ defined by Equations (3.1) and (3.2) respectively, we have

$$T_m(z) = zB(mz) \qquad \text{and} \qquad T_{m,p} = B_{p-1}m^p,$$

thus

$$N_m^{(2)}(n,z) = \frac{mz}{2}N_2(n-1,mz)$$

with $N_2(n,z)$ defined by (3.9). Hence the estimate (3.15) gives

$$N_m^{(2)}(n,z) = \frac{(4mz)^{n+1}}{\sqrt{\pi}\sqrt{1-4mz}\sqrt{n}}\left(1+\mathcal{O}(\sqrt{n(1-4mz)})+\mathcal{O}(n^{-1})\right). \tag{3.25}$$

**Step 6: an upper bound of $N_m^{(4)}$.** Given that $m|z|^2 < 1/(4m) \leq 1/4$, we can write

$$\begin{aligned}\left|N_m^{(4)}(n,z)\right| &\leq \frac{m}{2}\,|T(mz)| + \frac{1}{4\sqrt{|1-4mz|}}\sum_{p>n} T_{m,p}|z|^{2p}\\ &= \mathcal{O}\left(1+\frac{T_{m,n}z^{2n}}{\sqrt{|1-4mz|}}\right)\end{aligned}$$

for $z$ in a neighborhood of $1/(4m)$. The argument is the same as for the bound (3.13).

**Step 7: the threshold $n(z)$.** Using the same threshold function as in the binary case, we see that $N_m(n,z)$ is dominated by $N_m^{(2)}(n,z)$, and more precisely, that (3.23) holds.

□

Thus, on average, the DAG of a full binary tree of size $n$ has twice as many nodes as the DAG of an unranked tree of size $n$. Note that the same ratio holds between the heights of these trees: The average height of an unranked tree of size $n$ is $\sqrt{\pi n}$ [dBKR72], whereas the average height of a binary tree of size $n$ is $2\sqrt{\pi n}$ [FO82].

**Theorem 3.26.** *The average number of edges in the minimal DAG of an m-labelled unranked tree of size n satisfies*

$$\bar{E}_n^{\mathcal{T}_m} = 3\kappa_m \frac{n}{\sqrt{\ln n}}\left(1+\mathcal{O}\left(\frac{1}{\ln n}\right)\right),$$

*with $\kappa_m$ as in Theorem 3.23.*

*Proof.* Comparing the two series of Lemma 3.22 shows that it suffices to replace the numbers $T_p$ by

$$\bar{T}_p = \frac{3p}{p+2}T_p,$$

with generating function

$$\sum_{p\geq 0}\bar{T}_p z^p = \frac{1-3z-(1-z)\sqrt{1-4z}}{2z},$$

to go from $N^{\mathcal{T}_m}(z)$ to $E^{\mathcal{T}_m}(z)$. One can then adapt the proof of Proposition 3.25 without any difficulty. The only significant change is in the estimate (3.25) of $N_m^{(2)}(n,z)$, which is multiplied by a factor 3. This leads to the factor 3 in Theorem 3.26. □

While unranked trees and binary trees have the same asymptotic average number of edges in their respective DAGs, we show in Chapter 4 that the BDAG of an unranked tree $t$ (i.e. the DAG of the FCNS-encoding of $t$) can be quadratically smaller than the tree's DAG. On the other hand, bdag($t$) is at most twice the size of dag($t$).

| | $\mathcal{B}_m$ | $\mathcal{T}_m$ |
|---|---|---|
| $\bar{N}_{m,n}$ | $2\kappa_m \frac{n}{\sqrt{\ln n}}\left(1+\mathcal{O}\left(\frac{1}{\ln n}\right)\right)$ | $\kappa_m \frac{n}{\sqrt{\ln n}}\left(1+\mathcal{O}\left(\frac{1}{\ln n}\right)\right)$ |
| $\bar{E}_{m,n}$ | $3\kappa_m \frac{n}{\sqrt{\ln n}}\left(1+\mathcal{O}\left(\frac{1}{\ln n}\right)\right)$ | $3\kappa_m \frac{n}{\sqrt{\ln n}}\left(1+\mathcal{O}\left(\frac{1}{\ln n}\right)\right)$ |

Table 3.2: Overview over the different asymptotics. Recall that $\kappa_m = \sqrt{\frac{\ln 4m}{\pi}}$.

**Overview of the results.** Table 3.2 contains an overview of the results of this chapter.

## 3.6 On proving the main result by Mellin transform

In this section we discuss the possibility of an alternative proof of Theorem 3.1 (and the theorems from Section 3.5) via the Mellin transform. While we surely cannot prove that such a proof can't exist, we provide arguments why it is unlikely to be significantly easier than the existing proof (if it is possible at all).

The *Mellin transform* of a function sometimes provides an alternative way to derive an asymptotic of a function. Let $f(z)$ be a function defined over the positive reals. The transform of $f(z)$ is defined as the complex function $f^*(s)$, where

$$\mathcal{M}[f(z); s] = f^*(s) = \int_0^\infty f(z) z^{s-1}\, dz.$$

See [FGD95] and [Szp01, Chapter 9] for treatments of the Mellin transform in the context of generating functions.

The importance of the Mellin transform relies on the fact that it provides a correspondence between the behavior of a function at zero and infinity and the singularities of the transformed function. In other words, to understand how $f(z)$ behaves as $z \to 0$ resp. $z \to \infty$, we may also analyze the singularities of $f^*(z)$.

It follows directly from the linearity of the integral that

$$\mathcal{M}\left[\sum_{k\in\mathcal{K}} \lambda_k f(\mu_k\, z); s\right] = \left(\sum_{k\in\mathcal{K}} \lambda_k \mu_k^{-s}\right) \cdot f^*(s) \tag{3.26}$$

whenever the index set $\mathcal{K}$ is finite. For infinite sums, Chapter 3 in [FGD95] shows which conditions must be met for Equation (3.26) to still be applicable. Sums of the form $F(z) = \sum_k \lambda_k g(\mu_k z)$ are there called *harmonic sums*, with $\lambda_k$ being the *amplitudes*, $\mu_k$ the *frequencies* and $g(z)$ the *base function.*

It now becomes clear how Mellin transforms may be useful for the generating function $N(z) = \frac{1}{2z}\sum_{p\geq 0} B_p \left(\sqrt{1-4z+4z^{p+1}} - \sqrt{1-4z}\right)$ of the DAG: Though $N(z)$ is not of the form (3.26) (due to the term $z^{p+1}$ in the first square root), it may be possible to find an approximation $\tilde{N}(z)$ of $N(z)$ such that $N(z)$ may be factored into a base function and a sum of frequencies and amplitudes (the sum is called *Dirichlet series* in [FGD95]).

Indeed [Dis12] offers an analysis of a very similar generating function via Mellin transform: There, the function[1]

$$U(z) = \frac{1}{2}\sum_{k\geq 1}\left(\sqrt{1-4z+2^{k+1}z^{k+1}} - \sqrt{1-4z}\right)$$

is first approximated by a function $\tilde{U}(z)$ of the form (3.26), after which the Mellin transform conveniently provides the desired asymptotic behavior of $\tilde{U}(z)$ and its coefficients. Yet despite the similarities of the functions $U(z)$ and $N(z)$, it does not seem to be possible to follow the approach from [Dis12] for our function. To approximate $U(z)$, the author from [Dis12] substitutes the term $\sqrt{1-4z+2^{k+1}z^{k+1}}$ close to the singularity $z = 1/4$ by the corresponding root $\sqrt{1-4z+2^{-k-1}}$ and shows that the induced error, up to first order, is finite. This form can then be used to factor the Mellin transform of the generating function into a Dirichlet series. Yet this step is not possible for $N(z)$. By substituting $\sqrt{1-4z+4z^{p+1}}$ by $\sqrt{1-4z+4^{-p}}$, the effect of the substitution is, up to first order,

$$\left|\sum_{p\geq 0} B_p\sqrt{1-4z+4z^{p+1}} - \sum_{p\geq 0} B_p\sqrt{1-4z+4^{-p}}\right| \sim |z-1/4|\sum_{p\geq 0} 4B_p(p+1)2^{-p},$$

which does not converge. It remains open whether there is a different applicable substitution or a different way to use the Mellin transform for an alternative proof of Theorem 3.1.

We now briefly describe other open problems related to the topic of this chapter.

## 3.7 Open problems related to DAGs

### 3.7.1 Variance

While this chapter answers the question on what the average size of the DAG of a binary or unranked tree is, we do not know the variances of these sizes. To postulate a variance distribution for the average size of the DAG, we generated $1,000,000$ unranked trees of size $10,000$ using an algorithm described in [AS92] and built the DAG and the BDAG for the trees. Let $n$ be the number of data points. We consider the following well-established parameters that describe the goodness of a particular fitting:

1. The *sum of squared error of prediction* (SSE), is given by

$$\text{SSE} = \sum_{i=1}^{n}(y_i - f(x_i))^2.$$

[1] $U(z)$ is the generating function for the accumulated size of the *largest caterpillar subtree* of binary trees, where a *caterpillar tree* is defined as a tree in which every node has no more than one non-leaf successor.

2. Let $\bar{y} = \sum_{i=1}^{n} y_i$ and let $SST = \sum_i (y_i - \bar{y}_i)^2$ be the total sum of squares. The *Coefficient of determination* (R-squared) is the proportion of the total sum of squares explained by the model and is defined as

$$1 - \frac{\text{SSE}}{\text{SST}}.$$

3. The *root-mean-square error* (RMSE) is an estimate of the standard deviation of the random component in the data, and is defined as

$$\text{RMSE} = \sqrt{\frac{\text{SSE}}{n-m}}$$

where $m$ is the number of fitted coefficients (in our case, $m = 1$).

All data sets clearly suggest a normal distribution. Thus the following table shows the coefficients and goodness if we fit the different variances according to the normal distribution

$$f(x) = a \cdot \exp\left(-\left(\frac{x-b}{c}\right)^2\right). \tag{3.27}$$

| | $a$, $b$, $c$ | SSE | R square | RMSE |
|---|---|---|---|---|
| Nodes BDAG | $8,486,\ 4,286,\ 66.5$ | $1.04 \cdot 10^6$ | 0.9997 | 47.66 |
| Edges BDAG | $810,\ 6,247,\ 69.6$ | $8.27 \cdot 10^4$ | 0.9976 | 13.62 |
| Nodes DAG | $10,089,\ 2,228,\ 51.8$ | $8.99 \cdot 10^5$ | 0.9998 | 49.63 |
| Edges DAG | $7,393,\ 6,045,\ 76.2$ | $8.57 \cdot 10^5$ | 0.9998 | 26.98 |

Table 3.3: The variance

We remark that the average node size of the DAG is very close to the value predicted by Theorem 3.25, namely $2,228$ vs. $2,188$ (the asymptotic value being off by about 2%), whereas the difference is more for the average edge size of the DAG is larger ($6,045$ vs. $6,567$, or about 9%). This is probably due to a larger error term in the asymptotic edge size (recall that the order of convergence $\mathcal{O}(\log(n)^{-1})$ is rather slow).

Finally, Figure 3.3 plots the distribution of the node sizes of the DAGs of the $1,000,000$ binary trees generated plus the fitted distribution according to the normal distribution (3.27) with values $a, b$ and $c$ from Table 3.3.

### 3.7.2 Unordered trees

It is unknown what the average DAG size of *unordered* binary trees are. These trees are enumerated by the so-called *Wedderburn-Etherington* numbers.

### 3.7.3 The average size of the DAG of regular tree languages

We define *regular tree languages* as in the book [CDG+07]. It would be interesting to generalize the presented results to all regular tree languages, in particular

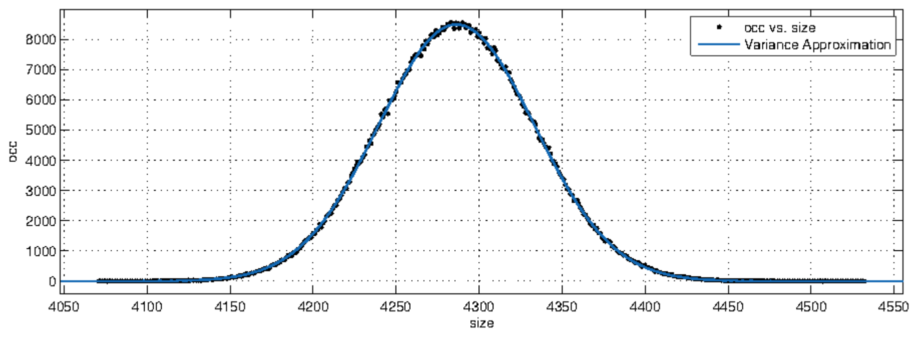


Figure 3.3: The distribution of the number of nodes in the DAG of 1, 000, 000 binary trees of size 10, 000.

to have a method that, given the grammar of a regular tree language $L$, calculates the average size of DAGs of trees of size $n$. Note that we assume that, for each $n$, the probability distribution of trees of size $n$ is uniform.

As an example, consider the grammars $G_1$ and $G_2$ with productions defined by

$$G_1: \begin{array}{l} S \to a(A,B) \\ A \to a(A) \mid a \\ B \to b(B,B) \mid b. \end{array} \qquad G_2: \begin{array}{l} S \to a(A,B) \\ A \to a_1(A) \mid a_2(A) \mid a \\ B \to b(B,B) \mid b. \end{array}$$

Because for a random tree from $L(G_1)$ of size $n$ the average number of nodes labelled by $a$ is negligible, the average DAG size of a tree from $L(G_1)$ is $\Theta(\frac{n}{\sqrt{\log n}})$.

On the other hand, on average more than half of the nodes in a tree from $L(G_2)$ are labelled by $a_1$ or $a_2$. Since the height of a tree is a trivial lower bound on the size of its DAG, it follows that the average DAG size of a tree from $L(G_2)$ is linear in the size of the tree.

So far, no regular tree language is known which has an average DAG size of neither $\Theta(\frac{n}{\sqrt{\log n}})$ nor $\Theta(n)$. This leads to the following conjecture.

**Conjecture 3.27.** *Let $\mathcal{A}$ be a regular tree language. Then the average size of the DAG of a tree of size $n$ from $\mathcal{A}$ is either $\Theta(\frac{n}{\sqrt{\log n}})$ or $\Theta(n)$.*

*Remark 1.* In [FSS90], Theorem 4 offers a generalization of Theorem 3.1 to a set of tree classes that contain an intersection with regular tree languages ($L(G_1)$ and $L(G_2)$ are not covered by [FSS90]). Yet the theorem also lacks a proof, and the derivation of the result remains unclear.

*Remark 2.* Binary trees with probability distribution induced by binary search trees have average DAG size $\Theta(\frac{n}{\log n})$, see [Dev98, FGM97], yet this tree class is not regular.

**On the average height.** For every tree classes $\mathcal{A}$ we considered when investigating the average DAG size of regular tree languages, the average size of the DAG of a tree from $\mathcal{A}$ was in $\Theta(\frac{n}{\sqrt{\log n}})$ if and only if the average height of a tree from $\mathcal{A}$ was in $\Theta(\sqrt{n})$. Trivially, tree classes with average height in $\Theta(n)$

also have an average DAG size in $\Theta(n)$, but also no tree class with average DAG size in $\Theta(n)$ was found that did not have an average height in $\Theta(n)$.

This is plausible, since DAG compression is better for shallow trees and because the height of a tree $t$ is a lower bound on $|\mathrm{dag}(t)|$. Thus we generalize Conjecture 3.27:

**Conjecture 3.28.** *Let $\mathcal{A}$ be a regular tree language. Then the average size of the DAG of a tree of size n from $\mathcal{A}$ is either $\Theta(\frac{n}{\sqrt{\log n}})$ or $\Theta(n)$. In the first case the average height of a tree from $\mathcal{A}$ is $\Theta(\sqrt{n})$ and in the second case the average height of a tree from $\mathcal{A}$ is $\Theta(n)$.*

# Chapter 4

# The hybrid DAG and worst-case sizes

In this chapter we introduce the so-called *hybrid DAG* (*HDAG*) of a tree, which shares features of both the DAG and the binary DAG: While the DAG of a tree shares repeated subtrees and the binary DAG shares repeated subtree sequences, the hybrid DAG achieves both. The HDAG is interesting from both a theoretical and a practical point of view: First of all, we show in Section 4.2 that the HDAG is smaller than both the DAG and the BDAG of a tree. This, among other inequalities will be used to show that the following holds for every unranked tree $t$ of size at least two:

$$\frac{1}{2}|\mathrm{bdag}(t)|_E \leq |\mathrm{dag}(t)|_E \leq \frac{1}{2}|\mathrm{bdag}(t)|_E^2.$$

Secondly, we use it in practice as a light-weight compressor for (XML)-tree data and perform some experiments on real-world data, which we discuss in Chapter 7. While the compression rates of the HDAG are not as good as TreeRePair, the HDAG has the advantage that some algorithmic problems may be solved easier for trees represented by HDAGs. We show such instances in Chapter 8. Furthermore, as we show in Chapter 7, heuristically the HDAG can be constructed a lot faster than other grammar-based compressors while only needing little more runtime than a DAG or BDAG compression, but is, guaranteed to be smaller than both.

The trees in this chapter are unranked, yet the HDAG may be defined for ranked trees as well.

## 4.1 The construction

Our starting point is the DAG grammar $\mathcal{G} = \mathbb{G}_{\mathrm{dag}}(t)$ of an unranked tree $t \in \mathcal{T}(\Sigma)$. We now want to share the repeated sibling end sequences in $\mathbb{G}$. As an example, consider the DAG grammar $\mathbb{G} = (\{S, A, B, C\}, \{f, g, a\}, P, S)$ where $P$ contains the rules

$$\begin{array}{lll} S & \to & f(A, B, C, B), \\ A & \to & f(C, B), \\ B & \to & g(a, a), \\ C & \to & g(a, a, a). \end{array} \tag{4.1}$$

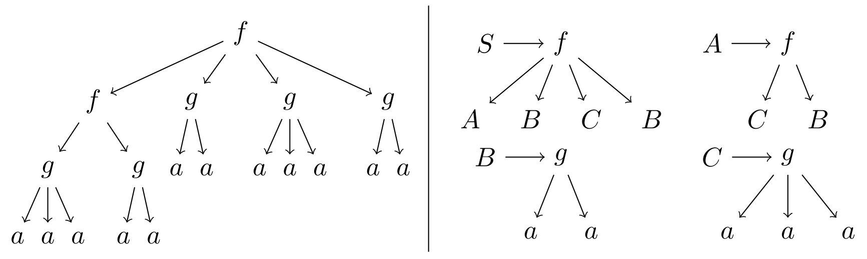


Figure 4.1: The tree $t$ and its DAG grammar $\mathbb{G}_{\text{dag}}(t)$.

The corresponding tree val($\mathbb{G}$) is depicted in Figure 4.1. We now apply the FCNS-encoding to the right-hand sides of the productions to obtain

$$\begin{array}{lcl} S & \rightarrow & f_l(A_r(B_r(C_r(B_0)))), \\ A & \rightarrow & f_l(C_r(B_0)), \\ B & \rightarrow & g_l(a_r(a_0)), \\ C & \rightarrow & g_l(a_r(a_r(a_0))). \end{array} \tag{4.2}$$

As is, this is not a correct binary tree grammar since we would need to define two different versions for $B$, one as rank 0, one as rank 1 nonterminal. But for now we simply view it as a binary encoding of the rules from $P$. We now construct the minimal DAG of the forest obtained by taking the disjoint union of the right-hand sides of (4.2), which yields

$$\begin{array}{lcl} S & \rightarrow & f_l(A(B_r(D))), \\ A & \rightarrow & f_l(D), \\ B & \rightarrow & g_l(E), \\ C & \rightarrow & g_l(a_r(E)), \\ D & \rightarrow & C_r(B_0), \\ E & \rightarrow & a_r(a_0). \end{array} \tag{4.3}$$

These rules constitute the *hybrid DAG* (or *HDAG* for short). The total number of edges in its right-hand sides is 9, as opposed to 11 in both the DAG and the BDAG of $t$.

In general, the hybrid DAG of a tree $t$ is produced by first constructing the minimal DAG grammar $\mathbb{G}_{\text{dag}}(t)$ of $t$ and then building the minimal DAG of the FCNS-encoding of the forest of right-hand sides of $\mathbb{G}_{\text{dag}}(t)$. Formally, assume that the DAG grammar $\mathbb{G}_{\text{dag}}(t)$ contains the rules $A_1 \rightarrow t_1, \ldots, A_n \rightarrow t_n$. Recall that every tree $t_i$ has height 1 (the grammar is reduced), and that the trees are pairwise different, and that it is also possible to construct the DAG of a forest of trees. Let $t'_i$ be the tree that is obtained from $t_i$ by adding $A_i$ as an additional label to the root of $t_i$. Then

$$\text{hdag}(t) = \text{dag}(\text{fcns}(t_1), \ldots, \text{fcns}(t_n)).$$

The HDAG is unique up to isomorphism (re-ordering of the nonterminals). Its size is defined as the number of edges it contains. Instead of adding an additional label to the root nodes, one could also assume that the root nodes $t_1, \ldots, t_n$ are ordered.

The HDAG construction may also be viewed as sharing the repeated suffixes that appear in the FCNS-encoding of the right-hand sides of the DAG grammar.

We will generalize this approach in Chapter 5, where we use other compression techniques on the right-hand sides of the DAG grammar. Though these may result in smaller grammars, the HDAG has the advantage of sharing properties of the BDAG and DAG, and can thus be used to show inequalities involving the worst-case sizes of those, which we will do in the next section.

**Building a TSLP from an HDAG.** *Unfolding* an HDAG, i.e. given an HDAG $h$, find the tree $t$ such that $\mathrm{hdag}(t) = h$, can be done in a two-step way: First we expand all rules that were introduced in the HDAG step. Then we reverse the fcns-encoding of the forest, which yields the DAG grammar $\mathbb{G}_{\mathrm{dag}}(t)$ of $t$.

On the other hand, the HDAG can also be converted easily to a monadic TSLP with only a modest size increase. We show this in a more general context in Theorem 5.5 of Chapter 5.

**The reverse hybrid DAG.** Just like the reverse binary DAG (see Section 2.2), one may also define the *reverse hybrid DAG* of a tree $t$ as the DAG of the last-child-previous-sibling encoding of the forest of right-hand sides of $\mathbb{G}_{\mathrm{dag}}(t)$. Again, while worst-case and average-case discussions are symmetrical to the HDAG case, this representation may be beneficial if has more repeated prefixes than suffixes. As with the reverse binary DAG, the reverse hybrid DAG performs better than the hybrid DAG on “real” data [BLMN15].

## 4.2 Comparison of the worst-case sizes of DAG, BDAG and HDAG

We compare the worst-case node size and the worst-case edge size of the DAG, the BDAG and the HDAG for an unranked tree $t$.

### 4.2.1 The number of nodes

**Lemma 4.1.** *Let $t$ be an unranked tree. Then $|dag(t)|_N \leq |bdag(t)|_N$.*

*Proof.* The lemma follows from Lemma 2.8 and the obvious fact that the number of different subtrees of $t$ (i.e., $|\mathrm{dag}(t)|_N$) is at most the number of different sibling sequences in $t$: $\mathrm{sibseq}(u) = \mathrm{sibseq}(v)$ implies $t/u = t/v$. □

**Example 4.2.** Consider the tree $t_n = f(a, a, \ldots, a)$ consisting of $n$ nodes (one labelled $f$ and the rest labelled $a$), where $n \geq 2$. Then $|\mathrm{dag}(t)|_N = 2$ and $|\mathrm{bdag}(t)|_N = n$, while $|\mathrm{dag}(t)|_E = |\mathrm{bdag}(t)|_E = n - 1$.

**Lemma 4.3.** *There exists a family of trees $(t_n)_{n\geq 2}$ such that $|dag(t)|_N = 2$ and $|t_n|_N = |bdag(t)|_N = n$.*

*Proof.* Take the family of trees $t_n$ from Example 4.2. □

Let us remark that the node size of the HDAG can be larger than the node size of the BDAG and of the DAG. The reason is that in $\mathbb{G}_{\mathrm{dag}}(t)$ there is a nonterminal for each node of the DAG (and hence the height of each right-hand side is at most one). This can be done differently of course; it was chosen so to simplify proofs and because our main goal is the reduction of the edge size.

### 4.2.2 The number of edges

We have just seen that the number of nodes of the minimal DAG is always at most the number of nodes of the BDAG, and that the gap can be maximal ($\mathcal{O}(1)$ versus $|t|$). For the number of edges, the situation is different. We show that

$$\frac{1}{2}|\mathrm{bdag}(t)|_E \leq |\mathrm{dag}(t)|_E \leq \frac{1}{2}|\mathrm{bdag}(t)|_E^2$$

for $|t| \geq 2$ and that these bounds are sharp up to the constant factor 1/2 in the second inequality. In fact, for $|t| \geq 2$ we show the following three inequalities

$$\begin{aligned} |\mathrm{hdag}(t)|_E &\leq \min(|\mathrm{dag}(t)|_E, |\mathrm{bdag}(t)|_E), \\ |\mathrm{bdag}(t)|_E &\leq 2|\mathrm{hdag}(t)|_E, \text{ and} \\ |\mathrm{dag}(t)|_E &\leq \frac{1}{2}|\mathrm{hdag}(t)|_E^2 \end{aligned}$$

which imply

$$\frac{1}{2}|\mathrm{bdag}(t)|_E \leq |\mathrm{dag}(t)|_E \leq \frac{1}{2}|\mathrm{bdag}(t)|_E^2.$$

Before we prove these bounds we need some definitions. Recall that the nodes of bdag($t$) are in one-to-one-correspondence with the different sibling sequences of $t$. In the following, let

$$\mathrm{sib}(t) = \{\mathrm{sibseq}(v) \mid v \text{ is a node of } t\}$$

be the set of all sibling sequences of $t$. To count the number of edges of bdag($t$), we have to count for each sibling sequence $w \in \mathrm{sib}(t)$ the number of outgoing edges in bdag($t$). We denote this number by $e(w)$; it can be computed as follows, where $w = s_1 s_2 \cdots s_m$ ($m \geq 1$) and the $s_i$ are trees:

- $e(w) = 0$ if $m = 1$ and $|s_1| = 0$,
- $e(w) = 1$ if either $m = 1$ and $|s_1| \geq 1$ (then $w$ only has a left child) or if $m \geq 2$ and $|s_1| = 0$ (then $w$ only has a right child),
- $e(w) = 2$ otherwise.

With this definition we obtain:

**Lemma 4.4.** *For every $t \in \mathcal{T}(\Sigma)$ we have*

$$|\mathit{bdag}(t)|_E = \sum_{w \in \mathit{sib}(t)} e(w).$$

The size of the HDAG can be computed similarly: Consider the reduced DAG grammar $\mathbb{G} = \mathbb{G}_{\mathrm{dag}}(t) = (N, \Sigma, P, S)$ of $t$. Recall that every right-hand side of $\mathbb{G}$ has the form $f(\alpha_1, \ldots, \alpha_n)$, where $\alpha_i \in \Sigma \cup N$ for every $i$. Let sib($\mathbb{G}$) be the set of all sibling sequences that occur in the right-hand sides of $\mathbb{G}$. Thus, for every right-hand side $f(\alpha_1, \ldots, \alpha_n)$ of $\mathbb{G}$, the sibling sequences $f(\alpha_1, \ldots, \alpha_n)$ (a sibling sequence of length 1) and $\alpha_i \alpha_{i+1} \cdots \alpha_n$ ($1 \leq i \leq n$) belong to sib($\mathbb{G}$). For such a sibling sequence $w$ we define $e(w)$ as above. Here, every $\alpha_i$ is viewed as a tree with a single node, i.e., $|\alpha_i| = 0$. Thus:

**Lemma 4.5.** *For every $t \in \mathcal{T}(\Sigma)$, we have*

$$|hdag(t)|_E = \sum_{w \in sib(\mathbb{G})} e(w).$$

For $w = s_1 \cdots s_m \in \mathrm{sib}(t)$ let $\tilde{w}$ be the string that results from $w$ by replacing every non-singleton tree $s_i \notin \Sigma$ by the unique nonterminal of $\mathbb{G}$ that derives to $s_i$.[1] Here are a few simple statements:

- For every $w \in \mathrm{sib}(t)$, the sibling sequence $\tilde{w}$ belongs to $\mathrm{sib}(\mathbb{G})$, except for the sequence $\tilde{w} = S$ of length 1 that is obtained from the sequence $w = t \in \mathrm{sib}(t)$.
- For every $w \in \mathrm{sib}(t)$, $\tilde{w} \in (N \cup \Sigma)^*$.
- For every $w \in \mathrm{sib}(t)$, $e(\tilde{w}) \leq e(w)$.
- The mapping $w \mapsto \tilde{w}$ is an injective mapping from $\mathrm{sib}(t) \setminus \{t\}$ to $\mathrm{sib}(\mathbb{G})$.

Using this mapping, the sums in Lemma 4.4 and 4.5 can be related as follows:

**Lemma 4.6.** *For every $t \in \mathcal{T}(\Sigma)$, we have*

$$|hdag(t)|_E = \sum_{w \in sib(\mathbb{G})} e(w) = |N| + \sum_{w \in sib(t)} e(\tilde{w}).$$

*Proof.* The first equality was shown in Lemma 4.5. The only sibling sequences in $\mathrm{sib}(\mathbb{G})$ that are not of the form $\tilde{w}$ for $w \in \mathrm{sib}(t)$ are the sequences (of length 1) that consist of the whole right-hand side $f(\alpha_1, \ldots, \alpha_m)$ of a nonterminal $A \in N$. For such a sibling sequence $u$ we have $e(u) = 1$ (since it has length 1 and $f(\alpha_1, \ldots, \alpha_m)$ is not a single symbol). Thus

$$\begin{aligned}\sum_{w \in \mathrm{sib}(\mathbb{G})} e(w) &= |N| + \sum_{w \in \mathrm{sib}(t) \setminus \{t\}} e(\tilde{w}) \\ &= |N| + \sum_{w \in \mathrm{sib}(t)} e(\tilde{w}),\end{aligned}$$

where the last equality follows from $e(\tilde{t}) = e(S) = 0$. □

**Theorem 4.7.** *For every $t \in \mathcal{T}(\Sigma)$, we have*

$$|hdag(t)|_E \leq \min(|dag(t)|_E, |bdag(t)|_E).$$

*Proof.* Since $\mathrm{hdag}(t)$ is obtained from $\mathrm{dag}(t)$ by sharing repeated suffixes of child sequences, we immediately get $|\mathrm{hdag}(t)|_E \leq |\mathrm{dag}(t)|_E$.

It remains to show that $|\mathrm{hdag}(t)|_E \leq |\mathrm{bdag}(t)|_E$. Thus by Lemma 4.4 and Lemma 4.6 we have to prove that

$$|N| + \sum_{w \in \mathrm{sib}(t)} e(\tilde{w}) \leq \sum_{w \in \mathrm{sib}(t)} e(w),$$

[1] Formally, $\tilde{w}$ depends on $t$ and thus it should be denoted by $\tilde{w}_t$. Since the context is unambiguous we omit the subscript.

where $N$ is the set of nonterminals of the DAG grammar. For every nonterminal $A \in N$ there exists a sibling sequence $w \in \mathrm{sib}(t)$ such that $\tilde{w}$ starts with $A$. For this sequence we have $e(w) = e(\tilde{w}) + 1$ (note that the right-hand side of $A$ is a non-singleton tree, hence $w$ starts with a tree of size at least 1). Recall that $e(\tilde{w}) \leq e(w)$ for all $w \in \mathrm{sib}(t)$.

Choose for every $A \in N$ a sibling sequence $w_A \in \mathrm{sib}(t)$ such that $\tilde{w}_A$ starts with $A$. Let $R = \mathrm{sib}(t) \setminus \{w_A \mid A \in N\}$ be the set of all other sibling sequences. We get

$$\begin{aligned}
|N| + \sum_{w \in \mathrm{sib}(t)} e(\tilde{w}) &= |N| + \sum_{A \in N} e(\tilde{w}_A) + \sum_{w \in R} e(\tilde{w}) \\
&= \sum_{A \in N} (e(\tilde{w}_A) + 1) + \sum_{w \in R} e(\tilde{w}) \\
&\leq \sum_{A \in N} e(w_A) + \sum_{w \in R} e(w) \\
&= \sum_{w \in \mathrm{sib}(t)} e(w).
\end{aligned}$$

This proves the theorem. □

**Theorem 4.8.** *For every $t \in \mathcal{T}(\Sigma)$ with $|t| \geq 2$, we have*

$$|dag(t)|_E \leq \frac{1}{2}|hdag(t)|_E^2.$$

*Proof.* Let $f_i(\alpha_{i,1}, \ldots, \alpha_{i,n_i})$ for $1 \leq i \leq k$ be the right-hand sides of $\mathbb{G}_{\mathrm{dag}}(t)$ with $\alpha_{i,j} \in \Sigma \cup N$ for all $i, j$. Recall that all the trees $f_i(\alpha_{i,1}, \ldots, \alpha_{i,n_i})$ are pairwise different, thus $|\mathrm{dag}(t)|_E = \sum_{i=1}^{k} n_i$. W.l.o.g. assume that $1 \leq n_1 \leq n_2 \leq \cdots \leq n_k$.

If $n_k = 1$, then every internal node in $t$ is unary. In this case, we get

$$|\mathrm{dag}(t)|_E = |t| \leq \frac{1}{2}|t|^2 = \frac{1}{2}|\mathrm{hdag}(t)|_E^2$$

since $|t| \geq 2$. Let us now assume that $n_k \geq 2$. Recall that we compute $\mathrm{hdag}(t)$ by taking the minimal DAG of the forest consisting of the binary encodings of the trees $f_i(\alpha_{i,1}, \ldots, \alpha_{i,n_i})$. The binary encoding of $f_i(\alpha_{i,1}, \ldots, \alpha_{i,n_i})$ has the form[2] $f_i(t_i, \Box)$, where $t_i$ is a chain of $n_i - 1$ many right pointers. Let $d$ be the minimal DAG of the forest consisting of all chains $t_i$. Since all the trees $f_i(\alpha_{i,1}, \ldots, \alpha_{i,n_i})$ are pairwise distinct, we have $|\mathrm{hdag}(t)| = k + |d|$. Since the chain $t_i$ consists of $n_i$ many nodes, we have $|d| \geq \max\{n_i \mid 1 \leq i \leq k\} - 1 = n_k - 1$. Hence, we have to show that $\sum_{i=1}^{k} n_i \leq \frac{1}{2}(k + n_k - 1)^2$. We have

$$\sum_{i=1}^{k} n_i \leq k \cdot n_k \leq (k-1)n_k + \frac{1}{2}n_k^2 = \frac{1}{2}(2(k-1)n_k + n_k^2) \leq \frac{1}{2}(k - 1 + n_k)^2,$$

which concludes the proof. For the second inequality note that $n_k \leq \frac{1}{2}n_k^2$, since $n_k \geq 2$. □

[2]We use $f(a, \Box)$ to denote $f_l(a)$ to avoid confusion when $f$ already has a subscript itself.

Consider the tree $s_n$ from Example 2.7 in Chapter 2: The tree $s_n$ is generated by the grammar with the productions

$$\begin{aligned} A_i &\to f(A_{i-1}, \underbrace{a, \dots, a}_{n-1 \text{ many}}) \quad \text{for } 2 \leq i \leq n, \\ A_1 &\to f(f, a, \dots, a), \end{aligned}$$

where $A_n$ is the start nonterminal (the tree is depicted in Figure 2.2). We have $|\mathrm{dag}(s_n)|_E = |s_n| = n^2$ and $|\mathrm{hdag}(s_n)|_E = |\mathrm{bdag}(s_n)|_E = 3n - 2$. Hence we get

$$|\mathrm{dag}(s_n)|_E = n^2 > \frac{1}{9}(3n-2)^2 = \frac{1}{9}|\mathrm{hdag}(s_n)|_E^2.$$

This shows that up to a constant factor, the bound in Theorem 4.8 is sharp. The constant 1/9 can be slightly improved:

**Theorem 4.9.** *There exists a family of trees $(s_n)_{n \geq 1}$ such that*

$$|dag(s_n)|_E > \frac{1}{6}|hdag(s_n)|_E^2.$$

*Proof.* We specify $s_n$ by the DAG grammar $\mathbb{G}_{\mathrm{dag}}(s_n)$. Let $\mathbb{G}_{\mathrm{dag}}(s_n)$ contain the following productions for $0 \leq i \leq n$:

$$A_i \to f(A_{i+1}, \dots, A_n, \underbrace{a, \dots, a}_{n \text{ many}}).$$

This is indeed the grammar obtained from the minimal DAG of a tree $s_n$ (of size exponential in $n$). We have

$$|\mathrm{dag}(s_n)|_E = \sum_{i=n}^{2n} i = n(n+1) + \sum_{i=0}^{n} i = n(n+1) + \frac{n(n+1)}{2} = \frac{3n(n+1)}{2}.$$

The hybrid DAG of $s_n$ consists of the child sequence $A_1 A_2 \cdots A_n a^n$ together with $n+1$ many left pointers into this sequence. Hence we have

$$|\mathrm{hdag}(s_n)|_E = 2n - 1 + n + 1 = 3n.$$

We obtain

$$\frac{1}{6}|\mathrm{hdag}(s_n)|_E^2 = \frac{1}{6}9n^2 = \frac{3}{2}n^2 < \frac{3n(n+1)}{2} = |\mathrm{dag}(s_n)|_E.$$

This proves the theorem. □

**Theorem 4.10.** *For every $t \in \mathcal{T}(\Sigma)$, we have*

$$|bdag(t)|_E + n \leq 2|hdag(t)|_E,$$

*where $n$ is the number of internal nodes of dag(t).*

*Proof.* We use the notations introduced before. In particular, $\mathbb{G}_{\mathrm{dag}}(t)$ is the DAG grammar of the tree $t$, $N$ the set of nonterminals in $\mathbb{G}_{\mathrm{dag}}(t)$, and $\tilde{w}$ refers to the word that is derived from the sibling sequence $w \in \mathrm{sib}(t)$ by replacing every non-singleton tree $s \in w$ by the nonterminal that derives to $s$.

By Lemma 4.4 we have $|\mathrm{bdag}(t)|_E = \sum_{w\in \mathrm{sib}(t)} e(w)$. By Lemma 4.6 we have $|\mathrm{hdag}(t)|_E = |N| + \sum_{w\in \mathrm{sib}(t)} e(\tilde{w})$. Hence we have to show that

$$|N| + \sum_{w\in \mathrm{sib}(t)} e(\tilde{w}) \geq \frac{1}{2} \sum_{w\in \mathrm{sib}(t)} e(w) + \frac{1}{2}|N|.$$

In order to prove this, we show the following for every sibling sequence $w \in \mathrm{sib}(t)$: Either $e(\tilde{w}) \geq \frac{1}{2}e(w)$ or $e(\tilde{w}) = 0$ and $e(w) = 1$. In the latter case, the sibling sequence $w$ consists of a single tree $s$ of size at least one (i.e., $s$ does not consist of a single node), and $\tilde{w}$ consists of a single nonterminal $A \in N$. So let $w = t_1 \cdots t_m \in \mathrm{sib}(t)$ and let $\tilde{w} = \alpha_1 \cdots \alpha_m$ with $\alpha_i \in \Sigma \cup N$. We consider the following four cases:

*Case 1.* $m > 1$ and $t_1 = \alpha_1 \in \Sigma$. We have $e(w) = e(\tilde{w}) = 1$.

*Case 2.* $m > 1$ and $|t_1| \geq 1$. We have $e(w) = 2$ and $e(\tilde{w}) = 1$.

*Case 3.* $m = 1$ and $t_1 = \alpha_1 \in \Sigma$. We have $e(w) = e(\tilde{w}) = 0$.

*Case 4.* $m = 1$ and $|t_1| \geq 1$. We have $e(w) = 1$, $e(\tilde{w}) = 0$, and $\tilde{w}$ consists of a single nonterminal $A \in N$.

□

For the tree $t_m$ from Figure 2.2 we have $n = |N| = 2$, $|\mathrm{bdag}(t_m)| = |t_m| = 2m$ and $|\mathrm{hdag}(s)_m| = |\mathrm{dag}(t)_m| = m + 1$. Hence Theorem 4.10 is optimal.

From the Theorems 4.7, 4.8, and 4.10 we immediately get:

**Corollary 4.11.** *For every $t \in \mathcal{T}(\Sigma)$ with $|t| \geq 2$, we have*

$$\frac{1}{2}|bdag(t)|_E \leq |dag(t)|_E \leq \frac{1}{2}|bdag(t)|_E^2.$$

## 4.3 The average size of the HDAG

Define

$$h_n = \sum_{t\in \mathcal{T}_{1,n}} |\mathrm{hdag}(t)|_E$$

as the accumulated (edge) size of the HDAGs of all unranked trees of size $n$ over a unary alphabet and let $\bar{h}_n = h_n/C_n$ be the corresponding average size. A closed formula for the generating function $H(z) = \sum_{n\geq 0} h_n z^n$ is not known. To postulate an asymptotic for the average size $\bar{h}_n$, we built the HDAG of for each of 50 trees chosen at random (using a method from [AS92]) for tree sizes $100, 200, 300, \ldots, 100,000$ and fitted the data with the software MATLAB, using different custom functions[3]. Thus a total of $50,000$ trees were generated.

[3] This means that the software is given different functions $c \cdot f(x)$ and uses curve fitting algorithms to find the optimal $c$ such that $c \cdot f(x)$ fits the data as good as possible.

The models we consider are

$$f_1(x) = \frac{cx}{\sqrt{\log x}}, \qquad f_2(x) = \frac{cx}{\log x},$$
$$f_3(x) = \frac{cx}{(\log x)^{2/3}}, \qquad f_4(x) = cx.s$$

Even though the asymptotic average size of the HDAG is surely not a linear function (the HDAG is always smaller than the DAG) we included this function to test the quality of the fitting for that case, too.

The functions $f_1(x)$ and $f_2(x)$ are likely the most reasonable candidates, as an asymptotic of type $f_1(x)$ would mean that the asymptotic average size of the DAG and the HDAG of a tree are only apart by a constant factor and because an asymptotic of type $f_2(x)$ is the information-theoretic minimum.

| Function type | Coeff. and bounds | SSE | R square | RMSE |
|---|---|---|---|---|
| $\frac{cx}{\sqrt{\log x}}$ | $1.628(1.628, 1.628)$ | $4.20 \cdot 10^5$ | 1 | 20.49 |
| $\frac{cx}{\log x}$ | $5.438(5.432, 5.444)$ | $2.20 \cdot 10^8$ | 0.9988 | 469.3 |
| $\frac{cx}{(\log x)^{2/3}}$ | $2.434(2.433, 2.435)$ | $2.66 \cdot 10^7$ | 0.9999 | 163.2 |
| $cx$ | $0.4871(0.4867, 0.4876)$ | $1.85 \cdot 10^8$ | 0.999 | 430 |

Table 4.1: Fitting the average HDAG size with different models. The bounds in the second column are within the 95% confidence intervals. See Section 3.7.1 for a description of the parameters SSE, R square and RMSE.

Though the results are best for the model $f_1(x) = \frac{cx}{\sqrt{\log x}}$, the other models are too good to be reasonably excluded from the statistical data alone (which demonstrates the limit of conjecturing an average size alone from statistical data). Yet $f_2(x)$ and $f_4(x)$ have a slightly worse fitting ($f_2(x)$ grows to slow while $f_4(x)$ grows too fast) than the models $f_1(x)$ and $f_3(x)$. Note that $\log(100,000)^{2/3} \approx 5.10$, while $\log(100,000)^{1/2} \approx 3.39$, so it is natural that the models $f_1(x)$ and $f_3(x)$ both offer a similarly good fitting. Figure 4.2 plots the average HDAG size as evaluated in the experiment together with the models $f_1(x)$ and $f_2(x)$. The lines for $f_1(x)$ and the average HDAG size are indistinguishable, whereas the line for $f_2(x)$ is above the HDAG line up to $n \approx 70,000$ and below that line afterwards. Figure 4.3 plots the residuals of the models $f_1(x)$ and $f_2(x)$. Note that the $y$-axis are scaled differently. Recall that the average edge size of the DAG of an unranked tree of size $n$ is $c_1 \cdot \frac{n}{\sqrt{\log n}}$ with $c_1 = 3\sqrt{\log 4/\pi} \approx 1.99$. Assuming that the average HDAG size is also of the form $c \cdot \frac{n}{\sqrt{\log n}}$, the experimentally evaluated $c = 1.628$ would imply that the HDAG is on average about 18.3 percent smaller.

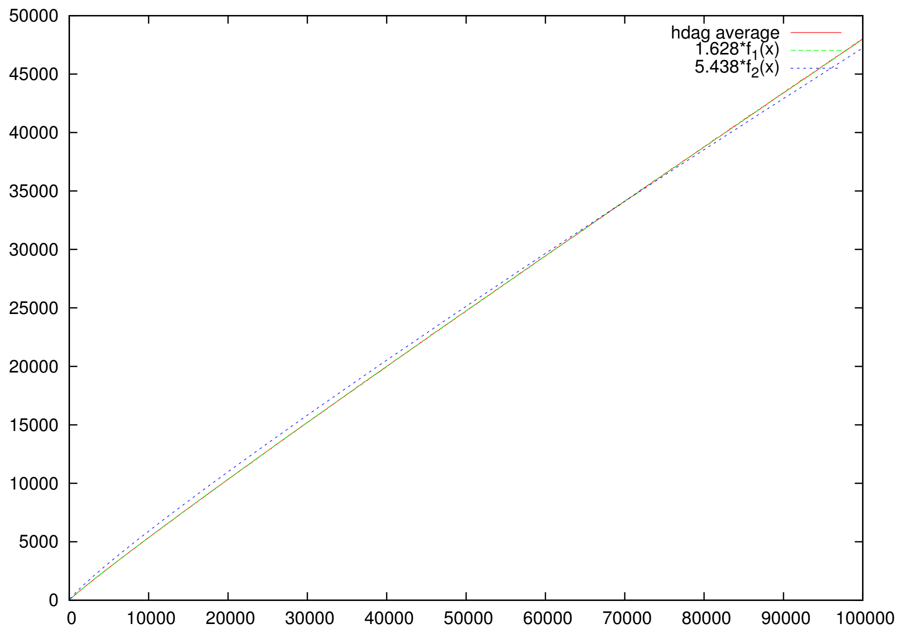


Figure 4.2: Comparing the experimentally evaluated average size of the HDAG with the functions $f_1(x) = \frac{x}{\sqrt{\log x}}$ and $f_2(x) = \frac{x}{\log x}$.

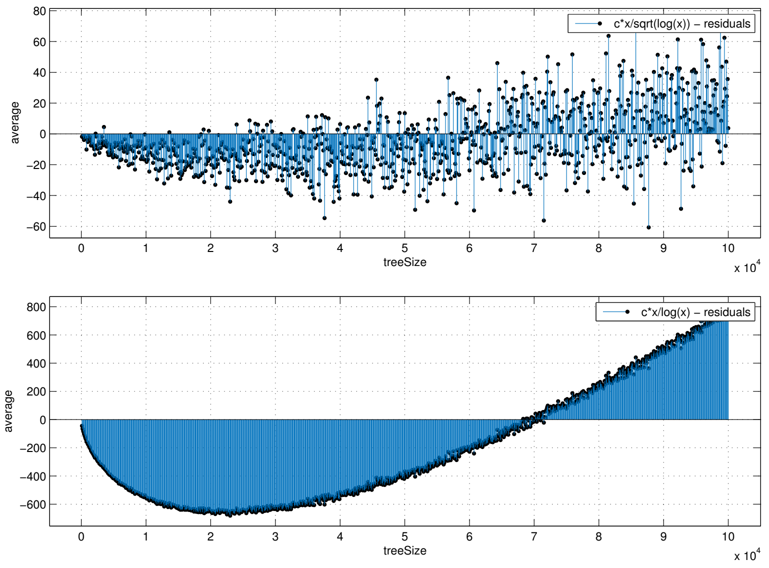


Figure 4.3: Residual plots of $f_1(x)$ and $f_2(x)$.

# Chapter 5

# DAG compression using string grammars

Let $t$ be a tree and consider its DAG grammar $\mathbb{G}_{\text{dag}}(t)$. The HDAG shares repeated suffixes of child sequences among the binary encodings of the right-hand sides of $\mathbb{G}_{\text{dag}}(t)$. These child sequences can easily be encoded by strings and so the HDAG can be viewed as sharing repeated string suffixes that are generated by the child sequences[1].

In this chapter we want to generalize the sharing of suffixes to other sharing schemes.

**Definition 5.1.** An *SLP-compressed DAG grammar* is a tuple

$$D = (N_t, N_s, \Sigma, P_t, P_s, S)$$

where $N_t$ is a set of *tree nonterminals*, $N_s$ is a set of *string nonterminals*, $\Sigma$ is a finite alphabet of unranked symbols, $P_t$ is a set of *tree productions*, $P_s$ is a set of *string productions* and finally $S \in P_t$ is the start nonterminal.

The productions in $P_t$ are of the form

$$N_t \to \mathcal{T}(\Sigma \cup N_t \cup N_s)$$

and the productions in $N_s$ are of the form

$$N_s \to (\Sigma \cup N_t \cup N_s)^*.$$

Both production sets are straight-line, meaning that there is at most one production per nonterminal (as usual, we assume without loss of generality that there is exactly one such production) and that the binary relation

$$\{(A, B) \in (N_t \cup N_s) \times (N_t \cup N_s) \mid (A \to t,\, B \in \text{rhs}(A)\}$$

is acyclic. Without loss of generality, we assume that the right-hand sides of the productions in $P_t$ to have height 1.

We apply a rule $p \in P_s$ to a tree $t \in \mathcal{T}(N_t \cup N_s \cup \Sigma)$ as follows: Let $p = (A \to A_1 \cdots A_n)$ and let $t = f(\alpha_1, \ldots, \alpha_{i-1}, A, \alpha_{i+1}, \ldots, \alpha_n)$. Then applying $p$

[1] Note that we implicitly used this view in Chapter 4.

to $t$ yields $t' = f(\alpha_1, \dots, \alpha_{i-1}, A_1, \dots, A_n, \alpha_{i+1}, \dots, \alpha_n)$ (thus we replace $A$ by the $n$ nonterminals $A_1, \dots, A_n$, effectively increasing the number of $f$'s children by $n-1$). The *size* of such an SLP-compressed DAG grammar is defined as the sum of the production sizes in $P_t$ plus the sum of the production sizes in $P_s$.

Note that the tuple $\mathbb{A} = (N_s, (\Sigma \cup N_t), P_s)$ is an SLP without a start rule (it may be viewed as a grammar which generates a finite set of strings) and that expanding all string productions yields a regular tree grammar.

**Lemma 5.2.** *Let $D = (N_t, N_s, \Sigma, P_t, P_s, S)$ be an SLP-compressed DAG. For a production $p \in P_t$, denote by $p'$ the production in which every string nonterminal in rhs$(p)$ has been expanded and define $P' := \bigcup_{p \in P_t} p'$ Then $D' = (N_t, \Sigma, P', S')$ is a regular tree grammar.*

Using these definitions, the HDAG of a tree may be viewed as an SLP-compressed DAG in which the set of string productions has been generated by suffix sharing among the child sequences of the right-hand sides of the tree's DAG.

**Example 5.3.** Consider the tree depicted in Figure 4.1 and its HDAG grammar in Equation (4.3) (which produces the FCNS-encoding of the tree $t$):

$$\begin{array}{lll} S \to f_l(A(B_r(D))), & A \to f_b(D), & B \to g_l(E), \\ C \to g_l(a_r(E)), & D \to C_r(B_0), & E \to a_r(a_0). \end{array}$$

An SLP-compressed DAG grammar for $t$ is $(N_t, N_s, \Sigma, P_t, P_s, S)$ with $N_s = \{D, E\}$, $N_t = \{S, A, B, C\}$ and

$$\begin{array}{lllllll} S & \to & f(A, B, D), & & D & \to & CB, \\ A & \to & f(D), & & E & \to & aa, \\ B & \to & g(E), & & & & \\ C & \to & g(a, E). & & & & \end{array}$$

**Example 5.4.** Let $t = f(a, \dots, a)$ with $\text{rank}(f) = n$. Then a smallest SLP-compressed DAG for $t$ has size $\mathcal{O}(\log n)$, while a TSLP for $t$ necessarily has size $n$.

Example 5.4 shows that a transformation of an SLP-compressed DAG to a TSLP grammar may induce an exponential size blow-up when the maximal rank of a terminal is unbounded. This may be avoided if we choose to transform it to a grammar for the FCNS-encoding of the tree instead. Next we show how to convert an SLP-compressed DAG for a tree $t$ to a monadic grammar for fcns$(t)$ with only a moderate size increase.

**Transforming to a monadic grammar.**

**Theorem 5.5.** *An SLP-compressed DAG $D = (N_t, N_s, \Sigma, P_t, P_s, S)$ can be transformed in time $\mathcal{O}(|D|)$ into a monadic TSLP $\mathbb{G}$ such that eval$(\mathbb{G})$ = fcns(eval$(D)$) and $|\mathbb{G}| \leq |D| + 2(N_t + N_s)$.*

Before we prove Theorem 5.5, we first show our procedure with a small example. Assume that the SLP-compressed DAG which we would like to convert contains the (tree) productions

$$A_1 \to f(B, C) \quad \text{and} \quad A_2 \to f(C, B),$$

where $B$ and $C$ are arbitrary nonterminals. In the FCNS-encoding of $f(B, C)$ and $f(C, B)$, the nonterminals $B$ and $C$ now each appear once as rank-1 nonterminal and once as rank-0 nonterminal. If we simply duplicated the rules for $B$ and $C$, we would double the size of the grammar in the worst-case. The same situation arises for string productions. Here, with $A_1 \to BC$ and $A_2 \to CB$, the last child of eval($B$) (resp. eval($C$)) has a right child in one case (namely the first child of eval($C$), resp. the first child of eval($B$)) and does not in the other.

Instead of duplicating the complete rules, we split the productions into parts that are the same for both productions and parts that need to be duplicated.

*Proof.* Let $D = (N_t, N_s, \Sigma, P_t, P_s, S)$ be an SLP-compressed DAG grammar. We create of each $N_t$ and $N_s$ two disjoint copies,

$$\hat{N}_t = \{\hat{X} \mid X \in N_t\}, \quad R_t = \{X_t \mid X \in N_t\},$$
$$\hat{N}_s = \{\hat{X} \mid X \in N_s\}, \quad R_s = \{X_s \mid X \in N_s\}.$$

Nonterminals in $N_t$, $N_s$ and $R_t$ have rank 0, while the nonterminals in $R_s$, $\hat{N}_s$ and $\hat{N}_t$ have rank 1. The set of nonterminals in $\mathbb{G}$ is $N_t \cup N_t' \cup \hat{N}_t \cup N_s \cup N_s' \cup \hat{N}_s$. Let $\alpha \in N_t \cup N_s$ be a tree nonterminal or a string nonterminal. The idea is that $\hat{\alpha}$ will appear at positions in which the FCNS-encoding has exactly one child (a right child), whereas the original $\alpha$ will appear when $\alpha$ is at a leaf position. Nonterminals in $R_t \cup R_s$ are used to keep the grammar small (they are nonterminals for the rest of the production).

The productions in $\mathbb{G}$ are defined as follows. For every tree production $X \in N_t$ with $X \to f(\alpha_1, \ldots, \alpha_n)$ ($n \geq 0$, $\alpha_1, \ldots, \alpha_n \in N_t \cup N_s$) we set

- If $n = 0$, then:

$$X \to f_0 \quad \text{and} \quad \hat{X}(y) \to f_r(y). \; X_t \text{ is not needed in this case.}$$

- Otherwise:

$$X \to f_l(X_t), \quad \hat{X}(y) \to f_b(X_t, y) \quad \text{and} \quad X_t \to \hat{\alpha}_1(\hat{\alpha}_2(\cdots \hat{\alpha}_{k-1}(\alpha_k)).$$

Note that the total size of these productions is at most $n + 2$.

For every string production $X \in N_s$ with $X \to \beta_1 \cdots \beta_n$ ($n \geq 2$ (wlog), $\beta_1, \ldots, \beta_n \in N_t \cup N_s$) we set

$$X \to X_s(\beta_n), \quad \hat{X}(y) \to X_s(\hat{\beta}_n(y)) \quad \text{and} \quad X_s(y) \to \hat{\beta}_1(\ldots(\hat{\beta}_{n-1}(y))).$$

These rules have total size $n + 2$. Hence the size of the grammar $\mathbb{G}$ is $|D| + 2(|N_t| + |N_s|)$ and the time needed to construct the grammar is bounded by $\mathcal{O}(|\mathbb{G}|) = \mathcal{O}(|D|)$. □

Note that all rules in $\mathbb{G}$ that have a parameter are of the form

$$A(y) \to f(t, y)$$

where $f \in \Sigma$ and $t \in \mathcal{T}(\Sigma \cup N)$. Thus parameters may only appear as the right child of the root node. The HDAG can be seen as a particular SLP-compressed

DAG $D = (N_t, N_s, \Sigma, P_t, P_s, S)$ where the SLP $\mathbb{A} = (N_s, (\Sigma \cup N_t), P_s)^2$ is *right regular*, thus for every $X \in N_s$ we have $\mathrm{rhs}(X) \in N_t^* N_s \cup N_t^*$ and similarly for every $V \in N_t$ we have $\gamma(V) \in N_s^* N_t \cup N_s^*$. When transforming such an SLP-compressed DAG into a monadic TSLP following the proof above, we do not need the sets $\hat{N}_s$ and $R_s$ because the nonterminals from $N_s$ always produce suffixes of child sequences in the DAG. This implies the following:

**Lemma 5.6.** *An HDAG that is represented as an SLP-compressed DAG $D = (N_t, N_s, \Sigma, P_t, P_s, S)$ can be transformed in time $\mathcal{O}(|D|)$ into a monadic tree grammar $\mathbb{G}$ such that eval$(\mathbb{G}) =$ fcns(eval$(D)$) and $|\mathbb{G}| \leq |D| + 2|N_t|$.*

**SLP-compressed DAG grammar construction.** In this section we present a possible algorithm to construct an SLP-compressed DAG in linear time (assuming the string compression algorithm needs linear time). To construct an SLP-compressed DAG for a tree $t$, we first construct $\mathbb{G}_{\mathrm{dag}}(t)$. Then we apply a grammar-based string compressor (e.g. Re-Pair [LM99] or Sequitur [NW97]) to the child sequences of the DAG (which we view as strings). Since we want to compress a set of strings instead of a single string, we concatenate all child sequences separated by unique symbols. Thus if the child sequences of $t$ are given by $s_1, s_2, \ldots, s_n$ we compress

$$s_1 \$_1 s_2 \$_2 \ldots \$_{n-1} s_n,$$

where $\$_i \notin \Sigma$ for $i \in [1..n-1]$ with a grammar-based string compressor, yielding an SLP $\mathbb{A}$. Since the symbols $\$_i$ are unique, they do not appear twice in the right-hand sides of the rules in $\mathbb{A}$ and thus the start production of $\mathbb{A}$ can be transformed without size increase to the form

$$S \to A_1 \$_1 A_2 \$_2 \ldots \$_{n-1} A_n$$

where $A_1 \to_{\mathbb{A}}^* s_1, \ldots, A_n \to_{\mathbb{A}}^* s_n$. Then the productions associated with the nonterminals $A_1, \ldots, A_n$ and the other productions introduced by the grammar compressor yield the string grammar of the SLP-compressed DAG.

Algorithm 1 summarizes the procedure described above using an arbitrary grammar-based string compressor.

```
input: tree $t$
let $((A_1 \to f_1(s_1)), \ldots, (A_n \to f_n(s_n)))$ be the rules in $\mathbb{G}_{\mathrm{dag}}(t)$;
let $s = s_1 \$_1 s_2 \$_2 \ldots \$_{n-1} s_n$;
compress $s$, yielding SLP $\mathbb{A}$;
add rules $B_1, \ldots, B_n$ to $\mathbb{A}$ such that $s$ is of form $B_1 \$_1 B_2 \$_2 \ldots \$_{n-1} B_n$;
return $((A_1 \to f_1(B_1)), \ldots, (A_n \to f_n(B_n)) \mid \mathbb{A})$;
```

**Algorithm 1:** DAG-and-String-Repair

**Example 5.7.** Consider the DAG grammar $\mathbb{G}_{\mathrm{dag}}(t) = (\{S, A, B\}, \{g, a\}, P, S)$ where the productions in $P$ are given by

$$S \to g(A, B, A, A, B, A), \quad A \to g(a, a, a) \quad \text{and} \quad B \to g(a, a, a, a, a, a).$$

$^2$As mentioned before, $\mathbb{A}$ may be treated as an SLP despite not having a start rule.

We then use a string compressor on the string

$$ABAABA\$_1aaa\$_2aaaaaa$$

which may yield the string grammar $\mathbb{S}$ with the productions

$$S' \to CC\$_1D\$_2DD, \quad C \to ABA \quad \text{and} \quad D \to aaa.$$

Using $\mathbb{S}$, we find that $D = (\{S, A, B\}, \{C, D\}, \{g, a\}, P_t, P_s, S)$ is an SLP-compressed DAG for $t$, where

$$\begin{aligned} P_t &= \{S \to g(C, C),\ A \to g(D),\ B \to g(D, D)\} \text{ and} \\ P_s &= \{C \to ABA,\ D \to aaa\}. \end{aligned}$$

With Theorem 5.5 we get a monadic grammar $\mathcal{G}$ with $\mathcal{F} = \{g_l, g_b, a_r, a_0\}$ and $N = \{S, S_t, A, \hat{A}, \hat{B}, B_s, C, \hat{C}, C_s, D, \hat{D}, D_s\}$ such that $\text{eval}(\mathcal{G}) = \text{fcns}(\text{eval}(D))$:

$$\begin{array}{rrl} S \to g_l(S_t), & S_t \to \hat{C}(C), & \\ A \to g_l(D), & \hat{A}(y) \to g_b(D, y), & \\ \hat{B}(y) \to g_b(B_s, y), & B_s \to \hat{D}(D), & \\ C \to \hat{C}_s(A), & \hat{C}(y) \to C_s(\hat{A}(y)), & C_s(y) \to \hat{A}(\hat{B}(y)), \\ D \to \hat{D}_s(a), & \hat{D}(y) \to D_s(a_r(y)), & D_s(y) \to a_r(a_r(y)). \end{array}$$

Note that we omitted unused productions and productions of the form $F \to G$.

# Chapter 6

# Tree compression using string grammars

In the previous chapter we investigated mixing string grammar and tree grammar formalisms to compress a tree $t$. Given a tree $t$, Algorithm 1 first computes the DAG grammar $\mathbb{G}$ of $t$, then calculates a string $s$ from $\mathbb{G}$, which is then compressed by a grammar-based string compressor.

In this chapter we drop the intermediate step of constructing the DAG grammar and examine the compression of the string given by the preorder traversal. This necessitates that trees are ranked, since otherwise a tree cannot be uniquely reconstructed from its traversal. As always, we may make an unranked tree ranked by introducing ranked symbols or by using the FCNS-encoding.

At first it might seem that this approach is equivalent to an SLP-compressed DAG in which the tree compression is trivial (i.e. every tree nonterminal occurs on exactly one right-hand side), but this is not the case. Consider for example the occurrence of $ba$ in the preorder traversal of the tree $f(f(a,b),a)$: This pattern cannot be shared in an SLP-compressed DAG. Thus, on the one hand, compressing the traversal pattern directly has the advantage that we can compress additional patterns, namely such patterns that are divided among different subtrees. On the other hand, the grammar introduced contains less information about the structure of the tree and thus certain algorithmic problems that are easier for TSLPs might be significantly more complicated for SLP-compressed traversals (like the evaluation of tree automata, which we show in Section 8.2).

## 6.1 Trees as preorder traversals

For the purpose of this section, we choose to define trees as particular strings over the alphabet $\mathcal{F}$, namely as those that represent preorder traversals: We define the set $\mathcal{T}(\mathcal{F})$ as the subset of $\mathcal{F}^*$ defined inductively as follows: If $f_n \in \mathcal{F}_n$ with $n \geq 0$ and $t_1, \ldots, t_n \in \mathcal{T}(\mathcal{F})$, then also $f_n t_1 \cdots t_n \in \mathcal{T}(\mathcal{F})$. We will also use the usual term notation of a tree if it improves readability. We address nodes by their position in the tree's preorder traversal, thus the root node is addressed by 1, its first child by 2 and so forth.

We call a string $s \in \mathcal{F}^*$ a *fragment* if there exists a tree $t \in \mathcal{T}(\mathcal{F})$ and a non-empty string $x \in \mathcal{F}^+$ such that $sx = t$. Note that the empty string $\varepsilon$ is a

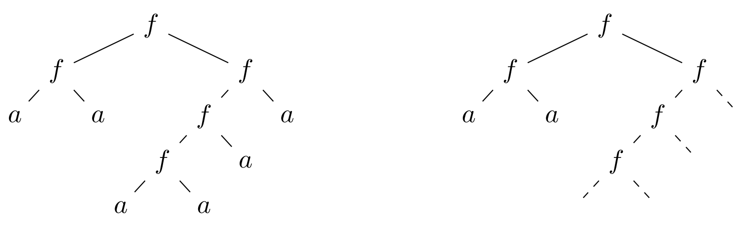


Figure 6.1: The tree $t$ from Example 6.3 and the tree fragment corresponding to the fragment $ffaafff$.

fragment. The number of *gaps* of a fragment $s \in \mathcal{F}^+$ is defined as the number $n$ of trees $t_1, \dots, t_n \in \mathcal{T}(\mathcal{F})$ such that $st_1 \cdots t_n \in \mathcal{T}(\mathcal{F})$, and is denoted by $\text{gaps}(s)$. The number of gaps of the empty string is defined as 0. Intuitively, $\text{gaps}(s)$ denotes the number of trees that are missing in order to make $s$ a tree. The following lemma shows that $\text{gaps}(s)$ is indeed well-defined.

**Lemma 6.1.** *The following statements hold:*

- *The set $\mathcal{T}(\mathcal{F})$ is prefix-free, i.e. $t \in \mathcal{T}(\mathcal{F})$ and $tv \in \mathcal{T}(\mathcal{F})$ imply $v = \varepsilon$.*
- *If $t \in \mathcal{T}(\mathcal{F})$, then every suffix of $t$ factors uniquely into a concatenation of trees from $\mathcal{T}(\mathcal{F})$.*
- *For every fragment $s \in \mathcal{F}^+$ there is a unique $n \geq 1$ such that*

$$\{x \in \mathcal{F}^* \mid sx \in \mathcal{T}(\mathcal{F})\} = (\mathcal{T}(\mathcal{F}))^n.$$

Since $\mathcal{T}(\mathcal{F})$ is prefix-free we immediately get:

**Lemma 6.2.** *For every $w \in \mathcal{F}^*$ there exist unique $n \geq 0$, $t_1, \dots, t_n \in \mathcal{T}(\mathcal{F})$ and a unique fragment $s$ such that $w = t_1 \cdots t_n s$.*

Let $w \in \mathcal{F}^*$ and let $w = t_1 \cdots t_n s$ be as in Lemma 6.2. We define $c(w) = (n, \text{gaps}(s))$. The number $n$ counts the number of complete trees in $w$ and $\text{gaps}(s)$ is the number of trees missing to make the fragment $s$ a tree, too.

**Example 6.3.** Let $t = ffaafffaaaa = f(f(a,a), f(f(f(a,a),a),a))$ be the tree depicted in Figure 6.1 with $f \in \mathcal{F}_2$ and $a \in \mathcal{F}_0$. Its height is 4. All prefixes (including the empty word, excluding the full word) of $t$ are fragments. The fragment $s = ffaafff$ is also depicted in Figure 6.1 in a graphical way, with dashed edges visualizing the gaps. We have $\text{gaps}(s) = 4$. For the factor $u = aafffa$ of $t$ we have $c(u) = (2,3)$. The children of node 5 (the third $f$-labelled node) are the nodes 6 and 11.

## 6.2 Checking whether an SLP produces a tree

Let $\mathcal{F}$ be a ranked alphabet and let $\mathbb{A}$ be an SLP. We show that we can verify in time $\mathcal{O}(|\mathbb{A}|)$ whether $\text{val}(\mathbb{A}) \in \mathcal{T}(\mathcal{F})$. In other words, we present a linear-time algorithm for the compressed membership problem for the language $\mathcal{T}(\mathcal{F}) \subseteq \mathcal{F}^*$. We remark that $\mathcal{T}(\mathcal{F})$ is a context-free language, which can be seen by considering the grammar with productions $S \to fS^n$ for all symbols $f \in \mathcal{F}_n$. In general the compressed membership problem for context-free languages can be solved in PSPACE and there exists a deterministic context-free language with a PSPACE-complete compressed membership problem [CMTV98, Loh11].

**Theorem 6.4.** *Given an SLP $\mathbb{A}$, one can check whether val$(\mathbb{A}) \in \mathcal{T}(\mathcal{F})$ in time $\mathcal{O}(|\mathbb{A}|)$.*

*Proof.* Let $\mathbb{A} = (N, \mathcal{F}, P, S)$ be an SLP in Chomsky normal form (recall that we can construct the Chomsky normal form of an arbitrary SLP in linear time) and let $A \in N$. Due to Lemma 6.2, we know that val$(A)$ is the concatenation of trees and a (possibly empty) fragment. Define $c(A) := c(\text{val}(A))$. Then val$(\mathbb{A}) \in \mathcal{T}(\mathcal{F})$ if and only if $c(S) = (1, 0)$. Hence, it suffices to compute $c(A)$ for all nonterminals $A \in N$, which we do bottom-up:

If $(A \to f) \in P$ with $f \in \mathcal{F}_n$, then we have

$$c(A) = \begin{cases} (1,0) & \text{if } n = 0 \\ (0,n) & \text{otherwise.} \end{cases}$$

Now consider a nonterminal $A$ with the rule $(A \to BC) \in P$, with val$(B) = t_1 \cdots t_{b_1} s$ and val$(C) = t'_1 \cdots t'_{c_1} s'$, where $t_1, \ldots, t_{b_1}, t'_1, \ldots, t'_{c_1} \in \mathcal{T}(\mathcal{F})$ and $s, s'$ are fragments with gaps$(s) = b_2$ and gaps$(s') = c_2$. Thus $c(B) = (b_1, b_2)$ and $c(C) = (c_1, c_2)$. We claim that

$$c(A) = \begin{cases} (b_1 + c_1 - \max\{1, b_2\} + 1, c_2) & \text{if } b_2 \le c_1 \\ (b_1, c_2 + b_2 - c_1 - \min\{1, c_2\}) & \text{otherwise.} \end{cases}$$

We distinguish two cases:

**Case $\mathbf{b_2 \le c_1}$:** If $b_2 \ge 1$, then the string $st'_1 \cdots t'_{b_2}$ is a tree, and thus val$(A)$ contains $b_1 + 1 + (c_1 - b_2)$ complete trees and the fragment $s'$ with $c_2$ many gaps. On the other hand, if $b_2 = 0$, then val$(A)$ contains $b_1 + c_1$ many complete trees.

**Case $\mathbf{b_2 \le c_1}$:** The trees $t'_1, \ldots, t'_{c_1}$ fill $c_1$ many gaps of $s$, and if $s' \neq \varepsilon$, then the fragment $s'$ fills one more gap, while creating another $c_2$ gaps. In total there are $b_2 - (c_1 + 1) + c_2$ gaps if $c_2 > 0$ and $b_2 - c_1$ gaps if $c_2 = 0$.

□

We define a *traversal SLP* to be an SLP that evaluates to the traversal of a tree.

## 6.3 Comparison with other grammar-based tree representations

In this section we compare the worst-case size of traversal SLPs with the following two grammar-based tree representations:

- TSLPs, and
- SLPs for balanced parenthesis representation [BLR+15].

### 6.3.1 Traversal SLPs versus TSLPs

In [BLM08] it is shown that a TSLP $\mathbb{A}$ producing a tree $t \in \mathcal{T}(\mathcal{F})$ can always be transformed into a traversal SLP of size $\mathcal{O}(|\mathbb{A}| \cdot r)$ for $t$, where $r$ is the maximal rank of a label occurring in $t$ (thus for binary trees the size at most doubles). We discuss the other direction in this section, i.e. transforming a traversal SLP into a TSLP.

As shown already in Example 5.4, the gap between traversal SLPs and TSLPs can trivially become exponential if the maximal rank of symbols is unbounded (we considered the tree family $(f_n a^n)_{n \geq 0}$, where $f_n$ is a symbol of rank $n$ and $a$ is a symbol of rank 0). It is less obvious that such an exponential gap can be also realized with trees of bounded rank. In the following we construct a family of binary trees $(t_n)_{n \in \mathbb{N}}$ where a smallest TSLP for $t_n$ is exponentially larger than the size of a smallest traversal SLP for $t_n$. Afterwards we show that it is always possible to transform a traversal SLP $\mathbb{A}$ for $t$ into a TSLP of size $\mathcal{O}(|\mathbb{A}| \cdot h \cdot r)$ for $t$, where $h$ is the height of $t$ and $r$ is the maximal rank of a label occurring in $t$.

**Worst-case comparison of SLPs and TSLPs.** Let $w \in \Sigma^*$. With $\mathrm{rev}(w) = a_n \cdots a_1$ we denote $w$ reversed. Given two strings $u, v \in \Sigma^*$, the *convolution* $u \otimes v \in (\Sigma \times \Sigma)^*$ is the string of length $\min\{|u|, |v|\}$ defined by $(u \otimes v)[i] = (u[i], v[i])$ for $1 \leq i \leq \min\{|u|, |v|\}$.

We use the following result from [BCR08] for the previously mentioned worst-case construction of a family of binary trees:

**Theorem 6.5** (Theorem 2 from [BCR08])**.** *For every $n > 0$, there exist words $u_n, v_n \in \{0,1\}^*$ with $|u_n| = |v_n|$ such that $u_n$ and $v_n$ have SLPs of size $n^{\mathcal{O}(1)}$, but the smallest SLP for the convolution $u_n \otimes v_n$ has size $\Omega(2^{n/2})$.*[1]

For two given words $u = i_1 \cdots i_n \in \{0,1\}^*$ and $v = j_1 \cdots j_n \in \{0,1\}^*$ we define the *comb tree*

$$t(u, v) = f_{i_1}(f_{i_2}(\dots f_{i_n}(\$, j_n) \dots j_2), j_1)$$

over the ranked alphabet $\{f_0, f_1, 0, 1, \$\}$ where $f_0, f_1$ have rank 2 and $0, 1, \$$ have rank 0. See Figure 6.2 for an illustration.

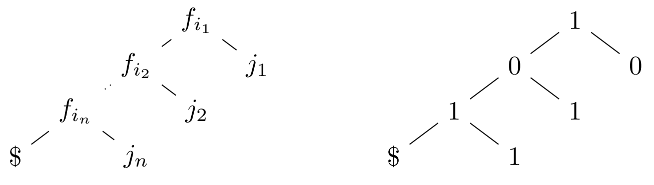


Figure 6.2: The comb tree $t(u, v)$ for $u = i_1 \cdots i_n$, $v = j_1 \cdots j_n$ (left) and for $s = 101$, $r = 011$ (right).

[1]Actually, in [BCR08] the result is not stated for the convolution $u_n \otimes v_n$, but for the literal shuffle of $u_n$ and $v_n$ which is $u_n[1]v_n[1]u_n[2]v_n[2] \cdots u_n[m]v_n[m]$. But this makes no difference, since the sizes of the smallest SLPs for the convolution and literal shuffle, respectively, of two words differ only by multiplicative constants.

**Theorem 6.6.** *For every $n > 0$ there exists a tree $t_n$ such that the size of a smallest traversal SLP for $t_n$ is polynomial in $n$, but the size of a smallest TSLP for $t_n$ is in $\Omega(2^{n/2})$.*

*Proof.* Let us fix an $n$ and let $u_n$ and $v_n$ be the aforementioned strings from Theorem 6.5. Let $|u_n| = |v_n| = m$. Consider the comb tree $t_n := t(u_n, v_n)$. Note that $t_n = f_{i_1} \cdots f_{i_m} \$ \operatorname{rev}(v_n)$, where $u_n = i_1 \cdots i_m$. By Theorem 6.5 there exist SLPs of size $n^{\mathcal{O}(1)}$ for $u_n$ and $v_n$, and these SLPs easily yield a traversal SLP of size $n^{\mathcal{O}(1)}$ for $t_n$.

Next, we show that a TSLP $\mathbb{A}$ for $t_n$ yields an SLP of size $\mathcal{O}(|\mathbb{A}|)$ for the string $u_n \wedge v_n$. Since a smallest SLP for $u_n \wedge v_n$ has size $\Omega(2^{n/2})$ by Theorem 6.5, the same bound must hold for the size of a smallest TSLP for $t_n$.

Let $\mathbb{A}$ be a TSLP for $t_n$. By [LMS12] we can transform $\mathbb{A}$ into a monadic TSLP $\mathbb{A}'$ for $t_n$ of size $\mathcal{O}(|\mathbb{A}|)$, which then can be transformed into an SLP $\mathbb{S}$ for $u_n \otimes v_n$, also of size $\mathcal{O}(|\mathbb{A}|)$ (using [BLM08]).

We can assume that every nonterminal in $\mathbb{S}$ except for the start nonterminal $S$ occurs in a right-hand side and in the derivation starting from $S$. At first we delete all rules of the form $A \to j$ ($j \in \{0,1\}$) and replace the occurrences of $A$ by $j$ in all right-hand sides. Now every nonterminal $A \neq S$ of rank 0 derives to a subtree of $t_n$ that contains the unique \$-leaf of $t_n$. Hence, $t_n$ contains a unique subtree $\operatorname{val}(A)$. This implies that $A$ occurs exactly once in a right-hand side. We can therefore, without size increase, replace this occurrence of $A$ by the right-hand side of $A$. After this step, $S$ is the only rank-0 nonterminal in the TSLP. With the same argument, we can also eliminate rank-1 nonterminals that derive to a tree containing the unique leaf \$. After this step, every rank-1 nonterminal $A(x)$ derives a tree of the form $g_1(g_2(\dots(g_k(x, j_k)\dots), j_2), j_1)$ ($g_i \in \{f_0, f_1\}$ and $j_i \in \{0,1\}$).

Now, if a right-hand side contains a subtree $f_i(s_1, s_2)$, then $s_2$ must be either 0 or 1. Similarly, for every occurrence of $i \in \{0,1\}$ in a right-hand side, the parent node of that occurrence must be either labelled by $f_0$ or by $f_1$ (note that the parent node exists and cannot be a nonterminal). Therefore we can obtain an SLP for $u_n \otimes v_n$ by replacing every production $A(x) \to t(x)$ by $A \to \lambda(t(x))$, where $\lambda(t(x))$ is the string obtained inductively by $\lambda(x) = \varepsilon$, $\lambda(B(s(x)) = B\lambda(s(x))$ for nonterminals $B$, and $\lambda(f_i(s(x), j)) = (i,j)\lambda(s(x))$. The production for $S$ must be of the form $S \to t(\$)$ for a term $t(x)$ and we replace it by $S \to \lambda(t(x))\$$.

□

**Conversion of traversal SLPs to TSLPs.** Note that the height of the tree $t_n$ in Theorem 6.6 is linear in the size of $t_n$. By the following result, large height or rank are always responsible for the exponential succinctness gap between traversal SLPs and TSLPs.

**Theorem 6.7.** *Let $t \in \mathcal{T}(\mathcal{F})$ be a tree of height $h$ and maximal rank $r$, and let $\mathbb{A}$ be a traversal SLP for $t$ with $|\mathbb{A}| = m$. Then there exists a TSLP $\mathbb{B}$ with val($\mathbb{B}$) $= t$ such that $|\mathbb{B}| \in \mathcal{O}(m \cdot h \cdot r)$, which can be constructed in time $\mathcal{O}(m \cdot h \cdot r)$.*

*Proof.* Without loss of generality we assume that $\mathbb{A}$ is in Chomsky normal form. For every rule of the form $A \to f$ with $f \in \mathcal{F}_n$ we add to $\mathbb{B}$ the TSLP-rule $A_1 \to f$ if $n = 0$ or $A'(x_1, \dots, x_n) \to f(x_1, \dots, x_n)$ if $n \geq 1$. For every nonterminal

$A$ of $\mathbb{A}$ with $c(A) = (a_1, a_2)$ we introduce $a_1$ nonterminals $A_1, \ldots, A_{a_1}$ of rank 0 (these produce one tree each) and, if $a_2 > 0$, one nonterminal $A'$ of rank $a_2$ for the fragment encoded by $A$. Now consider a rule of the form $A \to BC$ with $c(B) = (b_1, b_2)$ and $c(C) = (c_1, c_2)$.

**Case 1:** If $b_2 = 0$ we add the following rules to $\mathbb{B}$:

$$\begin{aligned} A_i &\to B_i && \text{for } 1 \le i \le b_1, \\ A_{b_1+i} &\to C_i && \text{for } 1 \le i \le c_1, \\ A'(x_1, \ldots, x_{c_2}) &\to C'(x_1, \ldots, x_{c_2}) && \text{if } c_2 > 0. \end{aligned}$$

**Case 2:** If $0 < b_2 \le c_1$ we add the following rules to $\mathbb{B}$:

$$\begin{aligned} A_i &\to B_i && \text{for } 1 \le i \le b_1, \\ A_{b_1+1} &\to B'(C_1, \ldots, C_{b_2}) && \\ A_{b_1+1+i} &\to C_{b_2+i} && \text{for } 1 \le i \le c_1 - b_2, \\ A'(x_1, \ldots, x_{c_2}) &\to C'(x_1, \ldots, x_{c_2}) && \text{if } c_2 > 0. \end{aligned}$$

**Case 3:** If $b_2 > c_1$ we add the following rules to $\mathbb{B}$, where $d = b_2 - c_1$:

$$A_i \to B_i \quad \text{for } 1 \le i \le b_1$$

and, depending on whether $c_2 = 0$ or not, either

$$\begin{aligned} A'(x_1, \ldots, x_d) &\to B'(C_1, \ldots, C_{c_1}, x_1, \ldots, x_d) \quad \text{or} \\ A'(x_1, \ldots, x_{c_2+d-1}) &\to B'(C_1, \ldots, C_{c_1}, C'(x_1, \ldots, x_{c_2}), x_{c_2+1}, \ldots, x_{c_2+d-1}). \end{aligned}$$

Chain productions, where the right-hand side consists of a single nonterminal, can be eliminated without size increase. Then, only one of the above productions remains and its size is bounded by $c_1 + 2$ (recall that we do not count parameters). Recall that $c_1$ is the number of complete trees produced by $C$. It therefore suffices to show that the number of complete trees of a factor $s$ of $t$ is bounded by $h \cdot r$, where $h$ is the height of $t$ and $r$ is the maximal rank of a label in $t$. Assume that $s = t[i : j] = t_1 \cdots t_n s'$, where $t_i \in \mathcal{T}(\mathcal{F})$ and $s'$ is a fragment. Let $k$ be the lowest common ancestor of $i$ and $j$. If $k = i$ (i.e., $i$ is an ancestor of $j$) then either $s = t_1$ or $s = s'$. Otherwise, the root of every tree $t_l$ ($1 \le l \le n$) is a child of a node on the path from $i$ to $k$. The length of the path from $i$ to $k$ is bounded by $h$, hence $n \le h \cdot r$. □

**Example 6.8.** Let $\mathcal{F} = \{a, f\}$ with $\text{rank}(a) = 0$, $\text{rank}(f) = 2$ and consider the tree $t = fafafafafaa$. A traversal SLP for $t$ is given by

$$S \to A_2 A_2 A_1 a,\ A_2 \to A_1 A_1,\ A_1 \to fa.$$

We obtain an TSLP with the following rules:

$$S \to A_2'(A_2'(A_1'(a)))),\ A_2'(x) \to A_1'(A_1'(x)),\ A_1'(x) \to f(a, x).$$

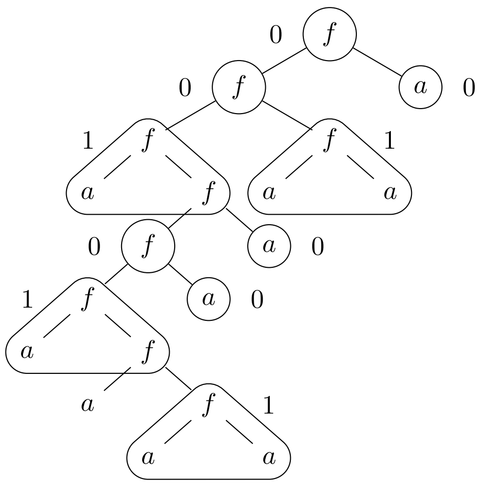


Figure 6.3: An example tree for the proof of Theorem 6.9

## 6.3.2 Traversal SLPs versus balanced parenthesis representation

*Balanced parenthesis sequences* are widely used as a succinct representation of ordered unranked unlabelled trees [MR01]. One defines the balanced parenthesis representation $\mathsf{bp}(t)$ of such a tree $t$ inductively as follows: If $t$ consists of a single node, then $\mathsf{bp}(t) = ()$. If the root of $t$ has $n$ children in which the subtrees $t_1, \dots, t_n$ are rooted (from left to right), then $\mathsf{bp}(t) = (\mathsf{bp}(t_1) \cdots \mathsf{bp}(t_n))$. Hence, a tree with $n$ nodes is represented by $2n$ bits, which is optimal in the information-theoretic sense. On the other hand, an unlabelled full binary tree $t$ (i.e., a tree where every non-leaf node has exactly two children) of size $n$ can be represented with $n$ bits by viewing $t$ as a ranked tree over $\mathcal{F} = \{a, f\}$, where $f$ has rank two and $a$ has rank zero.

**Theorem 6.9.** *For every $n > 0$ there exists an unlabelled full binary tree $t_n$ such that the size of a smallest traversal SLP for $t_n$ is polynomial in $n$, but the size of a smallest SLP for $\mathsf{bp}(t_n)$ is in $\Omega(2^{n/2})$.*

*Proof.* Let us fix an $n$ and let $u_n, v_n \in \{0,1\}^*$ be the strings from Theorem 6.5. Let $|u_n| = |v_n| = m$. We define $t_n$ by

$$t_n = \varphi_1(\mathrm{rev}(u_n))\, a\, \varphi_2(v_n)$$

where $\varphi_1, \varphi_2 : \{0,1\}^* \to \{a, f\}^*$ are the homomorphisms defined as follows:

$$\begin{aligned} \varphi_1(0) &= f, & \varphi_2(0) &= a, \\ \varphi_1(1) &= faf, & \varphi_2(1) &= faa. \end{aligned}$$

It is easy to see that $t_n$ is indeed a tree: $\mathrm{gaps}(\varphi_1(\mathrm{rev}(u_n))) = m{+}1$ and $\varphi_2(v_n)$ is a sequence of $m$ many trees. From the SLPs for $u_n$ and $v_n$ we obtain a traversal SLP for $t_n$ of size polynomial in $n$. It remains to show that the smallest traversal SLP for $\mathsf{bp}(t_n)$ has size $\Omega(2^{n/2})$. To do so, we show that from an SLP for $\mathsf{bp}(t_n)$ we can obtain with a linear size increase an SLP for the convolution of $u_n$ and $v_n$. In fact, we show the following claim:

*Claim.* The convolution $u_n \otimes v_n$ can be obtained from a suffix of $\mathsf{bp}(t_n)$ by a fixed rational transformation (i.e., a deterministic finite automaton that outputs along every transition a finite word over some output alphabet).

This claim proves the theorem using the following two facts:

- Let $\mathbb{A}$ be an SLP. An SLP for a suffix of $\mathrm{val}(\mathbb{A})$ can be produced by an SLP of size $\mathcal{O}(|\mathbb{A}|)$ [Loh14].
- For every fixed rational transformation $\rho$, an SLP for $\rho(\mathrm{val}(\mathbb{A}))$ can be produced by an SLP of size $\mathcal{O}(|\mathbb{A}|)$ [BCR08, Theorem 1] (the $\mathcal{O}$-constant depends on the rational transformation).

To see why the above claim holds, it is the best to look at an example. Assume that $u_n = 10100$ and $v_n = 10010$. Hence, we have

$$t_n = \varphi_1(\mathrm{rev}(u_n))\, a\, \varphi_2(v_n) \;=\; f\ f\ faf\ f\ faf\ a\ faa\ a\ a\ faa\ a.$$

This tree is shown in Figure 6.3. We have

$$\mathsf{bp}(t_n) = \underset{0}{(}\ \underset{0}{(}\ \underbrace{(()(}_{1}\ \underset{0}{(}\ \underbrace{(()(}_{1}\ \underset{a}{()}\ \underbrace{((()())))}_{(1,1)}\ \underbrace{())}_{(0,0)}\ \underbrace{()))}_{(1,0)}\ \underbrace{((()()))}_{(0,1)}\ \underbrace{())}_{(0,0)}\,.$$

Indeed, $\mathsf{bp}(t_n)$ starts with an encoding of the string $\mathrm{rev}(u_n)$ (here 00101) via the correspondence $0 \mathrel{\widehat{=}}$ "(" and $1 \mathrel{\widehat{=}}$ "(()(", followed by "()" (which encodes the single $a$ between $\varphi_1(\mathrm{rev}(u_n))$ and $\varphi_2(v_n)$ in $t_n$), followed by the desired encoding of the convolution $u_n \otimes v_n$. The latter is encoded by the following correspondence:

$$\begin{aligned}
(0,0) &\mathrel{\widehat{=}} ()),\\
(1,0) &\mathrel{\widehat{=}} ())),\\
(0,1) &\mathrel{\widehat{=}} ((()())),\\
(1,1) &\mathrel{\widehat{=}} ((()()))).
\end{aligned}$$

Thus a "0" (resp. a "1") in the second component is encoded by "()" (resp. by "((()())"), which corresponds to the tree $a$ (resp., $faa$). A "0" (resp. a "1") in the first component is encoded by one (resp. two) closing parenthesis.

Note that the strings

$$()),\ ())),\ ((()())),\ ((()())))$$

form a prefix code. This allows to replace these strings by the convoluted symbols $(0,0), (1,0), (0,1)$, and $(1,1)$, respectively, by a deterministic rational transducer. This shows the above claim. $\square$

The *depth-first-unary-degree-sequence* (DFUDS) of an unlabelled tree $t$ is defined as follows [BDM+05]: Traverse the tree in preorder and write down for every node with $d$ children the string "$(^d)$" ($d$ opening parenthesis followed by a closing parenthesis). Finally, add an additional opening parenthesis at the beginning of the string, which yields a well-balanced bracket expression. For instance, for the tree $g(f(a,a),a,h(a))$ we obtain the DFUDS-representation ( ((() (() ) ) ) () ).

Theorem 6.9 can be also interpreted as follows: For every $n > 0$ there exists a full binary tree $t_n$ such that the size of the smallest SLP for the DFUDS of $t_n$ is polynomial in $n$, but the size of the smallest SLP for the balanced parenthesis representation of $t_n$ is in $\Omega(2^{n/2})$. It remains open whether there is also a tree family where the opposite situation arises.

# Chapter 7

# Experiments

In this chapter we empirically compare the sizes of the different compression schemes discussed so far. We include

- the DAG, BDAG and HDAG,
- the SLP-compressed DAG as described in Algorithm 1 of Chapter 5 using RePair as string compressor (abbreviated *DAG-RP* for *DAG and Re-Pair*),
- traversal SLPs as described in Chapter 6, again using Re-Pair as string compressor (abbreviated *TravRP* for *traversal and Re-Pair*), and finally
- *TreeRePair* [LMM13], where we set *maxrank=1* since the other formalisms return one-parameter grammars when transformed to a TSLP (abbreviated *TreeRP*).

As noted in Chapter 6, *TravRP* may only be used for ranked trees. Since the trees in our corpora are unranked, we compress the traversals of the FCNS-encoding of the trees in that case.

## 7.1 Corpora

We use three corpora of XML files for our tests. For each XML document we consider the unranked tree of its element nodes; we ignore all other nodes such as texts, attributes, etc. One corpus (*Corpus I*) consists of XML documents that have been collected from the web, and which have often been used in the context of XML compression research, e.g. in [BGK03, BLM08, LMM13]. Each of these files is listed in Table 7.4 and the compressed sizes are listed in Table 7.4 and in Table 7.5. Precise references to the origin of these files can be found in [LMM13]. The second corpus (*Corpus II*) consists of all well-formed XML document trees with more than 10,000 edges and a depth of at least four from the *University of Amsterdam XML Web Collection*[1]. We decided on fixing a minimum size because there is no necessity to compress documents of very small size, and we chose a minimum depth because our subject is tree compression rather than list compression. Note that out of the over 180,000 documents

[1]`http://data.politicalmashup.nl/xmlweb/`, last visited 11 June 2016

| Corpus | Edges | Depth | Ch. p. Node |
|---|---|---|---|
| I | 1748169 | 4.0 | 5.4 |
| II | 79465 | 7.9 | 6.0 |
| III | 1531 | 18 | 1.5 |

Table 7.1: Average document characteristics: Average number of edges, average depth and the average number of nodes per child. The latter two values are unweighted.

| Corpus | Parse | DAG | HDAG | DAG-RP | TravRP | TreeRP |
|---|---|---|---|---|---|---|
| I | 35 | 43 | 46 | 48 | 91 | 175 |
| II | 85 | 105 | 120 | 117 | 247 | 310 |
| III | 6.9 | 8.7 | 9.2 | 10.0 | 8.0 | 14.8 |

Table 7.2: Cumulative Running times (in seconds).

of the collection, only 1,100 fit our criteria and are part of Corpus II (more than 27,000 were ill-formed and more than 140,000 had less than 10,000 edges). The documents in this corpus are somewhat smaller than those in Corpus 1, but otherwise have similar characteristics (such as average depth and average number of children), as can be seen in Table 7.1. The third corpus (*Corpus III*) consists of term rewriting systems[2] (see Chapter 9). Here we compress the entire XML-document tree structure (which also contains meta-information), not just the term rewriting system contained within it. We chose to add this third set of documents because its characteristics are atypical: While most XML-documents are typically very shallow (but wide), files from the TPDB can be quite the opposite (as the characteristics in Table 7.1 show).

## 7.2 Experiment Setup

For the DAG, BDAG and HDAG we built our own implementation. It is written in C++ (g++ version 4.9.2 with O3-switch) and uses Libxml 2.6 for XML parsing. For the Re-Pair-compressed DAG and traversal we use Gonzalo Navarro's implementation of Re-Pair[3], which is written in C. For TreeRePair we use Roy Mennicke's implementation[4] (which is written in C++) and run with *max_rank=1*, which produces monadic TSLP grammars. Since the implementations were written by different teams the accuracy of the running times is certainly limited.

Our test machine features an Intel Core i5 with 2.5Ghz and 4GB of RAM.

## 7.3 Comparison

**On the DAG, BDAG and HDAG.** Consider the accumulated numbers for the three corpora in Table 7.3. For the corpora I & II the HDAG is about

[2]`http://www.termination-portal.org/wiki/TPDB`
[3]`http://www.dcc.uchile.cl/~gnavarro/software/`
[4]`http://code.google.com/p/treerepair/`

| Corpus | Input | Dag | BDAG | HDAG | DAG-RP | TravRP | TreeRP |
|---|---|---|---|---|---|---|---|
| I | 36712 | 5769 | 7161 | 4607 | 2095 | 1256 | 1267 |
| II | 90036 | 13510 | 15950 | 10884 | 5162 | 3872 | 3957 |
| III | 2095 | 354 | 391 | 319 | 324 | 327 | 310 |

Table 7.3: Accumulated sizes (in thousand edges).

25% smaller than the DAG. The transformation to a monadic grammar (as in Theorem 5.5) is very inexpensive: Corpus I increases to 4657 edges (an increase of 1%) and Corpus II increases to 11109 edges. On the other hand Corpus III behaves differently: Here the improvement is only about 10%, and after transforming the HDAG to a monadic grammar the grammar size is larger than the DAG (362 vs. 354). This might be explained by the small number of average children in Corpus III: Not only is the additional child suffix compression inefficient, but also the transformation to a monadic grammar is rather expensive. Note that also DAG-RP, TravRP TreeRP do not perform much better than the DAG for Corpus III.

**On DAG-RP, TravRP and TreeRP.** The compression ratios of TravRP and TreeRP are very similar. While the prior is (slightly) smaller for Corpus I & II, the latter is (slightly) smaller for Corpus III. Yet the differences never exceed 5%. This is probably not too surprising: Consider the tree pattern $f(f(a,b),a)$. Opposed to TreeRP, TravRP may share the pattern $ba$ but fails to share $(f,2,b)$. In other words, TravRP can find patterns that are scattered across different trees but may fail to find a pattern from a node to a non-first child.

Concerning runtimes, TravRP is significantly faster for all three corpora (TravRP is between about 20% and 50% faster). The reason for this is most likely that TreeRP needs to keep track of an expensive tree pointer data structure.

The larger difference is between DAG-RP and the other RePair-variants. While the size of DAG-RP structures is significantly larger for Corpora I and II (around 67% for Corpus I and about 33% resp. 30% in Corpus II), the runtime is also equally better (about twice as fast as TravRP and three times as fast as TreeRP). A possible explanation is that Re-Pair (both in its tree and in its string version) is more expensive than a DAG construction, so when runtime is important it is advantageous to construct the DAG before applying Re-Pair.

| File | Edges | DAG | BDAG | HDAG |
|---|---:|---:|---:|---:|
| 1998statistics | 28305 | 1377 | 2403 | 1292 |
| catalog-01 | 225193 | 8554 | 6990 | 4555 |
| catalog-02 | 2240230 | 32394 | 52392 | 27457 |
| dictionary-01 | 277071 | 58391 | 77554 | 47418 |
| dictionary-02 | 2731763 | 545286 | 681130 | 414356 |
| EnWikiNew | 404651 | 35075 | 70038 | 35074 |
| EnWikiQuote | 262954 | 23904 | 47710 | 23903 |
| EnWikiVersity | 495838 | 43693 | 87276 | 43691 |
| EnWikTionary | 8385133 | 726221 | 1452298 | 726219 |
| EXI-Array | 226521 | 95584 | 128009 | 95563 |
| EXI-factbook | 55452 | 4477 | 5081 | 3847 |
| EXI-Invoice | 15074 | 1073 | 2071 | 1072 |
| EXI-Telecomp | 177633 | 9933 | 19808 | 9933 |
| EXI-weblog | 93434 | 8504 | 16997 | 8504 |
| JSTgene | 216400 | 9176 | 14606 | 7901 |
| JSTsnp | 655945 | 23520 | 40663 | 22684 |
| medline | 2866079 | 653604 | 740630 | 466108 |
| NCBIgene | 360349 | 16038 | 14356 | 11466 |
| NCBIsnp | 3642224 | 404704 | 809394 | 404704 |
| sprot39.dat | 10903567 | 1751929 | 1437445 | 1000376 |
| treebank | 2447726 | 1315644 | 1454520 | 1250741 |

Table 7.4: The XML documents in Corpus I and their compressed sizes, part 1

| File | Edges | DAG-RP | TravRP | TreeRP |
|---|---:|---:|---:|---:|
| 1998statistics | 28305 | 561 | 486 | 501 |
| catalog-01 | 225193 | 4372 | 4113 | 3965 |
| catalog-02 | 2240230 | 27242 | 29038 | 26746 |
| dictionary-01 | 277071 | 32139 | 21541 | 22375 |
| dictionary-02 | 2731763 | 267944 | 162658 | 167927 |
| EnWikiNew | 404651 | 9249 | 9394 | 9632 |
| EnWikiQuote | 262954 | 6328 | 6366 | 6608 |
| EnWikiVersity | 495838 | 7055 | 7067 | 7455 |
| EnWikTionary | 8385133 | 81781 | 81396 | 84107 |
| EXI-Array | 226521 | 905 | 908 | 1000 |
| EXI-factbook | 55452 | 1808 | 1417 | 1392 |
| EXI-Invoice | 15074 | 96 | 117 | 108 |
| EXI-Telecomp | 177633 | 110 | 132 | 140 |
| EXI-weblog | 93434 | 44 | 54 | 58 |
| JSTgene | 216400 | 3943 | 3691 | 4208 |
| JSTsnp | 655945 | 9809 | 10263 | 10327 |
| medline | 2866079 | 177638 | 119581 | 123817 |
| NCBIgene | 360349 | 6283 | 5108 | 5166 |
| NCBIsnp | 3642224 | 61 | 69 | 83 |
| sprot39.dat | 10903567 | 335756 | 268404 | 262964 |
| treebank | 2447726 | 1121566 | 524630 | 528372 |

Table 7.5: The XML documents in Corpus I and their compressed sizes, part 2

# Chapter 8

# Algorithmic Problems

In the previous chapters we have discussed different formalisms for the compact representation of (unranked) trees, namely the DAG, the BDAG, the HDAG, the SLP-compressed DAG representation, TSLPs and traversal SLP. In this section we want to discuss some algorithmic problems on these compressed tree representations.

We identify the nodes of a tree $t$ with the positions $1, \dots, |t|$ in the traversal of $t$. For a tree $t$ and an integer $m$, $t/m$ denotes the subtree of $t$ that is rooted at the $m^{\text{th}}$ node of $t$. Recall that $\gamma(v)$ returns the list of children of a node $v$.

## 8.1 Subtree equality check

We start with the following processing primitive, called *subtree equality check*: Given a tree $t$ and two nodes $p, q$, is $t/p = t/q$?

**Theorem 8.1.** *Let $t$ be an unranked tree with $N$ nodes. Given $g = dag(t)$ or $g = bdag(t)$ or an SLP-compressed DAG representation $g$ (this includes the HDAG) with $eval(g) = t$, one can, after $\mathcal{O}(|g|)$ time preprocessing, check for given $p, q$ whether $t/p = t/q$ in time $\mathcal{O}(\log N)$.*

*Proof.* Let $t \in \mathcal{T}(\Sigma)$. We first consider $g = \text{dag}(t)$. For $1 \le p \le N$ let $y_p$ be the unique node of $g$ such that $\text{eval}(y_p) = t/p$. Then $t/p = t/q$ if and only if $y_p = y_q$. Hence it suffices to show that $y_p$ can be computed in time $\mathcal{O}(\log N)$ after $\mathcal{O}(|g|)$ time preprocessing. For this we use techniques from [BLR+15]: We construct in time $\mathcal{O}(|g|)$ an SLP $\mathbb{G}$ for the word $y_1 y_2 \dots y_N \in \{1, \dots, N\}^*$: We introduce for every node $v \in g$ a nonterminal $\hat{v}$. If $\gamma(v) = v_1 \dots, v_n$, then we set $\hat{v} = v\hat{v_1} \dots \hat{v_n}$. Indeed we have $\text{eval}(\mathbb{G}) = y_1 y_2 \dots y_N$ and $|\mathbb{G}| = |g|$.

It now suffices to show that for a given number $p \in [1..N]$ the $p^{\text{th}}$ symbol of $\mathbb{G}$ can be computed in time $\mathcal{O}(\log N)$ after $\mathcal{O}(|g|) = \mathcal{O}(|\mathbb{G}|)$ time preprocessing. This is possible by [BLR+15, Theorem 1.1] (actually, $\mathbb{G}$ must be in Chomsky normal form in order to apply this result, but that can also be done in time $\mathcal{O}(|g|)$).

For an SLP-compressed DAG $D = (N_t, N_s, \Sigma, P_t, P_s, S)$ for $t$, essentially the same procedure as for the DAG applies. The set of nonterminals for the SLP $\mathbb{G}$ is $\{\hat{u} \mid u \in N_t\} \cup N_s$. For $u \in N_t$ with $\gamma(u) = \alpha_1 \dots \alpha_n$ (with $\alpha_i \in N_t \cup N_s$) we set $\hat{u} \to u\hat{\alpha_1} \dots \hat{\alpha_n}$, where $\hat{\alpha_i} = \hat{v}$ if $\alpha_i = v \in N_t$ and $\hat{\alpha_i} = \alpha_i$ otherwise.

The right-hand sides for the $\mathbb{G}$-nonterminals from $N_s$ are simply copied from the string productions with every occurrence of a symbol $u \in N_t$ replaced by $\hat{u}$.

Finally, for $g = \mathrm{bdag}(t)$ we proceed similarly. Let $U$ be the set of the nodes of $g$. Again we construct in time $\mathcal{O}(|g|)$ an SLP $\mathbb{G}$. The set of $\mathbb{G}$'s nonterminals is $\{\hat{u} \mid u \in U\}$. For every $u \in U$ with $\gamma(u) = u_1u_2$ we set $\hat{u} \to u\alpha_1\alpha_2$, where $\alpha_i = \varepsilon$ if $u$ does not have an $i^{\text{th}}$ child and $\alpha_i = \hat{u}_i$ otherwise. Note that for given preorder numbers $1 \le p, q \le N$, the $p^{\text{th}}$ symbol of $\mathrm{eval}(\mathbb{G})$ is equal to the $q^{\text{th}}$ symbol of $\mathrm{eval}(\mathbb{G})$ if and only if the sibling sequences at the nodes $p$ and $q$ are equal, yet we want to check whether the subtrees rooted at $p$ and $q$ are equal. For this, assume that we have computed the $p^{\text{th}}$ and the $q^{\text{th}}$ symbol $y_p$ and $y_q$ of $\mathrm{eval}(\mathbb{G})$ as above. Then $t/p = t/q$ if the following two conditions hold: (i) $\lambda(y_p) = \lambda(y_q)$ (recall that $\lambda(i)$ returns the label of the $i^{\text{th}}$ node) and (ii) either $y_p$ and $y_q$ do not have left children in $g$ or the left children coincide. Since these checks only require constant time, we obtain the desired time complexity. □

Note that the proof for the BDAG implies that we can check whether *sibling sequences* coincide in BDAGs. Lemma 8.2 below shows that we can do this for the DAG and the HDAG as well.

We observe that for TSLPs and for traversal SLPs a result such as the one of Theorem 8.1 is not known. To our knowledge, the fastest way of checking $t/p = t/q$ for a given TSLP $\mathbb{G}$ for $t$ works as follows: Assume that the subtree of $t$ rooted at $p$ (resp. $q$) consists of $m$ (resp. $n$) nodes. Then we have to check whether the substring of $\mathrm{eval}(G)$ from position $p$ to position $p+m-1$ is equal to the substring from position $q$ to position $q+n-1$. Using Plandowski's result [Pla94], this can be checked in time polynomial in the size of $G$.

Note that more efficient alternatives than Plandowski's algorithm exist (see [Loh12] for a survey), but all of them require at least quadratic time in the size of the SLP grammar.

As mentioned before, one can compute a traversal SLP for a tree $t$ from a TSLP for $t$, thus subtree equality checking can be done for TSLPs in polynomial time as well.

For DAGs, BDAGs and HDAGs we even have a stronger result, namely that we can check equivalence of sibling sequences:

**Lemma 8.2.** *Let $t$ be an unranked tree with $N$ nodes. Given $g = dag(t)$ or $g = bdag(t)$ or $g = hdag(t)$ we can, after $\mathcal{O}(|g|)$ time preprocessing, check for given $1 \le p, q \le N$, whether $sibseq(p) = sibseq(q)$ in time $\mathcal{O}(\log N)$.*

*Proof.* The result for the DAG follows from the HDAG-case, since the HDAG can be constructed from the DAG in time linear in the DAG's size. The BDAG case has already been proven in Theorem 8.1. Hence it remains to consider the HDAG $g$. For this we assume that $g$ is given as a SLP-compressed DAG $g = (N_t, N_s, \Sigma, P_t, P_s, S)$ which is *right regular* (see page 66 for a short discussion), meaning that for every nonterminal $X \in N_t$, $\mathrm{rhs}(X) \in N_s^* N_t \cup N_s^+$, and similarly for every $Y \in N_s$ we have $\gamma(Y) \in N_t^* N_s \cup N_t^*$. After introducing additional nonterminals, we can assume that for every $X \in N_s$ we have $\mathrm{rhs}(X) \in N_t N_s \cup N_t$, and for every $Y \in N_t$ we have $\gamma(Y) \in N_s \cup \{\varepsilon\}$ (this transformation can be done in time $\mathcal{O}(|g|)$). Then the elements of $\mathrm{sib}(t) \setminus t$ correspond to the elements of $N_s$ (recall that $\mathrm{sib}(t)$ is defined as the set of all sibling sequences of $t$, see Chapter 4).

We now construct an SLP $\mathbb{G}$ as follows: The set of nonterminals of $\mathbb{G}$ is $\{\hat{X} \mid X \in N_s\} \cup N_t$ and the set of terminals is $N_s$. The start nonterminal is the root $r \in N_t$ of the HDAG. For every $Y \in N_t$ we set

$$Y \to \begin{cases} \varepsilon & \text{if } \gamma(Y) = \varepsilon, \\ \hat{X} & \text{if } \gamma(Y) = X \in N_s. \end{cases}$$

For every $X \in N_s$ we set

$$Y \to \begin{cases} Xv\hat{Y} & \text{if } \mathrm{rhs}(X) = vY, v \in N_t, Y \in N_s, \\ Xv & \text{if } \mathrm{rhs}(X) = X \in N_s. \end{cases}$$

Then $\mathrm{sibseq}(p) = \mathrm{sibseq}(q)$ holds for $1 \leq p, q \leq N$ if and only if $p = q = 1$ or if $p > 1$, $q > 1$ and the $(p-1)^{\mathrm{st}}$ symbol of $\mathrm{eval}(\mathbb{G})$ is equal to the $(q-1)^{\mathrm{st}}$ symbol of $\mathrm{eval}(\mathbb{G})$. We deal with the case $p = q = 1$ separately because the sibling sequence $t$ corresponding to the root of $t$ is not represented in $\mathrm{eval}(\mathbb{G})$.

□

## 8.2 Tree Automata

*Tree automata* are a generalization of the ubiquitous string automata to trees. They play an important role in applications in which trees need to be processed automatically. In this context the *uniform membership problem* is of special interest: It asks whether a given tree automaton $\mathcal{A}$ and accepts a given tree $t \in \mathcal{T}(\mathcal{F})$.

**Definition 8.3** ([CDG+07])**.** A *nondeterministic (bottom-up) tree automaton* is a tuple $\mathcal{A} = (Q, \mathcal{F}, Q_F, \Delta)$ where $Q$ is a finite set of *states*, $\mathcal{F}$ is a ranked alphabet, $Q_F \subseteq Q$ is a set of *final states* and $\Delta$ is a set of *transition rules* of the form $f(q_1, \dots, q_n) \to q$, where $f \in \mathcal{F}_n$, and $q_1, \dots, q_n, q \in Q$. A tree $t \in \mathcal{T}(\mathcal{F})$ is *accepted* by $\mathcal{A}$ if $t \to_\Delta^* q$ for some $q \in Q_F$.

In [Loh01] it is shown that the uniform membership problem is complete for LogCFL, which is contained in P: It is the closure of context-free languages under logspace reductions. If the input tree is given by a TSLP, the uniform membership problem is P-complete [LM06].

Together with Theorem 5.5 we immediately get the following lemma:

**Lemma 8.4.** *Given a tree automaton $\mathcal{A}$ and an SLP-compressed DAG grammar $D$ for a tree $t$, we can decide in polynomial time whether $\mathcal{A}$ accepts $t$.*

In contrast, the problem is PSPACE-complete for traversal SLPs.

**Theorem 8.5.** *Given a tree automaton $\mathcal{A}$ and an SLP $\mathbb{A}$ for a tree $t \in \mathcal{T}(\mathcal{F})$, it is* PSPACE*-complete to decide whether $\mathcal{A}$ accepts $t$. Moreover,* PSPACE*-hardness already holds for a fixed tree automaton.*

*Proof.* For the upper bound we use the following lemma from [LM13]: If a function $f : \Sigma^* \to \Gamma^*$ is PSPACE-computable and $L \subseteq \Gamma^*$ is in $\mathsf{NSPACE}(\log^k(n))$ for some constant $k$, then $f^{-1}(L)$ belongs to PSPACE. Given an SLP $\mathbb{A}$ for the tree $t = \mathrm{val}(\mathbb{A})$, one can compute the tree $t$ by a PSPACE-transducer by computing the symbol $t[i]$ for every position $i \in \{1, \dots, |t|\}$. The current position can be stored in polynomial space and every query can be performed in polynomial time. As remarked above the uniform membership problem for explicitly given trees can be solved in $\mathsf{DSPACE}(\log^2(n))$.

For the lower bound we use a fixed regular language $L \subseteq (\{0,1\}^2)^*$ from [Loh11] such that the following problem is $\mathsf{PSPACE}$-complete: Given two SLPs $\mathbb{A}$ and $\mathbb{B}$ over $\{0,1\}$ with $|\mathrm{val}(\mathbb{A})| = |\mathrm{val}(\mathbb{B})|$, is $\mathrm{val}(\mathbb{A}) \otimes \mathrm{val}(\mathbb{B}) \in L$?[1]

Let $\mathcal{A} = (Q, \{0,1\}^2, \Delta, q_0, F)$ be a finite word automaton for $L$. Let $\mathbb{A}, \mathbb{B}$ be two SLPs over $\{0,1\}$ with $|\mathrm{val}(\mathbb{A})| = |\mathrm{val}(\mathbb{B})|$ and let $\mathbb{T}$ be an SLP for the comb tree $t(u,v)$ where $u = \mathrm{rev}(\mathrm{val}(\mathbb{A}))$ and $v = \mathrm{rev}(\mathrm{val}(\mathbb{B}))$. We transform $\mathcal{A}$ into a tree automaton $\mathcal{A}_T$ over $\{f_0, f_1, 0, 1, \$\}$ with the state set $Q \uplus \{p_0, p_1\}$, the set of final states $F$ and the following transitions:

$$\begin{aligned} \$ &\to q_0, \\ i &\to p_i \quad \text{for } i \in \{0,1\}, \\ f_i(q, p_j) &\to q' \quad \text{for } (q,(i,j),q') \in \Delta. \end{aligned}$$

The automaton $\mathcal{A}$ accepts the convolution $\mathrm{val}(\mathbb{A}) \otimes \mathrm{val}(\mathbb{B})$ if and only if the tree automaton $\mathcal{A}_T$ accepts $t(u,v)$. □

It may be worth noting that the uniform membership problem is $\mathsf{PSPACE}$-complete for nonlinear TSLPs [LM06].

## 8.3 Tree Navigation

In [BLR+15] it is shown that for a given SLP $\mathbb{A}$ of size $n$ that produces the balanced parenthesis representation (see Section 6.3.2) of an unranked tree $t$ of size $N$, one can produce in time $\mathcal{O}(n)$ a data structure of size $\mathcal{O}(n)$ that supports navigation as well as other important tree queries (e.g. lowest common ancestors queries) in time $\mathcal{O}(\log N)$. An analogous result is known for *top DAGs* [BGLW15], see also [HR15]. Both results use the *word RAM model*[2], where memory cells can store numbers with $\log N$ bits and arithmetic operations on $\log N$-bit numbers can be carried out in constant time. Here we show the same result for traversal SLPs.

**Theorem 8.6.** *Given a traversal SLP $\mathbb{A}$ of size $n$ for a tree $t \in \mathcal{T}(\mathcal{F})$ of size $N$, one can produce in time $\mathcal{O}(n)$ a data structure of size $\mathcal{O}(n)$ that allows to do the following computations in time $\mathcal{O}(\log N) \le \mathcal{O}(n)$ on a word RAM, where $i, j, k \in \mathbb{N}$ with $1 \le i, j \le N$ are given in binary notation:*

1. *Compute the parent node of the node $i > 1$ in $t$.*
2. *Compute the $k^{th}$ child of the node $i$ in $t$, if it exists.*
3. *Compute the number $k$ such that $i > 1$ is the $k^{th}$ child of its parent node.*
4. *Compute the size of the subtree rooted at the node $i$.*
5. *Compute the lowest common ancestor of the nodes $i$ and $j$ in $t$.*

[1] Recall that $v \otimes u$ denotes the convolution of the strings $u$ and $v$, see page 72.
[2] Also known as *transdichotomous model* [FW93].

Recall that in [BLM08] it is shown that a TSLP $\mathbb{A}$ producing a ranked tree $t \in \mathcal{T}(\mathcal{F})$ can always be transformed into a traversal SLP of size $\mathcal{O}(|\mathbb{A}| \cdot r)$ for $t$, where $r$ is the maximal rank of a label occurring in $t$. Thus, by prior transforming an TSLP to a traversal SLP, Theorem 8.6 also applies to TSLPs for ranked trees.

*Proof.* In [BLR+15] it is shown that for an SLP $\mathbb{A}$ of size $n$ that produces a well-parenthesized string $w \in \{(,)\}^*$ of length $N$, one can produce in time $\mathcal{O}(n)$ a data structure of size $\mathcal{O}(n)$ that allows to do the following computations in time $\mathcal{O}(\log N)$ on a word RAM, where $1 \leq k, j \leq N$ are given in binary notation and $b \in \{(,)\}$:

- Compute the number of positions $1 \leq i \leq k$ such that $w[i] = b$ ($\mathsf{rank}_b(k)$).
- Compute the position of the $k^{\text{th}}$ occurrence of $b$ in $w$ if it exists ($\mathsf{select}_b(k)$).
- Compute the position of the matching closing (resp., opening) parenthesis for an opening (resp., closing) parenthesis at position $k$ ($\mathsf{findclose}(k)$ and $\mathsf{findopen}(k)$).
- Compute the left-most position $i \in [k, j]$ having the smallest excess value in the interval $[k, j]$, where the excess value at a position $i$ is $\mathsf{rank}_((i) - \mathsf{rank}_)(i)$ ($\mathsf{rmqi}(k, j)$).

Let us now take a traversal SLP $\mathbb{A}$ of size $n$ for a tree $t \in \mathcal{T}(\mathcal{F})$ of size $N$ and let $s$ be the corresponding unlabelled tree.

Recall the definition of DFUDS-representation [BDM+05] of a tree $s$ (from page 76): Walk over the tree in preorder and write down for every node with $d$ children the string "$(^d)$". Finally add an additional opening parenthesis at the beginning of the resulting string. Clearly, from the traversal SLP $\mathbb{A}$ we can produce an SLP $\mathbb{B}$ for the DFUDS-representation of the tree $s$: Simply replace in the right-hand sides every occurrence of a symbol $f$ of rank $d$ by $(^d)$, and add an opening parenthesis in front of the right-hand side of the start nonterminal.

The starting position of the encoding of a node $i \in \{1, \ldots, N\}$ in the DFUDS-representation can be found as $\mathsf{select}_)(i-1)+1$ for $i > 1$, and for $i = 1$ it is 2. Vice versa if $k$ is the starting position of the encoding of a node in the DFUDS-representation, then the preorder number of that node is $\mathsf{rank}_)(k-1)+1$.

In [BDM+05, JSS12], it is shown that the tree navigation operations from the theorem can be implemented on the DFUDS-representation using a constant number of $\mathsf{rank}$, $\mathsf{select}$, $\mathsf{findclose}(k)$, $\mathsf{findopen}(k)$ and $\mathsf{rmqi}$-operations. Together with the above mentioned results from [BLR+15] this shows the theorem. □

The data structure of [BLR+15] allows to compute the depth and height of a given tree node in time $\mathcal{O}(\log N)$ as well. It is not clear, whether this result can be extended to our setting as well. In [JSS12] it is shown that the depth of a given node can be computed in constant time on the DFUDS-representation. But this uses an extra data structure, and it is not clear whether this extra data structure can be adapted so that it works for an SLP-compressed DFUDS-representation. On the other hand, in [GHLN15] it is shown that the height of the tree and the depth of a given node of a traversal SLP be computed in polynomial time.

# Chapter 9

# Accelerated computation by compression in term rewriting

In the last chapter we treated algorithms on compressed data. In this chapter we show an example on how compression can lead to faster runtimes. The application we highlight is *automatic termination proving of term rewriting systems* (TRS) and the results were first presented in [BLNW13]. Here one is given a term rewriting system, which consists of a set of rules on how terms can be manipulated (a precise definition is given below) and the question of interest is whether the given system terminates for every input. This question also arises in applied settings, e.g. compiler construction. Other important questions in this context are whether a given TRS is *normalizing*, meaning that every term has a unique normal form, or whether a given system is *confluent*, meaning that terms can be rewritten differently, yet yielding the same result. We do not address these properties in this chapter.

Term rewriting systems are Turing-complete, and thus it is clearly not surprising that the general question (whether a given TRS is terminating) is undecidable. Nevertheless, in some cases termination can be proven automatically.

**Example 9.1.** As a first example, consider the *string rewriting system* $S$ over $\Sigma = \{a, b\}$ with the rules $\{(abb \to b), (ba \to a)\}$. Since both rules reduce the length of the string, it is clear that the rules can only be applied a finite number of times for every string $s$, and thus this system is terminating.

**Example 9.2.** Let $P = \mathbb{Z}[x]$ be the set of all integer polynomials, let $a, b \in \mathbb{Z}$, $f, g \in \mathbb{Z}[x]$ and let $R$ be given by

$$\begin{aligned} R = \{d_x c \to 0, \qquad & d_x(ax^n + bx^m) \to d_x ax^n + d_x bx^m, \\ d_x x^n \to nx^{n-1}, \qquad & d_x(f \cdot g) \to d_x f \cdot g + f \cdot d_x g\}. \end{aligned}$$

Then applying rules from $R$ act as symbolic differentiation on $P$. Since every rule either reduces the number of $d_x$-symbols or the number of brackets, this system is also terminating.

For both examples we proved termination by a *monotone embedding* of the rules set into $(\mathbb{N}, >)$.

Other embedding options are polynomials [Lan79] and matrices [EWZ08]. We focus on matrix interpretations. Here, one interprets the function symbols of the term rewriting system as linear mappings represented by matrices. This way termination can be proven e.g. if every application of a rule lessens a (set of) parameter(s), much like every rule application from Example 9.1 makes the parameter size *smaller*.

The conditions of monotonicity and compatibility with the given rewriting system result in a constraint system for the coefficients of the matrix interpretation. Constraint solvers are used to obtain an actual interpretation. The aim of the compression here is to minimize the number of involved matrix multiplications in the interpretation. Our experiments show that this leads to smaller constraint systems, which result in faster runtimes for the constraint solvers. Since our focus here is on minimizing the number of matrix multiplications, we will treat the constraint system solver as a black box.

The rest of this chapter is structured as follows. In Section 9.1 we provide some context on term rewriting and termination proving, including the matrix interpretation we use here. In Section 9.2 we first recall the algorithm *TreeRePair* from [LMM13] and then show how TreeRePair can be adjusted to minimize the *matrix multiplication cost* of a term rewriting system. We call the new algorithm *MCTreeRePair* (*MC* for *matrix cost*). In Section 9.3 we discuss using MCTreeRePair together with the *dependency pairs transformation.* Section 9.4 shows the results of the experiments we made. Finally Section 9.5 poses some open problems.

## 9.1 Term Rewriting

In this section we provide the necessary background from term rewriting. For a detailed discussion we refer to [BN98]. Let $\mathcal{F}$ be a ranked alphabet (also called *signature* in the context of term rewriting) and let $k$ be maximal such that $\mathcal{F}_k \neq \emptyset$. In this section we will refer to elements from $\mathcal{T}(F)$ as *terms* rather than *trees* and view terms as prefix-closed sets:

**Definition 9.3.** A *term* over $\mathcal{F}$ is a pair $t = (D, \lambda)$ where $D$ is a finite prefix-closed and non-empty subset of $\{1, \ldots, k\}^*$ and $\lambda$ is a function from $D$ to $\mathcal{F}$ such that for all $p \in D$ and $1 \leq d \leq k$: $pd \in D$ iff $1 \leq d \leq \text{rank}(\lambda(p))$.

Elements from $D$ are also called *nodes* or *positions.* The size of a tree is defined as the number of nodes it contains. Let $V$ be a set of variables. We define $\mathcal{T}(\mathcal{F}, V) = \mathcal{T}(\mathcal{F} \cup V)$. For a term $t \in \mathcal{T}(\mathcal{F}, V)$, let $\text{Var}(t)$ be the set of all variables in $t$ that occur at least once in $t$.

**Definition 9.4.** A *term rewriting system* (TRS) over the signature $\mathcal{F}$ is a finite set $R \in \mathcal{T}(\mathcal{F}, V) \times \mathcal{T}(\mathcal{F}, V)$ of rules such that for every rule $(l \to r) \in R$ we have $l \notin V$ and $\text{Var}(r) \subseteq \text{Var}(l)$. The *one-step rewriting relation* is defined as usual.

### 9.1.1 Termination and matrix interpretations

**Definition 9.5** ([Zan94])**.** Let $\mathcal{F}$ be a ranked alphabet. A *well-founded monotone $\mathcal{F}$-algebra* $(S, >)$ is defined as an $\mathcal{F}$-algebra $S$, such that the underlying set is equipped with a well-founded order $>$ and each algebra operation is strictly monotone on all of its coordinates.

Next we define our matrix interpretations. We fix a semiring $S$ (the ring of matrix coefficients) and a dimension $n$. For a set of variables $U \subseteq V$ we denote by $(S^n)^U$ the set of all mappings from $U$ to $S^n$.

We want to interpret a term $t$ with $m$ different variables as an $m$-ary function $(S^n)^m \to S^n$. Moreover, we fix for every symbol $f \in \mathcal{F}_m$ matrices $F_1, \dots, F_m \in S^{n \times n}$ and a vector $F_0 \in S^n$. This allows us to define the linear function $[f]$ by

$$[f](x_1, \dots, x_m) = F_0 + F_1 x_1 + \dots + F_m x_m, \tag{9.1}$$

where $x_1, \dots, x_m \in S^n$. Now let $t \in \mathcal{T}(\mathcal{F}, V)$ be a term with $U = \mathrm{Var}(t)$. The interpretation $[t] : (S^n)^U \to S^n$ is computed by composing the interpretations for the ranked symbols in the natural way: Let $t = f(t_1, \dots, t_k)$. Then $[t](\bar{x}) = [f]([t_1](\bar{x}_1), \dots, [t_k](\bar{x}_k))$, where $\bar{x} \in (S^n)^U$ and $\bar{x}_i$ is the restriction of $\bar{x}$ to $\mathrm{Var}(t_i) \subseteq U$.

We say that a non-empty well-founded monotone $\mathcal{F}$-algebra $S$ is *compatible* with a TRS if $l >_A r$ for every rule $l \to r$ in the term rewriting system.

**Theorem 9.6** ([Zan94], Prop. 1)**.** *A TRS is terminating if and only if it admits a compatible non-empty well-founded monotone algebra.*

Define $>$ as follows:

$$(a_1, \dots, a_n) > (b_1, \dots, b_n) \Leftrightarrow a_1 > b_1 \wedge a_i \geq b_i \text{ for } i = 2, \dots, n.$$

Theorem 9.6 and the above definitions show a way to prove termination: If the TRS allows a compatible interpretation to $(\mathbb{N}^n, >)$ for a suitable dimension $n$, then it is terminating.

The following example is from the termination database [TPD].

**Example 9.7.** Consider the TRS $R$ consisting of the rules

$$f(x, c(y)) \to f(x, s(f(y, y))),$$
$$f(s(x), s(y)) \to f(x, s(c(s(y)))).$$

The matrix interpretations

$$[f](\bar{x}, \bar{y}) = \begin{pmatrix} 4 & 0 \\ 0 & 0 \end{pmatrix} \cdot \bar{x} + \begin{pmatrix} 1 & 4 \\ 0 & 0 \end{pmatrix} \cdot \bar{y} + \begin{pmatrix} 0 \\ 0 \end{pmatrix},$$
$$[s](\bar{x}) = \begin{pmatrix} 1 & 0 \\ 0 & 0 \end{pmatrix} \cdot \bar{x} + \begin{pmatrix} 1 \\ 0 \end{pmatrix} \text{ and}$$
$$[c](\bar{x}) = \begin{pmatrix} 1 & 3 \\ 1 & 1 \end{pmatrix} \cdot \bar{x} + \begin{pmatrix} 0 \\ 1 \end{pmatrix}$$

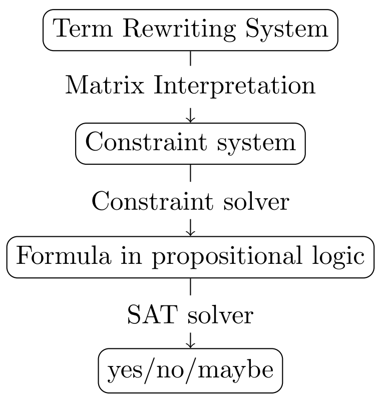


Figure 9.1: Automatic Termination proving

show that $R$ is terminating: We have

$$\begin{aligned}
[f(x, c(y))](\bar{x}, \bar{y}) &= \begin{pmatrix} 4 & 0 \\ 0 & 0 \end{pmatrix} \cdot \bar{x} + \begin{pmatrix} 5 & 7 \\ 0 & 0 \end{pmatrix} \cdot \bar{y} + \begin{pmatrix} 4 \\ 0 \end{pmatrix}, \\
[f(x, s(f(y, y)))](\bar{x}, \bar{y}) &= \begin{pmatrix} 4 & 0 \\ 0 & 0 \end{pmatrix} \cdot \bar{x} + \begin{pmatrix} 5 & 4 \\ 0 & 0 \end{pmatrix} \cdot \bar{y} + \begin{pmatrix} 1 \\ 0 \end{pmatrix}, \text{ and} \\
[f(s(x), s(y))](\bar{x}, \bar{y}) &= \begin{pmatrix} 4 & 0 \\ 0 & 0 \end{pmatrix} \cdot \bar{x} + \begin{pmatrix} 1 & 0 \\ 0 & 0 \end{pmatrix} \cdot \bar{y} + \begin{pmatrix} 5 \\ 0 \end{pmatrix}, \\
[f(x, s(c(s(y))))](\bar{x}, \bar{y}) &= \begin{pmatrix} 4 & 0 \\ 0 & 0 \end{pmatrix} \cdot \bar{x} + \begin{pmatrix} 1 & 0 \\ 0 & 0 \end{pmatrix} \cdot \bar{y} + \begin{pmatrix} 2 \\ 0 \end{pmatrix}.
\end{aligned}$$

Thus, given a TRS, a termination prover sets up a constraint system for the coefficients of the matrix interpretation. Typically, the dimensions of these matrices are small (e.g. $n \leq 5$, see Section 9.4). Note that this still results in a huge search space. Then the unknown coefficients are represented by sequences of boolean unknowns and the constraints are transformed to a formula in propositional logic, after which a SAT solver is used to find a satisfying assignment, from which the interpretation may be reconstructed. Figure 9.1 visualizes the process of automatic termination proving.

### 9.1.2 Cost of terms

We next define a cost measure for terms that approximates the amount of computation that is needed to calculate the coefficients of the linear function that is represented by the term. We are mainly interested in the number of necessary matrix-matrix multiplications. This is justified by the higher asymptotic runtime ($\mathcal{O}(n^3)$) of matrix multiplication as opposed to matrix-vector multiplication or matrix addition ($\mathcal{O}(n^2)$). Since the dimension of the chosen matrices is typically small (as mentioned before), matrix multiplication algorithms with asymptotic runtime better than $\mathcal{O}(n^3)$ (e.g. Strassen's algorithm) are not of interest here.

Let $t = f(t_1, \ldots, t_m) \in \mathcal{T}(\mathcal{F}, V)$ and $U = \mathrm{Var}(t) = \{x_1, \ldots, x_k\}$. Recall that $t$ represents a linear function $[t] : (S^n)^U \to S^n$, which can be written as $T_0 + T_1 x_1 + \cdots + T_k x_k$ with $T_0 \in S^n$ and $T_1, \ldots, T_k \in S^{n \times n}$. Define $\mathrm{coeff}_{x_i}(t) = T_i$

and assume that all the coefficient matrices of $t$, $\mathrm{coeff}_x(t_1), \ldots, \mathrm{coeff}_x(t_m)$, are already known. Then we can compute $\mathrm{coeff}_x(t)$ by

$$\mathrm{coeff}_x(t) = \sum_{i=1}^{m} F_i \cdot \mathrm{coeff}_x(t_i),$$

where $F_i$ is from (9.1), and we set $\mathrm{coeff}_x(t_i) = 0$ if $x \notin \mathrm{Var}(t_i)$. Note that the multiplication is trivial if $x = t_i$. This motivates the following definition:

**Definition 9.8.** The *(matrix multiplication) cost* of a term $t = (D, \lambda) \in \mathcal{T}(\mathcal{F}, V)$ is

$$\mathrm{cost}(t) = \sum_{p \in D \setminus \{\varepsilon\}, \lambda(p) \notin V} |\mathrm{Var}(t/p)|. \tag{9.2}$$

The cost of a tuple $(t_1, \ldots, t_m)$ of terms is $\sum_{i=1}^{m} \mathrm{cost}(t_i)$.

Note that this definition models a bottom-up evaluation where we do not use any caching, memoization, etc.

**Example 9.9.** Continuing Example 9.7, we have

$$\begin{aligned} \mathrm{cost}(f(x, c(y))) &= 1, & \mathrm{cost}(f(x, s(f(y, y)))) &= 2, \\ \mathrm{cost}(f(s(x), s(y))) &= 2, & \mathrm{cost}(f(x, s(c(s(y))))) &= 3. \end{aligned}$$

**Example 9.10.** As another example, let $\mathrm{rank}(h) = \mathrm{rank}(f) = 2$, $\mathrm{rank}(s) = 1$, and $\mathrm{rank}(0) = 0$ and consider the TRS

$$\begin{aligned} h(x, f(y, z)) &\to h(f(s(y), x), z), \\ h(f(s(x), f(s(0), y)), z) &\to h(y, f(s(0), f(x, z))). \end{aligned}$$

Then we have

$$\begin{aligned} \mathrm{cost}(h(x, f(y, z))) &= 2, & \mathrm{cost}(h(f(s(y), x), z)) &= 3, \\ \mathrm{cost}(h(f(s(x), f(s(0), y)), z)) &= 4, & \mathrm{cost}(h(y, f(s(0), f(x, z)))) &= 4. \end{aligned}$$

Figure 9.2 shows a detailed computation of the coefficients of the interpretation of the term $h(f(s(x), f(s(0), y)), z)$. This example has been taken from [EWZ08] and also appears in [BLNW13].

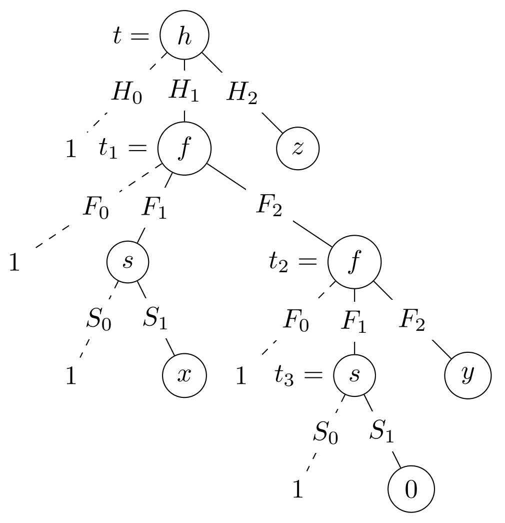
## 9.2 Term Compression with TreeRePair

### 9.2.1 TreeRePair

We first provide a detailed description of the TreeRePair-algorithm [LMM13] (see also Section 2.3.2). Let $\mathcal{F}$ be a ranked alphabet.

**Definition 9.11.** A *digram* over $\mathcal{F}$ is a triple $d = [f, i, g]$, where $f, g \in \mathcal{F}$ and $i \in [1..\mathrm{rank}(f)]$. The *rank* $\mathrm{rank}(d)$ of a digram is $\mathrm{rank}(f) + \mathrm{rank}(g - 1)$.

We regard a digram $d$ as a new symbol of rank $\mathrm{rank}(d)$.



coefficients of variables:

$$\begin{aligned}
\text{coeff}_x(t_1) &= F_1 \cdot S_1 \\
\text{coeff}_y(t_1) &= F_2 \cdot F_2 \\
\text{coeff}_x(t) &= H_1 \cdot \text{coeff}_x(t_1) \\
\text{coeff}_y(t) &= H_1 \cdot \text{coeff}_y(t_1)
\end{aligned}$$

absolute parts:

$$\begin{aligned}
\text{coeff}_1(t_3) &= S_1 \ \cdot 0_0 \\
\text{coeff}_1(t_2) &= F_0 \ + F_1 \cdot \text{coeff}_1(t_3) \\
\text{coeff}_1(t_1) &= F_0 \ + F_1 \cdot S_0 \\
&\qquad + F_2 \cdot \text{coeff}_1(t_2) \\
\text{coeff}_1(t) &= H_0 + H_1 \cdot \text{coeff}_1(t_1)
\end{aligned}$$

Figure 9.2: Bottom-up computation of the coefficient matrices of the term $h(f(s(x), f(s(0), y)), z)$.

**Definition 9.12.** To the digram $d = [f, i, g]$ with $\text{rank}(d) = n$ and $\text{rank}(g) = l$ we associate the rewriting rule

$$\text{rule}(d) = (d(x_1, \dots x_n) \to f(x_1, \dots x_{i-1}, g(x_i, \dots, x_{i+l-1}), x_{i+l}, \dots x_n)).$$

With $\text{rule}(d)^{-1}$ we denote the reverse rule

$$f(x_1, \dots x_{i-1}, g(x_i, \dots, x_{i+l-1}), x_{i+l}, \dots x_n)) \to d(x_1, \dots x_n).$$

The right-hand side of the rule $\text{rule}(d)$ can be seen as the tree pattern represented by the digram $d$.

**Definition 9.13.** A *compressed term list* is a list $(t_1, \dots, t_m \mid d_1, \dots, d_n)$, where

- for each $1 \leq i \leq n$, $d_i$ is a digram over the signature $\mathcal{F} \cup \{d_1, \dots, d_{i-1}\}$, and
- for each $1 \leq i \leq m$, $t_i \in \mathcal{T}(\mathcal{F} \cup \{d_1, \dots, d_n\}, V)$.

The idea is that a compressed term list $(t_1, \dots, t_m \mid d_1, \dots, d_n)$ represents the term list $(s_1, \dots, s_m)$ that is obtained by replacing nodes labelled with digram symbols by the corresponding tree patterns (of size 2). This motivates the following definition:

**Definition 9.14.** Let $(\bar{t} \mid \bar{d}) = (t_1, \dots, t_m \mid d_1, \dots, d_n)$ be a compressed term list. The *expansion* of $(\bar{t} \mid \bar{d})$ is the list $(s_1, \dots, s_m)$ where $s_i$ is the unique normal form of $t_i$ with respect to the (confluent and terminating) term rewriting system $\{\mathsf{rule}(d_1), \dots, \mathsf{rule}(d_n)\}$.

**Example 9.15.** As in Example 9.10, let $\text{rank}(h) = \text{rank}(f) = 2$, $\text{rank}(s) = 1$, $\text{rank}(0) = 0$, and consider the following compressed term list:

$$\begin{aligned}
&([h,2,f](x,y,z),\ [[h,1,f],1,s](y,x,z), \\
&\ [[h,1,f],1,s](x, f(s(0),y), z),\ [h,2,f](y, s(0), f(x,z)) \mid \\
&\ [h,1,f],\ [h,2,f],\ [[h,1,f],1,s]).
\end{aligned}$$

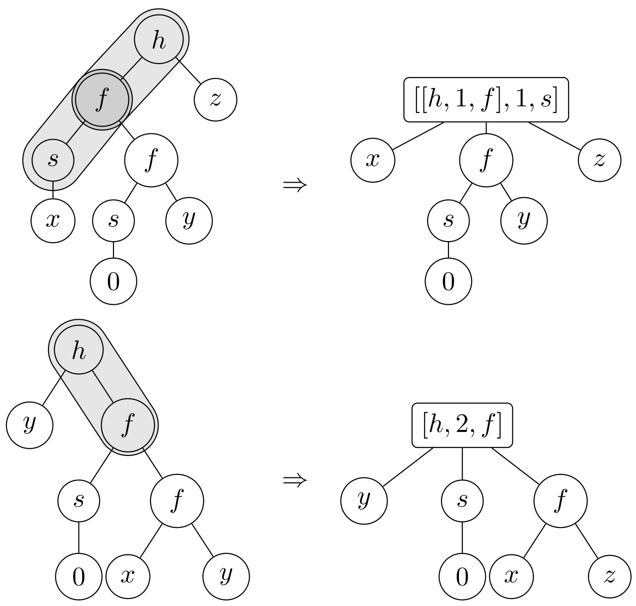


Figure 9.3: The replaced digrams from Example 9.15.

The expansion of this list is the following term list consisting of the terms from Example 9.10:

$$(h(x, f(y, z)),\ h(f(s(y), x), z),\ h(f(s(x), f(s(0), y)), z),\ h(y, f(s(0), f(x, z)))). \tag{9.3}$$

Figure 9.3 shows the replaced digrams of two terms from the above list.

In our applications, a term list will be a list of all left-hand and right-hand sides of a TRS. If term (list) compression is the main objective, then the goal is to compute a small compressed term list whose expansion is the input term list. For this let us define the size of a compressed term list $(t_1, \dots, t_m \mid d_1, \dots, d_n)$ as $\sum_{i=1}^{m} |t_i| + n$. This definition is justified by the fact that a digram can be stored by two symbols (either symbols from the initial signature or references to previously defined digrams) and an integer, which needs constant space in a uniform cost model. In Example 9.10 the compressed term list has size 25, whereas the expanded term list has size 28.

Let $d = [f, i, g]$ be a digram and let $\bar{t} = (t_1, \dots t_n)$ be a term list with $t_k = (D_k, \lambda_k)$ for $k \in [1..n]$. An *occurrence* of $d$ in $\bar{t}$ is a pair $(j, p)$, where $j \in [1..n]$ and $p$ is a position where $d$ occurs, i.e. $p \in D_j$, $\lambda(p) = f$ and $\lambda(pi) = g$. A set $\mathsf{Occ}_d(\bar{t})$ of occurrences of $d$ in $\bar{t}$ is *non-overlapping* if for every $(j, p) \in \mathsf{Occ}_d(\bar{t}) : (j, pi) \notin \mathsf{Occ}_d(\bar{t})$. If $f \neq g$ then every occurrence set is trivially non-overlapping. For non-overlapping occurrence sets we can apply the rewrite rule $\text{rule}^{-1}(d)$ at every occurrence of $d$ in $\mathsf{Occ}_d(\bar{t})$ simultaneously. Let $t'_j$ with $j \in [1..n]$ be the resulting terms. We write $(t_1, \dots, t_n) \rightarrow_{\mathsf{Occ}_d(\bar{t})} (t'_1, \dots, t'_n)$.

Let $\max_d(\bar{t})$ be the maximal size among all non-overlapping sets of occurrences of $d$ in $\bar{t}$. We can easily determine a set $\mathsf{maxOcc}_d(\bar{t})$ of non-overlapping occurrences of $d$ with $|\mathsf{maxOcc}_d(\bar{t})| = \max_d(\bar{t})$:

If $f \neq g$, we simply set $\mathsf{maxOcc}_d(\bar{t}) = \mathsf{Occ}_d(\bar{t})$. Otherwise we obtain

$\mathsf{maxOcc}_d(\bar{t})$ as follows. For every maximal chain $p, pi, \ldots pi^k$ of positions in $D_j$ such that $\lambda(pi^l) = f$ for every $f \in [1..k]$, we do the following: If $k$ is odd, we add the pairs $(j,p), (j,pi^2), \ldots, (j,pi^{k-1})$ to $\mathsf{maxOcc}_d(\bar{t})$. Otherwise we add $(j,p)$, $(j,pi^2)$, $\ldots, (j,pi^{k-2})$ to $\mathsf{maxOcc}_d(\bar{t})$. In the even-sized case we could have added $(j,pi)$, $(j,pi^3)$, $\ldots, (j,pi^{k-1})$ to $\mathsf{maxOcc}_d(\bar{t})$ instead and also obtain a set of size $\max(d,\bar{t})$. We chose the first option instead because it is the better choice for our adaption of TreeRePair, as will become clear in the next section. Setting $\bar{t} = (t_1, \ldots, t_n)$ and $\bar{d}$ as a set of digrams, a high-level description of the TreeRePair algorithm looks as follows:

```
**input**: term list $\bar{t} = (t_1, \ldots, t_n)$;
$\bar{d} = \emptyset$;
**while** $\exists\, d : \max_d(\bar{t}) > 1$ **do**
    let $d$ be a digram such that $\max_d(\bar{t}) \geq \max_{d'}(\bar{t})$ for all digrams $d'$;
    let $\bar{u}$ be such that $\bar{t} \rightarrow_{\mathsf{maxOcc}_d(\bar{t})} \bar{u}$;
    $\bar{t} := \bar{u}$;
    $\bar{d} := (\bar{d}, d)$;
**end**
**return** $(\bar{t} \mid \bar{d})$;
```

**Algorithm 2:** TreeRePair

**Bounding the maximal rank and self-overlapping digrams.** In the implementation of TreeRePair from [LMM13] the user can specify a parameter $r$ which bounds the maximal rank of the digrams considered (i.e. only digrams $d$ with $\mathrm{rank}(d) \leq r$ are replaced). This has two advantages:

1. This may lead to better compression rates, and actually did so for the test data in [LMM13] (which are large tree structures defined by XML files).
2. Bounding the maximal rank of digrams improves the runtime drastically.

Though the first point may seem surprising at first, it does make sense from a statistical point of view: By bounding the maximal rank of the digrams considered, we keep the ranks of the tree nodes smaller. In return this reduces the number of possible different digrams, which increases the probability that some digrams occur more frequently. See [LMM13] for a further explanation and a tree family where bounding the maximal rank leads to exponentially better compression.

Regarding the second point, recall that a naïve implementation of TreeRePair, which counts the digram frequencies anew at the beginning of each iteration of the **while**-loop, takes quadratic time. To achieve linear runtime instead, the implementation of [LMM13] chooses a pointer structure that only needs a constant amount of updating per **while**-iteration, yet possibly loses some occurrences of self-overlapping digrams. Since our adaption of TreeRePair uses the same principles, we elaborate on the respective implementation details.

The input terms of $\bar{t}$ are represented as pointer structures, where every node stores a pointer to its parent node and a list of pointers to its children. An occurrence $d = (j,p)$ of a digram in $\bar{t}$ is represented by a pointer to $p$ in $t_j$. Initially we save for each digram $d$ every occurrence from $\mathsf{maxOcc}_d(\bar{t})$ in a doubly linked list, one for every digram and the size of $\mathsf{maxOcc}_d(\bar{t})$ is counted.

This can be done in a single pass over the term list $\bar{t}$. Each time an occurrence $(j, p)$ of a digram is replaced, the following steps are done:

1. We delete the node $pi$ of $t_j$, set the parent pointer for every child $pik$ (with $k \in [1..\operatorname{rank}(g)]$) to $p$ and insert the list of $pi$'s children into the child list of $p$. Finally we change the label of $p$ to $d$.

2. We remove digrams $d'$ that overlap with the replaced occurrence of $d$ from their respective digram lists and decrement the count values accordingly.

3. We create new digram lists for digrams that are introduced by the replacement step (i.e. digrams of the form $[f, k, d]$ or $[d, k, f]$) and set their count values accordingly.

Assume that $\operatorname{rank}(f) = m$ and $\operatorname{rank}(g) = n$. Then at most $m + n$ digram occurrences overlap the replaced occurrence $d$ at position $p$. The rank of $d$ is $n + m - 1$. Hence, if we replace only digram occurrences of rank at most $r$, then at most $r + 1$ digram occurrences overlap the replaced occurrence of $d$ at position $p$. Thus only a constant number of updates is necessary per digram replacement. This way the runtime of TreeRePair stays linear.

A problem arises with self-overlapping digrams of the form $[f, i, f]$. As an example, consider a term list $\bar{t}$ including the term $f(a, f(b, f(c, d)))$. The occurrence of $d_f = [f, 2, f]$ at the root position $\epsilon$ would belong to $\mathsf{maxOcc}_{d_f}(\bar{t})$, whereas the second occurrence of $d_f$ at position 2 would not. Now assume that we replace the digram $[f, 1, a]$ in $\bar{t}$, obtaining $A_1(f(b, f(c, d)))$. If we would recompute the set $\mathsf{maxOcc}_{d_f}(\bar{t})$ anew from scratch, the occurrence of $d_f$ at position 1 would be inserted into the list $\mathsf{maxOcc}_{d_f}(\bar{t})$. Yet in the implementation above that occurrence is lost.

## 9.2.2 Minimizing the matrix multiplication cost

We now present our adaption of TreeRePair, *MCTreeRePair* (for *matrix cost* TreeRePair), which aims at reducing the matrix multiplication cost of a given term list. First we define the (matrix multiplication) cost of a digram and of a compressed term list:

**Definition 9.16.** The *(matrix multiplication) cost* of a digram $d$ is $\operatorname{cost}(d) = \operatorname{rank}(d)$.

The *(matrix multiplication) cost* of a compressed term list $(t_1, \ldots, t_m \mid d_1, \ldots, d_n)$ is

$$\operatorname{cost}(t_1, \ldots, t_m \mid d_1, \ldots, d_n) = \sum_{i=1}^{m} \operatorname{cost}(t_i) + \sum_{i=1}^{n} \operatorname{cost}(d_i). \tag{9.4}$$

Let $(s_1, \ldots, s_m)$ be the expansion of $(\bar{t} \mid \bar{d}) = (t_1, \ldots, t_m \mid d_1, \ldots, d_n)$. We can compute the coefficients of the linear functions $[s_1], \ldots, [s_m]$ with $\operatorname{cost}(\bar{t} \mid \bar{d})$ many matrix multiplications.

**Example 9.17.** Let us compute the cost of the compressed term list from Example 9.15. We have:

$$\begin{aligned} \mathrm{cost}([h,2,f](x,y,z)) &= 0, & \mathrm{cost}([h,1,f]) &= 2,\\ \mathrm{cost}([[h,1,f],1,s](y,x,z)) &= 0, & \mathrm{cost}([h,2,f]) &= 2,\\ \mathrm{cost}([[h,1,f],1,s](x,f(s(0),y),z)) &= 1, & \mathrm{cost}([[h,1,f],1,s]) &= 1,\\ \mathrm{cost}([h,2,f](y,s(0),f(x,z))) &= 2. \end{aligned}$$

Hence, the total cost is 8. In contrast, the cost of the expanded term list in (9.3) is 13.

**Definition 9.18.** Let $d = [f,i,g]$ be a digram and let $\bar{t} = (t_1,\dots,t_m)$ be a term list with $t_j = (D_j,\lambda_j)$. The *savings* of a non-overlapping set of occurrences $\mathsf{Occ}_d(\bar{t})$, briefly $\mathsf{save}(\mathsf{Occ}_d(\bar{t}))$, is defined as follows:

$$\mathsf{save}(\mathsf{Occ}_d(\bar{t})) = -\mathrm{cost}(d) + \sum_{(j,p)\in\mathsf{Occ}_d(\bar{t})} |\mathrm{Var}(t_j/pi)|. \tag{9.5}$$

Thus we add to the negative cost of $d$ for each digram occurrence $(j,p) \in \mathsf{Occ}_d(\bar{t})$ the number of different variables below the $i^{\text{th}}$ child of the node $p$ (which is the lower digram node). By the following lemma, $\mathsf{save}(\mathsf{Occ}_d(\bar{t}))$ is exactly the cost-reduction obtained by replacing all digram occurrences from $\mathsf{Occ}_d(\bar{t})$.

**Lemma 9.19.** *Let $(\bar{t} \mid \bar{d})$ be a compressed term list with $\bar{t} = (t_1,\dots,t_m)$, let $d = [f,i,g]$ be a digram, and let $\mathsf{Occ}_d(\bar{t})$ be a non-overlapping set of occurrences of $d$ in $\bar{t}$. Let $(\bar{t}) \to_{\mathsf{Occ}_d(\bar{t})} (t'_1,\dots,t'_m)$. Then we have*

$$\mathit{cost}(t'_1,\dots,t'_m \mid \bar{d}, d) = \mathit{cost}(\bar{t} \mid \bar{d}) - \mathit{save}(\mathit{Occ}_d(\bar{t})). \tag{9.6}$$

*Proof.* Let $\mathsf{Occ}_{j,d}(\bar{t}) = \{p \mid (j,p) \in \mathsf{Occ}_d(\bar{t})\}$. Using (9.4) and (9.5), it follows that (9.6) is equivalent to

$$\sum_{j=1}^{m} \mathrm{cost}(t_j) = \sum_{j=1}^{m} \mathrm{cost}(t'_j) + \sum_{(j,p)\in\mathsf{Occ}_d(\bar{t})} |\mathrm{Var}(t_j/pi)|.$$

This follows from

$$\mathrm{cost}(t_j) = \mathrm{cost}(t'_j) + \sum_{p\in\mathsf{Occ}_{j,d}(\bar{t})} |\mathrm{Var}(t_j/pi)|$$

for all $1 \le j \le m$. But this is a consequence of (9.2). Applying the rule $\mathsf{rule}(d)^{-1}$ at all positions $p \in \mathsf{Occ}_{j,d}(\bar{t})$ in $t_j$ means that we remove all nodes $pi$ with $p \in \mathsf{Occ}_{j,d}(\bar{t})$ from $t_j$. Moreover, for all other nodes of $t_j$ the number of different variables below the node does not change. Also note that for all $p \in \mathsf{Occ}_{j,d}(\bar{t})$, the node $pi$ of $t_j$ is not labelled with a variable. □

By Lemma 9.19, in order to reduce the cost of a (compressed) term list maximally, we have to find a non-overlapping set of occurrences (of some digram $d$) with maximal savings.

**Definition 9.20.** Let $d = [f,i,g]$ be a digram, let $\bar{t} = (t_1,\dots,t_m)$ be a term list. Define $\mathsf{maxsave}_d(\bar{t})$ as the maximum of $\mathsf{save}(\mathsf{Occ}_d(\bar{t}))$, where we maximize over all non-overlapping sets of occurrences $\mathsf{Occ}_d(\bar{t})$.

Recall the definition of the non-overlapping occurrence set $\mathsf{maxOcc}_d(\bar{t})$ from Section 9.2.

**Lemma 9.21.** *We have* $\mathsf{save}(\mathsf{maxOcc}_d(\bar{t})) = \mathsf{maxsave}_d(\bar{t})$.

*Proof.* Let $d = [f, i, g]$. The case $f \neq g$ is clear, since then $\mathsf{maxOcc}_d(\bar{t})$ is the set of all occurrences of $d$ in $\bar{t}$. Now assume that $f = g$. Recall that we obtain $\mathsf{maxOcc}_d(\bar{t})$ by considering all maximal chains of positions $p, pi, pii, \dots, pi^k \in D_j$ in a term $t_j = (D_j, \lambda_j)$ from our list such that $\lambda_j(pi^\ell) = f$ for all $0 \le \ell \le k$. If $k$ is odd, we put the occurrences $(j,p), (j,pi^2), \dots, (j, pi^{k-1})$ into $\mathsf{maxOcc}_d(\bar{t})$. If $k$ is even, we put the occurrences $(j,p), (j,pi^2), \dots, (j, pi^{k-2})$ into $\mathsf{maxOcc}_d(\bar{t})$. Note that for even $k$, the set of occurrences $\{(j,pi), (j,pi^3), \dots, (j,pi^{k-1})\}$ has the same size as the chosen set of occurrences $\{(j,p), (j,pi^2), \dots, (j,pi^{k-2})\}$. But the latter gives a larger (or the same) savings according to (9.5), since $\mathrm{Var}(t_j/pi^{\ell+1}) \subseteq \mathrm{Var}(t_j/pi^\ell)$ and thus $|\mathrm{Var}(t_j/pi^{\ell+1})| \le |\mathrm{Var}(t_j/pi^\ell)|$ for all $0 \le \ell \le k-1$. □

We include a high-level description of MCTreeRePair.

```
input: term list $\bar{t} = (t_1, \dots, t_n)$;
$\bar{d} = \emptyset$;
while $\exists\, d : \mathsf{maxsave}_d(\bar{t}) > 1$ do
    let $d$ be a digram s. th. $\mathsf{maxsave}_d(\bar{t}) \ge \mathsf{maxsave}_{d'}(\bar{t})$ for all digrams $d'$;
    let $\bar{u}$ be s. th. $\bar{t} \to_{\mathsf{maxOcc}_d(\bar{t})} \bar{u}$;
    $\bar{t} := \bar{u}$;
    $\bar{d} := (\bar{d}, d)$;
end
return $(\bar{t} \mid \bar{d})$;
```

**Algorithm 3:** MCTreeRePair

Here is a complete example run of MCTreeRePair.

**Example 9.22.** Let $\mathrm{rank}(h) = \mathrm{rank}(f) = 2$, $\mathrm{rank}(s) = 1$ and $\mathrm{rank}(0) = 0$. Consider the following term rewriting system (which consists of the terms from Example 9.10):

$$h(x, f(y,z)) \to h(f(s(y), x), z),$$
$$h(f(s(x), f(s(0), y)), z) \to h(y, f(s(0), f(x,z))).$$

The matrix multiplication cost of this TRS is 13, see Example 9.10. The $\mathsf{maxsave}$-values of the digrams in this system are (we omit the second parameter in $\mathsf{maxsave}$ for the term list):

$$\mathsf{maxsave}(h,1,f) = 2, \quad \mathsf{maxsave}(f,1,s) = 1, \quad \mathsf{maxsave}(s,1,0) = 0,$$
$$\mathsf{maxsave}(h,2,f) = 2, \quad \mathsf{maxsave}(f,2,f) = 1.$$

Let us decide to replace the digram $d := (h,1,f)$ (we could also choose $(h,2,f)$). We obtain the following system:

$$h(x, f(y,z)) \to d(s(y), x, z), \qquad d(s(x), f(s(0), y), z) \to h(y, f(s(0), f(x,z))).$$

The new $\mathsf{maxsave}$-values are:

$$\mathsf{maxsave}(h,2,f)=2,\quad \mathsf{maxsave}(f,2,f)=0,\quad \mathsf{maxsave}(f,1,s)=-1,$$
$$\mathsf{maxsave}(d,1,s)=1,\quad \mathsf{maxsave}(d,2,f)=-1.$$

Next we replace the digram $e := (h,2,f)$ and obtain the system:

$$e(x,y,z)\to d(s(y),x,z),\qquad d(s(x),f(s(0),y),z)\to e(y,s(0),f(x,z)).$$

At this point, $f := (d,1,s)$ is the only digram with a strictly positive $\mathsf{maxsave}$-value, namely 1. Hence, we replace this digram and get the final compressed system

$$e(x,y,z)\to f(y,x,z),\qquad f(x,f(s(0),y),z)\to e(y,s(0),f(x,z)).$$

Our implementation of MCTreeRePair follows the same principles as TreeRePair. First we compute for every node $p$ in a tree $t_j \in \bar{t}$ the number $|\mathrm{Var}(t_j/p)|$ of different variables in the subtree rooted at $p$. These numbers are necessary to compute the savings of a digram according to (9.5). Then we insert for every digram $d$ every element from the set $\mathsf{maxOcc}_d(\bar{t})$ into a doubly-linked list, one for each digram. We compute thereby also $\mathsf{maxsave}_d(\bar{t}) = \mathsf{save}(\mathsf{maxOcc}_d(\bar{t}))$, again using (9.5). Note that we do not need to recompute the numbers $|\mathrm{Var}(t_j/q)|$ for all the nodes $q$ in the new tree $t'_j$: In a digram replacement we remove the node $pi$ from the pointer structure representing $t_j$. The new parent node of $pik$ (for $k \in [1..\mathrm{rank}(g)]$) becomes the node $p$. Therefore the number of different variables below a certain node does not change.

## 9.3 Compression and the DP-transformation

The *dependency pairs transformation* (short: DP) [AG00] is another technique used for automated termination proofs. We first briefly explain the dependency pairs transformation, following [HM04].

The main idea of the DP is to compare left-hand sides of rules only with those subterms of the right-hand sides that may spawn a new reduction. Thus we introduce the notion of a *defined symbol*:

**Definition 9.23.** Let $R$ be a TRS over a signature $\mathcal{F}$. We call a symbol $f \in \mathcal{F}$ a *defined symbol* if it is the root symbol of a left-hand side in $R$.

Using this definition, we define the dependency pairs transformation as follows:

**Definition 9.24** ([HM04])**.** Let $R$ be a TRS over a signature $\mathcal{F}$. Let $\mathcal{F}^\#$ denote the union of $\mathcal{F}$ and $\{f^\# \mid f \text{ is a defined symbol in } R\}$, where $f^\#$ is a fresh function symbol with the same arity as $f$. We call these new symbols *dependency pair symbols*. Given a term $t = f(t_1,\dots,t_n) \in \mathcal{T}(\mathcal{F},V)$ with $f$ a defined symbol, we write $t^\#$ for the term $f^\#(t_1,\dots,t_n)$. If $(l \to r) \in R$ and if $u$ is a subterm of $r$ with defined root symbol such that $u$ is not a proper subterm of $l$, then the rewrite rule $(l^\# \to r^\#)$ is called a *dependency pair* of $R$. The set of all dependency pairs of $R$ is denoted by $\mathsf{DP}(R)$.

Since $h$ is the only defined symbol in the TRS from Example 9.1 and that symbol only appears as a root symbol on right-hand sides of rules, $\mathsf{DP}(R) = \emptyset$ for Example 9.1. We provide a different TRS with a non-trivial set of dependency pairs.

**Example 9.25.** Let $\neg$, $\vee$ and $\wedge$ be defined as usual and consider the term rewriting system

$$\begin{aligned} \neg\neg x &\to x, & \neg(x \vee y) &\to \neg x \wedge \neg y, \\ \neg(x \wedge y) &\to \neg x \vee \neg y, & x \wedge (y \vee z) &\to (x \wedge y) \vee (x \wedge z), \\ (y \vee z) \wedge x &\to (x \wedge y) \vee (x \wedge z). \end{aligned}$$

The defined symbols are $\neg^{\#}$ and $\wedge^{\#}$. Thus the dependency pairs are:

$$\begin{aligned} &\neg^{\#}(x \vee y) \to \neg x \wedge^{\#} \neg y, \\ &\neg^{\#}(x \vee y) \to \neg^{\#} x, && \neg^{\#}(x \vee y) \to \neg^{\#} y, \\ &\neg^{\#}(x \wedge y) \to \neg^{\#} x, && \neg^{\#}(x \wedge y) \to \neg^{\#} y, \\ &x \wedge^{\#} (y \vee z) \to x \wedge^{\#} y, && x \wedge^{\#} (y \vee z) \to x \wedge^{\#} z, \\ &(y \vee z) \wedge^{\#} x \to x \wedge^{\#} y, && (y \vee z) \wedge^{\#} x \to x \wedge^{\#} z. \end{aligned}$$

Let $R$ be a TRS. If $R$ is non-terminating, then there must exist a *minimal* non-terminating term $s$, minimal in the sense that every subterm of $s$ is terminating. We denote the set of all minimal non-terminating subterms by $\mathcal{T}_\infty$. It is clear that every term in $\mathcal{T}_\infty$ has a defined root symbol.

**Lemma 9.26** ([HM04]). *For every term $s \in \mathcal{T}_\infty$ there exist terms $t, u \in \mathcal{T}_\infty$ such that*

$$s^{\#} \to_R^* t^{\#} \to_{\mathsf{DP}(R)} u^{\#}.$$

A consequence of Lemma 9.26 is that for every non-terminating TRS $R$ there exists an infinite rewrite sequence of the form

$$t_1 \to_R^* t_2 \to_{\mathsf{DP}(R)} \to_R^* t_3 \to_{\mathsf{DP}(R)} t_4 \cdots$$

where the root symbol of $t_i$ is defined for every $i \geq 1$. Hence, to prove termination of a TRS $R$ it is sufficient to show that $R \cup \mathsf{DP}(R)$ does not admit such an infinite sequence. Thus the dependency pairs transformation converts the full termination problem of $R$ over $\mathcal{F}$ into the *relative termination problem* of $\mathsf{DP}(R)$, relative to $R$ over the signature $\mathcal{F} \cup \mathcal{F}^{\#}$.

When using monotone matrix interpretations to prove relative termination, one uses two-sorted interpretations. For this we define *weakly monotone $\mathcal{F}$-algebras.*

**Definition 9.27** ([EWZ08]). A *weakly monotone $\mathcal{F}$-algebra* $(A, [\cdot], >, \lesssim)$ is an $\mathcal{F}$-algebra $(A, >)$ equipped with two $\mathcal{F}$-sorted relations $>, \lesssim$ on $A$ such that

- $>$ is well-founded,
- $> \cdot \lesssim \subseteq >$
- For every $f \in \mathcal{F}$ the operation $[f]$ is monotone with respect to $\lesssim$.

An $m$-ary marked symbol $f^{\#}$ is interpreted by a linear function $[f^{\#}] : (S^n)^m \to S$, whereas an $m$-ary unmarked symbol $f$ is interpreted by a linear function $[f^{\#}] : (S^n)^m \to S^n$.

We now discuss how two-sortedness affects compression. We observe that the cost model that just counts matrix multiplications is wrong for computing interpretations for $\mathsf{DP}(R)$, and consequently MCTreeRePair produces inefficient results. This is shown in the following example.

**Example 9.28.** Let $f^{\#}$ be a marked binary symbol and let $g, h$ be unmarked unary symbols. The interpretation of $f^{\#}$ has coefficient matrices $F_1^{\#}, F_2^{\#}$ of dimension $1 \times n$, and the interpretation of $g$ (resp., $h$) has a coefficient matrix $G_1$ (resp., $H_1$) of dimension $n \times n$. Consider the computation of the coefficient for variable $x$ in the term

$$t = f^{\#}(g(h(x)), g(h(x))).$$

It is tempting to first replace the two occurrences of the digram $e = (g, 1, h)$. Thus we compute $E_1 = G_1 \cdot H_1$ with $n^3$ elementary multiplications. Then we compute the coefficient of $x$ in $t$ as $F_1^{\#}E_1 + F_2^{\#}E_1$, which needs another $2n^2$ elementary multiplications. Hence, in total we need $n^3 + 2n^2$ multiplications.

But there is a better way: Compute $(F_1^{\#}G_1)H_1$, that is, first multiply $F_1^{\#}$ by $G_1$ and then the result by $H_1$, and similarly $(F_2^{\#}G_1)H_1$, which needs in total only $4n^2$ multiplications. In terms of digrams, this means that we replace the following digrams (which occur only once) in this order: $c := (f^{\#}, 1, g)$, $d := (c, 1, h)$, $e := (d, 2, g)$, $f := (e, 2, h)$.

The example shows that the computation of the interpretation is best done top-down, as this can avoid expensive $(n \times n)$-multiplications. Thus when compressing, only $R$ and not $\mathsf{DP}(R)$ shall be compressed.

Nevertheless terms in $\mathsf{DP}(R)$ may be compressed as a side effect of the compression that takes place in $R$. Note that all terms in $\mathsf{DP}(R)$ are of the form $f^{\#}(t_1, \ldots, f_n)$, where each $t_i$ is a subterm of a term in $R$. This implies that we may be able to extract a compressed version of $t_i$. We compress $R$ over $\mathcal{F}$, obtaining a system $R_c$ over the extended alphabet $\mathcal{F}_c$, which consists of $\mathcal{F}$ and the introduced digrams. We then compute the compressed version of $\mathsf{DP}(R)$ by applying a modified operation $\mathsf{DP}_c$ on $R_c$. This operation $\mathsf{DP}_c(R_c)$ has two ingredients:

- computation of the set of all subterms (in compressed form) of a compressed term, and
- marking of the top symbol of a compressed term.

In both cases the output term(s) should be compressed, and be obtained without completely unpacking the input term. These operations can be realized in a straightforward manner by expanding digrams as needed. We make no attempts at constructing fresh digrams.

**Example 9.29.** Let $f$ be a unary symbol and $t = f^8(x)$. Consider the compressed term $t_c = d_3(x)$ with digrams $d_3 = (d_2, 1, d_2), d_2 = (d_1, 1, d_1), d_1 = (f, 1, f)$. The proper subterms of $t$ in compressed form are $fd_1d_2(x)$, $d_1d_2(x)$, $fd_2(x)$, $d_2(x)$, $fd_1(x)$, $d_1(x)$, $f(x)$, $x$, and $d_3(x)^{\#}$ is $f^{\#}fd_1d_2(x)$.

To compute the interpretations of marked terms efficiently, we work from the top, i.e., left-to-right. This can be modeled by the introduction of digrams $e_1 = (f^\#, 1, f)$, $e_2 = (e_1, 1, d_1)$, $e_3 = (e_2, 1, d_2)$. The interpretations of these digrams are linear functions from $S^n$ to $S$. For the computation of the coefficients of these linear functions we have to multiply a $(1 \times n)$-matrix with a $(n \times n)$-matrix (but never two $(n \times n)$-matrices). We therefore compress $\mathsf{DP}_c(R_c)$ by repeatedly replacing digrams that occur at the root of some term from $\mathsf{DP}_c(R_c)$. The algorithm stops when all children of all top symbols are variables.

We summarize the DP-MCTreeRePair algorithm:

```
**input**: term list $\bar{t} = (t_1, \ldots, t_n)$
$R_c :=$ MCTreeRePair($R$)
$D_c := \mathsf{DP}_c(R_c)$
$\bar{d} = \emptyset$;
**while** $\exists\, d$ *which occurs at the top of a rule in* $D_c$ **do**
  | $D_c :=$ replace every occurrence of $d$ in lhs and rhs of $D_c$
**end**
**return** $(D_c, R_c)$;
```

**Algorithm 4:** DP-MCTreeRePair. Note that expand($D_c$) = $\mathsf{DP}(R)$ and expand($R_c$) = $R$.

**Example 9.30.** For the symbolic evaluation of an $n$-dimensional matrix interpretation for the rewriting system from Example 9.10, Table 9.1 contains in column $(p, q, r)$ the number of multiplications of a $(p \times q)$-matrix by a $(q \times r)$-matrix.

| method | $(1, n, 1)$ | $(1, n, n)$ | $(n, n, 1)$ | $(n, n, n)$ |
|---|---|---|---|---|
| uncompressed ($\mathsf{DP}(R) \cup R$) | 4 | 8 | 20 | 18 |
| MCTreeRePair($\mathsf{DP}(R) \cup R$) | 4 | 8 | 13 | 12 |
| DP-MCTreeRePair($R$) | 9 | 11 | 9 | 8 |

Table 9.1: Number of matrix multiplications for the rewriting system from Example 9.10.

By applying algorithm DP-MCTreeRePair, the number of matrix-by-matrix multiplications is lowest—in fact it is equal to the number of matrix-by-matrix multiplications of MCTreeRePair($R$).

## 9.4 Experiments

We implemented a version of MCTreeRePair as described in Sections 9.2 and 9.3, and we evaluated our implementation in two settings:

- We evaluated how compression reduces the size of constraint systems for rewrite systems from the Termination Problems Data Base (more precisely, the SRS/TRS standard/relative subsets of TPDB version 8), which consists of 3027 files.

- We determined the influence of compression on the power of an actual termination prover.

Links to the source code[1] and the logfiles[2] are provided in the footnotes.

To measure the "compressibility" of the TRS problems provided by the TPDB, we used the matrix multiplication cost from (9.2) as well as the actual size of the resulting SAT constraint system with fixed parameters for the matrix dimension and the bit width of the matrix entries. We compared these measures for the settings with and without compression, for both the original systems and for their DP-transformed versions. Table 9.2 shows the results. Column *cost* shows the accumulated costs of all terms from the corpus. Column *CNF-size* shows the accumulated number of variables and clauses of the Boolean expression that the constraint solver generates and which are then given to the SAT-solver.

For *no compression* and *compression* we use $(3 \times 3)$-matrices with 3-bit entries, for *DP* and *DP and compression* we use $(3 \times 3)$-matrices with 5-bit entries. We obtain an overall compression ratio of about 3 for both the matrix multiplication cost and the actual CNF size (number of clauses). For DP, these ratios are 3.44 and 1.71, respectively. We conclude that our cost model gives, on average, a very good approximation of the real cost.

| method | cost $\cdot 10^5$ | CNF-size (v, c), both $\cdot 10^8$ |
|---|---|---|
| no compression | 16.1 | $4.0, 32.3$ |
| compression | 5.2 | $1.3, 10.4$ |
| dependency pairs (DP) | 15.1 | $19.2, 62.2$ |
| DP and compression | 4.4 | $11.1, 36.3$ |

Table 9.2: Total cost and CNF-size (variables and clauses) with and without compression, for 3027 TRS from the termination problem data base.

For estimating the effect of compression on the performance of a termination prover, we used a restricted version of *matchbox*. It optionally applies the dependency pairs transformation and then repeats the following steps until there are no more strict rules:

- If the system is linear, remove rules by additive weights (linear polynomials of slope 1 with absolute coefficients computed by the *GLPK*[3] solver for linear inequalities).
- For increasing matrix dimensions, try to remove rules by natural matrix interpretations for original systems [EWZ08] (solved by binary bit-blasting) and matrix interpretations for DP-transformed systems [KW09] (solved by unary bit-blasting [CFF+12]). In both cases, MINISAT [EB05] is used as the backend solver.

We apply the "cheap" method (additive weights) first so that the remaining constraint systems are non-trivial. We isolate the effect of compression by using matrix interpretations as the only non-cheap method.

[1] `https://github.com/jwaldmann/matchbox`
[2] `http://www.eti.uni-siegen.de/ti/mitarbeiter/noeth/`
[3] GNU Linear Programming Kit, `https://www.gnu.org/software/glpk/`

Our experiment then consists of a comparison between the performances of an implementation with and without compression. The following parameters are fixed at the beginning:

- the boolean encoding of numbers (in particular their bit width),
- the matrix dimensions that are being used,
- the compiler settings,
- the runtime settings and
- the resources of the execution platform (timeout, memory size, cores).

We choose "sensible" values for these parameters, but make no particular attempt to optimize them.

We also use parallelism: We search for matrix interpretations in dimensions $1, 2, \ldots, D$ in parallel (for some parameter $D$ that is fixed in advance), i.e., we generate constraint systems $C_1, C_2, \ldots, C_D$ and submit each of them to a separate instance of the SAT solver. As soon as one $C_i$ is solved, we stop the other computations, remove some rules from the input problem (according to the interpretation that was obtained as the solution of $C_i$), and start anew. This way we only measure the time that the constraint solver needs in the positive case(s). Compare this with a sequential implementation, where we would have to wait for $C_i$ to be recognized as unsolvable before attempting to solve $C_{i+1}$: In this case the total time would include several unsuccessful attempts as well. But when proving termination automatically we are not interested in unsolvable $C_i$, because they do not contain information on the termination problem. (We cannot distinguish between unsatisfiability due to non-termination, or due to insufficient bit width.)

Table 9.3 shows the results of our experiments. The column "# yes instances" shows the number of rewrite systems for which termination is successfully proven within one minute (the time-out). The column "average time yes" is the average time needed to prove termination for all yes instances. It shall be noted that the "yes instances" include the systems for which termination could be proven by "cheap" methods, as described above (there were 50 such cases without using the dependency pairs method, and 250 with it). As can be seen, the number of systems which can be proven to be terminating increases by about 7% (3, 5% for DP) when using MCTreeRePair-compression. We conclude that compression of rewriting systems using MCTreeRePair does improve the power of a termination prover that uses a constraint solver to find interpretations.

Table 9.3 also shows our results when "naïve" compression based on TreeRePair (as outlined in Section 9.2.1) instead of MCTreeRePair is used in the termination prover. Surprisingly, the number of systems for which termination can be proven is *less* than without any compression. Here is a possible explanation: When computing the interpretation of a term $t$ without variables bottom-up, only cheap matrix-by-vector multiplications are needed; expensive matrix-by-matrix multiplications do not occur. But compression based on ordinary TreeRePair only tries to reduce the size of $t$ and therefore may introduce digrams which lead to expensive matrix-by-matrix multiplications when evaluating the digrams.

| method | average time yes | # yes instances |
|---|---|---|
| no compression | 11.9 | 584 |
| MCTreeRePair | 12.2 | 628 |
| TreeRePair | 11.9 | 571 |
| Dependency pairs (DP) | 1.85 | 681 |
| DP and MCTreeRePair | 4.10 | 709 |

Table 9.3: Influence of compression on the matchbox termination prover.

All values in Table 9.3 were obtained for an unlimited maximal rank for digrams. We also experimented with bounded maximal ranks, and it turned out that the optimal value (w.r.t. resulting number of termination proofs for Matchbox using DP-MCTreeRePair) seems to be $r = 4$ (which is also the optimal value for XML-compression based on TreeRePair in [LMM13]): The number of proofs is slightly larger than with unbounded rank, and we have no explanation for that at the moment.

## 9.5 Discussion

**Does compression really preserve semantics?** For any given interpretation of function symbols, the interpretation of a compressed term is equivalent to the interpretation of the original term. The underlying reason is that matrix multiplication is associative: digrams correspond to sub-multiplications.

When solving matrix constraints by bit-blasting, the range for matrix elements is a finite set, prescribed by the bit width of the encoding. This implies that arithmetical operations may overflow, so they are partial functions. These partial functions are no longer associative: For instance, consider the integer product $abc$ for three bit integers $a, b$ and $c$. For $a = 7$, $b = 7$, $c = 0$ the product $a(bc)$ is representable, while the product $(ab)c$ is not. Now, $(ab)$ could be a digram that occurs during compression, while $(bc)$ could correspond to an evaluation of the uncompressed term. Then the constraint system generated from the compressed terms may be unsatisfiable, while the original system is satisfiable. Take the bit width $w$ as a parameter. It can be shown that the original system $O$ and the compressed system $C$ are equivalent in the sense that for each satisfying assignment $s$ of $O(w)$, there is some $w' \geq w$ such that a padded version $s'$ of $s$ satisfies $C(w')$.

**Does compression work with more advanced termination methods?** The basic dependency pairs method has many refinements [HM04, GTSF06], which we ignore here since they appear orthogonal to the topic of compression. For instance, by using (estimated) dependency graphs, one obtains termination subproblems that refer to subsets of $\mathsf{DP}(R)$. The "usable rules" method creates subproblems that contain subsets of $R$. In both cases, compression can be obtained by the methods shown in Section 9.3.

**Is the data base sufficient?** We ran experiments on problems contained in the Termination Problems Data Base (TPDB). It may be argued that most of the problems in TPDB are small, and do not need compression. Yet our experiments show that even for small problems, compression may help.

Another point is that TPDB problems might not be typical "real life" termination problems. It appears that most of the TPDB problems are hand-crafted: They are taken from publications, where they serve to illustrate certain isolated points. So, they tend to be small but hard (and trivial only when one applies a specific, advanced method). Application problems, on the other hand, may be large but "easy", and appear hard only because of their size, and compression can reduce the size.

**Extensions.** The method given in the paper counts matrix-by-matrix multiplications only. Our experiments confirm that this is a reasonable simplification. For still better compression, we additionally need to take into account the cost of vector-by-matrix multiplications at the top of DP-transformed rules, and also the matrix-by-vector multiplications for evaluating absolute parts. This implies extensions in the definition of the savings of a digram, and in the algorithm to incrementally update the savings information. In full generality, this includes the "matrix chain multiplication" optimization problem—and goes beyond it, since it is not just about parenthesizing matrix chains, but also about re-using subexpressions. We leave that as possible direction for future work.

# Chapter 10

# Conclusion and final remarks

In this thesis we discussed different aspects of grammar-based tree compression. In Chapter 3 we provided a full proof that the asymptotic average number of nodes in the minimal dag of a full binary tree is

$$2\sqrt{\frac{\ln 4}{\pi}}\frac{n}{\sqrt{\log n}}\left(1+\mathcal{O}\left(\frac{1}{\log n}\right)\right).$$

This proof has only been sketched previously in the literature. We presented a short take on the mathematical discipline of *Analytic Combinatorics*, which is necessary to understand the proof. We extended the result to the asymptotic average number of nodes and edges in dags of unranked trees and to a labelled setting.

In Chapter 4 we introduced a new data structure, which we called the *hybrid dag*. The hybrid dag of a tree $t$ is built by sharing common child end sequences from dag($t$). We used it to show that the bdag of an unranked tree may be quadratically smaller than the dag of that tree, while it is never more than twice the size of the dag (Corollary 4.11).

In Chapter 5 we generalized the hybrid dag from Chapter 4. Instead of sharing only common child end sequences, we showed how existing grammar-based string compression algorithms can be used to share arbitrary patterns among the child sequences of the dag. We proved that these structures can be transformed to a tree straight-line program with only little size increase (Theorem 5.5).

In Chapter 6 we investigated the use of straight-line programs to compress the *traversal string* of a tree. The traversal string of a tree $t$ is given by the label sequence when traversing the tree (depth-first, left to right). We showed that there exist trees such that this representation is exponentially more succinct than the smallest tree straight-line program (Theorem 6.6). Similarly we showed that there exist trees such that a smallest traversal SLP for those trees may be exponentially smaller than the smallest SLP for their balanced parenthesis representation (Theorem 6.9).

In Chapter 7 we used the tree compression formalism from the previous chapters as compression algorithms on real-world tree data.

In Chapter 8 we discussed two examples of algorithms on compressed trees. We introduced *subtree equality check*, which is particularly easy for DAG-compressed trees. We proved that it is PSPACE-hard to check whether a tree given by a traversal SLP is accepted by a tree automaton (Theorem 8.5). This is in strong contrast to trees that are given by tree straight-line programs, where membership is decidable in polynomial time.

In Chapter 9 we discussed at length an example where compression (heuristically) accelerates computation. We modified the tree compression algorithm *TreeRePair* [LMM13] to minimize the number of multiplications in the matrix interpretation of a given set of rewrite rules. We provided data that show that this approach indeed improves the running time of an automatic termination prover.

**Open problems.** While the asymptotic average size of a full binary tree's DAG has been fully discussed, the variance of that value is unknown. Simple experiments strongly suggest that the distribution is normal, see Section 3.7.1. Another interesting open problem is a full classification of the average DAG size for all regular tree languages. We conjectured (Conjecture 3.27) that every regular tree language has an asymptotic average DAG size which is either linear in the size of the tree or of the form $\Theta(n/\sqrt{\log n})$. Conjecture 3.28 states that all tree classes in the prior case have an average height that is also linear in the size of the tree while the other tree classes have an average height of $\Theta(\sqrt{n})$.

The average size of the HDAG remains an open problem, as it has not been possible to use the methods from Chapter 3 on the HDAG.

While there exists a tree family such that their traversal SLP is exponentially more succinct than their smallest SLP-compressed balanced parenthesis, there is no tree family known where the opposite situation arises.

There are several open problems related to the use of TreeRePair for accelerated termination proving, see Section 9.5. Among other problems, it remains open whether the compression algorithm may be combined with more advanced termination methods or whether a more precise cost model that also considers matrix-vector multiplications instead of only matrix-matrix multiplications may improve the algorithm.

# Bibliography

[AG00] Thomas Arts and Jürgen Giesl. Termination of term rewriting using dependency pairs. *Theor. Comput. Sci.*, 236(1-2):133–178, 2000.

[AHdLT05] Stephen Alstrup, Jacob Holm, Kristian de Lichtenberg, and Mikkel Thorup. Maintaining information in fully dynamic trees with top trees. *ACM Transactions on Algorithms*, 1(2):243–264, 2005.

[Aku10] Tatsuya Akutsu. A bisection algorithm for grammar-based compression of ordered trees. *Inf. Process. Lett.*, 110(18-19):815–820, 2010.

[AS92] Malcolm. D. Atkinson and J. R. Sack. Generating binary trees at random. *j-INFO-PROC-LETT*, 41(1):21–23, January 1992.

[ASU86] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools.* Addison-Wesley, 1986.

[BCR08] Alberto Bertoni, Christian Choffrut, and Roberto Radicioni. Literal shuffle of compressed words. In *Fifth IFIP International Conference On Theoretical Computer Science - TCS 2008, IFIP 20th World Computer Congress, TC 1*, pages 87–100, 2008.

[BDM$^+$05] David Benoit, Erik D. Demaine, J. Ian Munro, Rajeev Raman, Venkatesh Raman, and S. Srinivasa Rao. Representing trees of higher degree. *Algorithmica*, 43(4):275–292, 2005.

[BGK03] Peter Buneman, Martin Grohe, and Christoph Koch. Path queries on compressed XML. In *29th Conference on Very Large Data Bases (VLDB) 2003*, pages 141–152, 2003.

[BGLW15] Philip Bille, Inge Li Gørtz, Gad M. Landau, and Oren Weimann. Tree compression with top trees. *Inf. Comput.*, 243:166–177, 2015.

[BLM08] Giorgio Busatto, Markus Lohrey, and Sebastian Maneth. Efficient memory representation of XML document trees. *Inf. Syst.*, 33(4-5):456–474, 2008.

[BLMN15] Mireille Bousquet-Mélou, Markus Lohrey, Sebastian Maneth, and Eric Nöth. XML compression via directed acyclic graphs. *Theory Comput. Syst.*, 57(4):1322–1371, 2015.

[BLNW13] Alexander Bau, Markus Lohrey, Eric Nöth, and Johannes Waldmann. Compression of rewriting systems for termination analysis. In *24th International Conference on Rewriting Techniques and Applications,RTA 2013*, pages 97–112, 2013.

[BLR$^+$15] Philip Bille, Gad M. Landau, Rajeev Raman, Kunihiko Sadakane, Srinivasa Rao Satti, and Oren Weimann. Random access to grammar-compressed strings and trees. *SIAM J. Comput.*, 44(3):513–539, 2015.

[BN98] Franz Baader and Tobias Nipkow. *Term rewriting and all that.* Cambridge University Press, 1998.

[CDG$^+$07] H. Comon, M. Dauchet, R. Gilleron, F. Jacquemard, C. Löding, D. Lugiez, S. Tison, and M. Tommasi. Tree automata techniques and applications. Available at: http://www.grappa.univ-lille3.fr/tata, 2007.

[CFF$^+$12] Michael Codish, Yoav Fekete, Carsten Fuhs, Jürgen Giesl, and Johannes Waldmann. Exotic semi-ring constraints. In *10th International Workshop on Satisfiability Modulo Theories, SMT 2012*, pages 88–97, 2012.

[CLL$^+$05] 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.

[CMTV98] Hervé Caussinus, Pierre McKenzie, Denis Thérien, and Heribert Vollmer. Nondeterministic $NC^1$ computation. *J. Comput. Syst. Sci.*, 57(2):200–212, 1998.

[dBKR72] Nicolaas G. de Bruijn, Donald E. Knuth, and S. O. Rice. The average height of planted plane trees. In *Graph theory and computing*, pages 15–22. Academic Press, New York, 1972.

[Dev98] Luc Devroye. On the richness of the collection of subtrees in random binary search trees. *Inf. Process. Lett.*, 65(4):195–199, 1998.

[Dis12] Filipo. Disanto. Unbalanced subtrees in binary rooted ordered and un-ordered trees. *ArXiv e-prints*, February 2012.

[DST80] Peter J. Downey, Ravi Sethi, and Robert Endre Tarjan. Variations on the common subexpression problem. *J. ACM*, 27(4):758–771, 1980.

[DZ80] Nachum Dershowitz and Shmuel Zaks. Enumerations of ordered trees. *Discrete Mathematics*, 31(1):9–28, 1980.

[EB05] Niklas Eén and Armin Biere. Effective preprocessing in SAT through variable and clause elimination. In *Theory and Applications of Satisfiability Testing, 8th International Conference, SAT 2005*, pages 61–75, 2005.

[Ers58] Andrei P. Ershov. On programming of arithmetic operations. *Commun. ACM*, 1(8):3–9, 1958.

[EWZ08] Jörg Endrullis, Johannes Waldmann, and Hans Zantema. Matrix interpretations for proving termination of term rewriting. *J. Autom. Reasoning*, 40(2-3):195–220, 2008.

[FB93] Eberhard Freitag and Rolf Busam. *Funktionentheorie*. Springer Textbook. Springer-Verlag, Berlin, 1993.

[FGD95] Philippe Flajolet, Xavier Gourdon, and Philippe Dumas. Mellin transforms and asymptotics: Harmonic sums. *Theor. Comput. Sci.*, 144(1&2):3–58, 1995.

[FGM97] Philippe Flajolet, Xavier Gourdon, and Conrado Martinez. Patterns in random binary search trees. *Random Struct. Algorithms*, 11(3):223–244, 1997.

[FLMM09] Paolo Ferragina, Fabrizio Luccio, Giovanni Manzini, and Muthu Muthukrishnan. Compressing and indexing labeled trees, with applications. *J. ACM*, 57(1), 2009.

[FO82] Philippe Flajolet and Andrew M. Odlyzko. The average height of binary trees and other simple trees. *J. Comput. Syst. Sci.*, 25(2):171–213, 1982.

[FO90] Philippe Flajolet and Andrew M. Odlyzko. Singularity analysis of generating functions. *SIAM J. Discrete Math.*, 3(2):216–240, 1990.

[FS87] Philippe Flajolet and Jean-Marc Steyaert. A complexity calculus for recursive tree algorithms. *Mathematical Systems Theory*, 19(4):301–331, 1987.

[FS09] Philippe Flajolet and Robert Sedgewick. *Analytic Combinatorics*. Cambridge University Press, 2009.

[FSS90] Philippe Flajolet, Paolo Sipala, and Jean-Marc Steyaert. Analytic variations on the common subexpression problem. In *Automata, Languages and Programming, 17th International Colloquium, ICALP90*, pages 220–234, 1990.

[FW93] Michael L. Fredman and Dan E. Willard. Surpassing the information theoretic bound with fusion trees. *J. Comput. Syst. Sci.*, 47(3):424–436, 1993.

[GHLN15] Moses Ganardi, Danny Hucke, Markus Lohrey, and Eric Noeth. Tree compression using string grammars. *CoRR*, abs/1504.05535, 2015.

[GTSF06] Jürgen Giesl, René Thiemann, Peter Schneider-Kamp, and Stephan Falke. Mechanizing and improving dependency pairs. *J. Autom. Reasoning*, 37(3):155–203, 2006.

[HLN14] Danny Hucke, Markus Lohrey, and Eric Noeth. Constructing small tree grammars and small circuits for formulas. In *34th International Conference on Foundation of Software Technology and Theoretical Computer Science, FSTTCS 2014*, pages 457–468, 2014.

[HM04] Nao Hirokawa and Aart Middeldorp. Dependency pairs revisited. In *Rewriting Techniques and Applications, 15th International Conference, RTA 2004*, pages 249–268, 2004.

[HR15] Lorenz Hübschle-Schneider and Rajeev Raman. Tree compression with top trees revisited. In *Experimental Algorithms - 14th International Symposium, SEA 2015*, pages 15–27, 2015.

[Jez14] Artur Jez. A really simple approximation of smallest grammar. In *Combinatorial Pattern Matching - 25th Annual Symposium, CPM 2014*, pages 182–191, 2014.

[JL14] Artur Jez and Markus Lohrey. Approximation of smallest linear tree grammar. In *31st International Symposium on Theoretical Aspects of Computer Science (STACS 2014)*, pages 445–457, 2014.

[JSS12] Jesper Jansson, Kunihiko Sadakane, and Wing-Kin Sung. Ultra-succinct representation of ordered trees with applications. *J. Comput. Syst. Sci.*, 78(2):619–631, 2012.

[Knu68] Donald E. Knuth. *The Art of Computer Programming, Volume I: Fundamental Algorithms*. Addison-Wesley, 1968.

[Koc03] Christoph Koch. Efficient processing of expressive node-selecting queries on XML data in secondary storage: A tree automata-based approach. In *29th Conference on Very Large Data Bases (VLDB) 2003*, pages 249–260, 2003.

[KW09] Adam Koprowski and Johannes Waldmann. Max/plus tree automata for termination of term rewriting. *Acta Cybern.*, 19(2):357–392, 2009.

[KY00] 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.

[KYNC00] 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.

[Lan79] Dallas S Lankford. On proving term rewriting systems are noetherian. *Memo MTP-3, Mathematics Department, Louisiana Tech. University, Ruston, LA*, 1979.

[LM99] N. Jesper Larsson and Alistair Moffat. Offline dictionary-based compression. In *Data Compression Conference, DCC 1999*, pages 296–305. IEEE Computer Society, 1999.

[LM06] Markus Lohrey and Sebastian Maneth. The complexity of tree automata and XPath on grammar-compressed trees. *Theor. Comput. Sci.*, 363(2):196–210, 2006.

[LM13] Markus Lohrey and Christian Mathissen. Isomorphism of regular trees and words. *Inf. Comput.*, 224:71–105, 2013.

[LMM13] Markus Lohrey, Sebastian Maneth, and Roy Mennicke. XML tree structure compression using RePair. *Inf. Syst.*, 38(8):1150–1167, 2013.

[LMS12] Markus Lohrey, Sebastian Maneth, and Manfred Schmidt-Schauß. Parameter reduction and automata evaluation for grammar-compressed trees. *J. Comput. Syst. Sci.*, 78(5):1651–1669, 2012.

[Loh01] Markus Lohrey. On the parallel complexity of tree automata. In *Rewriting Techniques and Applications, 12th International Conference, RTA 2001*, pages 201–215, 2001.

[Loh11] Markus Lohrey. Leaf languages and string compression. *Inf. Comput.*, 209(6):951–965, 2011.

[Loh12] Markus Lohrey. Algorithmics on slp-compressed strings: A survey. *Groups Complexity Cryptology*, 4(2):241–299, 2012.

[Loh14] Markus Lohrey. *The Compressed Word Problem for Groups.* Springer, 2014.

[MR01] J. Ian Munro and Venkatesh Raman. Succinct representation of balanced parentheses and static trees. *SIAM J. Comput.*, 31(3):762–776, 2001.

[MT98] Christoph Meinel and Thorsten Theobald. *Algorithms and Data Structures in VLSI Design: OBDD - Foundations and Applications.* Springer, 1998.

[Muc97] Steven S. Muchnick. *Advanced Compiler Design and Implementation.* Morgan Kaufmann, 1997.

[NW97] Craig G. Nevill-Manning and Ian H. Witten. Identifying hierarchical strcture in sequences: A linear-time algorithm. *J. Artif. Intell. Res. (JAIR)*, 7:67–82, 1997.

[Pla94] Wojciech Plandowski. Testing equivalence of morphisms on context-free languages. In *Algorithms - ESA '94, Second Annual European Symposium*, pages 460–470, 1994.

[Ryt03] Wojciech Rytter. Application of lempel-ziv factorization to the approximation of grammar-based compression. *Theor. Comput. Sci.*, 302(1-3):211–222, 2003.

[Sal07] David Salomon. *Data compression - The Complete Reference, 4th Edition.* Springer, 2007.

[Say06] Khalid Sayood. *Introduction to Data Compression, Third Edition.* Morgan Kaufmann, 2006.

[Sch07] Thomas Schwentick. Automata for XML - A survey. *J. Comput. Syst. Sci.*, 73(3):289–315, 2007.

[Sta99] Richard P. Stanley. *Enumerative combinatorics. Volume 2.* Cambridge studies in advanced mathematics. Cambridge university press, Cambridge, New York, 1999. Errata et addenda : p. 583-585.

[Szp01] Wojciech Szpankowski. *Average Case Analysis of Algorithms and Sequences.* John Wiley and Sons, 2001.

[TPD] Termination problem data base. `http://termination-portal.org/wiki/TPDB`. Accessed: 2015-10-02.

[Zan94] Hans Zantema. Termination of term rewriting: Interpretation and type elimination. *J. Symb. Comput.*, 17(1):23–50, 1994.

[ZHC$^+$15] Yang Zhao, Morihiro Hayashida, Yue Cao, Jaewook Hwang, and Tatsuya Akutsu. Grammar-based compression approach to extraction of common rules among multiple trees of glycans and rnas. *BMC Bioinformatics*, 16:128, 2015.

[ZL77] Jacob Ziv and Abraham Lempel. A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory*, 23(3):337–343, 1977.

[ZL78] Jacob Ziv and Abraham Lempel. Compression of individual sequences via variable-rate coding. *IEEE Transactions on Information Theory*, 24(5):530–536, 1978.