Diplomarbeit

Re-pair for Trees

Leipzig, im Juni, 2010

Universität Leipzig
Fakultät für Mathematik und Informatik
Institut für Informatik

Autor: Roy Mennicke
Studiengang: Diplom-Informatik
Matrikelnummer: 9 12 15 18

Betreuer: Prof. Dr. rer. nat. habil. Markus Lohrey
Abt. Algebraische und logische Grundlagen der Informatik
Universität Leipzig, Institut für Informatik

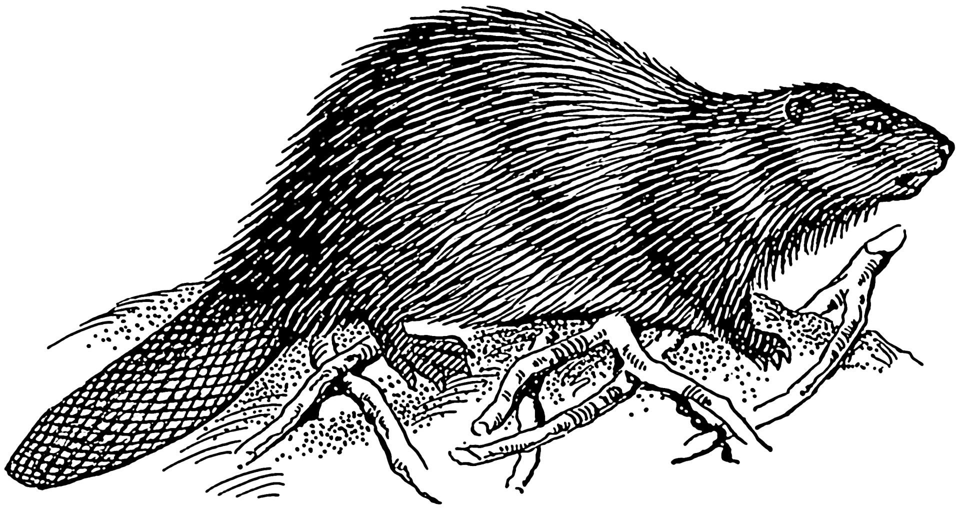
Re-pair for Trees

“No profit grows where no pleasure is taken; In brief, sir, study what you most affect.”

– William Shakespeare

Roy Mennicke

Re-pair for Trees



Department of Computer Science
University of Leipzig

Front page line art drawing by Pearson Scott Foresman.

While every precaution has been taken in the preparation of this work, the author assumes no responsibility for errors or omissions, or for damages resulting from the use of the information contained herein.

First printing, June 2010

Contents

Abstract

We introduce a new linear time compression algorithm, called "Re-pair for Trees", which compresses ordered trees over a ranked alphabet using linear straight-line context-free tree grammars. Such grammars generalize straight-line context-free string grammars and allow basic tree operations, like traversal along edges, to be executed without prior decompression. Our algorithm can be considered as a generalization of the "Re-pair" algorithm developed by N. Jesper Larsson and Alistair Moffat in 2000. The latter algorithm is a dictionary-based compression algorithm for strings.

We also introduce a succinct coding which is specialized in further compressing the grammars generated by our algorithm. This is accomplished without loosing the ability do directly execute queries on this compressed representation of the input tree. Finally, we compare the grammars and output files generated by a prototype of the Re-pair for Trees algorithm with those of similar compression algorithms. The obtained results show that that our algorithm outperforms its competitors in terms of compression ratio, runtime and memory usage.

1 Introduction

Motivation

Trees are nowadays a common data structure used in computer science to represent data hierarchically. This is, for instance, evidenced by XML documents which are widely used after their introduction in 1996. They are sequential representations of ordered unranked trees. When processing trees it is often convenient to hold the tree structure in memory in order to retain fast and random access to its nodes. However, this often leads to a heavy resource consumption in terms of memory usage due to the necessary pointer structure which represents the tree structure. The space needed to load an entire XML document into main memory in order to access it through a DOM proxy is usually 3–8 times larger than the size of the document itself [WLH07]. Therefore, it is essential for very large tree structures to use a memory efficient representation.

In [FGK03, BGK03] directed acyclic graphs (DAGs) were proposed to overcome this problem. By sharing common subtrees one is able to reduce the size of the in-memory representation by a factor of about 10 [BGK03]. One of the most appealing properties of this representation is that queries like the ones of the XPath language can be directly executed on the compressed representation, *i.e.*, it is not necessary to completely unfold the DAG.

Later, in [BLM08] so called linear straight-line context-free tree grammars were proposed as a more succinct representation of an input tree. These grammars represent exactly one tree and generalize the concept of sharing common subtrees to the sharing of repeating tree patterns. Most important, this new representation is still queryable, *i.e.*, queries can be evaluated without prior decompression. At the same time, the complexity of querying, e.g., using XQuery, stays the same as for DAGs [LM06].

However, finding the smallest linear straight-line context-free tree grammar generating a given tree is NP-hard. Already finding the smallest context-free string grammar for a given string is NP-complete [CLL+05]. In [BLM08] an algorithm called BPLEX was introduced which generates a small linear straight-line context-free tree grammar for a given input tree. On average, the resulting grammar is 3.5–4 times smaller than the minimal DAG (in terms of the number of edges). An implementation of this algorithm, which processes the underlying tree structure of XML documents, was also provided.

Main Contribution

Our main contribution is a compression algorithm, called "Re-pair for Trees", which is based on linear straight-line context-free tree grammars. Our investigations show that, regarding our test data, the grammars generated by Re-pair for Trees are always smaller than the grammars produced by the BPLEX algorithm. In addition, our algorithm outperforms BPLEX in terms of runtime and memory usage. Note that especially runtime was a huge drawback of the BPLEX implementation.

The Re-pair for Trees algorithm is a generalization of the "Re-pair" algorithm which was developed by Larsson and Moffat in [LM00]. The latter algorithm is an offline dictionary-based compression method for strings consisting of a simple but powerful phrase derivation method and a compact dictionary encoding. A *dictionary-based* compression algorithm is an algorithm where the input message is parsed into a sequence of phrases selected from a dictionary. Since the reference to a phrase in the dictionary is more compact than the phrase itself often a considerable compression can be achieved. Re-pair's dictionary is inferred *offline* since it is generated by considering the whole input message and since it is written out as a part of the compressed data so that it is available to the decoder.

Source message:

a	b	c	d	a	b	c

After the replacement of (a, b):

A	c	d	A	c

Figure 1.1: The pair (a, b) is replaced by the new symbol A.

The name Re-pair stands for "recursive pairing" and describes the idea of the algorithm. The latter is to count the frequencies of all pairs formed by two adjacent symbols of the source message, replacing the most frequent pair by a new symbol (see Fig. 1.1), updating the frequency counters of all involved pairs and repeating this process until there are no pairs occurring twice in the source message. This compression technique allows searching the compressed data without prior decompression.

Organization of this Work

In Chap. 3 we explain in detail the two steps of which the Re-pair for Trees algorithm consists. We also present a complete example of a run of our algorithm and consider the compressibility of special types of trees depending on the maximal rank allowed for a nonterminal. Chapter 4 gives an insight into our implementation of the Re-pair for Trees algorithm which is called TreeRePair. In particular, we elaborate on its linear runtime, the internal data structures used and its efficient in-memory representation of the input tree. Moreover, in Chap. 5 we present a succinct coding which is specialized in further compressing the grammars generated by the Re-pair for Trees algorithm without loosing the ability to directly execute queries on this compressed representation of the input tree. By using a combination of multiple Huffman codings, a run-length coding and a fixed-length coding the resulting file sizes are always smaller than the sizes of the files generated by competing compression algorithms when executed on our test data. In Chap. 6 we compare the compression results of our implementation of the Re-pair for Trees algorithm with several other compression algorithms. In particular, we consider BPLEX and "Extended-Repair". The latter algorithm is also based on the Repair for strings algorithm and was independently developed at the University of Paderborn, Germany [Kri08, BHK10].

2
Preliminaries

In the following, $\mathbb{N}$ denotes the set of natural numbers. We define $\mathbb{N}_{>0} = \mathbb{N} \setminus \{0\}$. For a set X we denote by X^* the set of all finite words over X. By $X \times Y = \{(x,y) \mid x \in X, y \in Y\}$ we denote the cartesian product of X and Y. For $w = x_1 x_2 \dots x_n \in X^*$ we define $|w| = n$. The empty word is denoted by ε.

We sometimes surround an element of $\mathbb{N}$ by square brackets in order to emphasize that we currently consider it a character instead of a number. For instance, for the sequence of integers 222221 we shortly write $[2]^5[1]$ instead of $2^5 1$ to clarify that we are not dealing with the fifth power of 2.

2.1 Labeled Ordered Tree

Definition 1 (Ranked alphabet) A *ranked alphabet* denotes a tuple $(\mathcal{F}, \mathsf{rank})$, where $\mathcal{F}$ is a finite set of *function symbols* and the function $\mathsf{rank} : \mathcal{F} \to \mathbb{N}$ assigns to each $\alpha \in \mathcal{F}$ its *rank*. Furthermore, we define $\mathcal{F}_i = \{a \in \mathcal{F} \mid \mathsf{rank}(\alpha) = i\}$.

We fix a ranked alphabet $(\mathcal{F}, \mathsf{rank})$ in the following.

Definition 2 ($\mathcal{F}$-labeled ordered tree) An *$\mathcal{F}$-labeled ordered tree* is a pair $t = (\mathsf{dom}_t, \lambda_t)$, where

(1) $\mathsf{dom}_t \subseteq \mathbb{N}_{>0}^*$ is a finite set of *nodes*,

(2) $\lambda_t : \mathsf{dom}_t \to \mathcal{F}$,

(3) if $w = vv' \in \mathsf{dom}_t$, then also $v \in \mathsf{dom}_t$, and

(4) if $v \in \mathsf{dom}_t$ and $\lambda_t(v) \in \mathcal{F}_n$, then $vi \in \mathsf{dom}_t$ if and only if $1 \le i \le n$.

The node $\varepsilon \in \mathsf{dom}_t$ is called the *root* of t. By $\mathsf{index}(w)$, where $w = vi \in \mathsf{dom}_t \setminus \{\varepsilon\}$ and $i \in \mathbb{N}_{>0}$, we denote the *index* i of the node w, *i.e.*, w is the i-th child of its parent node. Furthermore, we define $\mathsf{parent}(w) = v$. The *size* of t is given by the number of edges of which it consists, *i.e.*, we have $|t| = |\mathsf{dom}_t| - 1$. The *depth* of t is $\mathsf{depth}(t) = \max\{|u| \mid u \in \mathsf{dom}_t\}$. We identify an $\mathcal{F}$-labeled tree t with a term in the usual way: if $\lambda_t(\varepsilon) = \alpha \in \mathcal{F}_i$, then this term is $\alpha(t_1, \ldots, t_i)$, where t_j is the term associated with the subtree of t rooted at node j, where $j \in \{1, \ldots, i\}$. The set of all $\mathcal{F}$-labeled trees is $T(\mathcal{F})$.

Example 3 In Fig. 2.1 an $\mathcal{F}$-labeled ordered tree t is shown. We have

$$\mathsf{dom}_t = \{\varepsilon, 1, 2, 3, 11, 12, 21, 22, 31, 111, 112, 121, 122, \\ 211, 212, 221, 222\} \ .$$

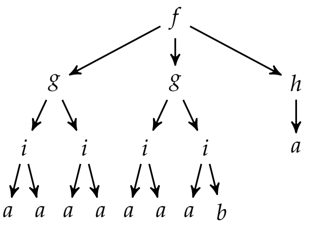


Figure 2.1: $\mathcal{F}$-labeled ordered tree t

Definition 4 (Parameters) We fix a countable set $\mathcal{Y} = \{y_1, y_2, \ldots\}$ with $\mathcal{Y} \cap \mathcal{F} = \emptyset$ of *(formal context-) parameters* (below we also use a distinguished parameter $z \notin \mathcal{Y}$). The set of all $\mathcal{F}$-labeled trees with parameters from $Y \subseteq \mathcal{Y}$ is denoted by $T(\mathcal{F}, Y)$. Formally, we consider parameters as function symbols of rank 0 and define $T(\mathcal{F}, Y) = T(\mathcal{F} \cup Y)$.

The tree $t \in T(\mathcal{F}, Y)$ is said to be *linear* if every parameter $y \in Y$ occurs at most once in t. By $t[y_1/t_1, \ldots, y_n/t_n]$ we denote the tree that is obtained by replacing in t for every $i \in \{1, 2, \ldots, n\}$ every y_i-labeled leaf with t_i, where $t \in T(\mathcal{F}, \{y_1, \ldots, y_n\})$ and $t_1, \ldots, t_n \in T(\mathcal{F}, Y)$.

Definition 5 (Context) A *context* is a tree $C \in T(\mathcal{F}, \mathcal{Y} \cup \{z\})$ in which the distinguished parameter z appears exactly once. Instead of $C[z/t]$ we write briefly $C[t]$.

Definition 6 (Tree pattern) Let $t = (\mathsf{dom}_t, \lambda_t) \in T(\mathcal{F}, \{y_1, \ldots, y_n\})$ such that for every y_i there exists a node $v \in \mathsf{dom}_t$ with $\lambda_t(v) = y_i$. We say that *t is a tree pattern occurring in $t' \in T(\mathcal{F}, \mathcal{Y})$* if there exist a context $C \in T(\mathcal{F}, \mathcal{Y} \cup \{z\})$ and trees $t_1, \ldots, t_n \in T(\mathcal{F}, \mathcal{Y})$ such that

$$C\big[t[y_1/t_1, y_2/t_2, \ldots, y_n/t_n]\big] = t' \ .$$

2.2 *SLCF Tree Grammar*

For further consideration, let us fix a countable infinite set $\mathcal{N}_i$ of symbols of rank $i \in \mathbb{N}$ with $\mathcal{F}_i \cap \mathcal{N}_i = \emptyset$ and $\mathcal{Y} \cap \mathcal{N}_0 = \emptyset$. Hence, every finite subset $N \subseteq \bigcup_{i \geq 0} \mathcal{N}_i$ is a ranked alphabet.

Definition 7 (Context-free tree grammar) A *context-free tree grammar (over the ranked alphabet $\mathcal{F}$)* or short *CF tree grammar* is a triple $\mathcal{G} = (N, P, S)$, where

(1) $N \subseteq \bigcup_{i \geq 0} \mathcal{N}_i$ is a finite set of *nonterminals*,

(2) P (the set of *productions*) is a finite set of pairs $(A \to t)$, where $A \in N$, $t \in T(\mathcal{F} \cup N, \{y_1, \ldots, y_{\mathsf{rank}(A)}\})$, $t \notin \mathcal{Y}$, each of the parameters $y_1, \ldots, y_{\mathsf{rank}(A)}$ appears in t, and[1]

[1] In contrast to [LMSS09], our definition of a context-free tree grammar inherits productivity, *i.e.*, $t \notin \mathcal{Y}$ and each parameter $y_1, \ldots, y_{\mathsf{rank}(A)}$ appears in t for every $(A \to t) \in P$. This is justified by the fact that the grammars generated by the Re-pair for Trees algorithm are always productive.

(3) $S \in N$ is the *start nonterminal* of rank 0.

We assume that every nonterminal $B \in N \setminus \{S\}$ as well as every terminal symbol from $\mathcal{F}$ occurs in the right-hand side t of some production $(A \to t) \in P$.

Let us define the derivation relation $\Rightarrow_{\mathcal{G}}$ on $T(\mathcal{F} \cup N, \mathcal{Y})$ as follows: $s \Rightarrow_{\mathcal{G}} s'$ iff there exists a production $(A \to t) \in P$ with $\mathsf{rank}(A) = n$, a context $C \in T(\mathcal{F} \cup N, \mathcal{Y} \cup \{z\})$, and trees $t_1, \ldots, t_n \in T(\mathcal{F} \cup N, \mathcal{Y})$ such that we have $s = C[A(t_1, \ldots, t_n)]$ and $s' = C[t[y_1/t_1 \cdots y_n/t_n]]$. Let

$$L(\mathcal{G}) = \{t \in T(\mathcal{F}) \mid S \Rightarrow_{\mathcal{G}}^{*} t\} \subseteq T(\mathcal{F}) \ .$$

Definition 8 (Size of a CF tree grammar) The *size* $|\mathcal{G}|$ of the CF tree grammar $\mathcal{G}$ is defined by

$$|\mathcal{G}| = \sum_{(A \to t) \in P} |t| \ .$$

That means that $|\mathcal{G}|$ equals the sum of the numbers of edges of the right-hand sides of P's productions.

Definition 9 (Restrictions on CF tree grammars) We consider the following restrictions on context-free tree grammars:

- $\mathcal{G}$ is *k-bounded* (for $k \in \mathbb{N}$) if $\mathsf{rank}(A) \leq k$ for every $A \in N$.
- $\mathcal{G}$ is *monadic* if it is 1-bounded.
- $\mathcal{G}$ is *linear* if for every $(A \to t) \in P$ the term t is linear.

Definition 10 (References of a production) Let $\mathcal{G} = (N, P, S)$ be a CF tree grammar. We denote the set of all nodes in the right-hand sides of $\mathcal{G}$'s productions which are labeled by the nonterminal $A \in N$ by $\mathsf{ref}_{\mathcal{G}}(A)$, *i.e.*,

$$\mathsf{ref}_{\mathcal{G}}(A) = \{(t, v) \mid \exists (B \to t) \in P : v \in \mathsf{dom}_t \wedge \lambda_t(v) = A\} \ .$$

Furthermore, let us define the following relation:

$$\rightsquigarrow_{\mathcal{G}} = \{(A, B) \in N \times N \mid (B \to t) \in P \wedge A \text{ occurs in } t\}$$

Definition 11 (SLCF tree grammar) A *straight-line context-free tree grammar (SLCF tree grammar)* is a CF tree grammar $\mathcal{G} = (N, P, S)$, where

(1) for every $A \in N$ there is *exactly one* production $(A \to t) \in P$ with left-hand side A, and

(2) the relation $\rightsquigarrow_{\mathcal{G}}$ is acyclic.

The conditions (1) and (2) ensure that $L(\mathcal{G})$ contains exactly one tree, which we denote by $\mathsf{val}(\mathcal{G})$.

Definition 12 (Hierarchical order) Let $\mathcal{G}$ be an SLCF tree grammar. We call the reflexive transitive closure of $\rightsquigarrow_{\mathcal{G}}$ the *hierarchical order* of $\mathcal{G}$ and denote it by $\rightsquigarrow_{\mathcal{G}}^{*}$.

Example 13 Consider the (linear and monadic) SLCF tree grammar $\mathcal{G} = (N, P, S)$ given by the following productions:

$$\begin{aligned} S &\to f\big(A(a), A(b), B\big) \\ A(y_1) &\to g\big(i(a,a), i(a,y_1)\big) \\ B &\to h(a) \end{aligned}$$

We have $\mathsf{val}(\mathcal{G}) = t$, where $t \in T(\mathcal{F})$ is the tree from Example 3 on page 16.

SLCF tree grammars can be considered as a generalization of the well-known DAGs (see, for instance, [LM06] for a common definition). Whereas the latter is a structure preserving compression of a tree by sharing common subtrees (see Fig. 2.2 for a depiction), SLCF tree grammars broaden this concept to the sharing of repeated tree patterns in a tree (see Fig. 2.3). Actually, a DAG can be considered as a 0-bounded SLCF tree grammar.

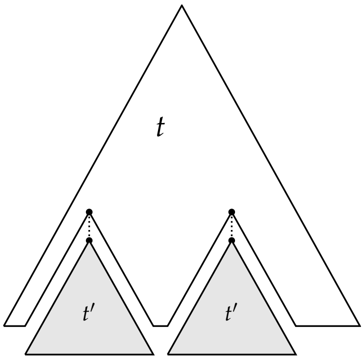


Figure 2.2: A tree t containing two occurrences of the very same subtree t'.

Definition 14 (Contribution of a production) Let $\mathcal{G} = (N, P, S)$ be a linear SLCF tree grammar. We define the function

$$\mathsf{sav}_{\mathcal{G}}(A) = |\mathsf{ref}_{\mathcal{G}}(A)| \cdot (|t| - \mathsf{rank}(A)) - |t|$$

which computes for every production $(A \to t) \in P$ its contribution to a small representation of the tree $\mathsf{val}(\mathcal{G})$ by the linear SLCF tree grammar $\mathcal{G}$.

The value $\mathsf{sav}_{\mathcal{G}}(A)$ specifies the number of edges by which the production with left-hand side A reduces the size of the grammar $\mathcal{G}$. However, $\mathsf{sav}_{\mathcal{G}}$ is not restricted to positive values. In particular, for a production $(A \to t) \in P$ with $|\mathsf{ref}_{\mathcal{G}}(A)| = 1$ we have $\mathsf{sav}_{\mathcal{G}}(A) = -\mathsf{rank}(A)$. Thus, a production which is only referenced once can be safely removed from the grammar without increasing the size of $\mathcal{G}$.

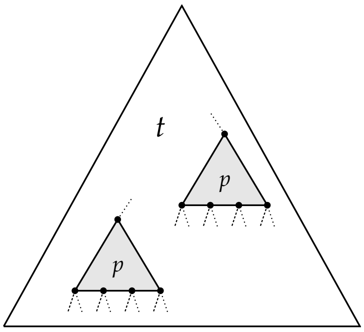


Figure 2.3: A tree t containing two occurrences of the tree pattern p.

Context-free tree grammars [CDG$^+$07] and especially SLCF tree grammars have been thoroughly studied recently. Lohrey and Maneth have shown in [LM06] that SLCF tree grammars in theory can be exponentially more succinct than DAGs which already can achieve exponential compression ratios.

Furthermore, in [LM06] various membership and complexity problems were considered. It was shown that in many cases the same complexity bounds hold as for DAGs. In particular, it was pinpointed that for a given nondeterministic tree automaton $\mathcal{A}$ and a linear, k-bounded SLCF tree grammar $\mathcal{G}$ it can be checked in polynomial time if $\mathsf{val}(\mathcal{G})$ is accepted by $\mathcal{A}$ – provided that k is a constant. This is a worth mentioning result since in the context of XML, for instance, tree automata are used to type check XML documents against an XML schema (*cf.* [MLMK05, Nev02]).

Moreover, this result was further improved in [LMSS09]. It was proved that every linear SLCF tree grammar can be transformed in polynomial time into a monadic (and linear) one. Together with the above mentioned result from [LM06] Lohrey, Maneth and Schmidt-Schauss were able to present a polynomial time algorithm for testing if a given nondeterministic tree automaton accepts a tree given by a linear SLCF tree grammar $\mathcal{G}$ – no matter what the maximum rank of all nonterminals from $\mathcal{G}$ is.

In [BLM08] the so called BPLEX algorithm was presented. It produces for a given 0-bounded SLCF tree grammar $\mathcal{G}_1$, *i.e.*, $\mathcal{G}_1$ represents a DAG, in time $O(|\mathcal{G}_1|)$ an equivalent linear SLCF tree grammar $\mathcal{G}_2$, where $\mathsf{val}(\mathcal{G}_2) = \mathsf{val}(\mathcal{G}_1)$ and $\mathcal{G}_2$ is k-bounded (k is an input parameter). Experiments have shown that $|\mathcal{G}_2|$ is approximately 2–3 times smaller than $|\mathcal{G}_1|$.

Moreover, in [LMSS09] it was proved that the evaluation problem for core XPath (the navigational part of XPath) over SLCF tree grammars is PSPACE-complete just as this was proved earlier for DAG-compressed trees by Frick, Grohe and Koch in [FGK03]. The evaluation problem for XPath asks whether a given node in a given tree is selected by a given XPath expression. This result is remarkable since with SLCF tree grammars one achieves better compression ratios than with DAGs.

2.3 XML Terminology

Regarding XML documents, we use the official terminology introduced in [BPSM$^+$08]. Thus an XML document contains one or more *elements* which are either delimited by *start-tags* and *end-tags* or by an *empty-element tag*. The text between the start-tag and the end-tag of an element is called the element's *content*. An element with no content is said to be *empty*. There is exactly one element, called *root*, which does not appear in the content of any other element.

Example 15 The simplified XML document from Fig. 2.4 consists of 21 elements of the five types `books`, `book`, `author`, `title` and `isbn`. The elements of type `books` and `book` are delimited by start- and end-tags and exhibit element content. The remaining elements are empty elements delimited by empty-element tags. The root of the XML document is the element of type `books`.

```
<books>
  <book>
    <author/><title/><isbn/>
  </book>
  ...
  <book>
    <author/><title/><isbn/>
  </book>
</books>
```

5 times

Figure 2.4: An simplified XML document.

The *name* in the start- and end-tags of an element give the element's *type*. Elements can specify *attributes* by using name-value pairs. Consider for instance the element

```
<phone prefix="012">3456</phone>
```

exhibiting one attribute specification with attribute name `prefix` and attribute value `012`.

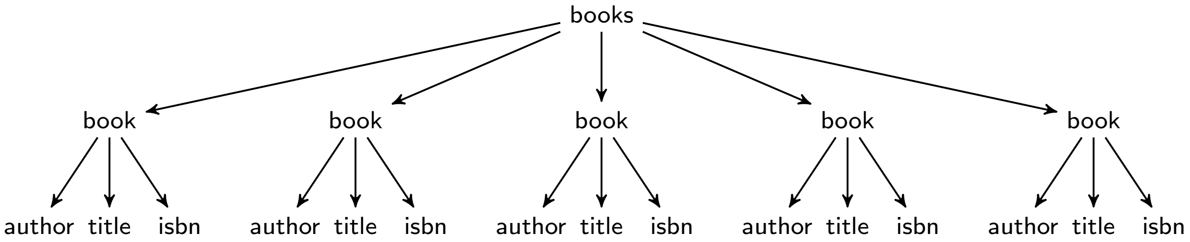


Figure 2.5: XML document tree of the XML document listed in Fig. 2.4

In addition to these terms we denote by *XML document tree* the nested structure of elements which is left after removing all character data and attribute specifications from an XML document.

2.4 *Binary Tree Model*

An XML document tree can be considered as an unranked tree, *i.e.*, nodes with the same label possibly have a varying number of children. Figure 2.5 shows the XML document tree of the XML document from Example 15. In our case, the XML document tree is a ranked tree, *i.e.*, all nodes with the same label exhibit the same number of children. However, the XML document might as well have contained an element of type `book` exhibiting a second `author` child element. In this case, we would have not obtained a ranked tree.

In the next chapter we will learn that our Re-pair for Trees algorithm operates on ranked trees only. Therefore, in general, a transformation of an XML document tree becomes necessary. A common way of modeling such a tree in a ranked way is to transform it into a binary $\mathcal{F}$-labeled ordered tree t by encoding first-child and next-sibling relations. In fact,

- the first child element of an XML element becomes the left child of the node representing its parent element and
- the right sibling element of another element becomes the right-child of the node representing its left sibling (*cf.* Fig. 2.6).

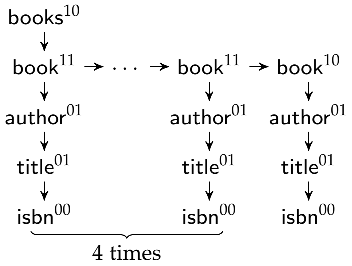


Figure 2.6: Binary tree representation of the XML document tree from Fig. 2.5.

Note that a node representing a leaf (resp. a last sibling) of the XML document has no left (resp. no right) child in the binary tree model representation. Therefore $\mathcal{F}$ does not consist of the element types of the XML document but of special versions of the element types indicating that the left, the right, both or no children are missing. In Fig. 2.6 this is denoted by superscripts at the end of the element types. These superscripts are listed in Table 2.1 together with their meanings.

Superscript	Meaning
00	no children
10	no right child
01	no left child
11	two children

Table 2.1: The superscripts and their meanings.

Let us point out that another way of preserving the rankedness along with circumventing the introduction of special labels with a lower rank is the introduction of placeholder nodes. These can be used to indicate missing left or right children. However, our experiments showed that our implementation of Re-pair for Trees achieves slightly less competitive compression results in this setting.

In [BLM08] it was stated that the binary tree model allows access to the next-in-preorder and previous-in-preorder node in $O(\mathit{depth})$, where *depth* refers to the longest path from the root of the XML document to one of its leaves. Furthermore, in [MSV03] it was demonstrated that XML query languages can be readily evaluated on the binary tree model.

3
Re-Pair for Trees

In this chapter we examine the Re-pair for Trees algorithm in detail. We will learn that it consists of two steps, namely, a *replacement step* and a *pruning step*. Furthermore, a detailed example of a run of our algorithm is presented. Finally, we investigate the impact of a possible restriction on the maximal rank allowed for nonterminals.

3.1 Definitions

In order to be able to elaborate on our Re-pair for Trees algorithm we need the following definitions. Recall that we have fixed a ranked alphabet $\mathcal{F}$ of function symbols, a set $\mathcal{N}$ of nonterminals and a set $\mathcal{Y}$ of parameters. We define the set of triples

$$\Pi = \bigcup_{a \in \mathcal{F} \cup \mathcal{N}} \{a\} \times \{1, 2, \ldots, \mathsf{rank}(a)\} \times (\mathcal{F} \cup \mathcal{N}) \ .$$

Definition 16 (Digram) A *digram* is a triple $\alpha = (a, i, b) \in \Pi$. The symbol a is called the *parent symbol of the digram* α and b is called the *child symbol of the digram* α, respectively. We define

$$\mathsf{par}(\alpha) = \mathsf{rank}(a) + \mathsf{rank}(b) - 1 \text{ and}$$
$$\mathsf{pat}(\alpha) = a\big(y_1, \ldots, y_{i-1}, b(y_i, \ldots, y_{j-1}), y_j, \ldots, y_{\mathsf{par}(\alpha)}\big) \ ,$$

where $j = i + \mathsf{rank}(b)$ and $y_1, y_2, \ldots, y_{\mathsf{par}(\alpha)} \in \mathcal{Y}$. Let $m \in \mathbb{N} \cup \{\infty\}$. We further define the set

$$\Pi_m = \{\alpha \in \Pi \mid \mathsf{par}(\alpha) \leq m\} \ .$$

Obviously, it holds that $\Pi_\infty = \Pi$.

We can consider $\mathsf{pat}(\alpha)$ as the tree pattern which is represented by the digram α. We usually denote digrams by possibly indexed lowercase letters $\alpha, \alpha_1, \alpha_2, \ldots, \beta, \ldots$ of the Greek alphabet.

Definition 17 (Occurrence) An *occurrence* of the digram $\alpha \in \Pi$ within the tree $t = (\mathsf{dom}_t, \lambda_t) \in T(\mathcal{F} \cup \mathcal{N}, \mathcal{Y})$ is a node $v \in \mathsf{dom}_t$ at which a subtree

$$\mathsf{pat}(\alpha)[y_1/t_1, y_2/t_2, \dots, y_{\mathsf{par}(\alpha)}/t_{\mathsf{par}(\alpha)}] \ ,$$

where $t_1, t_2, \dots, t_{\mathsf{par}(\alpha)} \in T(\mathcal{F} \cup \mathcal{N}, \mathcal{Y})$, is rooted. The *set of all occurrences of the digram α in t* is denoted by $\overline{\mathsf{occ}}_t(\alpha) \subseteq \mathsf{dom}_t$.

Definition 18 (Overlapping occurrences) Let $\alpha = (a, i, a') \in \Pi$ and $\beta = (b, j, b') \in \Pi$. Let $t \in T(\mathcal{F} \cup \mathcal{N}, \mathcal{Y})$. Furthermore, let $v \in \overline{\mathsf{occ}}_t(\alpha)$ and $w \in \overline{\mathsf{occ}}_t(\beta)$ be two occurrences. The occurrences v and w are *overlapping* if one of the following equations holds: $v = w$, $vi = w$ or $wj = v$. We denote this by $v \parallel w$. Otherwise, *i.e.*, if v and w are not overlapping, v and w are said to be *non-overlapping* (denoted by $v \nparallel w$).

Definition 19 (Overlapping set of occurrences) Let $\alpha \in \Pi$ and $t \in T(\mathcal{F} \cup \mathcal{N}, \mathcal{Y})$. A subset $\sigma \subseteq \overline{\mathsf{occ}}_t(\alpha)$ is said to be overlapping if there exist $v, w \in \sigma$ such that $v \parallel w$ holds. In contrast, σ is *non-overlapping* if $v \nparallel w$ for all $v, w \in \sigma$.

Let $\alpha = (a, i, b) \in \Pi$ be a digram an let $t \in T(\mathcal{F} \cup \mathcal{N}, \mathcal{Y})$. It is easy to see that the set $\overline{\mathsf{occ}}_t(\alpha)$ is non-overlapping if $a \neq b$. In contrast, if we have $a = b$, the set $\overline{\mathsf{occ}}_t(\alpha)$ potentially contains overlapping occurrences. Consider the following example:

Example 20 Let $t \in T(\mathcal{F})$ be the tree depicted in Fig. 3.1 and let $\alpha = (f, 2, f)$. We have $\{\varepsilon, 2, 22\} \subseteq \overline{\mathsf{occ}}_t(\alpha)$ where on the one hand ε and 2 and on the other hand 2 and 22 are overlapping occurrences of α.

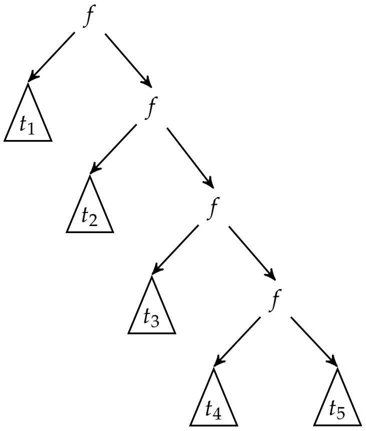


Figure 3.1: Tree $t \in T(\mathcal{F})$ consisting of nodes labeled by the terminal $f \in \mathcal{F}_2$ and the subtrees $t_1, t_2, \dots, t_5 \in T(\mathcal{F})$. We have to deal with overlapping occurrences of the digram $(f, 2, f)$.

Let $\alpha \in \Pi$ and $t \in T(\mathcal{F} \cup \mathcal{N}, \mathcal{Y})$. Let $\sigma \subseteq \overline{\mathsf{occ}}_t(\alpha)$ be a non-overlapping set. Furthermore, let us assume that $\sigma \cup \{v\}$ is overlapping for all $v \in \overline{\mathsf{occ}}_t(\alpha) \setminus \sigma$, *i.e.*, σ is maximal with respect to inclusion among non-overlapping subsets. Then σ is not necessarily maximal with respect to cardinality.

Example 21 Consider the tree $t \in T(\mathcal{F})$ which is depicted in Fig. 3.1. Let $\alpha = (f, 2, f) \in \Pi$. We have $\overline{\mathsf{occ}}_t(\alpha) = \{\varepsilon, 2, 22\}$. Let $\sigma = \{2\} \subseteq \overline{\mathsf{occ}}_t(\alpha)$. The set σ is non-overlapping and $\sigma \cup \{v\}$ is overlapping for all $v \in \overline{\mathsf{occ}}_t(\alpha) \setminus \sigma$. However, σ is not maximal with respect to cardinality. Consider the non-overlapping subset $\sigma' = \{\varepsilon, 22\} \subseteq \overline{\mathsf{occ}}_t(\alpha)$. We have $|\sigma| < |\sigma'|$.

Example 21 shows us that we cannot choose an arbitrary subset $\sigma \subseteq \overline{\mathsf{occ}}_t(\alpha)$ which is non-overlapping and maximal with respect to inclusion to obtain a set which is maximal with respect to cardinality. Let us also point out that the set $\overline{\mathsf{occ}}_t(\alpha)$ may contain more than one maximal (with respect to cardinality) non-overlapping subset.

Example 22 Consider the tree $f(f(f(a)))$ over the ranked alphabet $\mathcal{F}$. The sets $\{\varepsilon\}$ and $\{1\}$ are both maximal with respect to cardinality.

```
FUNCTION next-in-postorder($t$,$v$) // let $t = (\mathsf{dom}_t, \lambda_t)$
   if ($v = \varepsilon$) then
      $v :=$ walk-down($t$, $v$);
   else
      $i := \mathsf{index}(v) + 1$;
      $v := \mathsf{parent}(v)$;

      if ($\mathsf{rank}(\lambda_t(v)) \geq i$) then
         $v := vi$;
         $v :=$ walk-down($t$, $v$);
      endif
   endif
   return $v$;
ENDFUNC

FUNCTION walk-down($t$, $v$) // let $t = (\mathsf{dom}_t, \lambda_t)$
   while (true) do
      if ($\mathsf{rank}(\lambda_t(v)) > 0$) then
         $v := v1$;
      else
         return $v$;
      endif
   endwhile
ENDFUNC
```

Figure 3.2: The algorithm which is used to traverse a tree in postorder.

```
FUNCTION retrieve-occurrences($t$,$\alpha$) // let $\alpha = (a, i, b)$
   $\mathsf{occ}_t(\alpha) := \emptyset$; $v := \varepsilon$;
   while (true) do
      $v :=$ next-in-postorder($t$, $v$);
      if ($v \in \overline{\mathsf{occ}}_t(\alpha) \wedge vi \notin \mathsf{occ}_t(\alpha)$) then
         $\mathsf{occ}_t(\alpha) := \mathsf{occ}_t(\alpha) \cup \{v\}$
      endif
      if ($v = \varepsilon$) then
         return $\mathsf{occ}_t(\alpha)$;
      endif
   endwhile
ENDFUNC
```

Figure 3.3: The function `retrieve-occurrences` which is used to construct the set $\mathsf{occ}_t(\alpha)$ for a digram $\alpha \in \Pi$ and a tree $t \in T(\mathcal{F} \cup \mathcal{N})$.

The algorithm from Fig. 3.3 computes one non-overlapping subset of $\overline{\mathsf{occ}}_t(\alpha)$ which we denote by $\mathsf{occ}_t(\alpha)$. Lemma 24 ascertains that this subset is maximal with respect to cardinality.

Definition 23 (Set of non-overlapping occurrences) Let $\alpha \in \Pi$ be a digram and $t = (\mathsf{dom}_t, \lambda_t) \in T(\mathcal{F} \cup \mathcal{N}, \mathcal{Y})$. By $\mathsf{occ}_t(\alpha)$ we denote the subset of $\overline{\mathsf{occ}}_t(\alpha)$ which is computed by the algorithm `retrieve-occurrences(`t,α`)` from Fig. 3.3.

With the help of the function `next-in-postorder` listed in Fig. 3.2 we are able to traverse the tree t in postorder. We begin by passing the parameters t and ε and obtain the first node $u \in \mathsf{dom}_t$ of t in postorder. The second node in post order is obtained by passing the parameters t and u. This step can be repeated to traverse the whole tree t in postorder.

For every node v which is encountered during the postorder traversal it is checked if v is an occurrence of α and if it is non-overlapping with all occurrences already contained in the current set $\mathsf{occ}_t(\alpha)$. If both conditions are fulfilled, the node v is added to $\mathsf{occ}_t(\alpha)$.

Now, let us assume that we have constructed the set $\mathsf{occ}_t(\alpha)$ using the function `retrieve-occurrences`. It is obvious that the inclusion $\mathsf{occ}_t(\alpha) \subseteq \overline{\mathsf{occ}}_t(\alpha)$ holds. If $a \neq b$ in $\alpha = (a, i, b)$ we have $\mathsf{occ}_t(\alpha) = \overline{\mathsf{occ}}_t(\alpha)$. In the following, we show that the subset $\mathsf{occ}_t(\alpha) \subseteq \overline{\mathsf{occ}}_t(\alpha)$ is maximal with respect to cardinality.

Lemma 24 *Let $\alpha \in \Pi$ and $t \in T(\mathcal{F} \cup \mathcal{N}, \mathcal{Y})$. Let $\sigma \subseteq \overline{\mathsf{occ}}_t(\alpha)$ be non-overlapping and maximal with respect to cardinality. Then the equation $|\mathsf{occ}_t(\alpha)| = |\sigma|$ holds.*

Proof In the following we briefly write "maximal" for "maximal with respect to cardinality". Let $\alpha = (a, i, b) \in \Pi$, $t \in T(\mathcal{F} \cup \mathcal{N}, \mathcal{Y})$ and σ as above. The graph (V, E), where

$$\begin{aligned} V &= \overline{\mathsf{occ}}_t(\alpha) \cup \{vi \mid v \in \overline{\mathsf{occ}}_t(\alpha)\} \text{ and} \\ E &= \{(v, vi) \mid v \in \overline{\mathsf{occ}}_t(\alpha)\} \ , \end{aligned}$$

is a disjoint union of paths. Maximal non-overlapping subsets of $\overline{\mathsf{occ}}_t(\alpha)$ exactly correspond to maximum matchings in (V, E). Clearly, a path with an odd number of edges has a unique maximum matching, whereas a path with an even number of edges has two maximum matchings: one containing the first edge (in direction from the root) and one containing the last edge on the path. Intuitively, the algorithm from Fig. 3.3 finds the maximum matching in (V, E) which contains for every path with an even number of edges the last edge in direction from the root. □

Definition 25 (Replacement of a digram) Let $t \in T(\mathcal{F} \cup \mathcal{N}, \mathcal{Y})$, $\alpha = (a, i, b) \in \Pi$ and $A \in \mathcal{N}_{\mathsf{par}(\alpha)}$. By $t[\alpha/A]$ we denote the tree which is obtained by replacing all occurrences from $\mathsf{occ}_t(\alpha)$ in the tree t by the nonterminal A (in parallel). More precisely, we replace every subtree

$$\mathsf{pat}(\alpha)[y_1/t_1, y_2/t_2, \ldots, y_{\mathsf{par}(\alpha)}/t_{\mathsf{par}(\alpha)}],$$

where $t_1, t_2, \ldots, t_{\mathsf{par}(\alpha)} \in T(\mathcal{F} \cup \mathcal{N}, \mathcal{Y})$, which is rooted at an occurrence $v \in \mathsf{occ}_t(\alpha)$ by a new subtree $A(t_1, t_2, \ldots, t_{\mathsf{par}(\alpha)})$.

Example 26 Consider the tree $t \in T(\mathcal{F})$ which is depicted in Fig. 3.4.1. We have $\mathsf{occ}_t(\alpha) = \{\varepsilon, 11, 12, 21, 22\}$, where $\alpha = (f, 2, f)$. By replacing the digram α in t by a nonterminal $A \in \mathcal{N}_3$ we obtain the tree $t[\alpha/A]$ which is depicted in Fig. 3.4.2.

Definition 27 (Most frequent digram) Let $t \in T(\mathcal{F} \cup \mathcal{N})$ and let $m \in \mathbb{N} \cup \{\infty\}$. We define

$$\mathsf{max}_m(t) = \begin{cases} \alpha \in \Pi_m & \text{if } \mathsf{occ}_t(\alpha) \neq \emptyset \text{ and} \\ & \qquad \forall \beta \in \Pi_m : |\mathsf{occ}_t(\beta)| \leq |\mathsf{occ}_t(\alpha)| \\ \mathsf{undefined} & \text{if } \forall \alpha \in \Pi_m : \mathsf{occ}_t(\alpha) = \emptyset \end{cases}$$

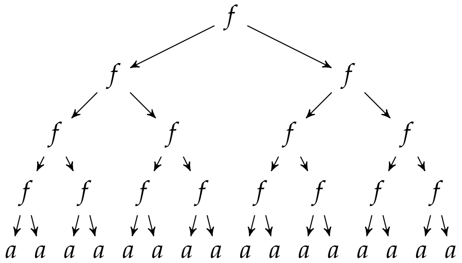


Figure 3.4.1: The tree $t \in T(\mathcal{F})$.

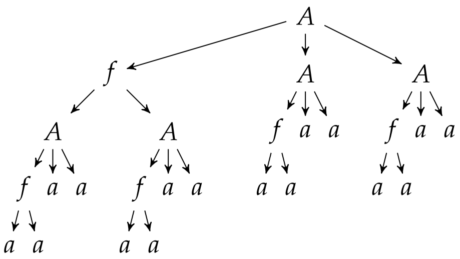


Figure 3.4.2: The tree $t[\alpha/A]$.

The function $\mathsf{max}_m : T(\mathcal{F} \cup \mathcal{N}) \to \Pi$ associates with every tree $t \in T(\mathcal{F} \cup \mathcal{N})$ a digram $\alpha \in \Pi_m$ which occurs in t most frequently (with respect to all digrams from Π_m). If there are multiple most frequent digrams, we can choose any of them. In contrast, we have $\mathsf{max}_m(t) = \mathsf{undefined}$ if there is no most frequent digram. If $m = \infty$ there is no most frequent pair if and only if the tree t consists of exactly one node. Now let us assume that $m \neq \infty$. We have $\mathsf{max}_m(t) = \mathsf{undefined}$ if and only if t consists of exactly one node or if for all digrams α occurring in t it holds that $\alpha \notin \Pi_m$.

In the sequel, if we do not specify the maximal rank allowed for a nonterminal, we always assume that $m = \infty$. For convenience we write $\mathsf{max}(t)$ instead of $\mathsf{max}_\infty(t)$, *i.e.*, we omit the symbol ∞.

3.2 *Replacement of Digrams*

In this section we give an insight into the first step of our Re-pair for Trees algorithm, namely, the *replacement step*. Let $m \in \mathbb{N} \cup \{\infty\}$ be the maximal rank allowed for a nonterminal[1] and let the tree $t = (\mathsf{dom}_t, \lambda_t) \in T(\mathcal{F})$ be the input of our algorithm.

We describe a run of the Re-pair for Trees algorithm by a sequence of $h+1$ linear SLCF tree grammars $\mathcal{G}_0, \mathcal{G}_1, \ldots, \mathcal{G}_h$, where $h \in \mathbb{N}$. For every $i \in \{0, 1, \ldots, h\}$ we have $\mathcal{G}_i = (N_i, P_i, S_i)$, $(S_i \to t_i) \in P_i$, $\alpha_i = \mathsf{max}_m(t_i)$ and $\mathsf{val}(\mathcal{G}_i) = t$. The grammar $\mathcal{G}_0$ contains solely the start production $(S_0 \to t_0)$, where $t_0 = t$. We obtain the grammar $\mathcal{G}_{i+1}$ by replacing the digram α_i in the right-hand side of $\mathcal{G}_i$'s start production t_i by a new nonterminal $A_{i+1} \in \mathcal{N}_{\mathsf{par}(\alpha_i)} \setminus N_i$ $(0 \le i \le h-1)$. We set

$$\begin{aligned} N_{i+1} &= (N_i \setminus \{S_i\}) \cup \{S_{i+1}, A_{i+1}\} \text{ and} \\ P_{i+1} &= (P_i \setminus \{(S_i \to t_i)\}) \cup \{(A_{i+1} \to \mathsf{pat}(\alpha_i)), (S_{i+1} \to t_{i+1})\} \ , \end{aligned}$$

where $t_{i+1} = t_i[\alpha_i / A_{i+1}]$.

The computation stops if there is no digram $\alpha \in \Pi_m$ occurring at least twice in the start production of the current grammar, *i.e.*, either $|\mathsf{occ}_{t_h}(\mathsf{max}_m(t_h))| = 1$ or $\mathsf{max}_m(t_h) = \mathsf{undefined}$ holds. In contrast, for all $0 \le i \le h-1$ we have $|\mathsf{occ}_{t_i}(\mathsf{max}_m(t_i))| > 1$.

Note that the linear SLCF tree grammar $\mathcal{G}_h$ is almost in Chomsky normal form (CNF) as it is defined in [LMSS09]. By appropriately transforming the right-hand side of S_h (as it is described in

[1] Regarding our implementation of the Re-pair for Trees algorithm which is described in Chap. 4, m is a parameter which can be specified by the user.

the proof of Proposition 5 of [LMSS09]) and introducing a production with right-hand side $a(y_1,\ldots,y_n)$ for every terminal $a \in \mathcal{F}_n$ ($n \in \mathbb{N}$) we would obtain a linear SLCF tree grammar which perfectly meets the requirements of the CNF.

The linear SLCF tree grammar $\mathcal{G}_h$ can only be considered an intermediate result, since it potentially consists of productions which do not contribute to a compact representation of the input tree t. Therefore, we get rid of unprofitable productions by eliminating them during the so-called *pruning step*. The latter, which is described in the next section, is executed directly after the replacement step.

3.3 *Pruning the Grammar*

Definition 28 (Elimination of a production) Let $\mathcal{G} = (N, P, S)$ be a linear SLCF tree grammar. We *eliminate* a production $(A \to t)$ from P as follows:

(1) For every reference $(t', v) \in \mathsf{ref}_{\mathcal{G}}(A)$ we replace the subtree $A(t_1, t_2, \ldots, t_n)$ rooted at $v \in \mathsf{dom}_{t'}$ by the tree

$$t[y_1/t_1, y_2/t_2, \ldots, y_n/t_n] \ ,$$

where $t_1, \ldots, t_n \in T(\mathcal{F} \cup \mathcal{N}, \mathcal{Y})$ and $n = \mathsf{rank}(A)$.

(2) We update the set of productions by setting

$$P := P \setminus \{(A \to t)\} \ .$$

Let $\mathcal{G} = (N, P, S)$ be the linear SLCF tree grammar generated in the replacement step of our algorithm, *i.e.*, we have $\mathcal{G} = \mathcal{G}_h$. Let $n = |N|$ and let

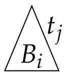
$$\omega = B_1, B_2, \ldots, B_{n-1}, B_n$$

be a sequence of all nonterminals of N in hierarchical order, *i.e.*, the following conditions hold:

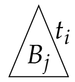
(i) $B_n = S$

(ii) $\forall 1 \leq i < j \leq n : B_j \not\rightsquigarrow^*_{\mathcal{G}} B_i$

Let $(B_i \to t_i), (B_j \to t_j) \in P$, where $1 \leq i, j < n$ and $i \neq j$. If we eliminate B_i this may have an impact on the value of $\mathsf{sav}_{\mathcal{G}}(B_j)$. We need to differentiate between two cases:

(1) $B_j \to$ 

If B_i occurs in t_j, *i.e.*, $B_i \rightsquigarrow_{\mathcal{G}} B_j$, then $|t_j|$ is increased because of the elimination of B_i. At the same time, $\mathsf{sav}_{\mathcal{G}}(B_j)$ goes up if we have $|\mathsf{ref}_{\mathcal{G}}(B_j)| > 1$. The increase of $|t_j|$ is due to the fact that we can assume that the inequality $|\{v \in \mathsf{dom}_{t_i} \mid \lambda_{t_i}(v) \notin \mathcal{Y}\}| \geq 2$ holds. Every production which was introduced in the replacement step represents a digram and therefore consists of at least two nodes labeled by the parent and child symbol, respectively, of this digram.

(2) $B_i \to$ 

If B_j occurs in t_i, *i.e.*, $B_j \rightsquigarrow_{\mathcal{G}} B_i$, then $|\mathsf{ref}_{\mathcal{G}}(B_j)|$ and therefore $\mathsf{sav}_{\mathcal{G}}(B_j)$ are possibly increased by eliminating B_i. In fact, both values go up if $|\mathsf{ref}_{\mathcal{G}}(B_i)| > 1$.

First phase In the first phase of the pruning step, we eliminate every production $(A \to t) \in P$ with $|\mathsf{ref}_{\mathcal{G}}(A)| = 1$. That way we achieve not only a possible reduction of the size of $\mathcal{G}$ (because we have $\mathsf{sav}_{\mathcal{G}}(A) = -\mathsf{rank}(A)$ for every $A \in N$ referenced only once) but we also decrement the number of nonterminals $|N|$ each time we eliminate such a production.

Second phase In the second phase of the pruning step we eliminate all remaining inefficient productions. We consider a production $(A \to t) \in P$ as inefficient if $\mathsf{sav}_{\mathcal{G}}(A) \leq 0$. Unfortunately, this time we have to deal with a rather complex optimization problem. In contrast to the first phase, the decision whether to eliminate a production $(A \to t) \in P$ or not does now depend on the value $\mathsf{sav}_{\mathcal{G}}(A)$. However, the latter may be increased by eliminating other nonterminals (see the above case distinction). This forces us to use a heuristic to decide what productions to remove next from the grammar. In fact, after completing the first phase, we cycle through the remaining productions in their reverse hierarchical order. For every $(A \to t) \in P$ we check if $\mathsf{sav}_{\mathcal{G}}(A) \leq 0$. If this proves to be true, we eliminate $(A \to t)$. That way $|\mathcal{G}|$ and $|N|$ are possibly further reduced.

The following example shows that the size of the final grammar generated by the Re-pair for Trees algorithm may depend on the order in which possible inefficient productions are eliminated.

Example 29 Consider the linear SLCF tree grammar $\mathcal{G} = (N, P, S)$, where $N = \{S, A, B\}$ and P is the following set of productions:

$$\begin{aligned} S &\to f(A(a,a), B(A(a,a))) \\ A(y_1, y_2) &\to f(B(y_1), y_2) \\ B(y_1) &\to f(y_1, a) \end{aligned}$$

Let us assume that the grammar $\mathcal{G}$ was generated by the replacement step of our algorithm and that we now want to remove all inefficient productions. We have $\mathsf{sav}_{\mathcal{G}}(A) = -1$ and $\mathsf{sav}_{\mathcal{G}}(B) = 0$, *i.e.*, the productions with left-hand sides A and B do not contribute to a small representation of the input tree $\mathsf{val}(\mathcal{G})$. Let us consider the following two cases:

(1) If we eliminate the production with left-hand side A, we obtain the grammar $\mathcal{G}_1 = (N_1, P_1, S_1)$, where $N_1 = \{S_1, B_1\}$ and P_1 is the following set of productions:

$$\begin{aligned} S_1 &\to f(f(B_1(a), a), B_1(f(B_1(a), a))) \\ B_1(y_1) &\to f(y_1, a) \end{aligned}$$

We have $|\mathcal{G}_1| = 11$ and $\mathsf{sav}_{\mathcal{G}_1}(B_1) = 1$, *i.e.*, the production with left-hand side B_1 is not considered inefficient.

(2) In contrast, the elimination of the production with left-hand side B yields the linear SLCF tree grammar $\mathcal{G}_2 = (N_2, P_2, S_2)$, where $N_2 = \{S_2, A_2\}$ and P_2 is the following set of productions:

$$\begin{aligned} S_2 &\to f(A_2(a,a), f(A_2(a,a),a)) \\ A_2(y_1,y_2) &\to f(f(y_1,a),y_2) \end{aligned}$$

We also eliminate the production with left-hand side A_2 since we have $\mathsf{sav}_{\mathcal{G}_2}(A_2) = 0$. This leads to an updated grammar $\mathcal{G}_2 = (N_2, P_2, S_2)$, where $N_2 = \{S_2\}$ and P_2 contains solely the production

$$S_2 \to f(f(f(a,a),a), f(f(f(a,a),a),a)) \ .$$

We have $|\mathcal{G}_2| = 12$.

This case distinction shows that the order in which inefficient productions are eliminated has an influence on the size of the final grammar (since $|\mathcal{G}_1| < |\mathcal{G}_2|$). Let us consider the sequence A, B, S which is the only way to enumerate the nonterminals from N in hierarchical order. Due the fact that the above described heuristic cycles through the productions in their reverse hierarchical order to eliminate inefficient productions we would obtain the larger grammar $\mathcal{G}_2$ if we would execute the pruning step with $\mathcal{G}$ as the input grammar.

Given the above example one might expect better compression results if the inefficient productions are eliminated in the order of their $\mathsf{sav}_{\mathcal{G}}$-values, *i.e.*, if we would proceed as follows: as long as their is a production whose left-hand side has a $\mathsf{sav}_{\mathcal{G}}$-value smaller or equal to 0 we remove a production whose left-hand side has the smallest occurring $\mathsf{sav}_{\mathcal{G}}$-value. However, our investigations showed that this approach leads to unappealing final grammars — at least for our set of test input trees. The grammars generated by this approach exhibit nearly the same number of edges but much more nonterminals (about 50% more) compared to the grammars obtained using the above heuristic.

Note that it is not possible to already detect digrams leading to inefficient productions during the replacement step. For instance, we would not act wisely if we would ignore digrams occurring only twice and exhibiting a large number of parameters a priori.

Example 30 Imagine an input tree $t \in T(\mathcal{F})$ comprising two instances of a large tree pattern $t' \in T(\mathcal{F}, \mathcal{Y})$. Let $\lambda_{t'}(v) \neq \lambda_{t'}(u)$ for all $v, u \in \mathsf{dom}_{t'}$, $u \neq v$. Furthermore, let us assume that all symbols in the tree pattern t' are not occurring outside of this pattern. For every digram α occurring in the tree pattern t' (whose

digram α	$\lvert\mathsf{occ}_{t_0}(\alpha)\rvert$
$(\mathsf{title}^{01}, 1, \mathsf{isbn}^{00})$	5
$(\mathsf{author}^{01}, 1, \mathsf{title}^{01})$	5
$(\mathsf{book}^{11}, 1, \mathsf{author}^{01})$	4
$(\mathsf{book}^{11}, 2, \mathsf{book}^{11})$	2
$(\mathsf{book}^{11}, 2, \mathsf{book}^{10})$	1
$(\mathsf{book}^{10}, 1, \mathsf{author}^{01})$	1
$(\mathsf{books}^{10}, 1, \mathsf{book}^{11})$	1

Table 3.1.1: All digrams encountered in the input tree t_0 and their number of non-overlapping occurrences.

digram α	$\lvert\mathsf{occ}_{t_1}(\alpha)\rvert$
$(\mathsf{author}^{01}, 1, A_1)$	5
$(\mathsf{book}^{11}, 1, \mathsf{author}^{01})$	4
$(\mathsf{book}^{11}, 2, \mathsf{book}^{11})$	2
$(\mathsf{book}^{11}, 2, \mathsf{book}^{10})$	1
$(\mathsf{book}^{10}, 1, \mathsf{author}^{01})$	1
$(\mathsf{books}^{10}, 1, \mathsf{book}^{11})$	1

Table 3.1.2: All digrams encountered in the tree t_1 and their number of non-overlapping occurrences.

digram α	$\lvert\mathsf{occ}_{t_2}(\alpha)\rvert$
$(\mathsf{book}^{11}, 1, A_2)$	4
$(\mathsf{book}^{11}, 2, \mathsf{book}^{11})$	2
$(\mathsf{book}^{11}, 2, \mathsf{book}^{10})$	1
$(\mathsf{book}^{10}, 1, A_2)$	1
$(\mathsf{books}^{10}, 1, \mathsf{book}^{11})$	1

Table 3.1.3: All digrams encountered in the tree t_2 and their number of non-overlapping occurrences.

replacement may firstly lead to a production with a large number of parameters) we would have $\lvert\mathsf{occ}_t(\alpha)\rvert = 2$. It becomes clear that this great redundancy in the input tree t, which can be represented by a production with right-hand side t', would not be detected if we would not carry out these initially anything but efficient seeming digram replacements.

3.4 *Complete Example*

Let the tree depicted in Fig. 2.6 be our input tree t_0 and let there be no restrictions on the maximal rank allowed for a nonterminal. We set $\mathcal{G}_0 = (N_0, P_0, S_0)$, where $N_0 = \{S_0\}$ and P_0 solely contains the production $(S_0 \to t_0)$. Table 3.1.1 shows every digram α encountered in t_0 along with its number of non-overlapping occurrences $\lvert\mathsf{occ}_{t_0}(\alpha)\rvert$. Furthermore, this table tells us that the two digrams $(\mathsf{title}^{01}, 1, \mathsf{isbn}^{00})$ and $(\mathsf{author}^{01}, 1, \mathsf{title}^{01})$ are the most frequent digrams occuring in t_0. We decide to replace the former digram and therefore have $\mathsf{max}(t_0) = (\mathsf{title}^{01}, 1, \mathsf{isbn}^{00}) =: \alpha_0$.

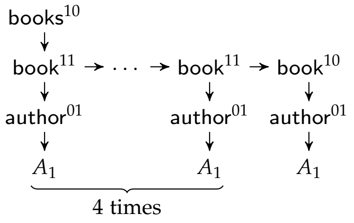


Figure 3.5: Tree t_1 which evolved from the input tree t_0 in the first iteration of our computation.

Now, in the first iteration of our computation, we generate a new grammar $\mathcal{G}_1 = (N_1, P_1, S_1)$ as follows. We introduce a new nonterminal $A_1 \in \mathcal{N}_0$ and set $N_1 = \{S_1, A_1\}$. We introduce the new production $\big(A_1 \to \mathsf{pat}(\alpha_0)\big)$, where $\mathsf{pat}(\alpha_0) = \mathsf{title}^{01}(\mathsf{isbn}^{00})$. Finally, we set $P_1 = \{(S_1 \to t_1), \big(A_1 \to \mathsf{pat}(\alpha_0)\big)\}$, where we have $t_1 = t_0[\alpha_0 / A_1]$. The tree t_1 is depicted in Fig. 3.5.

In the second iteration, during which we generate the grammar $\mathcal{G}_2 = (N_2, P_2, S_2)$, we have $\mathsf{max}(t_1) = (\mathsf{author}^{01}, 1, A_1) =: \alpha_1$ as it can be seen in Table 3.1.2. Again, we introduce a new nonterminal $A_2 \in \mathcal{N}_0$ with right-hand side $\mathsf{pat}(\alpha_1)$, set $N_2 = \{S_2, A_1, A_2\}$ and set $P = \{(S_2 \to t_2), \big(A_1 \to \mathsf{pat}(\alpha_0)\big), \big(A_2 \to \mathsf{pat}(\alpha_1)\big)\}$, where $t_2 = t_1[\alpha_1 / A_2]$ (see Fig. 3.6).

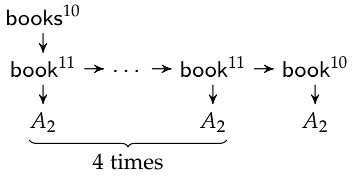


Figure 3.6: Tree t_2 which evolved from the tree t_1 in the second iteration of our computation.

We have $\mathsf{max}(t_2) = (\mathsf{book}^{11}, 1, A_2) =: \alpha_2$ (*cf.* Table 3.1.3) in the third iteration of our algorithm. This time, we need to introduce a new nonterminal $A_3 \in \mathcal{N}_1$, *i.e.*, a nonterminal with one parameter, with right-hand side $\mathsf{pat}(\alpha_2) = \mathsf{book}^{11}(A_2, y_1)$. We obtain the grammar $\mathcal{G}_3 = (N_3, P_3, S_3)$, where $N_3 = \{S_3, A_1, A_2, A_3\}$, $P_3 = (P_2 \setminus \{(S_2 \to t_2)\}) \cup \{(S_3 \to t_3), (A_3 \to \mathsf{pat}(\alpha_2))\}$ and $t_3 = \mathsf{books}^{10}(A_3(A_3(A_3(A_3(\mathsf{book}^{10}(A_2))))))$ by replacing the 4 occurrences of α_2.

In the fourth and last iteration the digram $(A_3, 1, A_3)$ is replaced by a new nonterminal $A_4 \in \mathcal{N}_1$. Therefore, we obtain the grammar $\mathcal{G}_4 = (N_4, P_4, S_4)$ with 10 edges and 5 nonterminals, where we have $N_4 = \{S_4, A_1, A_2, A_3, A_4\}$ and P_4 is the following set of productions:

$$\begin{aligned} S_4 &\to \mathsf{books}^{10}(A_4(A_4(\mathsf{book}^{10}(A_2)))) \\ A_4(y_1) &\to A_3(A_3(y_1)) \\ A_3(y_1) &\to \mathsf{book}^{11}(A_2, y_1) \\ A_2 &\to \mathsf{author}^{01}(A_1) \\ A_1 &\to \mathsf{title}^{01}(\mathsf{isbn}^{00}) \end{aligned}$$

Finally, in the pruning step, we begin with merging the right-hand side of A_1 with the right-hand side of A_2 since $|\mathsf{ref}_{\mathcal{G}_4}(A_1)| = 1$, *i.e.*, it is only referenced once. This yields the updated production $(A_2 \to \mathsf{author}^{01}(\mathsf{title}^{01}(\mathsf{isbn}^{00})))$. Furthermore, we roll back the replacement of the digram $(A_3, 1, A_3)$ due to the fact that it does not contribute to the reduction of the total number of edges. Although the production with left-hand side A_4 is referenced twice in the right-hand sides of $\mathcal{G}_4$ and removes redundancy this gain is neutralized by the necessary edge to the parameter node. This is indicated by the $\mathsf{sav}_{\mathcal{G}_4}$ value of A_4:

$$\begin{aligned} \mathsf{sav}_{\mathcal{G}_4}(A_4) &= |\mathsf{ref}_{\mathcal{G}_4}(A_4)| \cdot (|A_3(A_3(y_1))| - \mathsf{rank}(A_4)) \\ &\qquad - |A_3(A_3(y_1))| \\ &= 2 \cdot (2-1) - 2 = 0 \end{aligned}$$

With these adjustments we obtain the linear SLCF tree grammar $\mathcal{G} = (N, P, S_4)$, where $N = \{S_4, A_2, A_3\}$ and P is the following set of productions:

$$\begin{aligned} S_4 &\to \mathsf{books}^{10}(A_3(A_3(A_3(A_3(\mathsf{book}^{10}(A_2)))))) \\ A_3(y_1) &\to \mathsf{book}^{11}(A_2, y_1) \\ A_2 &\to \mathsf{author}^{01}(\mathsf{title}^{01}(\mathsf{isbn}^{00})) \end{aligned}$$

Compared to the grammar $\mathcal{G}_4$ it has the same number of edges (namely 10) but nearly half as much nonterminals only.

3.5 Another Example

It is very unlikely to be confronted with an XML document tree which, in the binary tree model, is represented by a perfect binary tree[2]. Nevertheless we want to investigate the compression performance of our algorithm on this kind of trees since it is an interesting aspect from a theoretical point of view. Last but not least our undertaking is justified by the fact that the actual Re-pair for Trees algorithm is not restricted to applications processing XML files but can be used in other applications as well. The latter, in turn, may exhibit ranked trees similar to full binary trees.

[2] A *perfect binary tree* is a binary tree in which every node is either of rank 2 or 0 and all leaves are at the same level (*i.e.*, the paths to the root are of the same length). In contrast, a *full binary tree* has no restrictions on the level of the leaves, *i.e.*, the only requirement is that every node is either of rank 2 or 0.

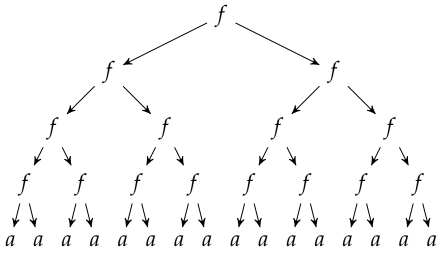


Figure 3.7.1: The perfect binary tree $t \in T(\mathcal{F})$ consisting of 15 inner nodes and 16 leaves.

$$\begin{aligned} A_1(y_1) &\to f(y_1, a) \\ A_2 &\to A_1(a) \\ A_3(y_1) &\to f(y_1, A_2) \\ A_4 &\to A_3(A_2) \\ A_5(y_1) &\to f(y_1, A_4) \\ A_6 &\to A_5(A_4) \end{aligned}$$

Figure 3.7.2: Production generated before the pruning step without the start production.

$$\begin{aligned} A_2 &\to f(a, a) \\ A_4 &\to f(A_2, A_2) \\ A_6 &\to f(A_4, A_4) \\ S &\to f(A_6, A_6) \end{aligned}$$

Figure 3.7.3: The set of productions P after executing the pruning step.

Let $t \in T(\mathcal{F})$ be a sufficiently large perfect binary tree of which each inner node is labeled by a terminal $f \in \mathcal{F}_2$ and each leaf is labeled by a terminal $a \in \mathcal{F}_0$. A run of Re-pair for Trees on t consists of $2 \cdot (d-1)$ iterations folding the input tree beginning at its leaves, where $d = \mathsf{depth}(t)$. Thus, in the first two iterations, the digrams formed by the leaf nodes and their parents are replaced. We obtain the productions $A_1(y_1) \to f(y_1, a)$ and $A_2 \to A_1(a)$ each occurring 2^{d-1} times. Now, we undertake further digram replacements in a bottom up fashion. In the $(2i-1)$-th and $2i$-th iteration we replace two digrams resulting in the productions $A_{2i-1}(y_1) \to f(y_1, A_{2(i-1)})$ and $A_{2i} \to A_{2i-1}(A_{2(i-1)})$, respectively, where $2 \leq i \leq d-1$.

The production with left-hand side A_{2k-1} is only referenced once for every $1 \leq k \leq d-1$. Therefore, in the pruning step, for every $1 \leq k \leq d-1$ the production with left-hand side A_{2k-1} is eliminated by merging its right-hand side with the right-hand side of the production with left-hand side A_{2k}. In particular, the production with left-hand side A_1 is merged with the production for A_2 resulting in a production $A_2 \to f(a, a)$.

Finally, we obtain a linear SLCF tree grammar with d nonterminals — including the left-hand side of the start production $S \to f(A_{2(d-1)}, A_{2(d-1)})$ — and a total of $2 \cdot d$ edges. Note that even though some of the intermediate productions exhibit parameters the final grammar consists only of nonterminals of rank 0. Thus, the generated grammar is a DAG and in this particular case the minimal DAG of the input tree.

Example 31 Let $t \in T(\mathcal{F})$ be the perfect binary tree from Fig. 3.7.1 with 30 edges and $\mathsf{depth}(t) = 4$. A run of Re-pair for Trees initially generates the 6 productions listed in Fig. 3.7.2. After the pruning step we finally obtain the linear SLCF tree grammar $\mathcal{G} = (N, P, S)$, where $N = \{A_2, A_4, A_6, S\}$ and the set of productions P consists of the productions from Fig. 3.7.3. The size of $\mathcal{G}$ is $|\mathcal{G}| = 8$.

3.6 Unlimited Maximal Rank

It seems natural to assume that, in general, trees can be compressed best by the Re-pair for Trees algorithm if there are no restrictions on the maximal rank of a nonterminal. However, it turns out that there are (not so uncommon) types of trees for which the opposite is true. Firstly, in this section, we will construct a set of trees whose compressibility is best if there are no restrictions on the maximal rank of a nonterminal. After that, in the succeeding section, we will present a set of trees whose compressibility is best when restricting the maximal rank to 1.

Let us consider the infinite set $M = \{t_1, t_2, t_3, \ldots\} \subseteq T(\mathcal{F})$ of trees, where for all $i \in \mathbb{N}_{>0}$ the tree t_i has the following properties:

- The tree t_i is a perfect binary tree of depth 2^i.
- Each inner node of t_i is labeled by the terminal $f \in \mathcal{F}_2$.
- Each leaf of t_i is labeled by a unique terminal from $\mathcal{F}_0$, *i.e.*, there do not exist two different leaves which are labeled by the same symbol.

Example 32 Figure 3.8.1 shows a simplified depiction of the tree $t_3 \in M$. The inner nodes labeled by the symbol $f \in \mathcal{F}_2$ are represented by a circle filled with paint. In contrast, the leaves, of which each is labeled by a unique symbol from $\mathcal{F}_0$, are depicted by a circle which is not filled with paint.

The tree t_3 is compressed by a run of our algorithm as follows. The digrams $(f, 1, f)$ and $(f, 2, f)$ occur equally often in t_3. It makes no difference to the size of the final grammar whether we replace the former or the latter. Let us replace the digram $(f, 2, f)$ (whose occurrences are painted in green in Fig. 3.8.1) by a nonterminal $A_1 \in \mathcal{N}_3$ with right-hand side $f(y_1, f(y_2, y_3))$. We obtain the tree of the form shown in Fig. 3.8.2. After that, the digram $(A_1, 1, f)$, which occurs the same number of times as $(f, 2, f)$ did, is replaced by the nonterminal $A_2 \in \mathcal{N}_4$ with right-hand side $A_1(f(y_1, y_2), y_3, y_4)$. The occurrences of $(A_1, 1, f)$ are marked with green paint in Fig. 3.8.2. The right-hand side of the nonterminal A_1 is merged with the right-hand side of A_2 during the pruning step since A_1 is only referenced once. This yields the production with left-hand side A_2 and right-hand side $f\big(f(y_1, y_2), f(y_3, y_4)\big)$.

After the replacement of the above two digrams the right-hand side of the start production is a 4-ary tree of depth 4 whose inner nodes are labeled by A_2 (see Fig. 3.8.3). Now, the digrams

$$(A_2, 1, A_2), (A_2, 2, A_2), (A_2, 3, A_2), (A_2, 4, A_2)$$

occur equally often. Again, the order of the digram replacements makes no difference to the final grammar. Assuming that at first we replace the digram $(A_2, 4, A_2)$, which is marked with green paint in Fig. 3.8.3, by a new nonterminal A_3, we obtain the tree

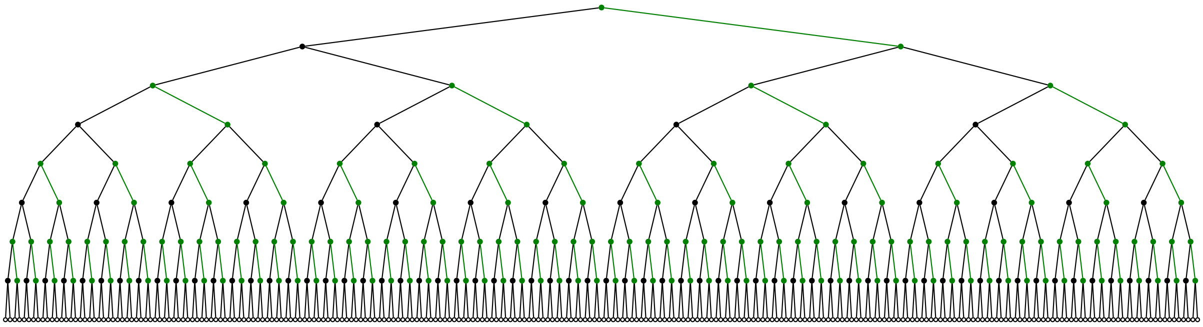


Figure 3.8.1: The tree $t_3 \in M$.

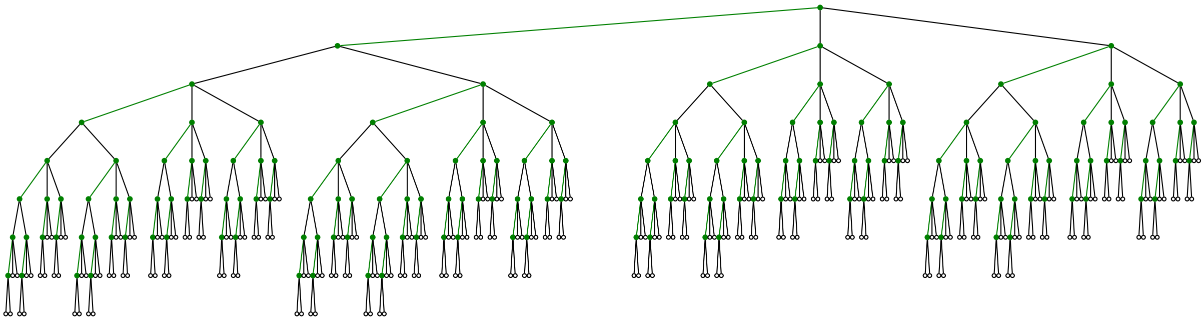


Figure 3.8.2: The tree $t_3 \in M$ after replacing the digram $(f, 2, f)$.

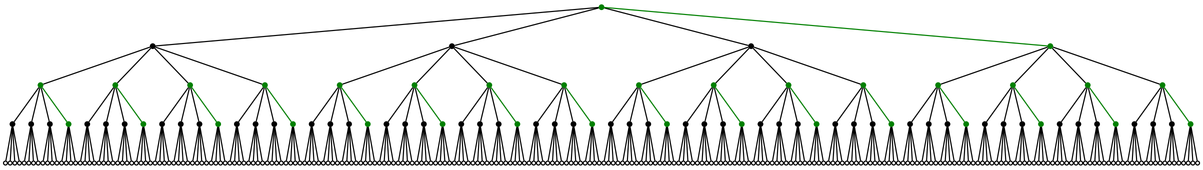


Figure 3.8.3: The tree $t_3 \in M$ after the second iteration, *i.e.*, after replacing the digrams $(f, 2, f)$ and $(A_1, 1, f)$.

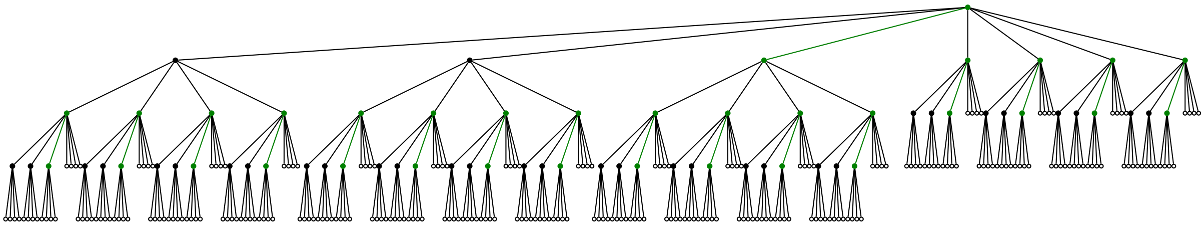


Figure 3.8.4: The tree which remains after replacing the digram $(A_2, 4, A_2)$ in the tree from Fig. 3.8.3.

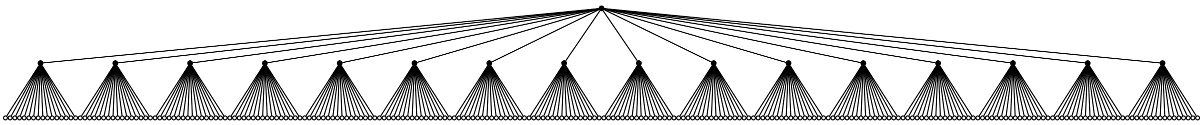


Figure 3.8.5: The tree $t_3 \in M$ after 6 iterations of our algorithm. We obtained a 16-ary tree whose inner nodes are labeled by the nonterminal A_6.

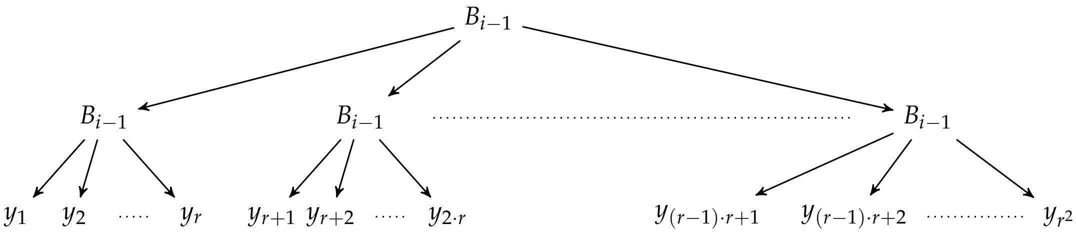

B_{i-1}
B_{i-1}
B_{i-1}
B_{i-1}
y_1 y_2 ····· y_r
y_{r+1} y_{r+2} ····· $y_{2\cdot r}$
$y_{(r-1)\cdot r+1}$ $y_{(r-1)\cdot r+2}$ ··············· y_{r^2}

Figure 3.9: Right-hand side s_i of the nonterminal B_i, where $r = \mathsf{rank}(B_{i-1})$ and $i > 1$.

shown in Fig. 3.8.4. After that, the digrams $(A_3, 3, A_2)$, $(A_4, 2, A_2)$ and $(A_5, 1, A_2)$ are replaced in three additional iterations. The above four digram replacements result in a new production

$$A_6(y_1, \dots, y_{16}) \to A_2\big(A_2(y_1, \dots, y_4), \dots, A_2(y_{13}, \dots, y_{16})\big)$$

after pruning the grammar. The remaining tree is a 16-ary tree of depth 2 (of the form depicted in Fig. 3.8.5) whose inner nodes are labeled by the nonterminal A_6. In this tree there is no digram occurring more than once. Therefore, the execution of our algorithm stops.

Now, we want to analyze the behavior of Re-pair for Trees on a tree from M in general. Let $x \in \mathbb{N}_{>0}$ and let it : $\mathbb{N}_{>0} \to \mathbb{N}_{>0}$ be the following function:

$$\mathsf{it}(x) = \sum_{i=0}^{x-1} 2^{2^i}$$

Let $B_1, B_2, B_3, \dots$ be a sequence of nonterminals where for all $i > 0$ the following conditions are fulfilled:

- $\mathsf{rank}(B_i) = 2^{2^i}$
- $s_i \in T(\mathcal{F} \cup \mathcal{N}, \mathcal{Y})$ is the right-hand side of B_i
- If $i = 1$, we have $s_i = f(f(y_1, y_2), f(y_3, y_4))$ and if $i > 1$, the tree s_i is of the form shown in Fig. 3.9, where $r = \mathsf{rank}(B_{i-1}) = 2^{2^{i-1}}$.

Regarding the nonterminals A_2 and A_6 from Example 32, we have $B_1 = A_2$ and $B_2 = A_6$, respectively. Let $i \in \{1, 2, \dots, k\}$. The following two equations hold:

$$\mathsf{rank}(B_i) = 2^{2^i} = 2^{2^{i-1}} \cdot 2^{2^{i-1}} = \mathsf{rank}(B_{i-1})^2 \tag{3.1}$$

$$|s_i| = \mathsf{rank}(B_i) + \mathsf{rank}(B_{i-1}) \tag{3.2}$$

For convenience, we define $\mathsf{rank}(B_0) = \mathsf{rank}(f) = 2$.

Now assume that we have an unlimited maximal rank allowed for a nonterminal. After $\mathsf{it}(n)$ iterations on $t_{n+1} \in M$ we have obtained the nonterminals $B_1, B_2, \dots, B_n$. The right-hand side of the start nonterminal is a $\mathsf{rank}(B_n)$-ary tree of height 2 (see also Example 32, where $n = 2$). At this point, no further replacements

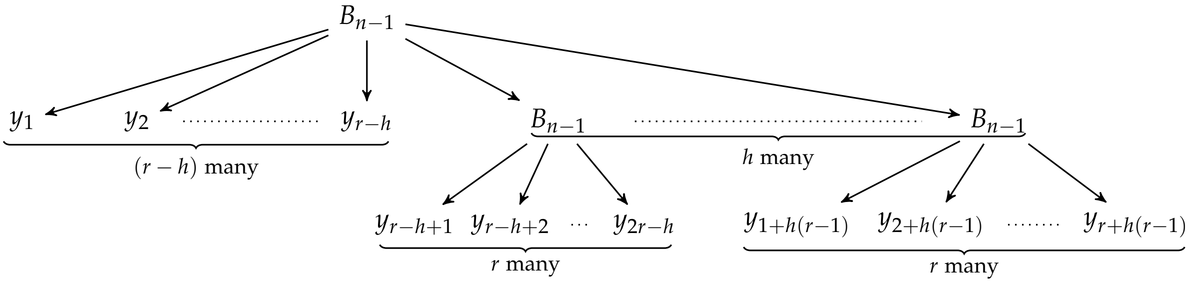


Figure 3.10: Right-hand side of the nonterminal C, where $r = \mathsf{rank}(B_{n-1})$.

are carried out. For each of the generated nonterminals $B_1, \ldots, B_n$ we have

$$|\mathsf{ref}_{\mathcal{G}}(B_i)| = \mathsf{rank}(B_i) + 1 \ , \tag{3.3}$$

where $i \in \{1, 2, \ldots, n\}$ (*cf.* Fig. 3.9). Hence, we have

$$\begin{aligned} \mathsf{sav}_{\mathcal{G}}(B_i) &= |\mathsf{ref}_{\mathcal{G}}(B_i)| \cdot \mathsf{rank}(B_{i-1}) - |s_i| \\ &\overset{(3.2)}{=} |\mathsf{ref}_{\mathcal{G}}(B_i)| \cdot \mathsf{rank}(B_{i-1}) - \mathsf{rank}(B_i) - \mathsf{rank}(B_{i-1}) \\ &\overset{(3.3)}{=} \mathsf{rank}(B_i) \cdot \mathsf{rank}(B_{i-1}) - \mathsf{rank}(B_i) \\ &\overset{(3.1)}{=} \mathsf{rank}(B_{i-1})^3 - \mathsf{rank}(B_{i-1})^2 \ > \ 0 \end{aligned}$$

since $\mathsf{rank}(B_{i-1}) \geq \mathsf{rank}(B_0) = 2$. Therefore, none of the nonterminals $B_1, \ldots, B_n$ will be eliminated in the pruning step.

Now assume that the maximal rank is $m \in \mathbb{N}$, *i.e.*, we have $m < \infty$. Choose the smallest $n \in \mathbb{N}$ such that

$$2^{2^n} > m \ . \tag{3.4}$$

Thus, B_n is the first nonterminal in the sequence $B_1, B_2, \ldots$ with a rank bigger than m. Let us consider a run of Re-pair for Trees on a tree $t_j \in M$ with $j \geq n+1$. Then, as above, the nonterminals $B_1, \ldots, B_{n-1}$ will be obtained after $\mathsf{it}(n-1)$ iterations (if we would prune the corresponding grammar by now). At this point, the right-hand side of the start production is a $2^{2^{n-1}}$-ary tree of height $2^j/2^{n-1} \geq 4$, where all inner nodes are labeled by the nonterminal B_{n-1}. Now, we can carry out h additional digram replacements leading to the nonterminals $C_1, C_2, \ldots, C_h \in \mathcal{N}$, where

$$h = \max\{l \in \mathbb{N} \mid r + l \cdot (r-1) \leq m\} \tag{3.5}$$

and $r = \mathsf{rank}(B_{n-1}) = 2^{2^{n-1}}$. We claim that

$$r = \mathsf{rank}(B_{n-1}) > h \tag{3.6}$$

holds. To see this, let us assume that $r = \mathsf{rank}(B_{n-1}) \leq h$. We have

$$m \overset{(3.5)}{\geq} r + h \cdot (r-1) \geq r + r \cdot (r-1) = r^2 = 2^{2^n} \ .$$

However, this contradicts (3.4).

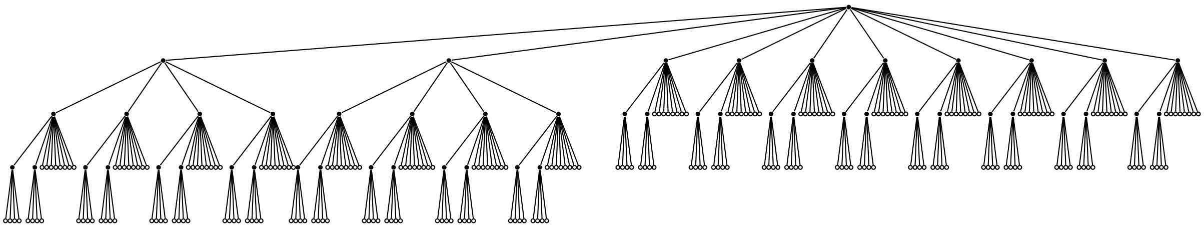


Figure 3.11: Right-hand side of the current start production after replacing the digram $(A_3, 3, A_2)$.

If we have $h > 0$, then we need to consider the following: After the pruning step, the nonterminals $C_1, C_2, \dots, C_h$ form one nonterminal $C \in \mathcal{N}$ with $\mathsf{rank}(C) = h \cdot r + r - h = r + h \cdot (r - 1)$ (see Fig. 3.10). It occurs at least $2^{2^n} + 1 = r^2 + 1$ many times according to (3.3) (the nonterminal C occurs as often as B_n does after $\mathsf{it}(n)$ iterations on t_j in the unlimited case). Each occurrence of C reduces the size of the corresponding grammar by h edges and the right-hand side of C consists of $r + h \cdot r$ edges (see Fig. 3.10). Now, let us consider the sav-value of C (assuming that $\mathcal{G}$ is the current grammar after $\mathsf{it}(n) + h$ iterations):

$$\begin{aligned}
\mathsf{sav}_{\mathcal{G}}(C) &= |\mathsf{ref}_{\mathcal{G}}(C)| \cdot h - (r + h \cdot r) \\
&\geq (r^2 + 1) \cdot h - h \cdot r - r \\
&= (r^2 - r + 1) \cdot h - r \\
&\geq r^2 - 2r + 1 \\
&= (r - 1)^2
\end{aligned}$$

Thus, we have $\mathsf{sav}_{\mathcal{G}}(C) > 0$, *i.e.*, the nonterminal C is not eliminated during the pruning step.

Example 33 Let us assume that the maximal rank for a nonterminal is restricted to 10 in Example 32. In this case we are able to undertake exactly one additional digram replacement in the tree from Fig. 3.8.4 resulting in a new nonterminal $A_4 \in \mathcal{N}_{10}$. If we replace the digram $(A_3, 3, A_2)$, we obtain the tree shown in Fig. 3.11. We have $n = 2$, $h = 2$ and $C = A_4$. After the pruning step, the right-hand side of A_4 is of the form

$$A_2(y_1, y_2, A_2(y_3, y_4, y_5, y_6), A_2(y_7, y_8, y_9, y_{10})) \enspace .$$

We can further state that the nonterminal B_{n-1} is not eliminated since it occurs $h + 1$ times in the right-hand side of C (see Fig. 3.10) and $(r - h) \cdot |\mathsf{ref}_{\mathcal{G}}(C)| \geq (r - h) \cdot (r^2 + 1)$ times in the right-hand side of the current start production (below each occurrence of C there are $r - h$ occurrences of B_{n-1} and C occurs at least $r^2 + 1$ times). Therefore, we have

$$\begin{aligned}
|\mathsf{ref}_{\mathcal{G}}(B_{n-1})| &\geq h + 1 + (r - h) \cdot (r^2 + 1) \\
&= h + 1 + r^3 - hr^2 + r - h \\
&= r^3 - hr^2 + r + 1 \enspace .
\end{aligned}$$

Tree t_i	$\mathsf{depth}(t_i)$	$\lvert t_i \rvert$	$\lvert \mathcal{G}^4 \rvert$	$\lvert \mathcal{G}^\infty \rvert$
t_2	4	30	26	26
t_3	8	510	346	298
t_4	16	131070	87386	66090

Table 3.2: Comparison of the sizes of the final grammars.

Because of (3.6), the inequality $|\mathsf{ref}_{\mathcal{G}}(B_{n-1})| > r+1$ holds. As shown before for the unlimited rank, in this case B_{n-1} has a sav-value bigger than 0 and therefore the nonterminals $B_1, B_2, \dots, B_{n-1}$ are not eliminated.

Let $\mathcal{G}^{m\prime}$ be the grammar which is obtained after $\mathsf{it}(n-1)+h$ iterations on the tree t_j when restricting the maximal rank to m and let $\mathcal{G}^{\infty\prime}$ be the current grammar after $\mathsf{it}(n)$ iterations on t_j when an unlimited rank is allowed. We can conclude that $|\mathcal{G}^{m\prime}| > |\mathcal{G}^{\infty\prime}|$ holds — no matter whether we have $h > 0$ or $h = 0$ — because of the following two facts:

(1) Each occurrence of B_n saves $\mathsf{rank}(B_{n-1})$ edges (see Fig. 3.9) and therefore according to (3.6) more than an occurrence of C does. The nonterminals B_n and C occur equally often. However, C is only existent if $h > 0$.

(2) The nonterminals $B_1, B_2, \dots, B_{n-1}$ (which are existent in both grammars, $\mathcal{G}^{m\prime}$ and $\mathcal{G}^{\infty\prime}$) and the nonterminals B_n and C are not eliminated during the pruning step.

Let $\mathcal{G}^m$ ($\mathcal{G}^\infty$) be the final grammar which is generated by a run of Re-pair for Trees on the tree t_j when restricting the maximal rank of a nonterminal to m (not restricting the maximal rank). We have $\mathcal{G}^m = \mathcal{G}^{m\prime}$ and $|\mathcal{G}^\infty| \leq |\mathcal{G}^{\infty\prime}|$. The latter holds because with every additional digram replacement at least one edge is absorbed and because during the pruning step only nonterminals with a sav-value smaller than or equal to 0 are eliminated. Therefore $|\mathcal{G}^m| > |\mathcal{G}^\infty|$ holds. Thus, we have shown that, in general, the trees from M can be compressed best if there are no restrictions on the maximal rank allowed for a nonterminal.

Example 34 Table 3.2 shows a comparison of the grammars generated by different runs of our algorithm on the trees t_2, t_3 and t_4 from M. By $\mathcal{G}^4$ ($\mathcal{G}^\infty$) we denote the final grammar which is generated when restricting the maximal rank to 4 (not restricting the maximal rank).

3.7 *Limiting the Maximal Rank*

In the preceding section we investigated a set of trees whose compressibility was best if we did not restrict the maximal rank of a nonterminal. Now, we want to construct a set of trees which behaves contrarily, *i.e.*, we construct trees which can be compressed best if we limit the maximal rank of a nonterminal to 1. In order to make it easier to quickly understand the following definition we want to refer the reader to Fig. 3.12.1 which shows one of the trees we define in the sequel.

First of all, let us define a labeling function $l : \mathbb{N} \to \mathcal{F}_0$, where

$$l(i) = \begin{cases} a & \text{if } i \equiv 0 \mod 5 \\ b & \text{if } i \equiv 1 \mod 5 \\ c & \text{if } i \equiv 2 \mod 5 \\ d & \text{if } i \equiv 3 \mod 5 \\ e & \text{if } i \equiv 4 \mod 5 \end{cases}$$

and $i \in \mathbb{N}$. Now, we are able to define for all integers $n \in \mathbb{N}$ the ordered tree $s_n = (\mathsf{dom}_{s_n}, \lambda_{s_n}) \in T(\mathcal{F})$, where

$$\mathsf{dom}_{s_n} = \left(\bigcup_{i=0}^{2^n} [2]^i \right) \cup \left(\bigcup_{i=0}^{2^n-1} [2]^i [1] \right)$$

and

$$\lambda_{s_n}(v) = \begin{cases} f \in \mathcal{F}_2 & \text{if } v = [2]^i,\ 0 \leq i < 2^n \\ l(i) \in \mathcal{F}_0 & \text{if } v = [2]^i[1],\ 0 \leq i < 2^n \\ l(2^n) \in \mathcal{F}_0 & \text{if } v = [2]^{2^n} \ . \end{cases}$$

Let us define $U = \{s_n \mid n \in \mathbb{N}, n \geq 3\}$. In the following we will show that for every run of Re-pair for Trees on a tree $s \in U$ we have $|\mathcal{G}^1| < |\mathcal{G}^\infty|$, where $\mathcal{G}^1$ is the grammar generated when allowing a maximal rank of 1 for a nonterminal and $\mathcal{G}^\infty$ is the resulting grammar when there is no restriction on the maximal rank.

Let us consider a run $\mathcal{G}_0^\infty, \mathcal{G}_1^\infty, \ldots, \mathcal{G}_{n-1}^\infty$ of the Re-pair for Trees algorithm on the tree s_n with no restrictions on the maximal rank of a nonterminal, where $\mathcal{G}_i^\infty = (N_i, P_i, S_i)$, $(S_i \to t_i)$ is the start production of $\mathcal{G}_i^\infty$ and $i \in \{0, 1, \ldots, n-1\}$. In the first iteration of our computation the digram $(f, 2, f)$ is the most frequent digram, *i.e.*, $\mathsf{max}(t_0) = (f, 2, f)$. This is because of $|\mathsf{occ}_{s_n}((f, 2, f))| = 2^{n-1}$ whereas for every $x \in \{a, b, c, d, e\}$ the inequality

$$|\mathsf{occ}_{s_n}((f, 1, x))| \leq \lceil 2^n/5 \rceil$$

holds. Therefore, we replace the digram $(f, 2, f)$ by a new nonterminal A_1 and obtain $\mathcal{G}_1^\infty$. In every subsequent iteration i we replace the digram $\mathsf{max}(t_{i-1}) = (A_{i-1}, 2^{i-1}+1, A_{i-1})$ by a new nonterminal A_i, where $i \in \{2, 3, \ldots, n-1\}$. For every $1 \leq i \leq n-1$ the right-hand side of the start production of the grammar $\mathcal{G}_i^\infty$ is given by the tree $t_i = (\mathsf{dom}_{t_i}, \lambda_{t_i})$, where

$$\mathsf{dom}_{t_i} = \left(\bigcup_{j=0}^{2^{n-i}} \left[2^i + 1 \right]^j \right) \cup \left(\bigcup_{k=1}^{2^i} \bigcup_{j=0}^{2^{n-i}-1} \left[\left(2^i + 1 \right) \right]^j [k] \right)$$

and

$$\lambda_{t_i}(v) = \begin{cases} A_i \in \mathcal{N}_{2^i+1} & \text{if } v = [2^i+1]^j \text{ with } 0 \leq j \leq 2^{n-i}-1 \ , \\ l(j \cdot 2^i + k - 1) & \text{if } v = [2^i+1]^j[k] \text{ with } 0 \leq j \leq 2^{n-i}-1 \text{ and } 1 \leq k \leq 2^i \ , \\ l(2^n) & \text{if } v = [2^i+1]^{2^{n-i}} \ . \end{cases}$$

Example 35 The Figs. 3.12.1, 3.12.2, 3.12.3 and 3.12.4 show the right-hand sides of the start productions of the grammars $\mathcal{G}_0$, $\mathcal{G}_1$, $\mathcal{G}_2$ and $\mathcal{G}_3$ generated by a run of our algorithm on the tree s_4.

In order to argue that we have $\mathsf{max}(t_i) = (A_i, 2^i+1, A_i) =: \alpha_i$ for every $0 < i < n$, we investigate the number of occurrences of all digrams occurring in the right-hand side of $\mathcal{G}_i^\infty$'s start production. Firstly, it is easy to verify that $|\mathsf{occ}_{t_i}(\alpha_i)| = 2^{n-i-1}$. In contrast, for every $1 \leq k \leq 2^i$ and $x \in \{a,b,c,d,e\}$ the inequality $|occ_{t_i}\big((A_{i-1},k,x)\big)| \leq \lfloor 2^{n-i}/5 \rfloor$ holds. This is because every power of 2 is not divisible by 5, *i.e.*, for every $1 \leq k \leq 2^i$ and every $0 \leq j \leq 2^{n-i}-5$ we have

$$\begin{aligned} \lambda_{t_i}([2^i+1]^j[k]) \ \neq \ \lambda_{t_i}([2^i+1]^{j+1}[k]) \ \neq \ \lambda_{t_i}([2^i+1]^{j+2}[k]) \\ \neq \ \lambda_{t_i}([2^i+1]^{j+3}[k]) \ \neq \ \lambda_{t_i}([2^i+1]^{j+4}[k]) \ . \end{aligned}$$

Due to the fact that we do not replace digrams with child symbols a, b, c, d or e, the right-hand side of $\mathcal{G}_{n-1}^\infty$'s start production has to contain at least 2^n nodes labeled by these symbols, *i.e.*, we can conclude that $|\mathcal{G}_{n-1}^\infty| \geq 2^n$. Therefore the compression ratio cannot be better than 50%.

In contrast, a run $\mathcal{G}_0^1, \mathcal{G}_1^1, \ldots, \mathcal{G}_k^1$ of our algorithm on s_n leads to a significantly better compression ratio when restricting the maximal rank to 1, where $k \in \mathbb{N}_{>0}$, $\mathcal{G}_i^1 = (N_i, P_i, S_i)$, $(S_i \to t_i)$ is the start production of $\mathcal{G}_i^1$ and $i \in \{0, 1, \ldots, k\}$. In the first iteration we have $\mathsf{max}_1(t_0) \neq (f,2,f)$, since a replacement of $(f,2,f)$ would result in a nonterminal with a rank greater than 1. Therefore only the digrams $(f,1,a)$, $(f,1,b)$, $(f,1,c)$, $(f,1,d)$, $(f,1,e)$ and subsequent digrams can be replaced. It turns out that after the first nine iterations the pattern $f(a, f(b, f(c, f(d, f(e, \ldots)))))$ is represented by a new nonterminal A_9 with $\mathsf{rank}(A_9) = 1$. The actual order of the replacements within the first nine iterations depends on the method used to choose a most frequent digram when there are multiple most frequent digrams. Refer to Example 36 for one possible proceeding.

The right-hand side of $\mathcal{G}_9$'s start production is a degenerated tree mainly consisting of consecutive nonterminals A_9. The corresponding nodes — there are roughly $2^n/5$ of them — are then boiled down using approximately $\log_2(2^n/5)$ digram replacements. Therefore the number of total edges of the resulting grammar is in $\mathcal{O}(n)$, *i.e.*, it is of logarithmic size (the size of the input tree s_n is $2^{n+1}+1$). Thus, we were able to construct a set of trees which exhibit a better compressibility when restricting the maximal rank of a nonterminal to 1.

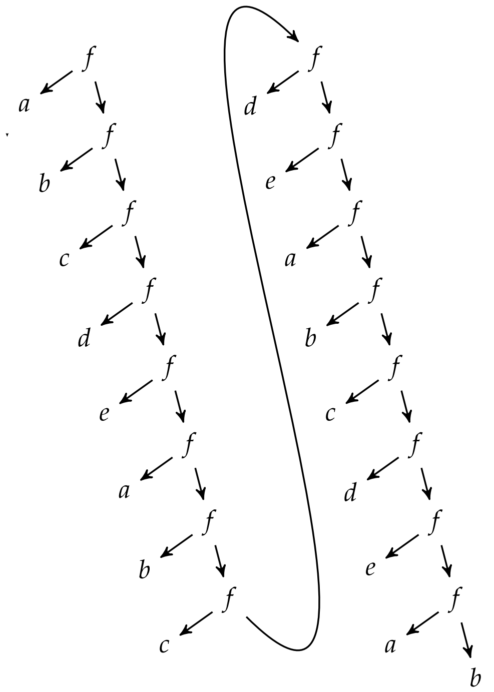


Figure 3.12.1: The tree $s_4 \in U$ which is the right-hand side of $\mathcal{G}_0$'s start production.

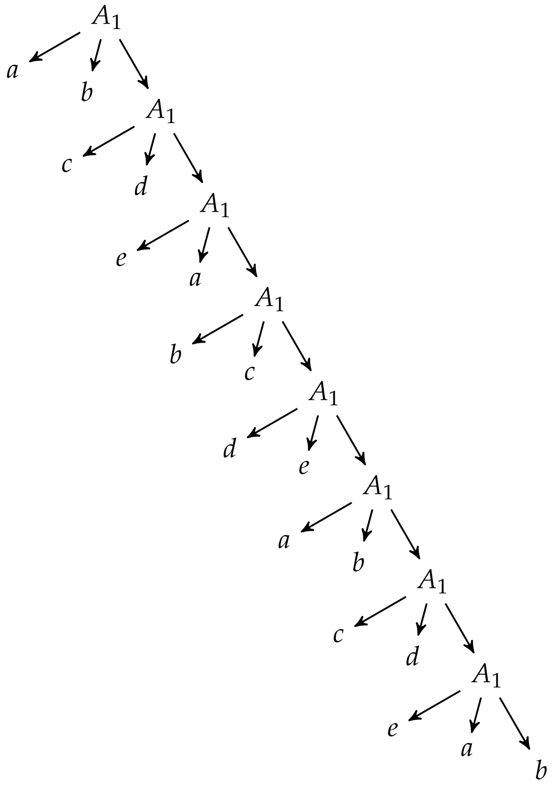


Figure 3.12.2: The right-hand side of $\mathcal{G}_1$'s start production.

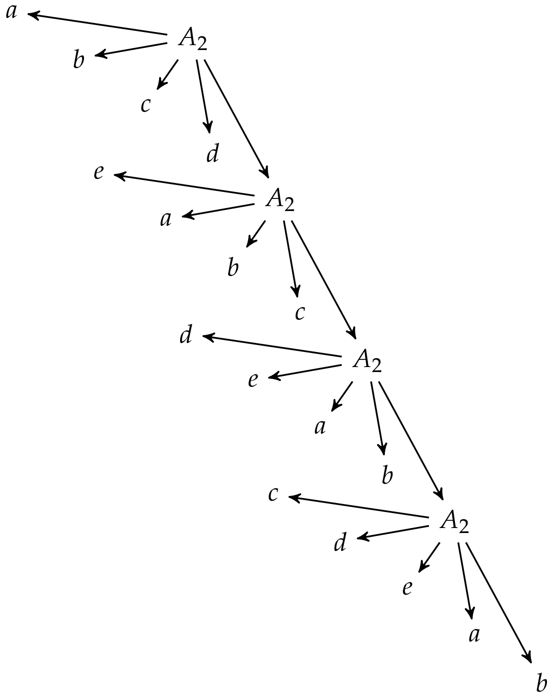


Figure 3.12.3: The right-hand side of $\mathcal{G}_2$'s start production.

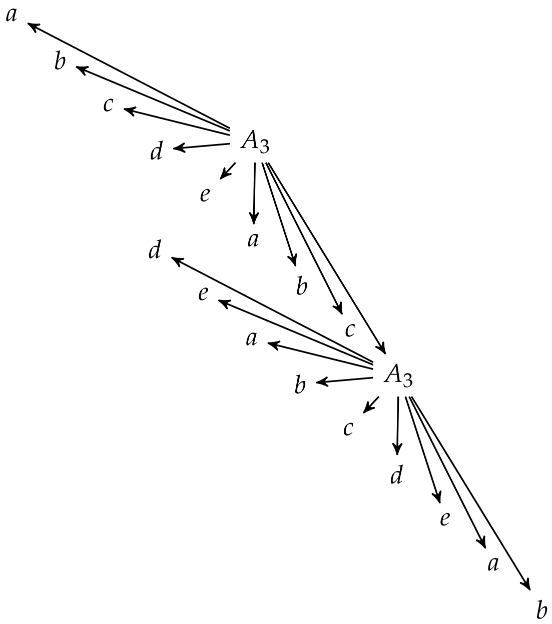


Figure 3.12.4: The right-hand side of $\mathcal{G}_3$'s start production.

Iteration	Replaced digram	New nonterminal	*cf.* Figure
1	$(f, 1, a)$	A_1	3.13.2
2	$(f, 1, b)$	A_2	3.13.3
3	$(A_1, 1, A_2)$	A_3	3.13.4
4	$(f, 1, c)$	A_4	3.13.5
5	$(f, 1, d)$	A_5	3.13.6
6	$(A_3, 1, A_4)$	A_6	3.13.7
7	$(A_6, 1, A_5)$	A_7	3.13.8
8	$(f, 1, e)$	A_8	3.13.9
9	$(A_7, 1, A_8)$	A_9	3.13.10

Table 3.3: A run of Re-pair for Trees on the tree $s_4 \in U$ with a maximal nonterminal rank of 1.

Example 36 Let us consider a run of Re-pair for Trees on the tree $s_4 \in U$ when restricting the maximal rank of a nonterminal to 1 (see Fig. 3.13.1 for a depiction of s_4). Table 3.3 shows one of several possible orders of digram replacements and the Figs. 3.13.2–3.13.10 show how the right-hand sides of the start productions evolve.

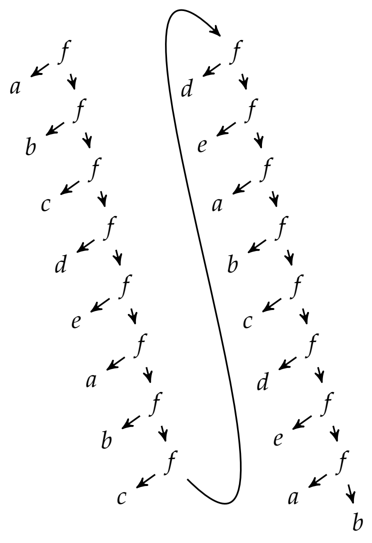


Figure 3.13.1: Right-hand side of S_0.

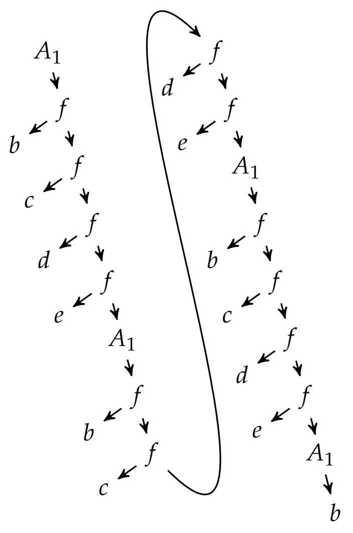


Figure 3.13.2: Right-hand side of S_1.

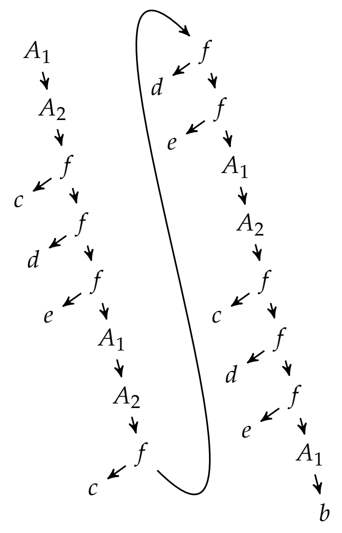


Figure 3.13.3: Right-hand side of S_2.

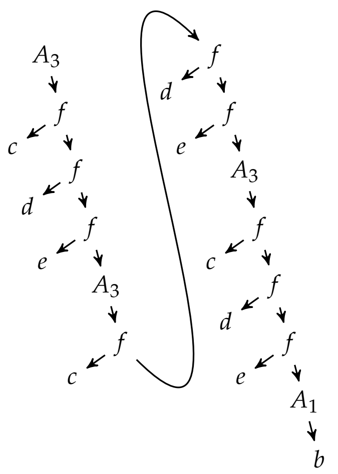


Figure 3.13.4: Right-hand side of S_3.

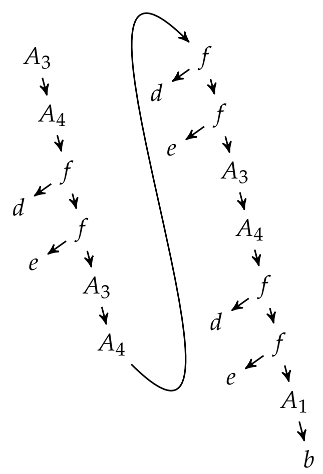


Figure 3.13.5: Right-hand side of S_4.

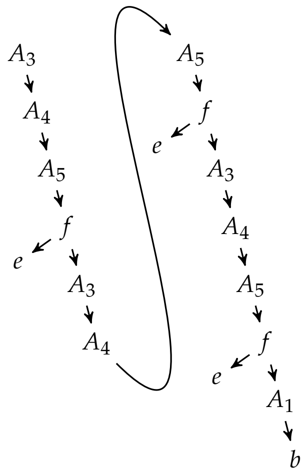


Figure 3.13.6: Right-hand side of S_5.

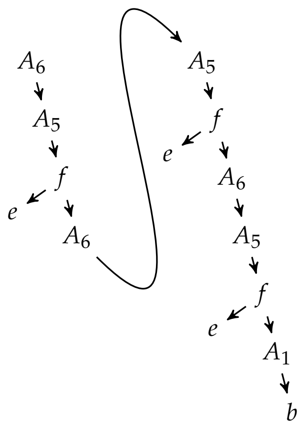


Figure 3.13.7: Right-hand side of S_6.

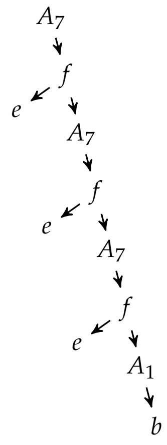


Figure 3.13.8: Right-hand side of S_7.

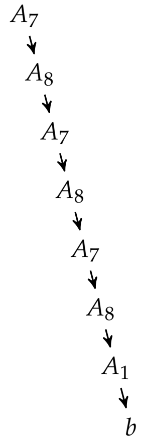


Figure 3.13.9: Right-hand side of S_8.

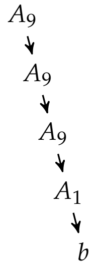


Figure 3.13.10: Right-hand side of S_9.

4

Implementation Details

We implemented a prototype of the Re-pair for Trees algorithm, named TreeRePair, running on XML documents. In the sequel, we demonstrate that it produces for any XML document tree in $\mathcal{O}(|t|)$ time a linear k-bounded SLCF tree grammar $\mathcal{G}$, where $k \in \mathbb{N}$ is a constant, $\mathsf{val}(\mathcal{G}) = t$ and $t \in T(\mathcal{F})$ is the binary representation of the input tree.

There are several reasons to restrict the maximal rank to a constant k. One of them is that only this way we are able to obtain a linear-time implementation. Another reason is that for every k-bounded linear SLCF tree grammar $\mathcal{G}$ generated by TreeRePair it can be checked in polynomial time if a given tree automaton accepts $\mathsf{val}(\mathcal{G})$ (using a result from [LM06]). Last but not least, Sect. 3.7 on page 39 showed us that for flat XML documents leading to a right-leaning binary tree it is quite promising to restrict the maximal rank. The latter reason is also supported by our experiments with different maximal ranks on our test set of XML documents.

On average, a maximal rank of 4 leads to the best compression performance (*cf.* Sect. 6.7 on page 84). Due to this fact TreeRePair generates 4-bounded linear SLCF tree grammars by default. This can be adjusted by using the `-max_rank` switch.

4.1 *Reading the Input Tree*

The XML document tree of the input file can be directly transformed into a binary $\mathcal{F}$-labeled tree $t = (\mathsf{dom}_t, \lambda_t) \in T(\mathcal{F})$.[1] The XML document is parsed by a SAX-like parser calling the functions `start-element` and `end-element` (see Figs. 4.1 and 4.2) of an object taking care of the tree construction. The latter is called *tree constructor* in the sequel.

[1] Refer to Sect. 2.4 on page 20 for an explanation of the binary tree model.

```
1  FUNCTION start-element(name)
2     if (hierarchy_stack is not empty) then
3        $i := $ index_stack.top() $+ 1$;
4        index_stack.pop();
5        index_stack.push($i$);
6
7        $v := $ hierarchy_stack.top();
8
9        if ($i = 1$) then $u := v1$
10       else  $u := v2$
11       endif
12
13       name_stack.push(name);
14    else
15       $u := \varepsilon$;
16       $\lambda_t(\varepsilon) := name^{10}$;
17    endif
18
19    $\mathsf{dom}_t := \mathsf{dom}_t \cup \{u\}$;
20
21    index_stack.push(0);
22    hierarchy_stack.push($u$);
23 ENDFUNC
```

Figure 4.1: The start-element function which is called for every start-tag.

The tree constructor uses three stacks to properly encode the SAX events. Firstly, the stack `index_stack` keeps track of the index[2] of the current element read. The stack `name-stack` stores the element types of the elements in order to be able to update the labeling function λ_t within the `end-element` function. Together with the stack `hierarchy_stack`, which is used to maintain the current sequence of parents within t, enough information stands by to encode the SAX events.

To be more precise, if the parser encounters a start-tag, it extracts the element type of the element and passes it to the tree constructor by calling the function `start-element`. If it is the first call of `start-element`, we must be dealing with the root of the document. Thus, the stack `hierarchy_stack` is empty and the `else`-part beginning in line 15 is processed. First of all, the variable u is identified with ε (and later added to the set dom_t). Afterwards, the labeling function λ_t is updated accordingly. Since, in the binary tree model, the root has no sibling nodes and since it is assumed that the input tree consists of at least two nodes, it is clear that the terminal symbol labeling the root node will have a left child but no right child (therefore the superscript 10 in line 16).

If we consider a subsequent call of `start-element`, the hierarchy stack is not empty and therefore the `if`-part is processed. Firstly, the index stack is updated in the lines 3–5 and after that the node $v \in \mathsf{dom}_t$ is retrieved from the hierarchy stack (line 7). The tree node v will be the parent of the node which is added in the following. We introduce a new node u which is later (but still in the same call of this function) added to dom_t (line 19). The node u becomes the left child of v if it represents the first child element

[2] Analogously to our definition for ranked trees: If an element is the n-th child of its parent element, then the index of this element is n.

```
FUNCTION end-element
    $i := index\_stack$.top();
    repeat $i$ times
        $v := hierarchy\_stack$.top();
        $name := name\_stack$.top();

        $l := 0$, $r := 0$;
        if ($v1 \in \mathrm{dom}_t$) then
            $l := 1$;
        endif
        if ($v2 \in \mathrm{dom}_t$) then
            $r := 1$;
        endif

        $\lambda_t(v) := name^{lr}$;

        $hierarchy\_stack$.pop();
        $name\_stack$.pop();
    endrepeat
    $index\_stack$.pop();
ENDFUNC
```

Figure 4.2: The end-element function which is called for every end-tag encountered in the input XML document.

of the element which is represented by v. In contrast, u becomes a right child if the current index i is greater than one, *i.e.*, if the element being processed is a sibling element of the element represented by v. Regarding the node u, we are unable to update the labeling function λ_t at this time since we do not know if the XML element being processed has children or sibling elements.

If an end-tag is encountered by the input parser, the function `end-element` listed in Fig. 4.2 is called. Now, the index of the current XML element is consulted in order to bubble up the sequence of parents stored by the hierarchy stack the correct number of times. Lastly, after processing the `repeat` loop, the node representing the first child element of the current XML element (the end-tag of its last child element was just read) is on top of the hierarchy stack. For every node $v \in \mathrm{dom}_t$ which is removed from the hierarchy stack within the `repeat` loop the labeling function λ_t is updated.

Example 37 Figure 4.3 considers the first calls to the functions `start-element()` and `end-element()`, respectively, when parsing the input tree from Fig. 2.5. It shows the content of the three stacks after the body of the corresponding function has been executed, where is denotes the index stack, hs denotes the hierarchy stack and ns denotes the name stack. Regarding Fig. 4.3, the element on top of the stack is always the upper element in the depiction of the corresponding stack. If there has not been assigned a label to a node, *i.e.*, the labeling function λ has not been updated accordingly yet, the node is depicted in brackets.

The binary representation of the input tree can be obtained in linear runtime since the function `start-element` and the function `end-element`, respectively, are each called only once for every node of the input tree. Furthermore, the body of the `repeat` loop of the latter function is executed once for every input node (except for the root node).

Re-pair for Trees on Multiary Trees Another way of modeling an XML document tree in a ranked way is the multiary tree model. In contrast to the binary tree model (which we described in Sect. 2.4 on page 20), this model does not encode the input tree by a binary tree but it turns the input tree into a ranked tree by introducing a terminal symbol for each element type/number of children combination which occurs in the input tree. Let us assume that an element type occurs three times and that there are three different numbers of children attach to the corresponding elements. In the multiary tree model, there are introduced three different terminal symbols.

During our investigations we also evaluated a TreeRePair version based on the multiary tree model. However, this modified version of our algorithm was outperformed by the original version in terms of compression ratio. This is due to the nature of typical XML documents. XML elements encountered in real-world XML documents often exhibit a long list of children elements. Therefore, compared to the binary tree model, a multiary tree model representation of an XML document leads to a higher number of different digrams occurring less often. This, in turn, reduces TreeRePair's ability to compress the XML document tree by the same degree as it is possible for the binary case.

Example 38 Consider for example the XML document tree from Fig. 2.5. The element of type books has five children elements of type book, *i.e.*, each of the five digrams

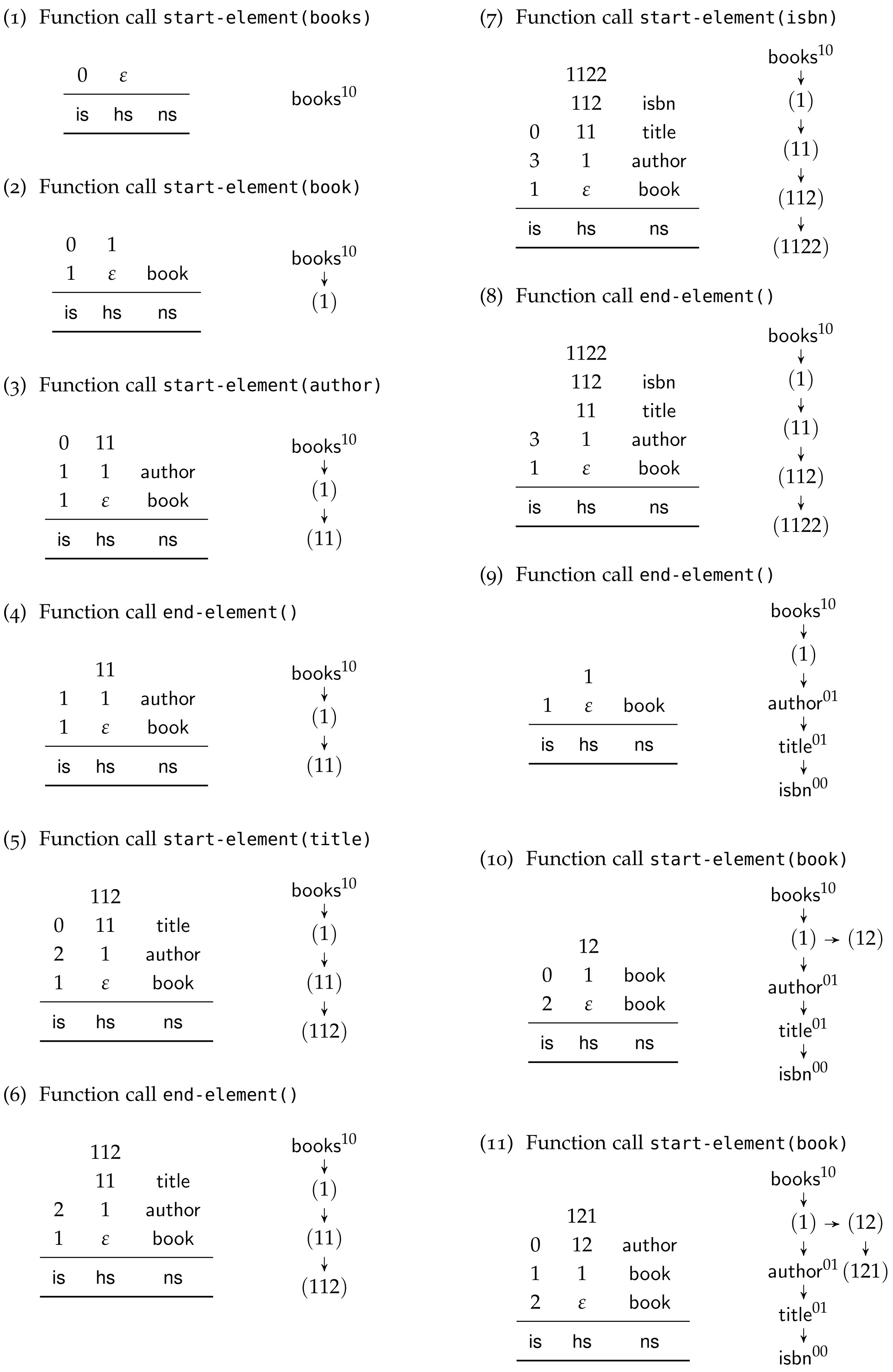
$$(\mathsf{books}, 1, \mathsf{book}), (\mathsf{books}, 2, \mathsf{book}), \ldots, (\mathsf{books}, 5, \mathsf{book})$$

occurs only once. None of these digrams is replaced by TreeRePair since a replacement is only reasonable if the corresponding digram occurs at least twice. In contrast, the binary tree model leads to two occurrences of the digram $(\mathsf{book}, 1, \mathsf{book})$ which can be replaced by a new nonterminal symbol in a run of TreeRePair (*cf.* Fig. 2.6).

4.2 *Representing the Input Tree in Memory*

In this section we show that the ranked input tree of our algorithm can be efficiently stored as a DAG in memory. This DAG representation can be made nearly transparent to the rest of the algorithm (*cf.* Sect. 4.5 on page 58).[3] Thus, by default, the tree constructor of our prototype does not only directly transform the XML document tree into a ranked representation but also infers the corresponding minimal 0-bounded SLCF tree grammar $\mathcal{G} = (N, P, S)$, *i.e.*, the minimal DAG, of the latter on the fly.

[3] Note that the DAG representation can also be circumvented by using the `-no_dag` switch. In this case the whole binary tree with all its possible redundancy is constructed in main memory.



(1) Function call `start-element(books)`

is	hs	ns
0	ε	

(2) Function call `start-element(book)`

is	hs	ns
0	1	
1	ε	book

(3) Function call `start-element(author)`

is	hs	ns
0	11	
1	1	author
1	ε	book

(4) Function call `end-element()`

is	hs	ns
	11	
1	1	author
1	ε	book

(5) Function call `start-element(title)`

is	hs	ns
	112	
0	11	title
2	1	author
1	ε	book

(6) Function call `end-element()`

is	hs	ns
	112	
	11	title
2	1	author
1	ε	book

(7) Function call `start-element(isbn)`

is	hs	ns
	1122	
	112	isbn
0	11	title
3	1	author
1	ε	book

(8) Function call `end-element()`

is	hs	ns
	1122	
	112	isbn
	11	title
3	1	author
1	ε	book

(9) Function call `end-element()`

is	hs	ns
	1	
1	ε	book

(10) Function call `start-element(book)`

is	hs	ns
	12	
0	1	book
2	ε	book

(11) Function call `start-element(book)`

is	hs	ns
	121	
0	12	author
1	1	book
2	ε	book

Figure 4.3: Content of the stacks after each call of the `start-element()` and `end-element()`, respectively, functions when parsing the tree from Fig. 2.5. In addition at each step their is a depiction of the binary tree which is constructed so far.

In [BGK03] it has been demonstrated that the representation of XML document trees based on the concept of sharing subtrees is highly efficient. Their experiments have shown that in several cases the size of the DAG was less than 10% of the uncompressed XML document tree. Therefore, the sharing of common subtrees enables us to load large XML documents trees which would have otherwise exceeded the computation resources. In addition to that it avoids time consuming swapping and the repetitive re-computation of the same results concerning subtrees that are shared.

Now, let us elaborate on how one can infer the DAG of the ranked representation $t = (\mathsf{dom}_t, \lambda_t) \in T(\mathcal{F})$ of the XML document tree. The tree constructor must check for every node which is removed from the hierarchy stack in the `end-element` function if the subtree rooted at this node can be shared. This can be accomplished by calling the function `share-subtree` listed in Fig. 4.4. To gain insight into this function let us assume that we want to check if the subtree $t' \in T(\mathcal{F})$ rooted at a node $v \in \mathsf{dom}_t$ can be shared. If we already encountered an exact copy of t' while reading the input tree, all subtrees of t' must have been shared before. Thus, the tree t' must be of depth 1 and all children nodes must be labeled by nonterminals of the DAG grammar $\mathcal{G}$. Therefore, it is only necessary to compare the labels of the root of t' and its direct children with those of all subtrees encountered until now. This can be done in constant time with the help of a hash table.

Now, let us assume that we have processed an exact copy of t' earlier, *i.e.*, t' can be shared. Thus, the condition in line 3 is evaluated to `true` and the hash table `subtrees_ht` contains t'. Hence, the `else`-part beginning in line 6 is processed. If there already exists a nonterminal $B \in N$ with right-hand side t' then we set $A := B$. We can check this in $\mathcal{O}(1)$ time because with each entry of the hash table `subtrees_ht` we can store a pointer to the corresponding production. Otherwise, *i.e.*, if there exists no $(B \to t'') \in P$ with $t' = t''$, we introduce a new nonterminal $A \in \mathcal{N}_0 \setminus N$ with right-hand side t' and replace the first occurrence u of the subtree t' by A. There can be only one earlier occurrence of the subtree t' since otherwise we would already have inserted a corresponding production. Furthermore, we can guarantee constant time access to u because with each entry in the hash table `subtrees_ht` we can store a pointer to the corresponding first occurrence. Finally, we add the subtree rooted at the node $\mathsf{parent}(u)$ to the hash table if all of its subtrees are shared. We do not need to insert the subtree rooted at the node $\mathsf{parent}(v)$ since we will process $\mathsf{parent}(v)$ in a later step (since we are traversing the input tree in postorder). In contrast, if t' was not encountered until now, we add it to the hash table `subtrees_ht` (line 5) in order to be able to share possible later occurrences of it.

Initially, *i.e.*, after reading the input tree, all shared subtrees are of depth 1. In order to reduce the number of nonterminals of

```
FUNCTION share-subtree($v$)
   let $t'$ be the subtree rooted at $v$;
   if ($\forall 1 \leq i \leq \mathsf{rank}(\lambda_t(v)) : \lambda_t(vi) \in \mathcal{N}_0$) then
      if (subtrees_ht does not contain $t'$) then
         insert $t'$ into subtrees_ht;
      else
         if ($\exists B \in \mathcal{N}_0 : (B \to t') \in P$) then
            $A := B$;
         else
            choose nonterminal $A \in \mathcal{N}_0 \setminus N$;
            $N := N \cup \{A\}$; $P := P \cup \{(A \to t')\}$;
            let $u$ be the node at which the first
                     occurrence of $t'$ is rooted;
            replace subtree rooted at $u$ by $A$;

            $w := \mathsf{parent}(u)$;
            if ($\forall 1 \leq i \leq \mathsf{rank}(\lambda_t(w)) : \lambda_t(wi) \in \mathcal{N}_0$) then
               let $t''$ be the subtree rooted at $w$;
               insert $t''$ into subtrees_ht;
            endif
         endif

         replace subtree rooted at $v$ by $A$;
      endif
   endif
ENDFUNC
```

Figure 4.4: The function share-subtree which checks for the subtree rooted at the node $v \in \mathsf{dom}_t$ if it can be shared. If this is the case then the sharing is performed.

the DAG grammar (without increasing the number of total edges) all productions referenced only once are eliminated. All in all, the inferring of the DAG grammar needs linear time and can be conveniently combined with the step of transforming the input tree into a ranked tree.

4.3 *Utilized Data Structures*

The data structures we use in our implementation are similar to those used in [LM00]. In order to be able to focus on the essentials, we do not pay attention to the fact that, internally, the input tree is represented by a DAG.

Let us assume that the binary input tree $t = (\mathsf{dom}_t, \lambda_t) \in T(\mathcal{F})$ has been generated by our implementation after reading a corresponding XML document tree. Hence, the tree t is the ranked representation of the latter. In main memory, every node $v \in \mathsf{dom}_t$ is represented by an object exhibiting several pointers. These allow constant time access to the parent and all children of the node v and to the possible next and previous occurrences of the digram $\alpha = \big(\lambda_t(v), i, \lambda_t(vi)\big)$, where $i \in \{1, 2, \ldots, \mathsf{rank}(\lambda_t(v))\}$. The pointers to the next and previous occurrences of α form a doubly linked list of all the occurrences in $\mathsf{occ}_t(\alpha)$. We call this type of list an *occurrences list (of α)* in the sequel.[4] The specific order of the occurrences in an occurrences list is not relevant.

Every digram is represented by a special object. It exhibits two

[4] During our investigations we also implemented a TreeRePair version avoiding these doubly linked lists of occurrences. Instead, for every digram, we used a hashed set storing pointers to all occurrences. However, this version had no benefits compared to the doubly linked list approach but lead to slightly longer runtimes. Considering the memory usage, in some cases it achieved better results while in others a substantial increase was noticed.

pointers which reference the first and the last element of the corresponding occurrences list. Let us consider a digram $\alpha \in \Pi$ with $|\mathsf{occ}_t(\alpha)| = m$, where $m < \lfloor\sqrt{n}\rfloor$ and $n = |t|$. Then the corresponding object exhibits two more pointers which point to the next and previous, respectively, digram $\beta \in \Pi$ with $|\mathsf{occ}_t(\beta)| = m$. These pointers form a doubly linked list of all digrams occurring m times. We denote this type of list the *m-th digram list*. In contrast, all digrams $\gamma \in \Pi$ with $|\mathsf{occ}_t(\gamma)| \geq \lfloor\sqrt{n}\rfloor$ are organized in one doubly linked list which is called the *top digram list*.

These doubly linked lists of digrams are again referenced by a *digram priority queue*. This queue consists of $\lfloor\sqrt{n}\rfloor$ entries. The i-th entry stores a pointer to the head of the i-th digram list, where $1 \leq i < \lfloor\sqrt{n}\rfloor$. The $\lfloor\sqrt{n}\rfloor$-th entry references the head of the top digram list. Refer to Sect. 4.4 on page 53 for an explanation on why we designed the digram lists and priority queue as described above. Lastly, there is a *digram hash table* storing pointers to all occurring digrams. It allows constant time access to all digrams and therefore constant time access to the first occurrence of each digram.

Let us consider the following example to gain an insight into the utilized data structures.

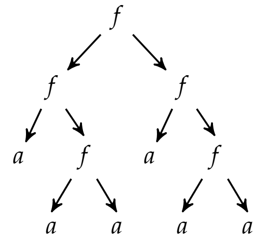


Figure 4.5: The tree $t \in T(\mathcal{F})$ modeled by the node objects from Fig. 4.8.

Example 39 Let us assume that the tree $t = (\mathsf{dom}_t, \lambda_t) \in T(\mathcal{F})$ shown in Fig. 4.5 has been generated by our implementation after reading a corresponding XML document tree. Then Fig. 4.8 shows a simplified depiction of the data structures used to efficiently replace the digrams in the replacement step. All non-null pointers are represented by arrows starting in a filled circle and ending in an empty circle. A filled circle without an outgoing arrow denotes a null pointer.

With respect to Fig. 4.8, there is a total of 11 node objects representing tree nodes labeled by the two symbols $f \in \mathcal{F}_2$ and $a \in \mathcal{F}_0$. An instance of a tree node $v \in \mathsf{dom}_t$ is represented by a tabular box as it is shown in Fig. 4.6. Unlike depicted, in our implementation a symbol is not directly stored within the node structure but for every unique symbol there is an object which is referenced by the corresponding nodes. The upper left empty circle of the box represents the memory address of the tree node instance. Thus, every arrow representing a pointer to the latter will end in this empty circle.

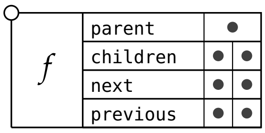


Figure 4.6: A graphical representation of an object representing a tree node labeled by $f \in \mathcal{F}$.

The filled circle in the first row of the tabular box represents the pointer to the possible parent node $\mathsf{parent}(v)$. The pointer to the i-th child vi of the node v is depicted by an arrow starting at the filled circle in the i-th column of the `children` row, where $i \in \{1, 2, \ldots, \mathsf{rank}(\lambda_t(v))\}$. Analogously, a pointer to a possible next (previous) occurrence of the digram $\alpha = \big(\lambda_t(v), i, \lambda_t(vi)\big)$ is represented by a filled circle in the i-th column of the row labeled by `next` (`previous`, respectively), where $i \in \{1, 2, \ldots, \mathsf{rank}(\lambda_t(v))\}$.

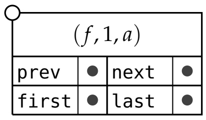


Figure 4.7: A graphical representation of a digram $(f, 1, a) \in \Pi$.

Each digram $(f, 1, f)$, $(f, 2, a)$, $(f, 2, f)$ and $(f, 1, a)$ is represented by a tabular box (see Fig. 4.7). Again, unlike depicted, in our implementation a symbol is not directly stored within the digram structure but the latter contains two pointers to the objects representing a and b. The first and the last element of the occurrences list of the digram α are referenced by the `first` and `last` pointers of the object representing the digram α. The pointers `prev` (previous) and `next` are part of the $|\mathsf{occ}_t(\alpha)|$-th digram list if $|\mathsf{occ}_t(\alpha)| < \lfloor\sqrt{n}\rfloor$ and $n = |t|$. Otherwise they belong to the top digram list.

The digram $(f, 1, f)$ forms a trivial doubly linked list, namely, the 1st digram list. The latter is referenced by the entry 1 of the priority queue. The digram $(f, 1, a)$ forms the (trivial) top digram list which is referenced by the entry 3 of the priority queue. In contrast, the digrams $(f, 2, a)$ and $(f, 2, f)$ each occur twice and therefore point to each other with their `next` and `previous` pointers, respectively. The first element of the resulting 2nd digram list is referenced by the entry 2 of the priority queue. The digram hash table stores the pointers to all four occurring digrams.

4.4 *Complexity of the TreeRePair Algorithm*

Theorem 40 *For any given input tree with n edges TreeRePair produces in time $\mathcal{O}(|t|)$ a k-bounded linear SLCF tree grammar $\mathcal{G}$, where $k \in \mathbb{N}$ is a constant, $\mathsf{val}(\mathcal{G}) = t$ and $t \in T(\mathcal{F})$ is the binary representation of the input tree.*

It is straightforward to come up with a linear time implementation of the pruning step of the Re-pair for Trees algorithm (*cf.* Sect. 3.3 on page 28). Therefore, we just want to investigate the complexity of the replacement step which was described in Sect. 3.2 on page 27.

With every replacement of a digram occurrence one edge of the input tree is absorbed. Therefore, a run of TreeRePair can consist of at most $n - 1$ iterations, where n is the size of the input tree. Each replacement of an occurrence can be accomplished in $\mathcal{O}(1)$ time since at most k children need to be reassigned — in our implementation, the reassignment of a child node is just a matter of updating two pointers.[5] For every production which is introduced during a run of our algorithm it holds that the right-hand side t is of size $|t| < 2 + k$, *i.e.*, it can be constructed in constant time.

However, to show that the replacement step can be performed in linear time two more aspects need to be considered. Imagine that we are in the i-th iteration of our algorithm (and $\mathcal{G}_{i-1}$ is the current grammar). Let $t \in T(\mathcal{F} \cup \mathcal{N})$ be the right-hand side of $\mathcal{G}_{i-1}$'s start production.

(1) *Updating the sets of non-overlapping occurrences*

In every iteration of our algorithm we need to know the number of occurrences of each digram. Only in that case we are

[5] As already mentioned at the beginning of this chapter on page 45: The maximal rank of a nonterminal of a grammar generated by TreeRePair is $k \in \mathbb{N}$. The constant k can be specified by a command line switch.

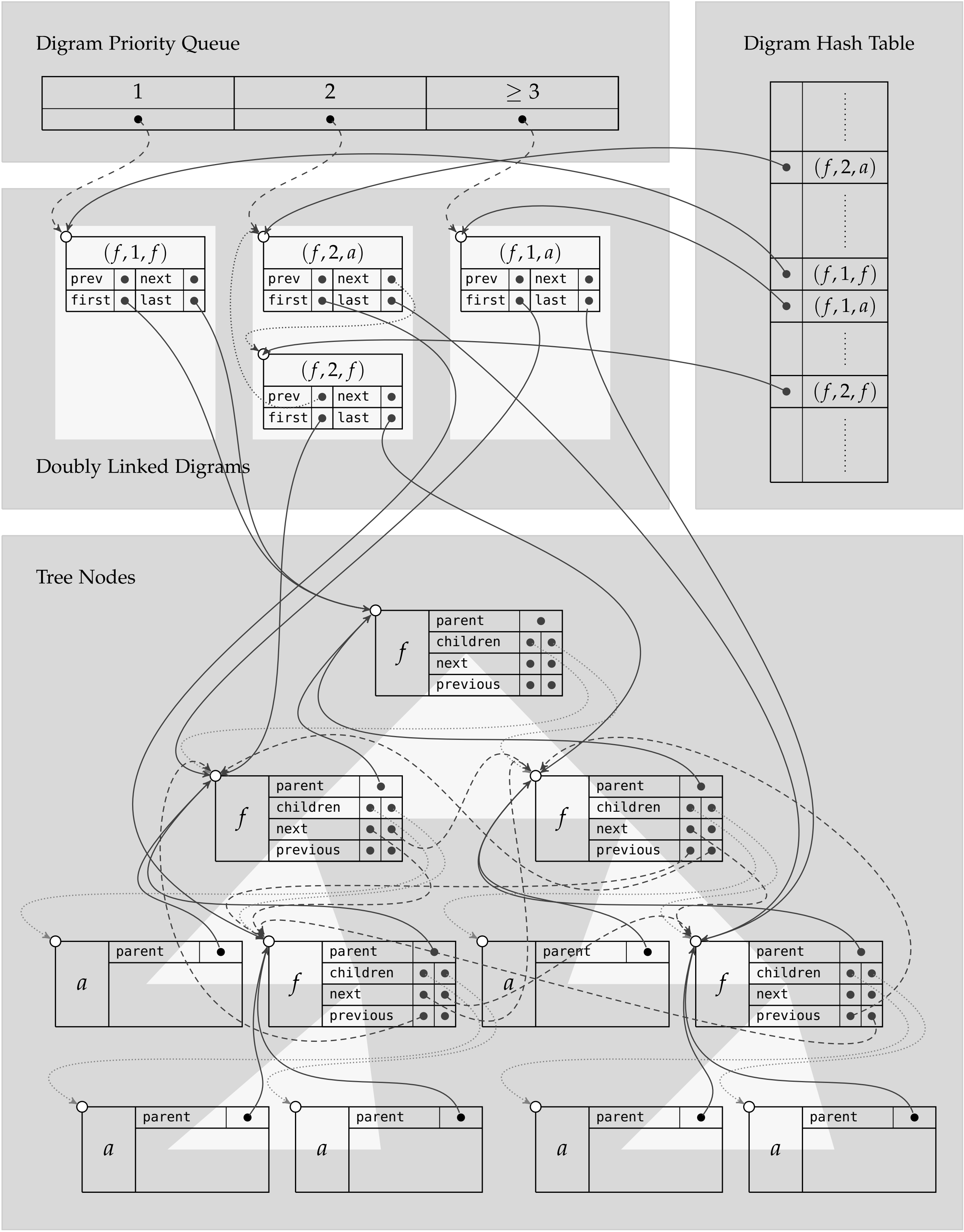


Figure 4.8: A simplified depiction of a part of the data structures used by our implementation.

```
FUNCTION retrieve-all-occs($t$)
    $v := \varepsilon$;
    while (true) do
        $v$ := next_in_postorder($t$, $v$);
        if ($v \neq \varepsilon$) then
            $\alpha := (\lambda_t(\mathsf{parent}(v)), \mathsf{index}(v), \lambda_t(v))$;
            if ($v \notin \mathsf{occ}_t(\alpha)$) then
                $\mathsf{occ}_t(\alpha) := \mathsf{occ}_t(\alpha) \cup \{\mathsf{parent}(v)\}$
            endif
        else
            return;
        endif
    endwhile
ENDFUNC
```

Figure 4.9: The function `retrieve-all-occs` which is used to construct the set $\mathsf{occ}_t(\alpha)$ for every digram $\alpha \in \Pi$ occurring in the tree $t \in T(\mathcal{F} \cup \mathcal{N})$. It uses the function `next-in-postorder` listed in Fig. 3.2.

able to determine the most frequent digram. In addition, for replacing the digram $\mathsf{max}_k(t)$, we need to know $\mathsf{occ}_t(\mathsf{max}_k(t))$. How can we compute the set $\mathsf{occ}_t(\alpha)$ for every digram $\alpha \in \Pi$ without traversing the whole right-hand side of the current start production in each iteration?

(2) *Retrieving the most frequent digram*

Let us assume that there is an up to date set $\mathsf{occ}_t(\alpha)$ available for every $\alpha \in \Pi$ occurring in t (in the form of occurrences lists). How do we determine the most frequent digram in constant time?

In the following we consider each of the above aspects in detail.

4.4.1 *Updating the Sets of Non-overlapping Occurrences*

Let the binary tree $t = (\mathsf{dom}_t, \lambda_t) \in T(\mathcal{F})$ be our input tree. At the beginning of the replacement step the set $\mathsf{occ}_t(\alpha)$ for every digram $\alpha \in \Pi$ occurring in t is initially constructed. This is done by parsing the tree t in a similar way as it is done in the function `retrieve-occurrences` which is listed in Fig. 3.3. However, during the traversal not only one digram is considered but for every encountered digram $\alpha \in \Pi$ the set $\mathsf{occ}_t(\alpha)$ is constructed. Figure 4.9 shows a possible function which accomplishes this task.

Therefore, in the first iteration of our computation we have up to date sets of non-overlapping occurrences at hand. However, we cannot afford to redo this traversal in every subsequent iteration. In this case we would not be able to achieve a linear runtime of our algorithm.

Fortunately, there is another way of keeping track of the sets of non-overlapping occurrences. It relies on the fact that every replacement of an digram occurrence v only involves those occurrences in the neighborhood of v which overlap with v.

Example 41 Let us consider the tree $t' = (\mathsf{dom}_{t'}, \lambda_{t'}) \in T(\mathcal{F})$ which is depicted in Fig. 4.10. The occurrences which would be

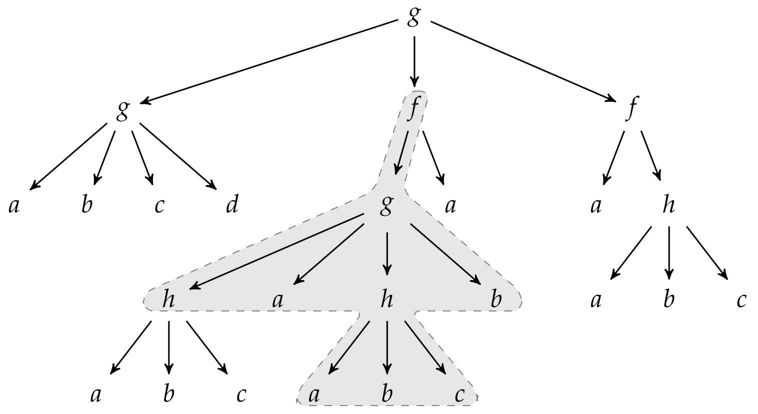


Figure 4.10: The tree $t' \in T(\mathcal{F})$. All occurrences which would be absorbed by the replacement are highlighted.

```
FUNCTION remove-absorbed-occs($t, v, j$)
    if ($v \neq \varepsilon$) then
        $\alpha := \big(\lambda_t(\mathsf{parent}(v)), \mathsf{index}(v), \lambda_t(v)\big)$;
        $\mathsf{occ}'_t(\alpha) := \mathsf{occ}'_t(\alpha) \setminus \{\mathsf{parent}(v)\}$;
    endif

    for ($l \in \{1,2,\ldots,\mathsf{rank}(\lambda_t(v))\}$) do
        $\alpha := \big(\lambda_t(v), l, \lambda_t(vl)\big)$;
        $\mathsf{occ}'_t(\alpha) := \mathsf{occ}'_t(\alpha) \setminus \{v\}$;
    endfor

    for ($l \in \{1,2,\ldots,\mathsf{rank}(\lambda_t(vj))\}$) do
        $\alpha := \big(\lambda_t(vj), l, \lambda_t(vjl)\big)$;
        $\mathsf{occ}'_t(\alpha) := \mathsf{occ}'_t(\alpha) \setminus \{vj\}$;
    endfor
ENDFUNC
```

Figure 4.11: Listing of the function `remove-absorbed-occs` which removes all absorbed occurrences from the occ'_t sets.

absorbed by the replacement of the occurrence $2 \in \mathsf{dom}_{t'}$ of the digram $(f, 1, g)$ are highlighted.

For every digram $\alpha \in \Pi$ we set $\mathsf{occ}'_t(\alpha) := \mathsf{occ}_t(\alpha)$ and base all upcoming computations on the set $\mathsf{occ}'_t(\alpha)$. In particular we use them to determine the most frequent digram in each iteration.

Let us consider the i-th iteration of a run $\mathcal{G}_0, \mathcal{G}_1, \ldots, \mathcal{G}_h$ of Re-pair for Trees on the input tree $t \in T(\mathcal{F})$, where $h \in \mathbb{N}$ and $i \in \{1, 2, \ldots, h\}$. Then $\mathcal{G}_{i-1} = (N_{i-1}, P_{i-1}, S_{i-1})$ is the current grammar. Let $t_{i-1} \in T(\mathcal{F})$ be the right-hand side of S_{i-1}. Let us assume that an up to date set $\mathsf{occ}'_{t_{i-1}}(\beta)$ for every $\beta \in \Pi$ which is occurring in t_{i-1} is at hand. Further, let us assume that $\mathsf{max}(t_{i-1}) = (a, j, b) =: \alpha$ and let $v \in \mathsf{occ}'_{t_{i-1}}(\alpha)$.

Before the actual replacement of the occurrence v we make use of the function `remove-absorbed-occs` listed in Fig. 4.11. The function call `remove-absorbed-occs(`t_{i-1}, v, j`)` removes all occurrences which will be absorbed by the upcoming replacement from the sets $\mathsf{occ}'_{t_{i-1}}$. After the replacement of v by a new node u with $\lambda_{t_i}(u) = A_i \in \mathcal{N}$ we call the function `add-new-occs` (which is

```
FUNCTION add-new-occs($t, u$)
    if ($u \neq \varepsilon$) then
        $\alpha := \big(\lambda_t(\mathsf{parent}(u)), \mathsf{index}(u), \lambda_t(u)\big)$;
        $\mathsf{occ}_t(\alpha)' := \mathsf{occ}'_t(\alpha) \cup \{\mathsf{parent}(u)\}$;
    endif

    for ($l \in \{1,2,\ldots,\mathsf{rank}(\lambda_t(u))\}$) do
        $\alpha := \big(\lambda_t(u), l, \lambda_t(ul)\big)$;
        $\mathsf{occ}'_t(\alpha) := \mathsf{occ}'_t(\alpha) \cup \{u\}$;
    endfor
ENDFUNC
```

Figure 4.12: Listing of the function add-new-occs which adds all newly created occurrences to the occ'_t sets.

listed Fig. 4.12) and pass the tree t_{i-1} and the node u. The function add-new-occs adds all new occurrences which arose by the introduction of u to the sets of non-overlapping occurrences. Finally, after all occurrences from $\mathsf{occ}'_{t_{i-1}}(\alpha)$ have been replaced, we set $\mathsf{occ}'_{t_i}(\beta) := \mathsf{occ}'_{t_{i-1}}(\beta)$ for all $\beta \in \Pi$ occurring in t_i.

Let $\alpha \in \Pi$ be a digram occurring in t_i. The above computed set $\mathsf{occ}'_{t_i}(\alpha)$ may not be equal to the actual set $\mathsf{occ}_{t_i}(\alpha)$ as it would be constructed by a complete postorder traversal of t_i using the function retrieve-occurrences from Fig. 3.3.

Example 42 Consider, for instance, the tree $t'' \in T(\mathcal{F})$ depicted in Fig. 4.13. Let $\alpha = (f, 2, f)$. In the first iteration of our algorithm, we would obtain $\mathsf{occ}'_{t''}(\alpha) := \mathsf{occ}_{t''}(\alpha) = \{2\}$. Now, let us assume that we want to replace the digram $(f, 1, c)$ (we could easily enlarge t'' such that $(f, 1, c)$ is the most frequent digram and still show the same). After performing this replacement and especially after calling the functions remove-absorbed-occs and add-new-occs we would have $\mathsf{occ}'_{t''}(\alpha) = \emptyset$. However, a postorder traversal of the updated tree t'' would result in $\mathsf{occ}_{t''}(\alpha) = \{\varepsilon\}$.

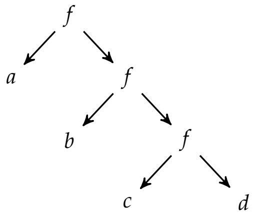


Figure 4.13: Tree $t'' \in T(\mathcal{F})$ consisting of nodes labeled by the terminal symbols $a, b, c, d, f \in \mathcal{F}$. We have to deal with three overlapping occurrences of the digram $(f, 2, f)$.

All in all, the update of the sets of non-overlapping occurrences consumes constant time per occurrence replacement. At most $2k + 1$ occurrences need to be removed by the remove-absorbed-occs function and at most $k + 1$ occurrences need to be added by the function add-new-occs. An occurrence v of a digram α can be removed from the occurrences list of α in constant time by setting the next and previous pointers of the corresponding node object to null. In addition, if v is the first (last) occurrence in the occurrence list of α the first (last) pointer of the object representing the digram α needs to be updated. This can also be accomplished in constant time by using the digram hash table. Analogously, an occurrence can be added to an occurrences list in $\mathcal{O}(1)$ time.

4.4.2 Retrieving the Most Frequent Digram

We now investigate the time needed to obtain the most frequent digram in an iteration of our algorithm. First of all, let us state the following fact: Let $m \in \mathbb{N} \cup \{\infty\}$ and let $\mathcal{G}_0, \mathcal{G}_1, \ldots, \mathcal{G}_n$ be a run of Re-pair for Trees, where $n \in \mathbb{N}_{>0}$, $\mathcal{G}_i = (N_i, P_i, S_i)$ and $(S_i \to t_i) \in P_i$ for every $i \in \{0, 1, \ldots, n\}$. Then

$$|\mathsf{occ}_{t_i}(\mathsf{max}_m(t_i))| \geq |\mathsf{occ}_{t_{i+1}}(\mathsf{max}_m(t_{i+1}))|$$

holds for every $i \in \{0, 1, \ldots, n-1\}$.[6] For every digram $\alpha \in \Pi$ occurring in t_i it holds that $|\mathsf{occ}_{t_i}(\alpha)| \geq |\mathsf{occ}_{t_{i+1}}(\alpha)|$ and for every digram $\beta \in \Pi$ which was introduced in $\mathcal{G}_{i+1}$ it holds that $|\mathsf{occ}_{t_{i+1}}(\beta)| \leq |\mathsf{occ}_{t_i}(\mathsf{max}_m(t_i))|$, where $i \in \{0, 1, \ldots, n-1\}$.

It is easy to see that, if the top digram list is empty, we can obtain the most frequent digram in constant time. We just need to walk down the remaining $\lfloor \sqrt{n} \rfloor - 1$ digram lists and choose the first element of the first non-empty list. In every iteration, after we have determined the most frequent digram, we remember the first non-empty digram list in order to save ourself the needless and time-consuming rechecking of the empty digram lists.

Now, let us assume that the top digram list, *i.e.*, the doubly linked list of all digrams occurring at least $\lfloor \sqrt{n} \rfloor$ times, is not empty. We need to scan all elements in it since the digrams contained are not ordered by their frequency. There can be roughly at most $\sqrt{n}$ digrams in the top digram list. Therefore, we need roughly $\mathcal{O}(\sqrt{n})$ time to retrieve the most frequent digram. However, by the replacement of this digram at least $\lfloor \sqrt{n} \rfloor$ edges are absorbed. It is easy to see that, all in all, obtaining the most frequent digram needs constant time on average.

In a run of TreeRePair we can replace at most $n-1$ digram occurrences and, as shown before, the replacement of each occurrence, the update of the sets of non-overlapping occurrences and the determination of the most frequent pair can be accomplished in constant time per occurrence replacement. Thus, the whole replacement step can be completed in linear time.

4.5 *Impact of the DAG Representation*

In the preceding section, dealing with the complexity of our implementation of the Re-pair for Trees algorithm, we did not pay attention to the underlying DAG representation of the input tree. This enabled us to concentrate on the essentials. Nevertheless, we have to clarify the impact of this representation, particularly concerning the compression performance and the runtime of our implementation, since TreeRePair uses it by default. Only by starting TreeRePair with the `-no_dag` switch it forgos the DAG representation and loads the whole input tree into main memory.

Definition 43 Let $\mathcal{G} = (N, P, S)$ be a 0-bounded SLCF tree grammar. We assume without loss of generality that for every $B \in N$ it holds that $B \leadsto_{\mathcal{G}}^{*} S$. Let $(A \to t) \in P$, $t = (\mathsf{dom}_t, \lambda_t) \in T(\mathcal{F})$ and $v \in \mathsf{dom}_t$. We define the function `unfold` using the algorithm listed in Fig. 4.14.

[6] Intuitively, we define

$$|\mathsf{occ}_{t_n}(\mathsf{max}_m(t_n))| = 0$$

if $\mathsf{max}_m(t_n) = \mathsf{undefined}$.

```
FUNCTION unfold($\mathcal{G}$,$t$,$v$)
    let $\mathcal{G} = (N,P,S)$ and $A \to t \in P$;
    if $\mathsf{ref}_{\mathcal{G}}(A) \neq \emptyset$ then
        $M := \emptyset$;
        for each $(t',v') \in \mathsf{ref}_{\mathcal{G}}(A)$ do
            $M := M \cup \{uv \mid u \in$ unfold($\mathcal{G}$,$t'$,$v'$)$\}$;
        endfor
    else
        $M := \{v\}$;
    endif
    return $M$;
ENDFUNC
```

Figure 4.14: The algorithm which computes unfold($\mathcal{G}$,t,v), where we have $t \in T(\mathcal{F} \cup \mathcal{N})$ and $v \in \mathsf{dom}_t$.

```
FUNCTION retrieve-all-occs-dag($\bar{t}$)
    $v := \varepsilon$;
    while (true) do
        $v :=$ next_in_postorder($\bar{t}$, $v$);
        if ($v \neq \varepsilon$) then
            if ($\lambda_{\bar{t}}(v) \notin \mathcal{N}$) then
                $\alpha := (\lambda_{\bar{t}}(\mathsf{parent}(v)), \mathsf{index}(v), \lambda_{\bar{t}}(v))$;
                if ($v \notin \mathsf{occ}'_{\bar{t}}(\alpha)$) then
                    $\mathsf{occ}'_{\bar{t}}(\alpha) := \mathsf{occ}'_{\bar{t}}(\alpha) \cup \{\mathsf{parent}(v)\}$;
                endif
            else
                let $\bar{t}'$ be the right-hand side of $\lambda_{\bar{t}}(v)$;
                if ($\lambda_{\bar{t}'}(\varepsilon) \neq \lambda_{\bar{t}}(\mathsf{parent}(v))$) then
                    $\alpha := (\lambda_{\bar{t}}(\mathsf{parent}(v)), \mathsf{index}(v), \lambda_{\bar{t}'}(\varepsilon))$;
                    $\mathsf{occ}'_{\bar{t}}(\alpha) := \mathsf{occ}'_{\bar{t}}(\alpha) \cup \{\mathsf{parent}(v)\}$;
                endif
            endif
        else
            return;
        endif
    endwhile
ENDFUNC
```

Figure 4.15: The function retrieve-all-occs listed in Fig. 4.9 adapted for the DAG case. For every $\alpha \in \Pi$ the set $\mathsf{occ}_{\bar{t}}(\alpha)$ is initially set to $\emptyset$.

It holds that unfold($\mathcal{G}$,t,v) $\subseteq \mathsf{dom}_{\mathsf{val}(\mathcal{G})}$ and it also holds that

$$\bigcup_{\substack{(A \to t) \in P, \\ v \in \mathsf{dom}_t}} \texttt{unfold}(\mathcal{G},t,v) = \mathsf{dom}_{\mathsf{val}(\mathcal{G})} \ .$$

Let us consider a run $\mathcal{G}_0, \mathcal{G}_1, \ldots, \mathcal{G}_h$ of TreeRePair, where we have $\mathcal{G}_i = (N_i, P_i, S_i)$, $(S_i \to t_i) \in P_i$, $h \in \mathbb{N}$ and $i \in \{0, 1, \ldots, h\}$. Then, in our implementation, t_i is represented by a 0-bounded (linear) SLCF tree grammar $\overline{\mathcal{G}}_i = (\overline{N}_i, \overline{P}_i, \overline{S}_i)$, *i.e.*, we have $\mathsf{val}(\overline{\mathcal{G}}_i) = t_i$, by default.

4.5.1 Constructing the Sets of Non-overlapping Occurrences

In the first iteration of TreeRePair we need to construct the set $\mathsf{occ}_{t_0}(\alpha)$ for every digram $\alpha \in \Pi$ occurring in t_0. Our first try to accomplish this could be a postorder traversal of all the right-hand sides of $\overline{P}_0$'s productions using the function retrieve-all-occs listed in Fig. 4.9 on page 55. However, when traversing the right-hand sides of the DAG grammar $\overline{\mathcal{G}}_0$ individually, we do not consider occurrences spanning two productions of the DAG.

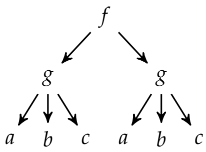


Figure 4.16: The tree $t \in T(\mathcal{F})$ which can be represented by a DAG grammar with productions $(S \to f(A,A))$ and $(A \to g(a,b,c))$.

Example 44 Consider the DAG grammar $\mathcal{G} = (N,P,S)$, where $N = \{S,A\}$ and P contains the productions $(S \to f(A,A))$ and $(A \to g(a,b,c))$. It is a compressed representation of the tree $t \in T(\mathcal{F})$ depicted in Fig. 4.16. If we would use the function `retrieve-all-occs` to determine all digram occurrences in the right-hand sides of P's productions, we would not capture the node $\varepsilon \in \mathsf{dom}_t$ which is an occurrence for both the digram $(f,1,g)$ and the digram $(f,2,g)$.

As we have seen, it is necessary to modify the `retrieve-all-occs` function slightly to also take occurrences spanning two productions into account. We use the algorithm listed in Fig. 4.15 to obtain the set $\mathsf{occ}'_{\overline{t}}(\alpha)$ for every right-hand side $\overline{t}$ of $\overline{\mathcal{G}}_0$'s productions and every digram $\alpha \in \Pi$ occurring in t_0. After that, we set

$$\mathsf{occ}'_{t_0}(\alpha) := \bigcup_{\substack{(\overline{A} \to \overline{t}) \in \overline{P}_0, \\ v \in \mathsf{occ}'_{\overline{t}}(\alpha)}} \mathtt{unfold}(\overline{\mathcal{G}}, \overline{t}, v) \enspace .$$

We test in line 13 of the `retrieve-all-occs` function if α has equal parent and child symbols. If this proves to be true, we do not add the corresponding occurrence to $\mathsf{occ}'_{\overline{t}}(\alpha)$, *i.e.*, we do not consider occurrences of a digram with equal parent and child symbols spanning two productions of the DAG. If we would do so, we would possibly register overlapping occurrences and run into problems during a later replacement of α. Consider the following example:

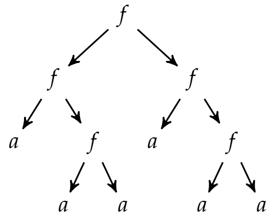


Figure 4.17: The tree $t' \in T(\mathcal{F})$ which can be represented by a DAG grammar with productions $A_1 \to f(A_2,A_2)$ and $A_2 \to f(a,f(a,a))$.

Example 45 Consider the DAG grammar $\mathcal{G} = (N,P,A_1)$ given by the productions $(A_i \to t_i) \in P$, where $i \in \{1,2\}$, $t_1 = f(A_2,A_2)$ and $t_2 = f(a,f(a,a))$. It is a compressed representation of the tree $t' \in T(\mathcal{F})$ depicted in Fig. 4.17. We use the algorithm from Fig. 4.15 to obtain the sets $\mathsf{occ}'_{t_i}(\alpha)$ for $i \in \{1,2\}$ and every digram $\alpha \in \Pi$ occurring t'. Let us assume that we omit the check in line 13, *i.e.*, we also consider occurrences of digrams with equal parent and child symbols spanning two productions. The union

$$\bigcup_{\substack{i \in \{1,2\}, \\ v \in \mathsf{occ}'_{t_i}((f,2,f))}} \mathtt{unfold}(\mathcal{G}, t_i, v) = \{\varepsilon, 1, 2\}$$

contains the overlapping occurrences ε and 2 of the digram $(f,2,f)$.

The precaution from line 13 leads sometimes to situations in which we replace fewer occurrences of a digram with equal parent and child symbols as we would replace when not using the DAG representation.

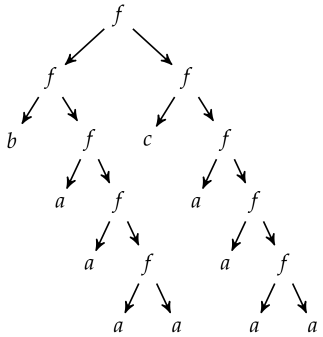


Figure 4.18: Tree $t'' \in T(\mathcal{F})$ with seven overlapping occurrences of the digram $(f,2,f)$.

Example 46 Consider the tree $t'' \in T(\mathcal{F})$ from Fig. 4.18 which can be represented by the DAG grammar consisting of the two productions $(S \to t_1)$ and $(A \to t_2)$, where $t_1 = f(f(b,A),f(c,A))$

and $t_2 = f(a, f(a, f(a,a)))$. After careful counting one can tell that t'' exhibits at most four non-overlapping occurrences of the digram $\alpha = (f, 2, f)$. However, if we use the above function `retrieve-all-occs-dag` we only capture three of them. We obtain $\mathsf{occ}'_{t_1}(\alpha) = \{\varepsilon\}$, $\mathsf{occ}'_{t_2}(\alpha) = \{2\}$ and therefore

$$\mathsf{occ}'_{t''}(\alpha) = \bigcup_{\substack{i \in \{1,2\}, \\ v \in \mathsf{occ}'_{t_i}(\alpha)}} \mathtt{unfold}(\mathcal{G}, t_i, v) = \{\varepsilon, 122, 222\} \ .$$

Even though this approach does not capture all the occurrences which could be captured when not using the DAG representation, it still achieves a competitive compression performance on our set of test files (*cf.* Sect. 6.6 on page 84). It seems that a more involved method of dealing with digrams with equal parent and child symbols spanning two productions would necessitate a partial unfolding of the DAG. The latter, however, would certainly result in a longer runtime.

4.5.2 *Updating the Sets of Non-overlapping Occurrences*

Considering the graph representation of a DAG, a tree node can exhibit multiple parent nodes. In fact, a node has multiple parent nodes if it is the root of the right-hand side of a production of the corresponding DAG grammar and if this production is referenced multiple times.

To capture all digram occurrences which are absorbed by the replacement of a digram we need to take care of the above fact. The `remove-absorbed-occs` function listed in Fig. 4.11 needs to be adapted accordingly. Instead of removing one occurrence formed by the node being replaced and its parent, we need to iterate over possibly multiple parents and remove all corresponding occurrences. In Fig. 4.19 the function `remove-absorbed-occs-dag` is listed which incorporates this necessary modification. Analogously, the function `add-new-occs` listed in Fig. 4.12 must be modified to work properly in the DAG mode. Figure 4.20 shows an adapted version.

It is easy to see that our linear runtime is not negatively affected by this loop over all parents. Far from it — as mentioned earlier, the DAG representation saves us time by avoiding repetitive recalculations.

4.5.3 *Replacing the Digrams*

The third and last scenario in which we have to take special care of the DAG representation is when replacing an occurrence of a digram $\alpha \in \Pi$ spanning two productions of the DAG grammar. Due to our restriction on digrams with equal parent and child symbols the digram α has to have different parent and child symbols. In the following we want to use an example to describe what needs to be done when replacing the digram α.

```
FUNCTION remove-absorbed-occs-dag($\bar{t}, v, j$)
  if ($v \neq \varepsilon$) then
    remove-occ-dag($\bar{t}$, $\mathsf{parent}(v)$, $\mathsf{index}(v)$);
  else
    let $\overline{A}$ be the right-hand side of $\bar{t}$;
    for each $(\bar{t}', u) \in \mathsf{ref}_{\overline{\mathcal{G}}_i}(\overline{A})$ do
      remove-occ-dag($\bar{t}'$, $\mathsf{parent}(u)$, $\mathsf{index}(u)$);
    endfor
  endif

  for ($l \in \{1, 2, \ldots, \mathsf{rank}(\lambda_{\bar{t}}(v))\}$) do
    remove-occ-dag($\bar{t}, v, l$);
  endfor

  for ($l \in \{1, 2, \ldots, \mathsf{rank}(\lambda_{\bar{t}}(vj))\}$) do
    remove-occ-dag($\bar{t}, vj, l$);
  endfor
ENDFUNC

FUNCTION remove-occ-dag($\bar{t}, v, j$)
  if ($\lambda_{\bar{t}}(vj) \notin \mathcal{N}$) then
    $\alpha := \big(\lambda_{\bar{t}}(v), j, \lambda_{\bar{t}}(vj)\big)$;
  else
    let $\bar{t}'$ be the right-hand side of $\lambda_{\bar{t}}(vj)$;
    $\alpha := \big(\lambda_{\bar{t}}(v), j, \lambda_{\bar{t}'}(\varepsilon)\big)$;
  endif
  $\mathsf{occ}'_{\bar{t}}(\alpha) := \mathsf{occ}'_{\bar{t}}(\alpha) \setminus \{v\}$;
ENDFUNC
```

Figure 4.19: Listing of the function `remove-absorbed-occs-dag` which removes all absorbed occurrences from the $\mathsf{occ}'_{\bar{t}}$ sets when using the DAG mode.

```
FUNCTION add-new-occs-dag($\bar{t}, v$)
  if ($v \neq \varepsilon$) then
    add-occ-dag($\bar{t}$, $\mathsf{parent}(v)$, $\mathsf{index}(v)$);
  else
    let $\overline{A}$ be the right-hand side of $\bar{t}$;
    for each $(\bar{t}', u) \in \mathsf{ref}_{\overline{\mathcal{G}}_i}(\overline{A})$ do
      add-occ-dag($\bar{t}'$, $\mathsf{parent}(u)$, $\mathsf{index}(u)$);
    endfor
  endif

  for ($l \in \{1, 2, \ldots, \mathsf{rank}(\lambda_{\bar{t}}(v))\}$) do
    add-occ-dag($\bar{t}, v, l$);
  endfor
ENDFUNC

FUNCTION add-occ-dag($\bar{t}, v, j$)
  if ($\lambda_{\bar{t}}(vj) \notin \mathcal{N}$) then
    $\alpha := \big(\lambda_{\bar{t}}(v), j, \lambda_{\bar{t}}(vj)\big)$;
  else
    let $\bar{t}'$ be the right-hand side of $\lambda_{\bar{t}}(vj)$;
    $\alpha := \big(\lambda_{\bar{t}}(v), j, \lambda_{\bar{t}'}(\varepsilon)\big)$;
  endif
  $\mathsf{occ}'_{\bar{t}}(\alpha) := \mathsf{occ}'_{\bar{t}}(\alpha) \cup \{v\}$;
ENDFUNC
```

Figure 4.20: Listing of the function `add-new-occs-dag` which adds all new occurrences to the $\mathsf{occ}'_{\bar{t}}$ sets when using the DAG mode.

Example 47 Consider the DAG grammar given by the two productions $S \to f\big(g(t_1, A), A\big)$ and $A \to h(t_2, t_3)$ which represents the $\mathcal{F}$-labeled tree t depicted in Fig. 4.21. Imagine that we want to replace the sole occurrence of the digram $(f, 2, h)$, *i.e.*, an occurrence spanning two productions.[7] In order to do that we mainly have to complete the following three steps.

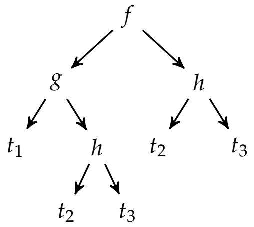


Figure 4.21: Depiction of the $\mathcal{F}$-labeled tree t. We have $t_1, t_2, t_3 \in T(\mathcal{F})$.

(1) We first have to introduce for every child of the node labeled by h a new production. Thus, we obtain two new productions $B \to t_2$ and $C \to t_3$. We can skip this step for every child node which is already labeled by a nonterminal of the DAG grammar.

(2) We need to update the production with left-hand side A to $A \to h(B, C)$.

(3) Finally, we introduce a new nonterminal D representing the digram $(f, 2, h)$ and update the production for S to

$$S \to D(g(t_1, A), B, C) \ .$$

The above steps are only necessary if the production with left-hand side A is referenced more than once. Otherwise we could have directly connected the children of h to the newly introduced node labeled by D and removed the production with left-hand side A from the grammar.

Since at most k new productions need to be introduced, the replacement of a digram occurrence can still be accomplished in constant time. All in all, it has become clear that even when representing the input tree of our algorithm as a DAG our implementation runs in linear time.

4.6 *Technical Details on the Prototype*

The source code of the TreeRePair prototype and its documentation is available at the Google Code™ open source developer site. It can be accessed by visiting the following web page:

```
http://code.google.com/p/treerepair
```

However, the implementation should be considered to be of alpha quality. There is still a lot of testing to be done.

We also implemented a decompressor called TreeDePair which is contained in the TreeRePair distribution. It is not optimized in terms of time and memory usage.

The software is licensed under the GPLv3 license which is available at

```
http://www.gnu.org/licenses/gpl-3.0.txt
```

[7] For the sake of convenience, our example uses a rather small tree and we decide to replace a digram occurring only once. We could easily enlarge t such that $(f, 2, h)$ occurs multiple times and still show the following.

It is implemented using the C++ programming language and can be compiled at least under the Windows and Linux operating systems. For compile instructions and library requirements, see the `README.txt` file in the root directory of the TreeRePair distribution.

5 Succinct Coding

In order to achieve a compact representation of the input tree of our TreeRePair algorithm we further compress the generated linear SLCF tree grammar by a binary succinct coding. The technique we use is loosely based on the DEFLATE algorithm described in [Deu96]. In fact, we use a combination of a fixed-length coding, multiple Huffman codings and a run-length coding to encode different aspects of the grammar (*cf.* Fig. 5.1).

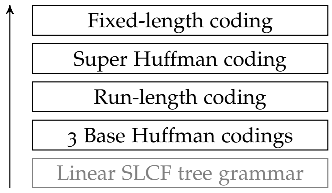


Figure 5.1: Hierarchy of the employed encodings.

In spite of the fact that we obtain an extremely compact binary representation of the generated SLCF tree grammar we are still able to directly execute queries on it with little effort. Basically, we only have to reconstruct the Huffman trees to be able to partially decompress the grammar on demand.

In [MMS08] many different variants of succinct codings specialized in SLCF tree grammars were investigated. Among them there was one encoding scheme which turned out to achieve the best compression performance in general — at least with respect to the set of sample SLCF tree grammars which was used in this work. However, our experiments show that, regarding the SLCF tree grammars generated by TreeRePair, this encoding is outperformed by the succinct coding which we present in this chapter.

5.1 General Remarks

In this section, we want to elaborate on the following topics: How do we need to modify the pruning step of our algorithm to make our succinct coding as efficient as possible? How does TreeRePair efficiently deal with parameter nodes? How can we serialize a Huffman tree in a compact way?

5.1.1 Inefficient Productions

Our experiments showed that, at least for our set of test XML documents, we achieve better compression results in terms of the size of the output file if we slightly modify the pruning step of our algorithm. It turns out that our succinct coding, which we describe in the following sections, is most efficient if we prune all productions with a sav-value smaller than or equal to 2 (instead of pruning all productions with a sav-value smaller than or equal to 0 as it is described in Sect. 3.3 on page 28). However, we use this modification only if we make the size of the output file a top priority (by using the switch `-optimize filesize`). Otherwise, when optimizing the number of edges of the final grammar (*i.e.*, when using the switch `-optimize edges`), we stick to the original version of the pruning step.

5.1.2 Handling of Parameter Nodes

Let $\mathcal{G} = (N, P, S)$ be the linear SLCF tree grammar which was generated by a run of TreeRePair. Then, for every production $(A \to t) \in P$ it holds that $y_i \in \mathcal{Y}$ labels the i-th parameter node of t in preorder, where $i \in \{1, 2, \dots, \mathsf{rank}(A)\}$. Due to this fact it is sufficient to represent the parameter symbols $y_1, y_2, \dots, y_{\mathsf{rank}(A)} \in \mathcal{Y}$ by a single parameter symbol $y \in \mathcal{Y}$. Let $(B \to t') \in P$ be another production and let $v \in \mathsf{dom}_{t'}$ with $\lambda_{t'}(v) = A$. Now, let us assume that we want to eliminate the production $(A \to t)$ and that we use only a single parameter symbol labeling all parameter nodes. It is clear that the i-th (in preorder) parameter node of t must be replaced by the subtree which is rooted at the i-th child of v.

Our implementation takes advantage of the above simplification, *i.e.*, it uses only one parameter symbol y for every occurring parameter node.

5.1.3 Serializing Huffman trees

As stated in [Deu96], it is sufficient to only write out the lengths of the generated codes to be able to reconstruct a Huffman tree at a later date. However, this requires the decompressor to be aware of the following.

- What symbols are encoded by the corresponding Huffman tree?
- In what order are their code lengths listed?

In our case only integers need to be encoded by Huffman codings because we will encode all symbols by integers (see Sect. 5.2 on page 67). Hence, it is obvious to use the natural order of integers to list the lengths of the generated codes. Let us assume that $n \in \mathbb{N}$ is the biggest integer which needs to be encoded and which was assigned a code to, respectively. We just need to loop over all integers $m \leq n$ in their natural order and print out the corresponding code length for each of it. For every $k < n$ for which no code was assigned to we print out a code length of 0.

In order to solely rely on the code lengths there is still something which needs to be considered. We are required to assign new codes to the integers based on the lengths of their original codes. More precisely, the new code assignment has to fulfill the following two requirements.

(1) All codes of the same code length exhibit lexicographically consecutive values when ordering them in the natural order of the integers they represent.

(2) Shorter codes lexicographically precede longer codes.

This reorganization of the Huffman codes does not affect the compression performance of the coding since only codes of the same length are swapped. The following example is based on an example from [Deu96].

Example 48 Imagine that we want to use a Huffman coding to encode the letters a, b, c and e which are each occurring multiple times in a data stream. Let us assume that we obtain the Huffman codes listed in Table 5.1. In order to be able to store the corresponding Huffman tree by only writing out the lengths of the Huffman codes we need to assign new codes to the letters. Table 5.2 shows the newly assigned codes which fulfill the above two requirements (1) and (2).

Now, let us assume that the decompressor expects the code lengths to be the lengths of codes assigned to the letters of the Latin alphabet and that these code lengths are ordered in the natural order of the letters they represent. Then, the corresponding Huffman tree can be unambiguously represented by the following sequence of code lengths: $2, 1, 3, 0, 3$. Note that we need to insert a code length of 0 at the position of the letter d since there is no code assigned to the letter d.

Symbol	Code
a	00
b	1
c	011
e	010

Table 5.1: Huffman coding before the reorganization of the codes. The letters are listed in their natural order, *i.e.*, in alphabetic order.

Symbol	Code
a	10
b	0
c	110
e	111

Table 5.2: Huffman coding from Table 5.1 after the reorganization of the codes.

5.2 *Contents of the Output File*

In this section we want to elaborate on the information which needs to be stored in the output file of our algorithm in order to be able to reconstruct the generated linear SLCF tree grammar at a later date. We also want to demonstrate how this data can be efficiently represented. However, at this time we do not pay attention to the fixed-length, run-length or Huffman codings which are employed in a subsequent step of the encoding process. For the sake of simplicity we consider these encodings in separate sections of this chapter.

Let $\mathcal{G} = (N, P, S)$ be the linear SLCF tree grammar over $\mathcal{F}$ which was generated by a run of TreeRePair. Before we are able to compile the information which needs to be written out we need to assign to every symbol from $\mathcal{F} \cup (N \setminus \{S\}) \cup \{y\}$ a unique integer. In fact, we assign to every symbol from $\mathcal{F}$ a unique ID from the set $\{1, 2, \ldots, |\mathcal{F}|\} \subset \mathbb{N}$. We assign the ID $|\mathcal{F}| + 1$ to y, *i.e.*, to the special symbol labeling all parameter nodes in the right-hand sides of P's productions. Finally, we associate with every symbol from the set of nonterminals $N \setminus \{S\}$ a unique ID from the set of integers $\{|\mathcal{F}| + 2, |\mathcal{F}| + 3, \ldots, |\mathcal{F}| + |N|\}$. The IDs are assigned to the nonterminals in such a way that the nonterminal $A \in N \setminus \{S\}$ has a higher ID than the nonterminal $B \in N \setminus \{S\}$ if $B \rightsquigarrow_{\mathcal{G}}^{+} A$ holds.

5.2.1 Writing out the Necessary Informations

Now, we are able to write out the information needed to reconstruct $\mathcal{G}$ in four steps. Bear in mind that the values mentioned below are not directly written to the output file but that they are additionally encoded by a combination of multiple Huffman codings, a run-length coding and a fixed-length coding later on.

First step In the first step, we write out the number of terminal symbols $|\mathcal{F}|$ and the number of introduced productions $|N| - 1$, *i.e.*, we are not counting the start production. By handing over this information to the decompressor we avoid the insertion of separators marking, for instance, the end of the enumeration of elements types (which are written out in the third step).

Second step In the second step, we directly append a representation of the children characteristics of the terminal symbols. By children characteristics we mean their rank and, concerning terminal symbols of rank 1, if we are dealing with a left or a right child.[1] Due to the fact that all terminal symbols have a rank of at most two, we can encode this information using two bits per symbol. Table 5.3 lists all the bit strings we use together with a brief description of their meanings.

Bit string	Description
00	rank 0
01	rank 1, right child
10	rank 1, left child
11	rank 2

Table 5.3: The bit strings encoding the children characteristics together with their meaning.

We write out the children characteristics as follows: Firstly, we print out a bit string from Table 5.3 representing a certain children characteristic. After that we append the number of corresponding terminal symbols and finally we enumerate their IDs. We do this for the characteristics 00, 01 and 10. We omit the enumeration of all terminal symbols with a rank of 2 since their IDs can be reconstructed with the information in hand. In fact, we just need to subtract the set of IDs of all terminal symbols with children characteristics 00, 01 and 10 from the set of IDs of all terminal symbols from $\mathcal{F}$ (which is $\{1, 2, \ldots, |F|\}$).

Furthermore, it is not necessary to print out the ranks of the nonterminals from N since these can be easily reconstructed by counting the number of parameter nodes in the corresponding right-hand sides. The latter are written to the output file in the fourth step.

[1] Consult Sect. 2.4 on page 20 for an explanation on why this information is necessary to reconstruct the input tree.

Third step In this step, we print the element types of the terminal symbols in the ascending order of their IDs to the output file. We do this by writing out the ASCII code of every single letter. The individual names are terminated by the ASCII character ETX which is assumed not to be used within the element types of the terminal symbols.

Fourth step In this last step we serialize the productions of $\mathcal{G}$ in the ascending order of the IDs of their left-hand sides. For every production $(A \to t) \in P$ we just write out the IDs of the labels of t's nodes in preorder. We do not need to use special marker symbols to indicate the nesting structure of the symbols and their IDs, respectively. When parsing the output file this hierarchy can be easily obtained by taking care of the individual ranks of the symbols.

We can also omit the specification of the left-hand side A since both, its ID and its rank, can be reconstructed with the information in hand. Imagine that we are parsing the output file to reconstruct the productions of $\mathcal{G}$. If we are parsing the i-th production, the ID of its left-hand side must be $|\mathcal{F}| + 1 + i$, where $i \in \{1, 2, \ldots, |N|\}$. As already mentioned, the rank of the left-hand side can be obtained by counting the parameter nodes in the right-hand side once this has been reconstructed.

Note that it is superfluous to insert separators between the representations of the productions from P since their boundaries can be calculated based on the ranks of the symbols. Again, imagine that we are trying to reconstruct the productions of P by parsing the output file of our algorithm. Let $(A \to t) \in P$ be the first production we encounter. The tree t can only consist of nodes labeled by terminal symbols, *i.e.*, we must have $t \in T(\mathcal{F})$.[2] The ranks of all symbols from $\mathcal{F}$ are known since the necessary information was written to the compressed file in the second step. Therefore, we can easily reconstruct t by iteratively parsing the corresponding IDs in the output file. While doing so we are also able to count the number of occurrences of the symbol $y \in \mathcal{Y}$ in t. Thus, we are aware of the value of $\mathsf{rank}(A)$. After that, we proceed with decoding the second production $(A' \to t') \in P$ by iteratively parsing the next IDs. We have $t' \in T(\mathcal{F} \cup \{A\})$, *i.e.*, the ranks of all occurring symbols are known. That way all productions from P can be reconstructed.

Example 49 In order to get a clear picture of the representation described above we apply the previous four steps to the linear SLCF tree grammar $\mathcal{G} = (N, P, S_4)$ over the ranked alphabet $\mathcal{F}$ from Sect. 3.4 on page 32, *i.e.*, we have $N = \{S_4, A_2, A_3\}$ and P is

[2] This is due to the fact that we have written out the productions in the ascending order of the IDs of their left-hand sides. These IDs were assigned to the nonterminals in such a way that the nonterminal $A \in N \setminus \{S\}$ has a higher ID than $B \in N \setminus \{S\}$ if $B \rightsquigarrow_{\mathcal{G}}^{+} A$ holds. Therefore, the right-hand side of $(A \to t)$, which is the first production which was written out, does not contain any node labeled by a nonterminal from N.

the following set of productions:

$$\begin{aligned} S_4 &\rightarrow \mathsf{books}^{10}(A_3(A_3(A_3(A_3(\mathsf{book}^{10}(A_2)))))) \\ A_3(y) &\rightarrow \mathsf{book}^{11}(A_2, y) \\ A_2 &\rightarrow \mathsf{author}^{01}(\mathsf{title}^{01}(\mathsf{isbn}^{00})) \end{aligned}$$

First of all, we assign to every symbol from $\mathcal{F} \cup (N \setminus \{S_4\}) \cup \{y\}$ a unique ID as it is shown in Fig. 5.2. After that we are able to write out the grammar exactly as described above resulting in the value sequence depicted in Fig. 5.3. We accomplish this task in four steps:

(1) We begin by writing out the number of terminals (6) directly followed by the number of nonterminals minus the start nonterminal (2) — see the values 0 and 1 in the depiction.

(2) After that the children characteristics of all terminal symbols are written to the file. We begin by specifying all terminal symbols of rank 0 (values 2–4). This is done by firstly writing out the bit string 00 and the number of corresponding symbols (1). Finally, the ID 2 of the terminal symbol isbn^{00}, which is the sole terminal symbol of rank 0, is listed.

Analogously, the terminal symbols with children characteristics 01 and 10 are enumerated (values 5–12).

(3) Now, the element types of all terminal symbols are exported to the output file (values 13–46). For each of them the decimal value of each ASCII character is written out. The element type *books*, for instance, is encoded by the sequence 98, 111, 111, 107, 115.

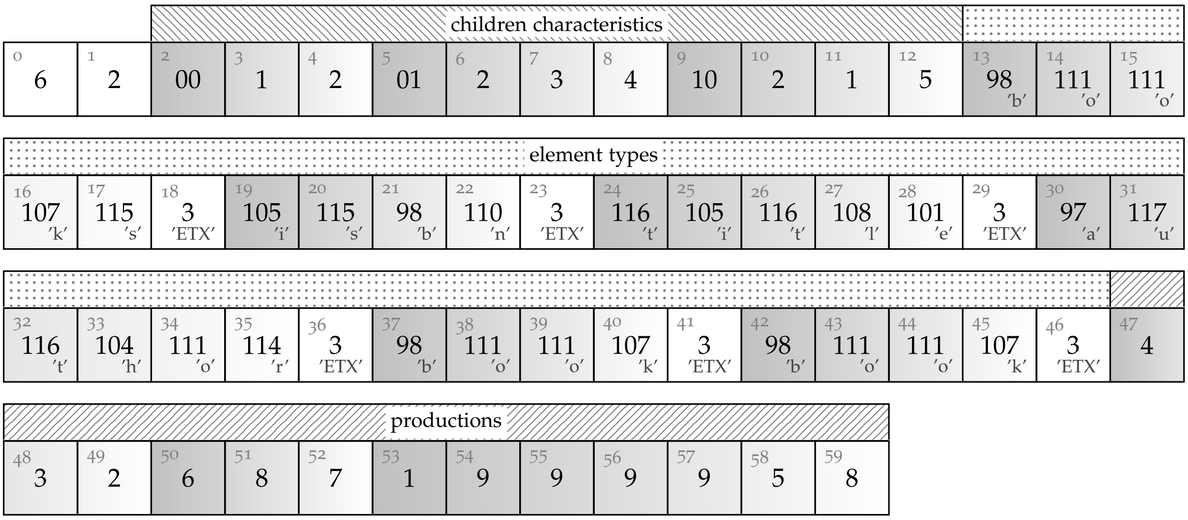
(4) Finally, the productions from P are written out in the ascending order of the IDs of their left-hand sides. Thus, the production with left-hand side A_2 is serialized as the very first production (values 47–49). It is encoded by the unambiguous sequence of IDs $4, 3, 2$ representing the terminal symbols author^{01}, title^{01} and isbn^{00} of the right-hand side of A_2 in preorder. Afterwards the remaining productions with left-hand sides A_3 (values 50–52) and S (values 53–59) are printed to the output file in this order.

Symbol	ID
books^{10}	1
isbn^{00}	2
title^{01}	3
author^{01}	4
book^{10}	5
book^{11}	6
y	7
A_2	8
A_3	9

Figure 5.2: All symbols with the ID assigned to them. The symbol y is the symbol used to label the parameter nodes in the right-hand sides of P's productions.

5.2.2 *Possible Optimizations*

Of course, there is still room to further reduce the data which needs to be written to the output file. Consider, for instance, terminal symbols of the same element type but different children characteristics. In the case of our implementation, the element type of these symbols is written to the file two or three times in the second step. However, an optimization with respect to this



0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
6	2	00	1	2	01	2	3	4	10	2	1	5	98	111	111
													'b'	'o'	'o'

16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
107	115	3	105	115	98	110	3	116	105	116	108	101	3	97	117
'k'	's'	'ETX'	'i'	's'	'b'	'n'	'ETX'	't'	'i'	't'	'l'	'e'	'ETX'	'a'	'u'

32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47
116	104	111	114	3	98	111	111	107	3	98	111	111	107	3	4
't'	'h'	'o'	'r'	'ETX'	'b'	'o'	'o'	'k'	'ETX'	'b'	'o'	'o'	'k'	'ETX'	

48	49	50	51	52	53	54	55	56	57	58	59
3	2	6	8	7	1	9	9	9	9	5	8

Figure 5.3: Representation of the grammar $\mathcal{G}$ from Example 49.

redundancy does only lead to marginally better compression results. This is due to the fact that typically the major part of the output file is the enumeration of the productions.

Still regarding the second step, we could at first determine the most frequent children characteristic and omit the enumeration of all corresponding terminal symbols. This dynamic approach certainly leads to a small reduction of the size of the output file compared to always skipping the children characteristic 11.

Another aspect which confesses optimization potential are possible long lists of the parameter symbol y which emerge when writing out the right-hand sides of productions with a higher rank. In this case, run-length coding can lead to a better compression performance. However, we did not further investigate this matter since we focus on generating grammars with nonterminals with a maximal rank of 4.

5.3 *Employing Multiple Types of Encodings*

Even though a Huffman tree has to be serialized for every Huffman coding used within our output file, we decided in favor of using four distinct Huffman codings. We use three of them for encoding

- the start production,
- the remaining productions, the children characteristics of the terminal symbols and the numbers of terminals and nonterminals, and finally
- the names of the terminals.

In the sequel, we call these three Huffman codings the *base Huffman codings*. The fourth Huffman coding, which we call *super Huffman coding*, is used to encode the Huffman trees of the above codings. Our tests with different numbers of Huffman codings revealed that, in general, the above approach leads to the best compression results. This is at least true for most of the XML test documents we used.

5.3.1 *Base Huffman Codings*

We serialize the three base Huffman codings by writing out the lengths of the generated codes as it is described in Sect. 5.1.3 on page 66. However, we additionally apply a run-length coding and the super Huffman coding to achieve a compact binary representation. In Sect. 5.3.3 on page 73 we elaborate on how exactly the run-length coding works. We briefly call the length of a code of a base Huffman coding a *base code length* in the sequel. Analogously, we denote the lengths of the codes of the super Huffman coding by the term *super code lengths*.

We output the number of base code lengths in front of every serialized base Huffman coding, *i.e.*, in front of every enumeration of base code lengths. That way the decompressor knows how many bits are part of this binary representation. Let us point out that this number of code lengths is encoded using k bits instead of using the super Huffman coding, where $k \in \mathbb{N}$ is a constant which is fixed at compile time. We do this due to the following fact. Let $n \in \mathbb{N}$ be the number of code lengths and let us assume that we encode n, which is usually many times larger than the maximum over all code lengths, using the super Huffman coding. This would result in a big gap of unused integers between the super code lengths and n. This again would lead to a long list of 0's when storing the super Huffman tree by enumerating its code lengths. In general, this leads to a reduced compression performance compared to a fixed-length coding of n using k bits.

5.3.2 *Super Huffman Coding*

The super Huffman coding will also be stored by the sequence of its code lengths. However, the relatively small set of integers is encoded by a fixed-length coding using $n \in \mathbb{N}$ bits, where n is the smallest possible number of bits which can be used to encode all super code lengths. More precisely, we serialize the super Huffman coding in three steps:

(1) First of all, we print out the binary representation of the number n using k bits, where $k \in \mathbb{N}$ is a fixed number of bits which is specified at compile time.

(2) Let $m \in \mathbb{N}$ be the biggest base code length. We print out the binary representation of m using k bits. With this information the decompressor knows that the next $n \cdot m$ bits make up the list of super code lengths.

(3) Finally, the binary representations of the m many super code lengths are written to the output file using n bits for each code

length. The super code lengths are printed in the natural order of the integers which are represented by the corresponding codes.

5.3.3 *Run-length Coding of the Base Code Lengths*

In this section we explain the run-length coding which is applied to the enumerations of code lengths used to write all base Huffman codings to the output file. This additional encoding marks a major contribution to the compactness of our representation. The bigger a code length is, the more different codes of that length are possible. At the same time a sequence of several occurrences of the same code length within the enumeration of all code lengths becomes more likely. In addition, our experience shows that it frequently happens that there is a longer run of 0's in the list of all code lengths due to symbols which no codes were assigned to.

Example 50 Consider, for instance, the example from Sect. 5.3.4 and in particular the base Huffman coding C_3 which is listed in Table 5.6 on page 74. This Huffman coding does not assign codes to the symbols 4–96. This results in a sequence of 94 zeros within the enumeration of the code lengths of C_3.

Definition 51 Let $m, k \in \mathbb{N}$, where $k \geq \lfloor \log_2(m) \rfloor + 1$. In the following we denote by $\mathsf{bin}_k(m)$ the (0-padded) binary representation $b_k b_{k-1} \ldots b_0$ of m, *i.e.*, the following holds:

$$m = \sum_{i=0}^{k} b_i \cdot 2^i$$

We encode an enumeration of code lengths using a run-length coding as follows: Let us assume that $n \in \mathbb{N}$ is the maximum code length. Then we use the three additional integers $n+1$, $n+2$ and $n+3$ to indicate certain types of runs — we call them *run indicators* in the sequel. Principally, all runs with a length less than or equal to 3 are straightly written to the output file. In contrast, a run of a code length $m \in \mathbb{N}$ exceeding this bound is encoded as follows:

- If we have $m > 0$, we use the run indicator $n+1$ and a bit string with a length of 2 to indicate 4–7 repetitions of the code length m. If $k > 3$ is the length of the run of m and $l = k \mod 7$ (*i.e.*, $l \in \{0, 1, \ldots, 6\}$), then this run is encoded as follows:
  - if $l > 3$:

$$m \; \underbrace{(n+1)\mathsf{bin}_2(3)}_{\lfloor k/7 \rfloor \text{ times}} \; (n+1)\mathsf{bin}_2(l-4)$$

  - if $l \leq 3$:

$$m \; \underbrace{(n+1)\mathsf{bin}_2(3)}_{\lfloor k/7 \rfloor \text{ times}} \; [m]^l$$

  Note that $[m]^l$ denotes l many consecutive m's.

- If we have $m = 0$, we use the run indicator $n+2$ with an appended bit string of length 3 to denote 4–11 repetitions of m. In contrast, we use the run indicator $n+3$ together with a bit string of length 7 to encode 12–139 repeated 0's.

  If $k > 3$ is the length of the run of 0's and $l = k \mod 139$ (*i.e.*, $l \in \{0, 1, \ldots, 138\}$), then this run is encoded as follows:

  – if $l > 11$:

  $$\underbrace{(n+3)\mathsf{bin}_7(127)}_{\lfloor k/139 \rfloor \text{ times}}\ (n+3)\mathsf{bin}_7(l-12)$$

  – if $3 < l \leq 11$:

  $$\underbrace{(n+3)\mathsf{bin}_7(127)}_{\lfloor k/139 \rfloor \text{ times}}\ (n+2)\mathsf{bin}_3(l-4)$$

  – if $l \leq 3$:

  $$\underbrace{(n+3)\mathsf{bin}_7(127)}_{\lfloor k/139 \rfloor \text{ times}}\ [m]^l$$

Symbol	Old code	New code
0	10	10
1	1110	1110
5	1111	1111
8	110	110
9	0	0

Table 5.4: Huffman coding C_1 used to encode the start production.

Symbol	Old code	New code
1	1010	1100
2	01	00
3	00	01
4	111	100
5	1011	1101
6	110	101
7	1001	1110
8	1000	1111

Table 5.5: Huffman coding C_2 used to encode the productions from $P \setminus \{S\}$, the children characteristics, and numbers of terminals and nonterminals.

Example 52 Consider the following sequence of integers:

$$122333\ 444444\ 555\ 000000000$$

Now, let us assume that we want to encode the above sequence using our run-length coding. Obviously, we have $n = 5$. The above run of 4's with a length of 6 is represented by the sequence 4610 since we have $n+1 = 6$ and $\mathsf{bin}_2(6-4) = 10$. In contrast, the run of 0's with a length of 9 leads to the sequence 7101 because it holds that $n+2 = 7$ and that $\mathsf{bin}_3(9-4) = 101$. All in all, we obtain the sequence 122333 4610 555 7101.

Surprisingly, our investigations evinced that an approach which dynamically adjusts the length of the bit strings used in the above encoding depending on the size of the input grammar does not lead to significantly better compression results.

Symbol	Old code	New code
3	111	010
97	00101	11010
98	101	011
101	00100	11011
104	00111	11100
105	1001	1010
107	1101	1011
108	110011	111110
110	110010	111111
111	01	00
114	11000	11101
115	1000	1100
116	000	100
117	00110	11110

Table 5.6: Huffman coding C_3 used to encode the names of the terminal symbols.

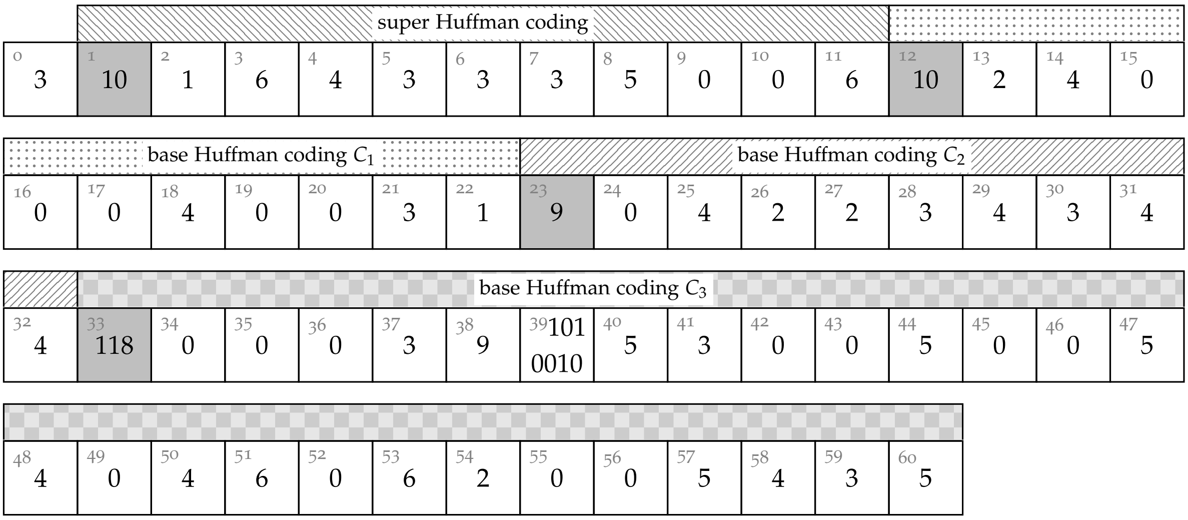
5.3.4 *Example*

This example continues the encoding of the linear SLCF tree grammar $\mathcal{G}$ from Example 49 on page 69. The Tables 5.4, 5.5 and 5.6 list the three base Huffman codings, called C_1, C_2 and C_3 in the sequel, which are calculated by our implementation. The columns labeled *Old code* show the initial Huffman codes while the columns labeled *New code* list the newly assigned codes after the necessary reorganization described in Sect. 5.1.3 on page 66.

While Fig. 5.3 on page 71 shows the second part of the output file as it is generated by a run of TreeRePair the Fig. 5.4 shows the first part of it. The latter stores the base Huffman codings C_1, C_2 and C_3 together with the corresponding super Huffman coding. For the sake of clarity the corresponding values are denoted

Symbol	Old code	New code
0	0	0
1	110000	111110
2	1101	1110
3	101	100
4	111	101
5	100	110
6	11001	11110
9	110001	111111

Table 5.7: Super Huffman coding used to encode the code lengths of the base Huffman codings.


super Huffman coding
base Huffman coding C_1
base Huffman coding C_2
base Huffman coding C_3

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
3	10	1	6	4	3	3	3	5	0	0	6	10	2	4	0

16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
0	0	4	0	0	3	1	9	0	4	2	2	3	4	3	4

32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47
4	118	0	0	0	3	9	101 0010	5	3	0	0	5	0	0	5

48	49	50	51	52	53	54	55	56	57	58	59	60
4	0	4	6	0	6	2	0	0	5	4	3	5

Figure 5.4: Depiction of the part of the output file which contains the serialized four Huffman codings.

by their integer representation instead of by their fixed-length or Huffman code. The Huffman coding C_1 from Table 5.4, for instance, is given by the sequence of code lengths ranging from value 13 to value 22, where value 12 informs us about the length of this sequence. Analogously, the code lengths of the Huffman codings C_2 and C_3 are given by the values 24–32 and 34–60, respectively. The sequence of code lengths of the Huffman coding C_3 exhibits a longer run, namely, 94 consecutive occurrences of the code length 0. This run is encoded by the run indicator $9 = n + 3$ and the bit string $\mathsf{bin}_7(94 - 12) = 1010010$, where $n = 6$ is the maximal length of a code from C_3.

The super Huffman coding listed in Table 5.7 is written to the output file (values 2–11) using 3 bits per integer as it is stated by the value 0 of the output file. There need to be enumerated 10 super code lengths since 10 values — the base code lengths $0, 1, \dots, 6$ and the run indicators $7, 8, 9$ which are used by the base Huffman coding C_3 — need to be encoded.

6
Experimental Results

In the following, we compare the compression performance of our implementation of the Re-pair for Trees algorithm with existing algorithms. Furthermore, we will check the impact of the DAG representation of the input tree on the compression factors achieved and we will learn about the influences of small changes to the maximal rank allowed for a nonterminal.

6.1 *XML Documents Used*

The set of XML documents we used for investigating the performance of TreeRePair consists of 23 files with different characteristics (*cf.* Table 6.1). Most of them were used in past papers evaluating various XML compressors and therefore may be familiar to the reader. The original files can be obtained from the sources listed in Table 6.2. In all cases character data, attributes, comments, namespace information were removed from the XML files, *i.e.*, the XML documents consist only of start tags, end tags and empty element tags. We do so, because, at this time, TreeRePair ignores this information and solely concentrates on the XML document tree.

6.2 *Algorithms Used in Comparison*

Basically, we compare our implementation of Re-pair for Trees with two other compression algorithms based on linear SLCF tree grammars, namely, BPLEX [BLM08] and Extended-Repair [Kri08, BHK10]. The former is a sliding-window based linear time approximation algorithm. It searches bottom-up in a fixed window for repeating tree patterns. The size of the sliding window, the maximal pattern size and the maximal rank of a nonterminal can be specified as input parameters. One of the main drawbacks of

XML document	File size (kb)	# Edges	Depth	# Element types	Source
1998statistics	349	28 305	5	46	1
catalog-01	4 219	225 193	7	50	9
catalog-02	44 656	2 390 230	7	53	9
dictionary-01	1 737	277 071	7	24	9
dictionary-02	17 128	2 731 763	7	24	9
dblp	117 822	10 802 123	5	35	2
EnWikiNew	4 843	404 651	4	20	3
EnWikiQuote	3 134	262 954	4	20	3
EnWikiSource	13 457	1 133 534	4	20	3
EnWikiVersity	5 887	495 838	4	20	3
EnWikTionary	99 201	8 385 133	4	20	3
EXI-Array	5 347	226 522	9	47	5
EXI-factbook	1 214	55 452	4	199	5
EXI-Invoice	266	15 074	6	52	5
EXI-Telecomp	3 700	177 633	6	39	5
EXI-weblog	1 104	93 434	2	12	5
JST_gene.chr1	4 202	216 400	6	26	8
JST_snp.chr1	13 795	655 945	7	42	8
medline02n0328	51 751	2 866 079	6	78	6
NCBI_gene.chr1	6 862	360 349	6	50	8
NCBI_snp.chr1	63 941	3 642 224	3	15	8
sprot39.dat	111 175	10 903 567	5	48	7
treebank	19 551	2 447 726	36	251	4

Table 6.1: Characteristics of the XML documents used in our tests. The values in the "Source"-column match the source IDs in Table 6.2. The depth of an XML document tree specifies the length (number of edges) of the longest path from the root of the tree to a leaf.

ID	Source
1	`http://www.cafeconleche.org/examples`
2	`http://dblp.uni-trier.de/xml`
3	`http://download.wikipedia.org/backup-index.html`
4	`http://www.cs.washington.edu/research/xmldatasets`
5	`http://www.w3.org/XML/EXI`
6	`http://www.ncbi.nlm.nih.gov/pubmed`
7	`http://expasy.org/sprot`
8	`http://snp.ims.u-tokyo.ac.jp`
9	`http://softbase.uwaterloo.ca/~ddbms/projects/xbench`

Table 6.2: Sources of the XML documents from Table 6.1.

BPLEX is that there exists only a slowly running implementation of it.

In contrast, Extended-Repair (which we sometimes call E-Repair in the sequel) is an algorithm developed by Christoph Krislin from the University of Paderborn, Germany as part of his Diplomarbeit [Kri08, BHK10]. This algorithm is, just like our Re-pair for Trees algorithm, based on the Re-pair algorithm introduced in [LM00]. However, it was independently developed and exhibits some fundamental differences to our algorithm. One of the main differences is that the Extended-Repair algorithm at first generates a DAG of the input tree and then processes each part of it individually, *i.e.*, it generates multiple grammars which are combined in the end. The individual parts of the input tree are called "repair packets". The maximal size of each packet can be specified by an input parameter (default is 20 000 edges). The author of [Kri08] points out that this packet-based behavior may have a negative impact on the compression performance of the Extended-Repair algorithm. Our own investigations concerning a TreeRePair version running on the DAG of the input tree instead of on the whole tree support this point of view.

The author of Extended-Repair shows that in the case of the XML document `NCBI_snp.chr1` the avoidance of breaking down the input tree into packets (by choosing the maximum packet size large enough) results in a much more competitive compression result. However, our experiments show that at the same time the memory requirements and the runtime of the Extended-Repair algorithm rise drastically. Note that, regarding our algorithm, the DAG representation is merely used to save memory resources and is almost completely transparent to the overlying digram replacement process (*cf.* Sect. 4.5 on page 58).

6.3 *Testing Environment*

Our experiments were done on a computer with an Intel® Core™ 2 Duo CPU T9400 processor, four gigabytes of RAM and the Linux operating system. Every algorithm was executed on a single processor core, *i.e.*, no algorithm was able to make use of multiprocessing. TreeRePair and BPLEX were compiled with the `gcc`-

	TreeRePair	BPLEX	E-Repair	mDAG	bin. mDAG
Edges (%)	2.9	3.4	4.1	12.8	18.3
# NTs	4 753	13 660	6 522	2 075	5 320
Time (sec)	10	322	63	-	-
Mem (MB)	47	536	401	-	-
File size (%)	0.46	0.71	0.61	-	-

Table 6.3: Average values of the characteristics of the generated grammars and of the corresponding runs of the algorithms.

compiler using the `-O3` (compile time optimizations) and `-m32` (*i.e.*, we generated them as 32bit-applications) switches. We were not able to compile the `succ`-tool of the BPLEX distribution with compile time optimizations (*i.e.*, using the `-O3` switch). This tool is used to apply a succinct coding to a grammar generated by the BPLEX algorithm. However, this should not have a great influence on the runtime measured for BPLEX since the `succ`-tool usually executes quite fast compared to the runtime of the actual BPLEX algorithm. In contrast, Extended-Repair is an application written in Java™ for which we only had the bytecode at hand, *i.e.*, we did not have access to the source code of it. We executed Extended-Repair using the Java SE Runtime Environment™ in version `1.6.0_15`.

During the execution of the algorithms we always measured their memory usage. We accomplished this by constantly polling the `VmRSS`-value which is printed out by executing the command `cat /proc/<pid>/status`, where `<pid>` is the process ID assigned to the algorithm. In the first second of the execution of an algorithm this value was checked every ten milliseconds and after that the frequency was slowly reduced to one second.

Every time we executed BPLEX we used its default input parameters, namely, window size: 20 000, maximal pattern size: 20, maximal rank: 10. In order to be able to test BPLEX together with every file of our set of test XML documents we needed to explicitly allow large stack sizes using the standard tool `ulimit`.

6.4 *Comparison of the Generated Grammars*

In this section, we compare the final grammars generated by the algorithms TreeRePair, BPLEX and Extended-Repair. All algorithms were instructed to minimize the number of edges of the generated grammar. For TreeRePair, we achieved this behavior by specifying the `-optimize edges` input parameter. Regarding Extended-Repair, we used the supplied `ConfEdges.xml` configuration file which is supposed to make Extended-Repair minimize the number of edges. The BPLEX algorithm was executed with its default input parameter values and no changes were made to the generated grammar (besides pruning nonterminals which are referenced only once by using the supplied `gprint` tool).

	TreeRePair	BPLEX	E-Repair	XMill	gzip	bzip2
File size (%)	0.45	0.57	0.61	0.47	1.36	0.58
Time (sec)	10	329	167	119	< 1	16
Mem (MB)	47	536	399	7	-	7
Edges (%)	3.0	3.9	4.1	-	-	-
# NTs	2 642	2 796	7 003	-	-	-

Table 6.4: Average values of the characteristics of the runs of the three algorithms when making a small size of the output file top priority.

Table 6.3 shows the average values of the essential characteristics of the final grammars generated by the three competing algorithms. The first row shows the average compression factors in terms of the number of edges in percent. The edge compression factor is computed as follows: if $t \in T(\mathcal{F})$ is the binary representation of the input tree and $\mathcal{G}$ is the final grammar, we obtain the edge compression factor by computing $|\mathcal{G}|/|t| \cdot 100$. The second row shows the average number of nonterminals of the final grammars. For the sake of completeness, the average runtimes (in seconds), the average memory usages (in megabytes) and the average file size compression factors are also listed. The compression factor in terms of file size specifies the ratio between the size of the input file and the file size of the succinct coding of the final grammar in percent.

We also added two columns to Table 6.3 showing the average number of edges and the average number of nonterminals of the minimal DAGs of the input trees (mDAG) and the minimal DAGs of the binary representations of the input trees (bin. mDAG).

As it can be seen, on average, TreeRePair generates the smallest linear SLCF tree grammars (in terms of the number of edges) compared to the other two algorithms. At the same time, its grammars exhibit a small number of nonterminals. It outperforms BPLEX and Extended-Repair in terms of runtime and memory usage. The speed and moderate requirements on main memory are a result of the transparent DAG representation of the input tree and the many optimizations we made to the source code of TreeRePair during our investigations.

Figure 6.1 on page 83 gives an impression on how each of the three algorithms performs on the individual XML documents in terms of the size of the final grammar in edges. For each file, the algorithm which generates the largest grammar is set to 100%. In Appendix A.1 on page 89 there is a detailed table listing all relevant characteristics of the runs of the algorithms on the set of test XML documents.

6.5 Comparison of Output File Sizes

In this section, we concentrate on the sizes of the files generated by the runs of the algorithms on our set of test XML documents. In fact, we execute each algorithm in a mode in which the size of the resulting file is made a top priority. For TreeRePair, we achieve this by specifying the input parameter `-optimize filesize` and for Extended-Repair, we get such a behavior by using the supplied `ConfSize.xml` configuration file and the `-s 4` switch. The latter chooses a certain succinct coding of the Extended-Repair distribution which is supposed to generate very small representations of the generated grammar. Regarding BPLEX, we first apply the supplied `gprint`-tool using the parameters `--prune` and `--threshold 14`. After that we use the `succ`-tool of the BPLEX distribution together with the parameter `--type 68` to generate a Huffman coding-based succinct coding of the corresponding grammar. In [MMS08] it is stated that this approach leads to the best compression performance of BPLEX in general (in terms of file size).

In addition to the above three algorithms, we also consider the compression results produced by gzip, bzip2[1] and XMill 0.8 [LS00]. We include them in our comparison to make it easier to get a handle for common compression rates and runtimes. The first two algorithms are widely used general purpose file compressors which, of course, produce a non-queryable compressed representation of the input file. In contrast, XMill is a compressor specialized in compressing the structure and, in particular, the character data of XML documents. In fact, it mainly concentrates on how to group the character data of an XML document in such a way that it can be efficiently compressed by general purpose compressors like gzip. Since its implementation does not exhibit a special "only consider the structure of the XML document" mode, it may be unfair to directly compare its compression results with those of TreeRePair, BPLEX or Extended-Repair. However, we included its compression results, which we obtained using its default input parameters, because we were interested in its performance in this setting.

Table 6.4 shows the average sizes of the output files generated by the six algorithms mentioned above. For the sake of completeness, the average runtime, the average memory usage, the average number of edges and the average number of nonterminals are also listed. Again, TreeRePair outperforms BPLEX and Extended-Repair regarding all considered characteristics. Surprisingly, its queryable output files are even smaller than the non-queryable ones produced by the highly optimized gzip and bzip2 algorithms. However, gzip (but interestingly not bzip2) runs much faster than TreeRePair on our test data.

Figure 6.2 gives an impression on how each of the six algorithms performs on the individual XML documents in terms of the size of the generated output file. For each file, the algorithm which generates the biggest output file is set to 100%. In Appendix A.2 on page 93 there is a detailed table listing all relevant characteristics of the runs of the algorithms on our set of test XML documents.

[1] For more information about the gzip algorithm, see `http://www.gzip.org`. For bzip2, see `http://www.bzip.org`.

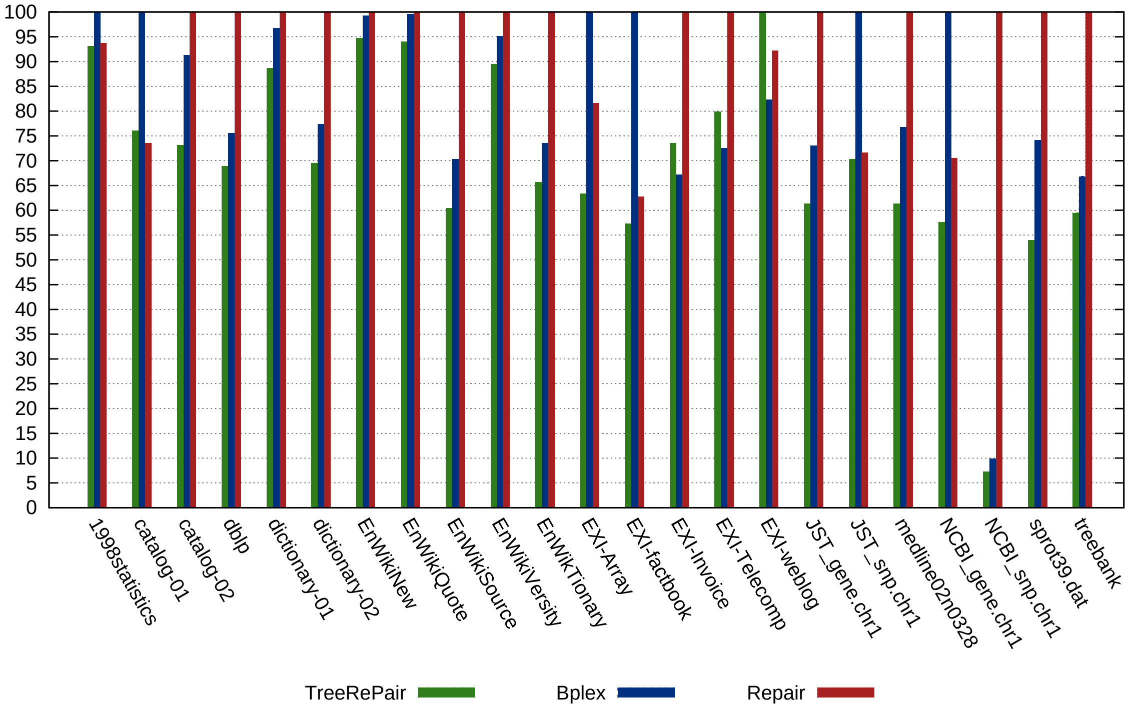


Figure 6.1: Comparison of the number of edges of the final grammars.

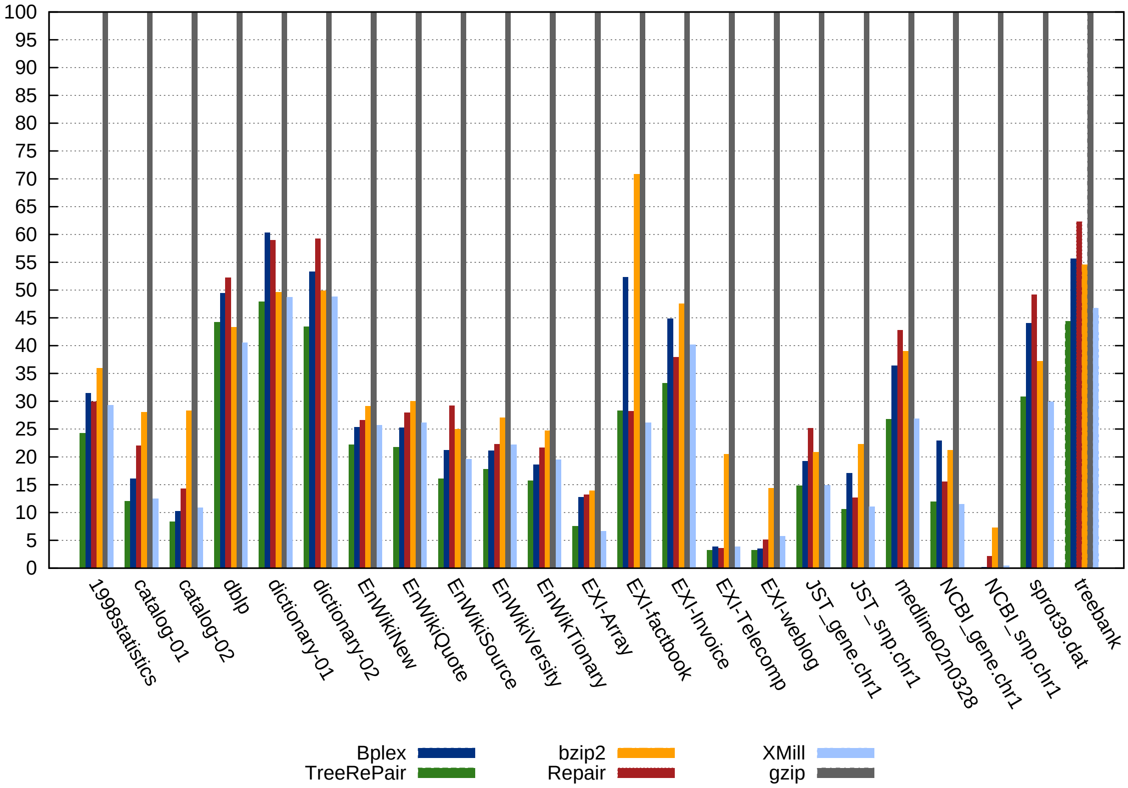


Figure 6.2: Comparison of the sizes of the output files.

	with DAG	without DAG
Edges (%)	2.86	2.84
# NTs	4 753	4 620
File size (%)	0.463	0.459
Time (sec)	9.8	11.2
Mem (MB)	47	188

Table 6.5: Average values of the characteristics of the runs of TreeRePair with and without the DAG representation of the input tree.

Max. rank	0	1	2	3	4	5	6
Edges (%)	55.02	3.29	2.92	2.89	2.86	2.89	2.89
# NTs	1 265	5 539	4 712	4 916	4 753	4 956	4 958
File size (%)	2.12	0.51	0.47	0.47	0.46	0.47	0.46
Time (sec)	7.0	8.4	9.3	9.5	9.6	9.8	9.8
Mem (MB)	44	44	45	47	47	47	47

Table 6.6: Average values of the characteristics of the runs of TreeRePair with different maximal ranks allowed for a nonterminal.

6.6 Results without DAG Representation

Table 6.5 shows a comparison between the compression results of TreeRePair when using and when not using, respectively, the DAG representation described in Sect. 4.2 on page 48. The left column shows the values obtained when executing TreeRePair with its default parameters in edge optimization mode, *i.e.*, we are only using the `-optimize edges` switch since our algorithm uses the DAG representation by default. In contrast, the right column is a result of running TreeRePair with the `-no_dag` and `-optimize edges` switches. Again, in Appendix A.3 on page 97, there is a detailed table listing all relevant characteristics of the runs of the two TreeRePair configurations on each test XML document.

Regarding the differences between the compression results of TreeRePair and the ones of the competing algorithms, it can be said that the DAG representation only has a minor impact on the compression performance of our algorithm. However, we can state that it drastically reduces the memory demands of TreeRePair — it slashes the memory consumption by a factor of 4. Interestingly, even without the DAG representation, TreeRePair uses only half as much main memory as Extended-Repair does (*cf.* Table 6.3). Furthermore, the DAG representation leads to a faster compression speed since it saves repetitive recalculations concerning equal subtrees.

6.7 Results with Different Maximal Ranks

We executed TreeRePair using the `-optimize edges` (*i.e.*, we enabled the edge optimization mode) and the `-max_rank` switches. Each time, we specified a different maximal rank for a nonterminal in order to get an insight into the influence of it concerning the compression performance. Table 6.6 shows that, regarding our set of test XML documents, a maximal rank of 4 leads to the best compression results on average.

At the same time, we can see that even when restricting the maximal rank to 1 TreeRePair performs better than BPLEX and Extended-Repair (*cf.* Table 6.3). The fact that large maximal ranks can lead to a worse compression ratio can be explained by the trees from Sect. 3.7 on page 39. Note that the trees from this section are basically long lists. Although this is not the case for our test trees, their shape is nevertheless similar to a list structure. In any case, its quite distinct from the shape of a full binary tree, where an unlimited maximal rank leads to the best compression ratio (*cf.* Sect. 3.6 on page 34).

Bibliography

[BGK03] Peter Buneman, Martin Grohe, and Christoph Koch. Path queries on compressed XML. In *VLDB 2003: Proceedings of the 29th international conference on very large data bases*, pages 141–152. VLDB Endowment, 2003.

[BHK10] Stefan Böttcher, Rita Hartel, and Christoph Krislin. CluX: Clustering XML sub-trees. In *ICEIS 2010: Proceedings of the 12th International Conference on Enterprise Information Systems*, 2010.

[BLM08] Giorgio Busatto, Markus Lohrey, and Sebastian Maneth. Efficient memory representation of XML document trees. *Information Systems*, 33(4-5):456 – 474, 2008.

[BPSM$^+$08] Tim Bray, Jean Paoli, C. Michael Sperberg-McQueen, Eve Maler, and François Yergeau. Extensible markup language (XML) 1.0. W3c recommendation, XML Core Working Group, World Wide Web Consortium, November 2008.

[CDG$^+$07] H. Comon, M. Dauchet, R. Gilleron, C. Löding, F. Jacquemard, D. Lugiez, S. Tison, and M. Tommasi. Tree automata techniques and applications. `http://www.grappa.univ-lille3.fr/tata`, 2007.

[CLL$^+$05] Moses Charikar, Eric Lehman, April Lehman, Ding Liu, Rina Panigrahy, Manoj Prabhakaran, Amit Sahai, and abhi shelat. The smallest grammar problem. *IEEE Trans. Inform. Theory*, 51(7):2554–2576, 2005.

[Deu96] P. Deutsch. DEFLATE compressed data format specification version 1.3. `http://tools.ietf.org/html/rfc1951`, 1996.

[FGK03] Markus Frick, Martin Grohe, and Christoph Koch. Query evaluation on compressed trees (extended abstract). In *LICS '03: Proceedings of the 18th Annual IEEE Symposium on Logic in Computer Science*, pages 188–197. IEEE Computer Society Press, 2003.

[Kri08] Christoph Krislin. Optimierung grammatik-basierter XML-Kompression. Diplomarbeit, Faculty for Electrical Engineering, Computer Science and Mathematics, University of Paderborn (Germany), 2008.

[LM00] N. Jesper Larsson and Alistair Moffat. Off-line dictionary-based compression. *Proceedings of the IEEE*, 88(11):1722–1732, 2000.

[LM06] Markus Lohrey and Sebastian Maneth. The complexity of tree automata and XPath on grammar-compressed trees. *Theoretical Computer Science*, 363(2):196 – 210, 2006. Implementation and Application of Automata, 10th International Conference on Implementation and Application of Automata (CIAA 2005).

[LMSS09] Markus Lohrey, Sebastian Maneth, and Manfred Schmidt-Schauss. Parameter reduction in grammar-compressed trees. In *Proceedings of FOSSACS 2009, number 5504 in Lecture Notes in Computer Science*, pages 212–226. Springer, 2009.

[LS00] H. Liefke and D. Suciu. XMill: an efficient compressor for XML data. In *Proceedings of the 2000 ACM SIGMOD international conference on Management of data*, page 164. ACM, 2000.

[MLMK05] Makoto Murata, Dongwon Lee, Murali Mani, and Kohsuke Kawaguchi. Taxonomy of XML schema languages using formal language theory. *ACM Transactions on Internet Technology*, 5(4):660–704, 2005.

[MMS08] Sebastian Maneth, Nikolay Mihaylov, and Sherif Sakr. XML tree structure compression. *International Workshop on Database and Expert Systems Applications*, pages 243–247, 2008.

[MSV03] Tova Milo, Dan Suciu, and Victor Vianu. Typechecking for XML transformers. *Journal of Computer and System Sciences*, 66(1):66 – 97, 2003.

[Nev02] Frank Neven. Automata theory for XML researchers. *SIGMOD Record*, 31(3):39–46, 2002.

[WLH07] Fangju Wang, Jing Li, and Hooman Homayounfar. A space efficient XML DOM parser. *Data & Knowledge Engineering*, 60(1):185 – 207, 2007.

Appendix A
Detailed Test Results

A.1 Optimization of Total Number of Edges

Algorithm	Edges	File size	#NTs	Time	Mem (MB)
		1998statistics			
TreeRePair	1.68%	0.20%	54	100ms	1
BPLEX	1.80%	0.34%	168	1.813s	295
E-Repair	1.69%	0.24%	37	7.518s	114
bin. mDAG	8.49%	-	31	-	-
mDAG	4.87%	-	15	-	-
		catalog-01			
TreeRePair	1.69%	0.10%	400	887ms	2
BPLEX	2.22%	0.22%	1251	6.548s	315
E-Repair	1.63%	0.12%	291	9.975s	279
bin. mDAG	3.10%	-	520	-	-
mDAG	3.80%	-	506	-	-
		catalog-02			
TreeRePair	1.11%	0.07%	965	9.409s	10
BPLEX	1.38%	0.11%	3045	30s	512
E-Repair	1.52%	0.11%	1499	42s	511
bin. mDAG	2.22%	-	805	-	-
mDAG	1.39%	-	792	-	-
		dblp			
TreeRePair	3.89%	0.59%	25250	43s	227
BPLEX	4.27%	0.73%	38712	57m 42s	1644
E-Repair	5.65%	0.68%	30430	4m 34s	510
bin. mDAG	19.36%	-	6592	-	-
mDAG	11.11%	-	3378	-	-
		dictionary-01			
TreeRePair	7.72%	1.54%	1676	1.010s	9
BPLEX	8.43%	2.37%	3994	44s	323
E-Repair	8.71%	1.83%	1248	16s	433
bin. mDAG	27.99%	-	2058	-	-
mDAG	21.07%	-	448	-	-

Algorithm	Edges	File size	#NTs	Time	Mem (MB)
		dictionary-02			
TreeRePair	5.92%	1.38%	9757	11s	69
BPLEX	6.58%	1.95%	23209	6m 12s	587
E-Repair	8.52%	1.83%	11672	1m 40s	494
bin. mDAG	24.93%	-	16281	-	-
mDAG	19.96%	-	2414	-	-
		EnWikiNew			
TreeRePair	2.29%	0.21%	667	1.585s	8
BPLEX	2.40%	0.30%	1369	35s	337
E-Repair	2.42%	0.24%	476	12s	347
bin. mDAG	17.31%	-	23	-	-
mDAG	8.67%	-	29	-	-
		EnWikiQuote			
TreeRePair	2.42%	0.21%	452	1.158s	7
BPLEX	2.56%	0.31%	985	25s	321
E-Repair	2.58%	0.26%	323	9.924s	290
bin. mDAG	18.14%	-	19	-	-
mDAG	9.09%	-	25	-	-
		EnWikiSource			
TreeRePair	1.10%	0.10%	861	4.927s	26
BPLEX	1.28%	0.16%	1895	1m 9s	418
E-Repair	1.82%	0.18%	1106	23s	500
bin. mDAG	17.52%	-	19	-	-
mDAG	8.77%	-	24	-	-
		EnWikiVersity			
TreeRePair	1.44%	0.13%	525	2.107s	12
BPLEX	1.53%	0.18%	1043	34s	347
E-Repair	1.61%	0.15%	423	12s	437
bin. mDAG	17.60%	-	19	-	-
mDAG	8.81%	-	24	-	-
		EnWikTionary			
TreeRePair	0.97%	0.11%	4535	36s	183
BPLEX	1.09%	0.14%	6402	8m 58s	1287
E-Repair	1.48%	0.15%	6315	1m 33s	540
bin. mDAG	17.32%	-	26	-	-
mDAG	8.66%	-	30	-	-
		EXI-Array			
TreeRePair	0.41%	0.03%	123	1.281s	14
BPLEX	0.65%	0.06%	383	42s	322
E-Repair	0.53%	0.05%	142	8.017s	320
bin. mDAG	56.51%	-	8	-	-
mDAG	42.20%	-	13	-	-

Algorithm	Edges	File size	#NTs	Time	Mem (MB)
EXI-factbook					
TreeRePair	2.35%	0.31%	145	271ms	2
BPLEX	4.11%	0.77%	1423	5.138s	298
E-Repair	2.58%	0.31%	146	11s	408
bin. mDAG	9.16%	-	236	-	-
mDAG	8.07%	-	293	-	-
EXI-Invoice					
TreeRePair	0.68%	0.21%	14	74ms	1
BPLEX	0.62%	0.30%	40	1.483s	293
E-Repair	0.93%	0.24%	20	4.689s	119
bin. mDAG	13.74%	-	6	-	-
mDAG	7.12%	-	15	-	-
EXI-Telecomp					
TreeRePair	0.07%	0.01%	21	780ms	3
BPLEX	0.06%	0.02%	47	9.684s	310
E-Repair	0.08%	0.02%	21	11s	452
bin. mDAG	11.15%	-	10	-	-
mDAG	5.59%	-	15	-	-
EXI-weblog					
TreeRePair	0.06%	0.01%	13	324ms	3
BPLEX	0.04%	0.01%	24	9.097s	303
E-Repair	0.05%	0.02%	11	7.868s	279
bin. mDAG	18.19%	-	2	-	-
mDAG	9.10%	-	2	-	-
JST_gene.chr1					
TreeRePair	1.84%	0.10%	354	874ms	3
BPLEX	2.19%	0.19%	1113	11s	315
E-Repair	2.99%	0.17%	126	8.006s	233
bin. mDAG	6.75%	-	114	-	-
mDAG	4.24%	-	76	-	-
JST_snp.chr1					
TreeRePair	1.51%	0.09%	856	3.150s	8
BPLEX	2.15%	0.21%	4193	31s	360
E-Repair	1.54%	0.10%	634	15s	445
bin. mDAG	6.20%	-	282	-	-
mDAG	3.59%	-	242	-	-
medline02n0328					
TreeRePair	4.13%	0.35%	9064	16s	79
BPLEX	5.17%	0.62%	33976	5m 52s	574
E-Repair	6.73%	0.54%	13010	1m 32s	479
bin. mDAG	25.84%	-	20013	-	-
mDAG	22.80%	-	3960	-	-

Algorithm	Edges	File size	#NTs	Time	Mem (MB)
		NCBI_gene.chr1			
TreeRePair	1.37%	0.09%	504	1.374s	4
BPLEX	2.38%	0.28%	3631	14s	327
E-Repair	1.68%	0.11%	328	10s	308
bin. mDAG	3.98%	-	605	-	-
mDAG	4.45%	-	436	-	-
		NCBI_snp.chr1			
TreeRePair	< 0.01%	< 0.01%	17	15s	80
BPLEX	< 0.01%	< 0.01%	23	2m 6s	770
E-Repair	0.03%	0.01%	291	37s	504
bin. mDAG	22.22%	-	2	-	-
mDAG	11.11%	-	2	-	-
		sprot39.dat			
TreeRePair	2.30%	0.38%	20224	43s	178
BPLEX	3.16%	0.79%	111167	14m 41s	1446
E-Repair	4.27%	0.59%	33102	3m 48s	499
bin. mDAG	13.18%	-	31116	-	-
mDAG	16.07%	-	10243	-	-
		treebank			
TreeRePair	20.72%	4.41%	32857	22s	164
BPLEX	23.29%	6.16%	76109	21m 27s	645
E-Repair	34.85%	6.03%	48358	6m 50s	526
bin. mDAG	59.42%	-	43586	-	-
mDAG	53.75%	-	24746	-	-

A.2 Optimization of File Size

Algorithm	Edges	File size	#NTs	Time	Mem (MB)
		1998statistics			
TreeRePair	1.77%	0.20%	35	109ms	1
BPLEX	2.19%	0.25%	27	2.018s	295
E-Repair	1.68%	0.24%	37	4.578s	108
bzip2	-	0.29%	-	229ms	4
gzip	-	0.81%	-	8ms	-
XMill	-	0.24%	-	2.728s	2
		catalog-01			
TreeRePair	1.76%	0.10%	279	898ms	2
BPLEX	2.23%	0.14%	342	6.834s	315
E-Repair	2.77%	0.19%	236	11s	349
bzip2	-	0.24%	-	2.701s	8
gzip	-	0.85%	-	51ms	-
XMill	-	0.11%	-	12s	2
		catalog-02			
TreeRePair	1.12%	0.07%	770	10s	10
BPLEX	1.27%	0.08%	948	32s	512
E-Repair	1.49%	0.12%	1692	47s	521
bzip2	-	0.23%	-	28s	8
gzip	-	0.81%	-	450ms	-
XMill	-	0.09%	-	1m 58s	12
		dblp			
TreeRePair	4.03%	0.58%	14533	43s	227
BPLEX	4.52%	0.65%	11693	61m 15s	1644
E-Repair	5.52%	0.68%	35125	42m 48s	516
bzip2	-	0.56%	-	1m 11s	8
gzip	-	1.30%	-	1.230s	-
XMill	-	0.53%	-	11m 36s	15
		dictionary-01			
TreeRePair	8.08%	1.47%	930	1.117s	9
BPLEX	9.67%	1.85%	1044	46s	323
E-Repair	8.51%	1.81%	1428	19s	462
bzip2	-	1.52%	-	1.313s	7
gzip	-	3.07%	-	39ms	-
XMill	-	1.49%	-	17s	2
		dictionary-02			
TreeRePair	6.15%	1.32%	5024	11s	69
BPLEX	7.56%	1.63%	5424	6m 12s	587
E-Repair	8.30%	1.81%	13698	1m 57s	475
bzip2	-	1.52%	-	15s	7
gzip	-	3.05%	-	279ms	-
XMill	-	1.49%	-	2m 41s	13

Algorithm	Edges	File size	#NTs	Time	Mem (MB)
		EnWikiNew			
TreeRePair	2.38%	0.20%	390	1.721s	8
BPLEX	2.63%	0.23%	335	35s	337
E-Repair	2.42%	0.24%	476	12s	369
bzip2	-	0.26%	-	2.999s	8
gzip	-	0.90%	-	57ms	-
XMill	-	0.23%	-	23s	2
		EnWikiQuote			
TreeRePair	2.51%	0.20%	274	1.195s	7
BPLEX	2.81%	0.23%	236	25s	321
E-Repair	2.58%	0.26%	323	10s	268
bzip2	-	0.28%	-	2.013s	8
gzip	-	0.93%	-	36ms	-
XMill	-	0.24%	-	15s	2
		EnWikiSource			
TreeRePair	1.14%	0.10%	515	5.025s	26
BPLEX	1.40%	0.13%	535	1m 10s	418
E-Repair	1.82%	0.18%	1127	23s	488
bzip2	-	0.16%	-	8.742s	8
gzip	-	0.63%	-	131ms	-
XMill	-	0.12%	-	1m 4s	9
		EnWikiVersity			
TreeRePair	1.50%	0.12%	303	2.244s	12
BPLEX	1.70%	0.15%	287	36s	347
E-Repair	1.61%	0.15%	423	13s	415
bzip2	-	0.19%	-	3.698s	8
gzip	-	0.69%	-	59ms	-
XMill	-	0.15%	-	28s	2
		EnWikTionary			
TreeRePair	1.00%	0.11%	2575	37s	183
BPLEX	1.15%	0.13%	2062	9m 13s	1287
E-Repair	1.48%	0.15%	6314	1m 40s	526
bzip2	-	0.17%	-	57s	8
gzip	-	0.68%	-	938ms	-
XMill	-	0.13%	-	7m 25s	15
		EXI-Array			
TreeRePair	0.44%	0.03%	75	1.393s	14
BPLEX	0.77%	0.05%	124	43s	322
E-Repair	0.51%	0.05%	155	7.833s	312
bzip2	-	0.05%	-	3.250s	8
gzip	-	0.37%	-	67ms	-
XMill	-	0.03%	-	10s	6

Algorithm	Edges	File size	#NTs	Time	Mem (MB)
EXI-factbook					
TreeRePair	2.51%	0.31%	99	356ms	2
BPLEX	6.44%	0.58%	170	5.333s	298
E-Repair	2.59%	0.31%	151	12s	438
bzip2	-	0.78%	-	854ms	8
gzip	-	1.10%	-	17ms	-
XMill	-	0.29%	-	5.248s	1
EXI-Invoice					
TreeRePair	0.72%	0.21%	11	147ms	2
BPLEX	0.78%	0.28%	8	1.406s	293
E-Repair	0.91%	0.24%	21	4.320s	113
bzip2	-	0.30%	-	191ms	3
gzip	-	0.64%	-	7ms	-
XMill	-	0.26%	-	1.256s	2
EXI-Telecomp					
TreeRePair	0.08%	0.01%	12	829ms	3
BPLEX	0.07%	0.02%	15	9.548s	310
E-Repair	0.08%	0.02%	24	13s	450
bzip2	-	0.09%	-	2.363s	8
gzip	-	0.45%	-	36ms	-
XMill	-	0.02%	-	11s	2
EXI-weblog					
TreeRePair	0.06%	0.01%	9	400ms	3
BPLEX	0.05%	0.01%	12	9.004s	303
E-Repair	0.05%	0.02%	12	7.942s	288
bzip2	-	0.06%	-	720ms	8
gzip	-	0.40%	-	14ms	-
XMill	-	0.02%	-	8.342s	2
JST_gene.chr1					
TreeRePair	1.91%	0.10%	227	906ms	3
BPLEX	2.42%	0.13%	211	11s	315
E-Repair	2.99%	0.17%	128	9.947s	211
bzip2	-	0.14%	-	2.599s	8
gzip	-	0.67%	-	43ms	-
XMill	-	0.10%	-	14s	2
JST_snp.chr1					
TreeRePair	1.58%	0.08%	537	3.213s	8
BPLEX	2.45%	0.14%	569	32s	360
E-Repair	1.51%	0.10%	673	15s	453
bzip2	-	0.18%	-	9.251s	8
gzip	-	0.79%	-	149ms	-
XMill	-	0.09%	-	40s	8

Algorithm	Edges	File size	#NTs	Time	Mem (MB)
		medline02n0328			
TreeRePair	4.32%	0.34%	4923	16s	79
BPLEX	6.47%	0.46%	6717	5m 45s	574
E-Repair	6.71%	0.54%	13243	1m 38s	477
bzip2	-	0.49%	-	31s	7
gzip	-	1.26%	-	544ms	-
XMill	-	0.34%	-	2m 13s	13
		NCBI_gene.chr1			
TreeRePair	1.43%	0.09%	354	1.442s	4
BPLEX	3.00%	0.16%	464	14s	327
E-Repair	1.66%	0.11%	342	10s	265
bzip2	-	0.15%	-	4.110s	8
gzip	-	0.71%	-	65ms	-
XMill	-	0.08%	-	21s	8
		NCBI_snp.chr1			
TreeRePair	< 0.01%	< 0.01%	11	15s	80
BPLEX	< 0.01%	< 0.01%	15	2m 6s	770
E-Repair	0.03%	0.01%	292	33s	465
bzip2	-	0.03%	-	40s	8
gzip	-	0.39%	-	578ms	-
XMill	-	0.00%	-	3m 45s	14
		sprot39.dat			
TreeRePair	2.41%	0.37%	11699	43s	178
BPLEX	4.33%	0.53%	11783	13m 43s	1446
E-Repair	4.25%	0.59%	33700	3m 59s	497
bzip2	-	0.45%	-	1m 11s	8
gzip	-	1.20%	-	1.122s	-
XMill	-	0.36%	-	9m 52s	15
		treebank			
TreeRePair	21.59%	4.28%	17186	22s	164
BPLEX	26.21%	5.37%	21302	21m 36s	646
E-Repair	34.53%	6.01%	51470	7m 44s	514
bzip2	-	5.26%	-	6.407s	7
gzip	-	9.65%	-	843ms	-
XMill	-	4.51%	-	1m 36s	12

A.3 Without Using DAG Representation

Algorithm	Edges	File size	#NTs	Time	Mem (MB)
1998statistics					
Without DAG	1.62%	0.20%	53	121ms	4
With DAG	1.68%	0.20%	54	214ms	1
catalog-01					
Without DAG	1.69%	0.10%	400	1.381s	20
With DAG	1.69%	0.10%	400	1.022s	3
catalog-02					
Without DAG	1.11%	0.07%	967	15s	199
With DAG	1.11%	0.07%	965	9.584s	10
dblp					
Without DAG	3.89%	0.59%	25039	55s	1015
With DAG	3.89%	0.59%	25250	44s	227
dictionary-01					
Without DAG	7.63%	1.51%	1622	1.238s	25
With DAG	7.72%	1.54%	1676	1.044s	9
dictionary-02					
Without DAG	5.88%	1.36%	9390	12s	238
With DAG	5.92%	1.38%	9757	11s	69
EnWikiNew					
Without DAG	2.28%	0.21%	656	2.042s	37
With DAG	2.29%	0.21%	667	1.732s	8
EnWikiQuote					
Without DAG	2.41%	0.21%	458	1.320s	24
With DAG	2.42%	0.21%	452	1.223s	7
EnWikiSource					
Without DAG	1.09%	0.10%	863	5.652s	101
With DAG	1.10%	0.10%	861	5.087s	26
EnWikiVersity					
Without DAG	1.43%	0.13%	522	2.472s	45
With DAG	1.44%	0.13%	525	2.229s	12
EnWikTionary					
Without DAG	0.97%	0.11%	4539	42s	743
With DAG	0.97%	0.11%	4535	38s	183
EXI-Array					
Without DAG	0.40%	0.03%	122	1.378s	21
With DAG	0.41%	0.03%	123	1.394s	14
EXI-factbook					
Without DAG	2.34%	0.31%	144	331ms	6
With DAG	2.35%	0.31%	145	330ms	2

Algorithm	Edges	File size	#NTs	Time	Mem (MB)
		EXI-Invoice			
Without DAG	0.61%	0.21%	12	85ms	3
With DAG	0.68%	0.21%	14	124ms	1
		EXI-Telecomp			
Without DAG	0.06%	0.01%	17	1.132s	17
With DAG	0.07%	0.01%	21	850ms	3
		EXI-weblog			
Without DAG	0.05%	0.01%	10	607ms	10
With DAG	0.06%	0.01%	13	400ms	3
		JST_gene.chr1			
Without DAG	1.73%	0.09%	299	1.365s	21
With DAG	1.84%	0.10%	354	910ms	3
		JST_snp.chr1			
Without DAG	1.50%	0.09%	841	4.187s	59
With DAG	1.51%	0.09%	856	3.287s	8
		medline02n0328			
Without DAG	4.11%	0.34%	8524	17s	235
With DAG	4.13%	0.35%	9064	17s	79
		NCBI_gene.chr1			
Without DAG	1.37%	0.09%	486	1.959s	32
With DAG	1.37%	0.09%	504	1.498s	4
		NCBI_snp.chr1			
Without DAG	< 0.01%	< 0.01%	13	18s	337
With DAG	< 0.01%	< 0.01%	17	15s	80
		sprot39.dat			
Without DAG	2.31%	0.37%	18516	55s	936
With DAG	2.30%	0.38%	20224	44s	178
		treebank			
Without DAG	20.71%	4.41%	32786	14s	215
With DAG	20.72%	4.41%	32857	22s	164

Eidesstattliche Erklärung

Ich versichere, dass ich die vorliegende Arbeit selbständig und nur unter Verwendung der angegebenen Quellen und Hilfsmittel angefertigt habe, insbesondere sind wörtliche oder sinngemäße Zitate als solche gekennzeichnet. Mir ist bekannt, dass Zuwiderhandlung auch nachträglich zur Aberkennung des Abschlusses führen kann.

Ort, Datum

Unterschrift