Forschungsbericht der
Technischen Fakultät
Abteilung Informationstechnik

Affix Trees

Jens Stoye

Report 2000-04

Universität Bielefeld • Postfach 10 01 31 • 33501 Bielefeld • Germany

Impressum: Herausgeber:
Robert Giegerich, Alois Knoll, Peter Ladkin,
Helge Ritter, Gerhard Sagerer, Ipke Wachsmuth

Technische Fakultät der Universität Bielefeld,
Abteilung Informationstechnik, Postfach 10 01 31,
33501 Bielefeld, FRG

ISSN 0946-7831

Diplomarbeit

Affix Trees

Jens Stoye

Technische Fakultät

Universität Bielefeld

This is the English translation of my Diploma Thesis "Affixbäume", written in 1995.

The translation of this work was initiated by the fact that Moritz Maass was able to solve the main open question posed in this thesis, namely to find a linear-time algorithm for the online construction of affix trees that is to be published in the Proceedings of the 11th Annual Symposium on Combinatorial Pattern Matching, CPM 2000.

Jens Stoye
April, 2000

Contents

1 Introduction **1**

2 Basic Data Structures **3**

2.1 Basic Notions . . . 3

2.2 $\mathcal{A}^+$-Trees and Related Data Structures . . . 4

2.2.1 $\mathcal{A}^+$-Trees . . . 4

2.2.2 Suffix Trees . . . 6

2.2.3 Suffix Link . . . 8

2.2.4 Duality of $\mathcal{A}^+$-trees . . . 9

2.2.5 Duality of Suffix Trees . . . 9

2.3 Bi-Trees . . . 10

2.3.1 Definition of the Data Structure . . . 10

2.3.2 Duality of Bi-Trees . . . 13

3 Affix Trees **15**

3.1 Definition of the Data Structure . . . 15

3.2 Space Usage of Affix Trees . . . 17

3.2.1 Asymptotic Space Usage . . . 17

3.2.2 Maximal Space Usage of Compact Affix Trees . . . 18

3.2.3 Expected Space Usage . . . 20

3.3 Representation of Affix Trees . . . 21

3.3.1 Edge vs. Node Labels . . . 22

3.3.2 Node Labels . . . 23

3.4 Neuralgic Points in the Affix Tree . . . 23

3.4.1 Active Suffix and Active Prefix . . . 23

3.4.2 Active Suffix Leaf and Active Prefix Leaf . . . 24

3.5 Construction of Compact Affix Trees . . . 25

3.5.1 Construction of Compact Affix Trees by Edge Contraction . . . 25

3.5.2 Construction of Compact Affix Trees by Reverse Union of Suffix and Reverse Prefix Tree 26
3.6 Applications of Affix Trees 26

4 Online Construction of Compact Affix Trees 28
4.1 Differences Between $cat(t)$ and $cat(ta)$ 28
4.1.1 Changes of the Active Suffix and the Active Prefix 29
4.1.2 Changes of the Node Set 32
4.2 Operational Consequences 33
4.2.1 Elongation of the Text and Insertion of Node $\overrightarrow{t}$ 34
4.2.2 Insertion of the Relevant Suffixes 36
4.2.3 Making the Active Suffix Explicit 40
4.2.4 Elongation of the Active Prefix 42
4.3 The Online-Algorithm (Unidirectional) 43
4.4 Comments on the Bidirectional Application 44

5 Complexity Analysis 45
5.1 Overview 45
5.2 Discussion of the Difficult Cases 47
5.2.1 *find_aLink* 47
5.2.2 *canonize_down* 49
5.2.3 *canonize_up* 49
5.2.4 *find_pLoc* 49
5.3 Empirical Results 50
5.4 Computation Times 55

6 The Program *bigrep* 57

7 Summary 58

Bibliography 60

Chapter 1

Introduction

Pattern matching is one of the classic fields in theoretical computer science. Pattern matching algorithms include methods for finding (short) patterns in (long) texts as they are used, for example, in linguistics and bioinformatics.

One usually distinguishes between exact and approximate pattern matching. In approximate matching one seeks for occurrences of the pattern in the text within a certain error bound, while exact matching requires that the pattern and a part of the text exactly coincide.

Within exact string matching, depending on the situation, different techniques are applied: If pattern and text are not known in advance, one uses methods like those based on the Boyer-Moore algorithm [BM77] where the pattern is preprocessed in a simple way to allow for an efficient search. If the pattern is known in advance and the same pattern is to be searched in different texts, techniques based on finite automata are the methods of choice. In the opposite case where the text is statically known and the pattern varies from search to search, a preprocessing of the text is worthwhile.

A data structure frequently used for preprocessing texts are (compact) suffix trees. They allow in a particularly useful way to store the subwords of the text (hence they are also called *subword trees*). Apostolico describes the virtues of this data structure as follows:

> "..., no digital index seems to outperform subword trees in versatility and elegance." [Apo85]

Different methods for the efficient construction of compact suffix trees are known, starting with the "classic" one by Weiner [Wei73], followed by methods of McCreight [McC76] and Chen and Seiferas [CS85], and finally the *online* construction of Ukkonen [Ukk93] where the text is read character by character from left to right and after the ith step the suffix tree of the first i characters is completed.

Upon closer inspection of this procedure, one may ask whether reading the text from right to left, or even in a bidirectional way, would be possible as well. In the latter case, the construction could start at any position in the text and then proceed "inside out" in both directions. In the present work we will show that an extension of the data structure is advantageous here. The graph obtained is called an affix tree and has a duality property described in [GK94a] that is not possessed by compact suffix trees. Since the size of affix trees does not significantly differ from the size of suffix trees, we will examine if these, like suffix trees, can be constructed in time proportional to the length of the text.

This work is based on ideas presented in a paper by R. Giegerich and S. Kurtz [GK94a]. For the extension of Ukkonen's algorithm we use a notation similar to the one in [Kur95].

The thesis has the following structure: We first introduce the basic data structures in Chapter 2. On the basis of these data structures we define affix trees in Chapter 3 and show a few of their properties. The fourth chapter is devoted to the online construction of affix trees. In the fifth chapter we analyze the complexity of this algorithm, without a satisfactory result, though. Finally, a short application of affix trees is presented in Chapter 6.

The code of the programs written as part of this thesis can be obtained from the author upon request.

Chapter 2

Basic Data Structures

Giegerich and Kurtz [GK94a] observe a duality of atomic suffix trees and the respective suffix link trees. To show that this duality is a property not only of suffix link trees, here we go a more general way than via suffix links. From the beginning we demand a symmetry from the resulting data structure that will be of high importance later on.

The notation and terminology introduced in this section follow [GK94a] and [Kur95].

2.1 Basic Notions

Let $\mathcal{A}$ be a finite set of characters, the *alphabet*. A sequence of characters from $\mathcal{A}$ is called a *string* or *text* over $\mathcal{A}$, $|t|$ denotes the *length* of text t, i. e., the number of characters t consists of. The empty string of length zero is denoted ε. $\mathcal{A}^m$ is the set of all strings of length m over $\mathcal{A}$, $\mathcal{A}^*$ is the set of all strings over $\mathcal{A}$, $\mathcal{A}^+ = \mathcal{A}^* \setminus \{\varepsilon\}$. Throughout this work, a, b, c, x, y denote characters from $\mathcal{A}$; s, t, u, v, w denote strings over $\mathcal{A}$. Characters themselves are written in type writer font: `a`, `b`, `c`, $\ldots$.

The m-fold repetition of a string t is denoted t^m, the *reverse* $a_n \ldots a_1$ of string $t = a_1 \ldots a_n$ is denoted t^{-1}. For any partitioning of $t = vwu$ into (possibly empty) v, w and u, v is called a *prefix* of t, w is called a *substring* of t (or a *t-word*), and u is called a *suffix* of t. If $v \neq t$ ($u \neq t$), v (u) is a *proper prefix* (*proper suffix*) of t. The set of all substrings of t is denoted *t-words*.

If the variable names for t-words are irrelevant, these will also be symbolized by underlined spaces:

$$\begin{aligned} t = v\_\_ &\iff v \text{ is a prefix of } t,\\ t = \_w\_ &\iff w \text{ is a } t\text{-word},\\ t = \_\_u &\iff u \text{ is a suffix of } t. \end{aligned}$$

A dot in this notation denotes a single character from $\mathcal{A}$:

$$\begin{aligned} t = v\_\_. &\iff v \text{ is a proper prefix of } t,\\ t = .\_\_u &\iff u \text{ is a proper suffix of } t. \end{aligned}$$

For the start/end of a substring $w = a_{l+1} \dots a_r$ of $t = a_1 \dots a_n$ we use the notation $t = \_\__l w_r\_\_$ where $0 \leq l \leq r \leq n$.[1]

A prefix or suffix of t is called *nested* if it also occurs at another position in t:[2]

$$t = \left\{ \begin{matrix} v\_\_ \\ .\_v\_ \end{matrix} \right\} \quad \Longleftrightarrow \quad v \text{ is a nested prefix of } t,$$

$$t = \left\{ \begin{matrix} \_\_u \\ \_u\_. \end{matrix} \right\} \quad \Longleftrightarrow \quad u \text{ is a nested suffix of } t.$$

A prefix (suffix) that is nested in t hence always is a proper prefix (suffix) of t. A *non-nested prefix* (*suffix*) of t is a prefix (suffix) of t that is not nested.[3]

A t-word w is called *right-branching* resp. *left-branching* if t is of the form

$$t = \left\{ \begin{matrix} \_wx\_\_ \\ \_\_wy\_ \end{matrix} \right\}, \; x \neq y \quad \text{resp.} \quad t = \left\{ \begin{matrix} \_xw\_\_ \\ \_\_yw\_ \end{matrix} \right\}, \; x \neq y.$$

By definition, the empty string ε is both left- and right-branching in every string.

2.2 $\mathcal{A}^+$-Trees and Related Data Structures

2.2.1 $\mathcal{A}^+$-Trees

Following [GK94a] we introduce a large part of the suffix tree terminology for a more general data structure, the $\mathcal{A}^+$-tree.

Definition 2.1 ($\mathcal{A}^+$-Tree)
An $\mathcal{A}^+$-tree T is a rooted tree whose edges are labeled with strings from $\mathcal{A}^+$. Each node k in T has at most one outgoing edge $k \xrightarrow{a\_} \bullet$[4] whose label starts with an a, a so-called *a-edge.* □

Terminology:

- The set of all edges of an $\mathcal{A}^+$-tree T is denoted *edges*(T), the set of all nodes is denoted *nodes*(T), and the root of T is denoted *root*(T). A node in *nodes*(T) is a *leaf* of T if there is no edge starting at that node. All other nodes are *internal nodes* of T.
- The number of nodes $|nodes(T)|$ of an $\mathcal{A}^+$-tree T is called the *size* of T, short $|T|$.
- An internal node is called *branching* if there are at least two edges starting at that node, otherwise it is called *non-branching.*

[1] Note that the indices l and r do not denote characters of t but the borders between characters. For a motivation of this notation see [Mei86].

[2] The notation $t = \left\{ \begin{matrix} w_1 \\ w_2 \end{matrix} \right\}$ denotes that t is both of the form $t = w_1$ and of the form $t = w_2$.

[3] But not an arbitrary string $w \in \mathcal{A}^*$ that is not a nested prefix (suffix) of t!

[4] We will use the variable k to denote nodes ("Knoten" in German), in some cases we will use longer names. In cases where the name of a node is irrelevant, it may be denoted by a bullet $\bullet$. Here this is the node where the a-edge ends.

- For a given $\mathcal{A}^+$-tree T, $s\text{-}path(k)$ denotes the concatenation of the edge labels on the (unique) path from the root to node k. By $p\text{-}path(k)$ we denote the concatenation of the reversed edge labels on the path from node k to the root of T.[5]

  Obviously, $s\text{-}path(k) = (p\text{-}path(k))^{-1}$.

- Conversely, due to the unique a-edges in an $\mathcal{A}^+$-tree T there exists for a given string w at most one node k in T with $s\text{-}path(k) = w$ and at most one node k with $p\text{-}path(k) = w$. Due to this correspondence every node k can be identified with its s-path or its p-path. We use two interchangeable notations:

  1. $k = \overrightarrow{w}$ if $w = s\text{-}path(k)$.
  2. $k = \overleftarrow{w}$ if $w = p\text{-}path(k)$.

- It is easy to see that for any node $\overrightarrow{v}$ on the path from the root to a node $\overrightarrow{w}$, v is a prefix of w. Moreover, due to the uniqueness of a-edges T does not contain other nodes that represent prefixes of w.
  Similar properties hold for $\overleftarrow{w}$: On the path from the root to node $\overleftarrow{w}$ we pass only and all nodes $\overleftarrow{v}$ in T with $w = \_v$.

- A string w is *represented as an s-word* in T if T contains a node $\overrightarrow{w\_}$. The set of all strings represented in T as s-words is denoted $s\text{-}words(T)$.
  Similarly, $p\text{-}words(T) = \{w \in \mathcal{A}^* | \overleftarrow{\_w} \in nodes(T)\}$.

  Obviously, we have that $w \in s\text{-}words(T)$ if and only if $w^{-1} \in p\text{-}words(T)$.

- Following [Ukk93], given a string $w \in s\text{-}words(T)$ we call the pair $(\overrightarrow{v}, u)$ a *reference pair* of w in T if $\overrightarrow{v}$ is a node in T and $w = vu$.[6] If v is the longest such prefix of w, $loc_T(w) = (\overrightarrow{v}, u)$ is called the *canonical reference pair* or *location* of w in T.

- A canonical reference pair of the form $(\overrightarrow{v}, \varepsilon)$ is called an *explicit node*; a canonical reference pair of the form $(\overrightarrow{v}, au)$ is called an *implicit node* because a node $\overrightarrow{vau}$ does not explicitly exist in T, but it can be interpreted as implicitly occurring "inside" the edge $\overrightarrow{v} \xrightarrow{au\_} \overrightarrow{vau\_}$.

  Sometimes it will be useful to extend the notation of canonic reference pairs to quadruples:
  $$loc_T(w) = (\overrightarrow{v}, au, s, \overrightarrow{vaus})$$
  if $w = vau$ and $(\overrightarrow{v}, au)$ is an implicit node "inside" the edge $\overrightarrow{v} \xrightarrow{aus} \overrightarrow{vaus}$ of T.

- The three trees T_a, T_i and T_c in Figure 2.1 show that the set of strings represented in an $\mathcal{A}^+$-tree T does not uniquely determine the structure of T. T_a, T_i and T_c represent the same sets of strings

$$s\text{-}words(T_a) = s\text{-}words(T_i) = s\text{-}words(T_c) = \{\varepsilon, \mathtt{a}, \mathtt{aa}, \mathtt{aab}, \mathtt{aaba}, \mathtt{b}, \mathtt{ba}, \mathtt{bab}, \mathtt{bb}\}$$

[5] The terms *s-path* and *p-path* are an anticipation of suffix and reverse prefix trees as they will be introduced in Section 2.2.2. Usually, one considers paths of the s-kind in suffix trees while paths of the p-kind are used in reverse prefix trees.

[6] Similarly one can define reference pairs for $w \in p\text{-}words(T)$. The result, though, does not differ significantly from the one above: $(\overleftarrow{u}, v)$ is a reference pair if $\overleftarrow{u}$ is a node in T and $w = vu$.

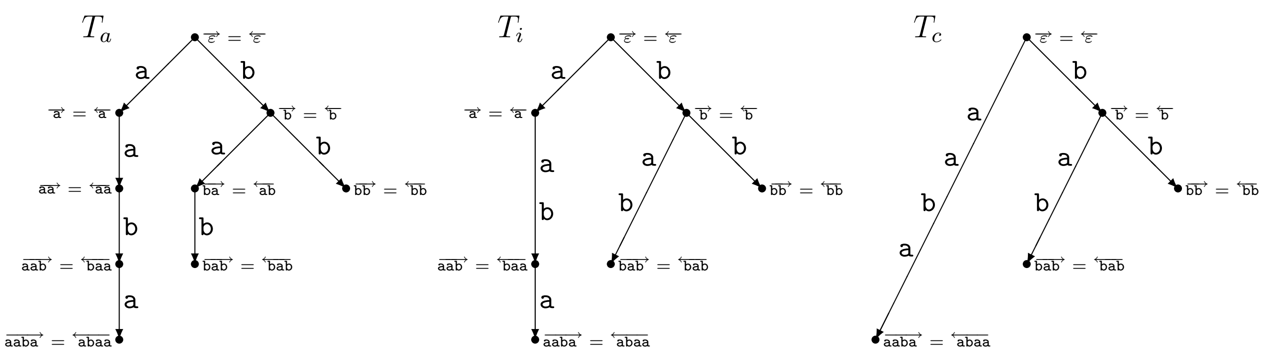


Figure 2.1: Three $\mathcal{A}^+$-trees that represent the same set of strings: on the left an atomic tree T_a, on the right a compact tree T_c, and in between an intermediate tree T_i.

and

$$p\text{-}words(T_a) = p\text{-}words(T_i) = p\text{-}words(T_c) = \{\varepsilon, \texttt{a}, \texttt{aa}, \texttt{ab}, \texttt{abaa}, \texttt{b}, \texttt{baa}, \texttt{bab}, \texttt{bb}\},$$

respectively.

- T_a only contains edges with labels of length 1. Such edges are called *atomic edges*. An $\mathcal{A}^+$-tree that only contains atomic edges is called an *atomic $\mathcal{A}^+$-tree*.

  On the other hand, an $\mathcal{A}^+$-tree that apart from the root only contains branching internal nodes is called a *compact $\mathcal{A}^+$-tree*. The tree T_c shown in Figure 2.1 (right) is compact.

- As shown in [GK95], among all $\mathcal{A}^+$-trees representing a given set of strings the atomic variant contains the maximal number of explicit nodes and the compact variant contains the minimal number of explicit nodes. The atomic $\mathcal{A}^+$-tree is a normal form under edge splitting, and the compact $\mathcal{A}^+$-tree is a normal form under edge contraction.

2.2.2 Suffix Trees

We introduce suffix trees as a special form of $\mathcal{A}^+$-trees. Hence the notation introduced above carries over automatically.

Definition 2.2 (Suffix Tree)
A *suffix tree* of a text t is an $\mathcal{A}^+$-tree T with $s\text{-}words(T) = t\text{-}words$. □

Remarks on Definition 2.2

1. Like all $\mathcal{A}^+$-trees, suffix trees can vary in the degree of compactness. Figure 2.2 shows the atomic suffix tree $ast(t)$ and the compact suffix tree $cst(t)$ of the text $t = \texttt{aababa}$.

2. As shown in [AHU74], using a proper representation of edge labels, suffix trees of a text t of length $n = |t|$ require the following amounts of space:

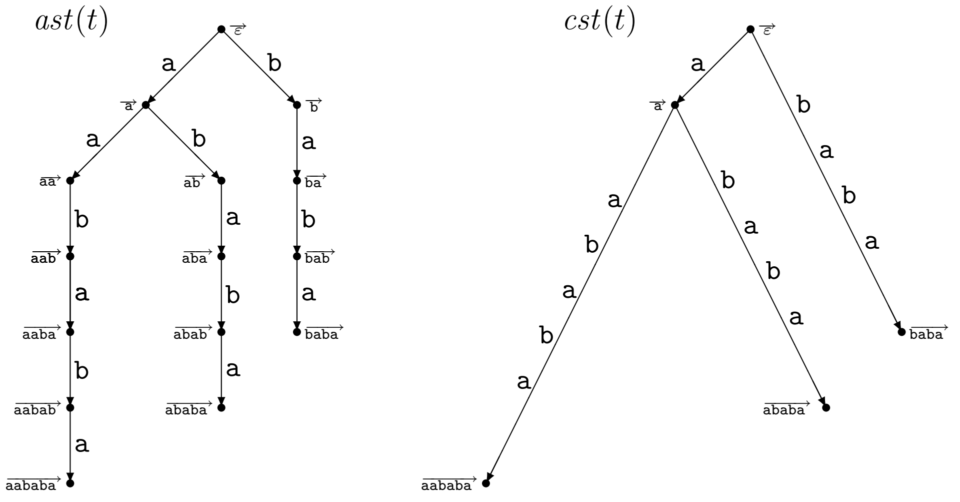


Figure 2.2: Atomic and compact suffix tree of $t = \texttt{aababa}$.

- $\mathcal{O}(n^2)$ for the atomic suffix tree $ast(t)$,
- $\mathcal{O}(n)$ for the compact suffix tree $cst(t)$.

That is why for realistic applications, especially for long texts, the compact variant is to be preferred.

3. An important property of the compact suffix tree, also mentioned by [CL94], is the following:

   I. Each internal node $\overrightarrow{w}$ of $cst(t)$ corresponds to a right-branching t-word w.[7]

   II. Each leaf $\overrightarrow{w}$ of $cst(t)$ corresponds to a non-nested suffix w of t.[8]

Similar to the suffix tree definition is the following:

Definition 2.3 (Reverse Prefix Tree)
A *reverse prefix tree* of a text t is an $\mathcal{A}^+$-tree T with $p\text{-}words(T) = t\text{-}words$. □

This definition is equivalent to the one given in [GK94a]:

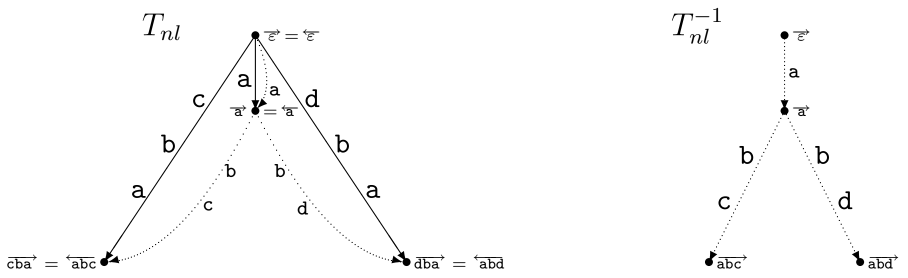
> A reverse prefix tree of a text t is a suffix tree for the reverse text t^{-1}

because a string $w \in \mathcal{A}^*$ is a t-word if and only if w^{-1} is a t^{-1}-word.

Hence we do not need to introduce a separate notation for prefix trees: The atomic prefix tree of a text t equals $ast(t^{-1})$, and the compact reverse prefix tree of t equals $cst(t^{-1})$.

[7]Because we consider ε being right-branching by definition, $root(cst(t))$ representing ε is automatically included in case I., whereas [CL94] introduce a special case for the root.

[8][CL94] only consider texts t that end with a special character \$ that does not occur elsewhere in t. Hence, they observe a general correspondence of the suffixes of t and the leaves of the suffix tree.



T_{nl}
$\overrightarrow{\varepsilon} = \overleftarrow{\varepsilon}$
$\overrightarrow{a} = \overleftarrow{a}$
$\overrightarrow{cba} = \overleftarrow{abc}$
$\overrightarrow{dba} = \overleftarrow{abd}$
T_{nl}^{-1}
$\overrightarrow{\varepsilon}$
$\overrightarrow{a}$
$\overrightarrow{abc}$
$\overrightarrow{abd}$

Figure 2.3: An $\mathcal{A}^+$-tree T_{nl} with suffix links and its suffix link tree T_{nl}^{-1} which is not an $\mathcal{A}^+$-tree.

2.2.3 Suffix Link

Important for the efficient construction of suffix trees and several applications is an additional kind of edges which lie somewhat orthogonally to the original tree structure. While these "suffix links" were first introduced by [McC76] only for suffix trees, like [GK94a] we define them more generally for $\mathcal{A}^+$-trees:

Definition 2.4 (Suffix Links)
Let $\overrightarrow{aw}$ be a node in an $\mathcal{A}^+$-tree T. The *suffix link* of $\overrightarrow{aw}$ is the edge $\overrightarrow{aw} \dashleftarrow \overrightarrow{v}$ where v is the longest suffix of w such that $\overrightarrow{v}$ is also a node in T.[9] For distinction we will print suffix links with dotted lines.

Suffix link $\overrightarrow{aw} \dashleftarrow \overrightarrow{v}$ may be labeled with the characters that are removed from node $\overrightarrow{aw} = \overrightarrow{uv}$ to node $\overrightarrow{v}$ in reverted order: $\overrightarrow{uv} \overset{u^{-1}}{\dashleftarrow} \overrightarrow{v}$. □

Remarks[10] on Definition 2.4

1. Suffix links play an important role in the efficient construction of suffix trees whose most efficient versions are [McC76] and [Ukk93]:

   > "All clever variations of subword trees are built in linear time by resorting to similar properties." [Apo85]

2. The suffix links of an $\mathcal{A}^+$-tree T themselves form a tree, the *suffix link tree* of T, denoted T^{-1}. In general the suffix link tree is not an $\mathcal{A}^+$-tree, as the example T_{nl}^{-1} in Figure 2.3 shows.

3. In an atomic suffix tree, all suffix links are atomic. In a compact suffix tree, the suffix links of internal nodes are atomic.

[9]Compared to the definition in [GK94a], our suffix links are in the opposite direction so that they form a tree without having to be turned. Nevertheless, we will say that the suffix link "starts" at node $\overrightarrow{aw}$ and "ends" at node $\overrightarrow{v}$.

[10]From [GK94a].

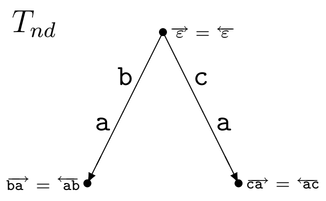

T_{nd}
$\overrightarrow{\varepsilon} = \overleftarrow{\varepsilon}$
b
a
c
a
$\overrightarrow{\mathtt{ba}} = \overleftarrow{\mathtt{ab}}$
$\overrightarrow{\mathtt{ca}} = \overleftarrow{\mathtt{ac}}$

Figure 2.4: An $\mathcal{A}^+$-tree T_{nd} for which a dual $\mathcal{A}^+$-tree does not exist.

2.2.4 Duality of $\mathcal{A}^+$-trees

This section introduces a general notion of duality for two $\mathcal{A}^+$-trees:

Definition 2.5 (Duality of two $\mathcal{A}^+$-trees)
Two $\mathcal{A}^+$-trees T_1 and T_2 are *dual* if for every node $\overrightarrow{w}$ in T_1 there exists a node $\overleftarrow{w}$ in T_2 and vice versa. □

Remarks on Definition 2.5

1. Not for every $\mathcal{A}^+$-tree there exists a dual $\mathcal{A}^+$-tree. For example, an $\mathcal{A}^+$-tree that is dual to the tree T_{nd} shown in Figure 2.4 would need to contain nodes $\overrightarrow{\mathtt{ab}}$ and $\overrightarrow{\mathtt{ac}}$. An $\mathcal{A}^+$-tree containing these two nodes, however, also contains the node $\overrightarrow{\mathtt{a}}$. But then it can not be dual to T_{nd} because $\overleftarrow{\mathtt{a}} \notin \mathit{nodes}(T_{nd})$.

2. If there exists a dual tree for an $\mathcal{A}^+$-tree T then this is the suffix link tree of T.

The proof of the second remark will be given in Section 2.3.2 (Lemma 2.12) because the argumentation will be easier with the notation introduced until there.

2.2.5 Duality of Suffix Trees

The duality properties described next are from [GK94a] where also the proof of Theorem 2.7 can be found.

Theorem 2.6 (Duality of Atomic Suffix Trees)
The dual $\mathcal{A}^+$-tree of an atomic suffix tree $\mathit{ast}(t)$ always exists and equals the atomic reverse prefix tree $\mathit{ast}(t^{-1})$. □

After the above remarks this is obvious since every t-word is represented by an explicit node $\overrightarrow{w}$ in $\mathit{ast}(t)$ and by an explicit node $\overleftarrow{w}$ in $\mathit{ast}(t^{-1})$.

For compact suffix trees only a weaker duality holds because in $\mathit{cst}(t)$ not for every node $\overrightarrow{w}$ there is a corresponding node $\overleftarrow{w}$ in $\mathit{cst}(t^{-1})$:

Theorem 2.7 (Weak Duality of Compact Suffix Trees)
The suffix link tree $(\mathit{cst}(t))^{-1}$ of a compact suffix tree $\mathit{cst}(t)$ is an $\mathcal{A}^+$-tree whose node set is a subset of the node set of $\mathit{cst}(t^{-1})$. Moreover, we have: $((\mathit{cst}(t))^{-1})^{-1} = \mathit{cst}(t)$. □

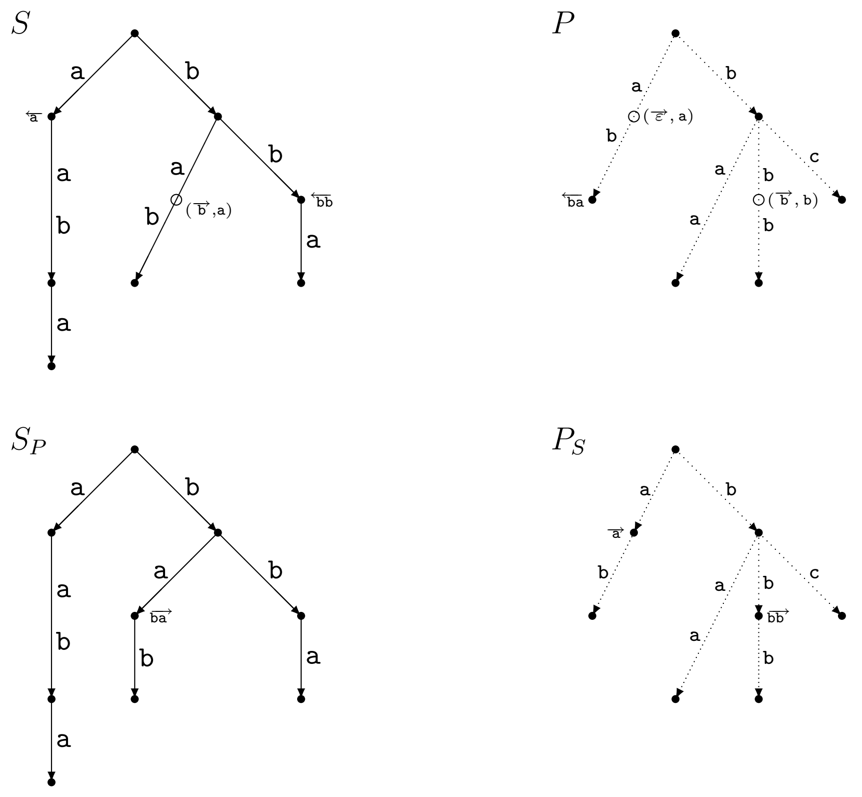


Figure 2.5: Two $\mathcal{A}^+$-trees S and P and their mutual extensions S_P and P_S.

This observation raises the question if there is a possibility to combine the efficiency of compact suffix trees with the symmetric properties of dual trees. As we will see in Chapter 3, affix trees are an elegant solution.

2.3 Bi-Trees

In spite of its name, strictly spoken the affix tree is not a tree but a special case of a bi-tree, a graph formed by combination of two trees. Bi-trees will now be introduced as the last of our basic data structures.

2.3.1 Definition of the Data Structure

Definition 2.8 (Extension of $\mathcal{A}^+$-Trees)
Let T and U be two $\mathcal{A}^+$-trees. The *U-extension* of T, denoted T_U, is the $\mathcal{A}^+$-tree that is obtained from T if one introduces in T an explicit node $\overrightarrow{wau}$ whenever $(\overrightarrow{w}, au)$ is an implicit node in T and $\overleftarrow{wau}$ is an explicit node in U. □

Figure 2.5 shows two $\mathcal{A}^+$-trees S and P and their mutual extensions S_P and P_S.

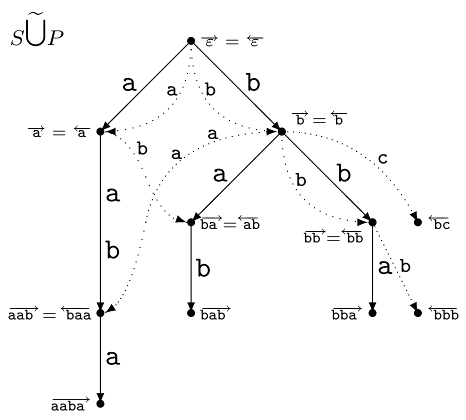

$_S\widetilde{\bigcup}_P$
$\overrightarrow{\varepsilon} = \overleftarrow{\varepsilon}$
$\overrightarrow{\mathtt{a}} = \overleftarrow{\mathtt{a}}$
$\overrightarrow{\mathtt{b}} = \overleftarrow{\mathtt{b}}$
$\overrightarrow{\mathtt{ba}} = \overleftarrow{\mathtt{ab}}$
$\overrightarrow{\mathtt{bb}} = \overleftarrow{\mathtt{bb}}$
$\overleftarrow{\mathtt{bc}}$
$\overrightarrow{\mathtt{aab}} = \overleftarrow{\mathtt{baa}}$
$\overrightarrow{\mathtt{bab}}$
$\overrightarrow{\mathtt{bba}}$
$\overleftarrow{\mathtt{bbb}}$
$\overrightarrow{\mathtt{aaba}}$
a
b
c

Figure 2.6: The reverse union $_S\widetilde{\bigcup}_P$ of the two $\mathcal{A}^+$-trees S and P from Figure 2.5.

Remarks on Definition 2.8

1. The set of strings represented by an $\mathcal{A}^+$-tree remains unchanged upon extension.
2. Atomic $\mathcal{A}^+$-trees are invariant upon extension because they do not contain implicit nodes.

Definition 2.9 (Reverse Union of two $\mathcal{A}^+$-trees, Bi-tree)
Let S and P be two $\mathcal{A}^+$-trees. The *reverse union* of S and P, denoted $_S\widetilde{\bigcup}_P$, is defined as follows:

1. $_S\widetilde{\bigcup}_P$ contains all nodes of S_P and P_S with the constraint that a node $\overrightarrow{w} \in \mathit{nodes}(S_P)$ is identified with a node $\overleftarrow{w} \in \mathit{nodes}(P_S)$ yielding a node $\overrightarrow{w} = \overleftarrow{w^{-1}}$:[11]

$$\mathit{nodes}(_S\widetilde{\bigcup}_P) = \mathit{nodes}(S_P) \mathbin{\tilde{\cup}} \mathit{nodes}(P_S),$$

   where

$$\begin{aligned} A \mathbin{\tilde{\cup}} B \;&=\; \{\overrightarrow{w} = \overleftarrow{w^{-1}} | \overrightarrow{w} \in A \wedge \overleftarrow{w} \in B\} \\ &\quad\; \cup \{\overrightarrow{w} | \overrightarrow{w} \in A \wedge \overleftarrow{w} \notin B\} \cup \{\overleftarrow{w^{-1}} | \overleftarrow{w} \in B \wedge \overrightarrow{w} \notin A\}. \end{aligned}$$

2. $_S\widetilde{\bigcup}_P$ inherits all edges of S_P and P_S:

$$\begin{aligned} s\text{-}\mathit{edges}(_S\widetilde{\bigcup}_P) &= \mathit{edges}(S_P), \\ p\text{-}\mathit{edges}(_S\widetilde{\bigcup}_P) &= \mathit{edges}(P_S). \end{aligned}$$

□

Figure 2.6 shows the reverse union $_S\widetilde{\bigcup}_P$ of the $\mathcal{A}^+$-trees S and P from Figure 2.5. s-edges are printed as solid lines, p-edges are printed as dotted lines.

[11]Here a little asymmetry regarding S and P is introduced in the notation: A node $\overrightarrow{w} \in \mathit{nodes}(S_P)$ remains $\overrightarrow{w}$ in $_S\widetilde{\bigcup}_P$ whereas $\overleftarrow{w} \in \mathit{nodes}(P_S)$ is turned into $\overleftarrow{w^{-1}}$. This, however, is only an artifact due to the notation. The structure of $_S\widetilde{\bigcup}_P$ is in perfect symmetry with regard to S and P.

Terminology:

- The bi-colored, rooted multigraph with edge labels from $\mathcal{A}^+$ formed this way is called a *bi-tree*[12], the node $\overrightarrow{\varepsilon} = \overleftarrow{\varepsilon}$ is called the *root* of ${}_S\widetilde{\bigcup}_P$, short *root*$({}_S\widetilde{\bigcup}_P)$, and $|\mathit{nodes}({}_S\widetilde{\bigcup}_P)|$ is the *size* of ${}_S\widetilde{\bigcup}_P$, short $|{}_S\widetilde{\bigcup}_P|$.
- The subgraph of ${}_S\widetilde{\bigcup}_P$ consisting only of s-edges is identical to S_P and is called the *s-subtree* (of ${}_S\widetilde{\bigcup}_P$). It represents the same strings as S:

  $$s\text{-}\mathit{words}({}_S\widetilde{\bigcup}_P) = \{w \in \mathcal{A}^* | \overrightarrow{w\_} \in \mathit{nodes}({}_S\widetilde{\bigcup}_P)\} = s\text{-}\mathit{words}(S).$$

  Similarly, the subgraph of ${}_S\widetilde{\bigcup}_P$ consisting only of p-edges equals P_S and is called the *p-subtree* (of ${}_S\widetilde{\bigcup}_P$). It represents

  $$p\text{-}\mathit{words}({}_S\widetilde{\bigcup}_P) = \{w \in \mathcal{A}^* | \overleftarrow{\_w} \in \mathit{nodes}({}_S\widetilde{\bigcup}_P)\} = p\text{-}\mathit{words}(P).$$

  Hence, the property $w \in s\text{-}\mathit{words}(T) \iff w^{-1} \in p\text{-}\mathit{words}(T)$ of $\mathcal{A}^+$-trees in general does not hold for bi-trees (but for dual bi-trees that we will define in the next section).
- Based on these two $\mathcal{A}^+$-trees that are contained in ${}_S\widetilde{\bigcup}_P$ we define the terms *leaf in the s-/p-subtree* and *internal branching/non-branching node in the s-/p-subtree* for bi-trees in the obvious way.
- In $T = {}_S\widetilde{\bigcup}_P$ we have two different types of reference pairs:
  - For $w \in s\text{-}\mathit{words}(T)$ we call $(\overrightarrow{v}, u)$ an *s-reference pair* of w if and only if $\overrightarrow{v} \in \mathit{nodes}(T)$ and $w = vu$.
  - For $w^{-1} \in p\text{-}\mathit{words}(T)$ we call $(\overleftarrow{u}, v)$ a *p-reference pair* of w if and only if $\overleftarrow{u} \in \mathit{nodes}(T)$ and $w = vu$.

  The terms *canonical s-/p-reference pair* and *s-/p-location* ($s\text{-}\mathit{loc}_T(w)$, $p\text{-}\mathit{loc}_T(w)$) are defined in the obvious way following the definition of these terms for $\mathcal{A}^+$-trees.
- As for $\mathcal{A}^+$-trees we have for bi-trees: On the path from the root to a node $\overrightarrow{w}$ following s-edges one finds only and all the nodes $\overrightarrow{v}$ in ${}_S\widetilde{\bigcup}_P$ that represent prefixes v of w. Similarly, following p-edges from the root to node $\overleftarrow{w}$ one finds only and all the nodes in $\overleftarrow{u}$ in ${}_S\widetilde{\bigcup}_P$ for which $w = \_u$.
- The atomic and compact bi-tree are defined in a similar way as for $\mathcal{A}^+$-trees: A bi-tree that only contains atomic edges is called an *atomic bi-tree*, a bi-tree that (with the possible exception of the root) does not contain internal nodes that are branching neither in the s- nor in the p-subtree is called a *compact bi-tree*.

[12]This term, not to be confused with a binary tree, has already been used by Weiner [Wei73] for the dual, atomic variant of our structure.

2.3.2 Duality of Bi-Trees

Definition 2.10 (Dual Bi-Tree)
A bi-tree ${}_S\widetilde{\bigcup}_P$ is called *dual* if each node of ${}_S\widetilde{\bigcup}_P$ is both a node in the s-subtree and in the p-subtree of ${}_S\widetilde{\bigcup}_P$. □

The connection between the duality of bi-trees and the duality of $\mathcal{A}^+$-trees is given by the following theorem:

Theorem 2.11 (Equivalence of Dualities)

$${}_S\widetilde{\bigcup}_P \text{ is dual} \iff S_P \text{ and } P_S \text{ are dual } \mathcal{A}^+\text{-trees.}$$

□

Proof

$$\begin{array}{lcl} {}_S\widetilde{\bigcup}_P \text{ is dual} & \iff & \text{each node } \overrightarrow{w} = \overleftarrow{w^{-1}} \in \mathit{nodes}({}_S\widetilde{\bigcup}_P) \text{ is a node both in} \\ & & \text{the } s\text{- and in the } p\text{-subtree of } {}_S\widetilde{\bigcup}_P \\ & \iff & \overrightarrow{w} \in \mathit{nodes}(S_P) \text{ if and only if } \overleftarrow{w} \in \mathit{nodes}(P_S) \\ & \iff & S_P \text{ and } P_S \text{ are dual.} \end{array}$$

□

Now it is easy to see the correctness of the above unproved second remark on Definition 2.5, namely that the $\mathcal{A}^+$-tree that is dual to a given $\mathcal{A}^+$-tree T equals the suffix-link tree of T; or equivalently in the terminology of bi-trees:

Lemma 2.12 (Dual Tree is Suffix Link Tree)
The p-Edges of a dual bi-tree ${}_S\widetilde{\bigcup}_P$ are the suffix links of the s-subtree in ${}_S\widetilde{\bigcup}_P$ and vice versa.
□

Proof
Assume a node $\overrightarrow{aw} \in \mathit{nodes}({}_S\widetilde{\bigcup}_P)$. Let u be the longest suffix of w such that $\overrightarrow{u}$ is a node in ${}_S\widetilde{\bigcup}_P$ as well. We show that ${}_S\widetilde{\bigcup}_P$ contains a p-edge that starts at node $\overrightarrow{u}$ and ends at node $\overrightarrow{aw}$: $\overrightarrow{u} \overset{\_a}{\dashrightarrow} \overrightarrow{aw} \in p\text{-}\mathit{edges}({}_S\widetilde{\bigcup}_P)$.

1. Due to the duality of ${}_S\widetilde{\bigcup}_P$, $\overrightarrow{aw}$ is a node in the p-subtree of ${}_S\widetilde{\bigcup}_P$. Hence, there exists a p-path from the root to node $\overrightarrow{aw}$.

2. Obviously, node $\overrightarrow{u}$ lies on this path because u is a suffix of aw.

3. It remains to be shown that there can not be another node between $\overrightarrow{u}$ and $\overrightarrow{aw}$ on this path. Such a node would be named $\overrightarrow{zu}$ with $w = \_.zu$, $z \in \mathcal{A}^+$, which is in contradiction to the assumption that u is the longest suffix of w that is represented by an explicit node in ${}_S\widetilde{\bigcup}_P$.

Due to its duality, ${}_S\widetilde{\bigcup}_P$ does not contain nodes that only exist in its p-subtree and hence there also can not be any additional p-edge that is not the suffix link of some node $\overrightarrow{aw}$.

Due to the symmetry of the bi-tree this proof also holds if the s- and the p-subtree are exchanged. $\square$

Hence a bi-tree can also be seen as an $\mathcal{A}^+$-tree with labeled suffix links. In this "suffix tree view" a p-edge $\bullet \dashrightarrow k$ will also be called the *suffix link* of node k.

Chapter 3

Affix Trees

3.1 Definition of the Data Structure

The terms and concepts introduced in Section 2 build the basis of the data structure introduced next:

Definition 3.1 (Affix Tree)
The reverse union of a suffix tree and a reverse prefix tree of the same text t is called an *affix*[1] *tree* of t. □

The s-subtree of an affix tree is a suffix tree, the p-subtree is a reverse prefix tree of t. Hence, the s-edges of an affix tree are also called *suffix (tree) edges* and the p-edges are also called *prefix (tree) edges*.

Remarks on Definition 3.1

1. The normal forms of affix trees under edge-splitting/edge-contraction are:

   - the atomic affix tree of t, short $aat(t)$, which is the reverse union of $ast(t)$ and $ast(t^{-1})$. It contains the same nodes as the atomic suffix tree of t.
   - the compact affix tree $cat(t)$ which is the reverse union of $cst(t)$ and $cst(t^{-1})$. Except for the root each internal node is branching in the suffix- or in the prefix-subtree.

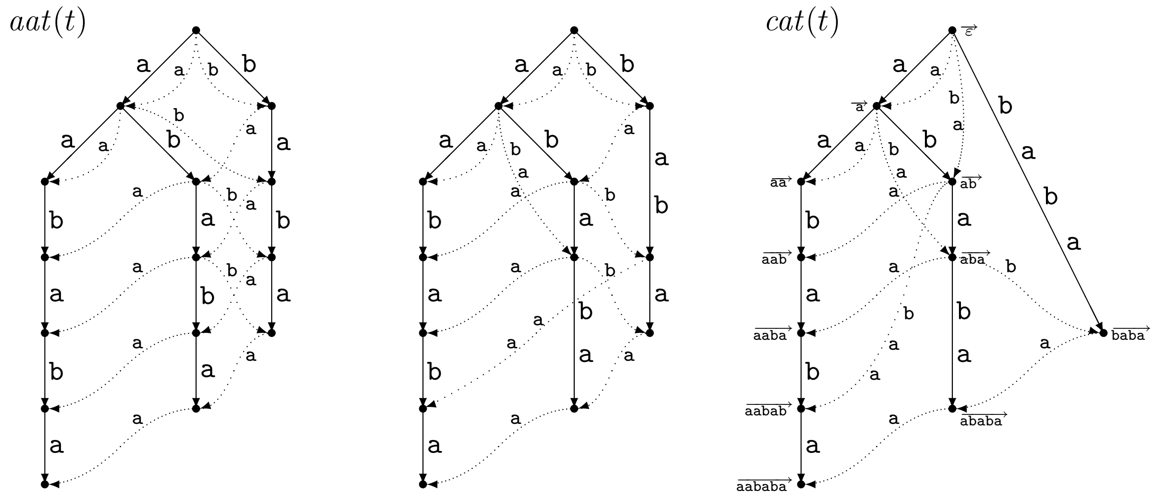
   In Figure 3.1 the atomic (left) and the compact affix tree (right) of $t =$ `aababa` are shown, as well as an intermediate one (center).

2. As for compact suffix trees, we find important correspondences between substrings of the text and the nodes in the compact affix tree:

   The compact affix tree $cat(t)$ contains

   I. an internal branching node in the suffix-subtree for every right-branching t-word,
   II. a leaf in the suffix-subtree for every non-nested suffix of t,

[1]In linguistics, "affix" is a generic term that denotes substrings of a string including prefixes and suffixes.



$aat(t)$
$cat(t)$
$\overrightarrow{\varepsilon}$
$\overrightarrow{\texttt{a}}$
$\overrightarrow{\texttt{aa}}$
$\overrightarrow{\texttt{ab}}$
$\overrightarrow{\texttt{aab}}$
$\overrightarrow{\texttt{aba}}$
$\overrightarrow{\texttt{aaba}}$
$\overrightarrow{\texttt{baba}}$
$\overrightarrow{\texttt{aabab}}$
$\overrightarrow{\texttt{ababa}}$
$\overrightarrow{\texttt{aababa}}$

Figure 3.1: Three affix trees of $t = \texttt{aababa}$.

III. an internal branching node in the prefix-subtree for every left-branching t-word, and

IV. a leaf in the prefix-subtree for every non-nested prefix of t.

Note that it is possible that some of the nodes are branching both in the suffix- and in the prefix-subtree (nodes of types I. and III.) or are leaves in both subtrees (types II. and IV.) as the compact affix tree of Figure 3.1 shows:

node in $cat(\texttt{aababa})$	$\overrightarrow{\varepsilon}$	$\overrightarrow{\texttt{a}}$	$\overrightarrow{\texttt{aa}}$	$\overrightarrow{\texttt{aab}}$	$\overrightarrow{\texttt{aaba}}$	$\overrightarrow{\texttt{aabab}}$	$\overrightarrow{\texttt{aababa}}$	$\overrightarrow{\texttt{ab}}$	$\overrightarrow{\texttt{aba}}$	$\overrightarrow{\texttt{ababa}}$	$\overrightarrow{\texttt{baba}}$
type	I./III.	I./III.	IV.	IV.	IV.	IV.	II./IV.	III.	III.	II.	II.

However, it is not possible that the same node is both an internal branching node in one subtree and a leaf in the other subtree. This would correspond to a right- or left-branching t-word that is a non-nested suffix or prefix at the same time, obviously an impossible situation.

To obtain disjoint types one has to introduce two additional classes:

V. doubly branching nodes for simultaneously right- and left-branching t-words;

VI. double-leaves for t-words that are simultaneously non-nested suffixes and non-nested prefixes of t.

Obviously, only the complete string t can be of type VI.

3. A compact affix tree may contain non-atomic edges that lead to internal nodes. An example is the edge $\overrightarrow{\texttt{a}} \overset{\texttt{ba}}{\dashrightarrow} \overrightarrow{\texttt{aba}}$ in $cat(\texttt{aababa})$, Figure 3.1 (right). In the suffix tree view this means that $cat(t)$ may contain internal nodes that have a non-atomic

suffix link. It is easy to see, though, that such a node $\overrightarrow{aw}$ can not be branching in the suffix-subtree of $cat(t)$: Otherwise (due to I.) aw would be a right-branching t-word, and so would be w, implying that the suffix link of $\overrightarrow{aw}$ is atomic because w is the longest suffix of aw that is represented by an explicit node: $\overrightarrow{aw} \overset{a}{\leftarrow\cdots\cdots} \overrightarrow{w}$.

A fundamental property of affix trees that underlines their symmetric structure is expressed by the following theorem:[2]

Theorem 3.2 (Duality of Affix Trees)
Affix trees are dual. □

Proof
Let S be a suffix tree and P be a reverse prefix tree of the same text t. We show that S_P and P_S are dual, i.e., $\overrightarrow{w} \in nodes(S_P) \iff \overleftarrow{w} \in nodes(P_S)$.

"$\Longrightarrow$" Let $\overrightarrow{w} \in nodes(S_P)$.
$\Rightarrow$ $w \in s\text{-}words(S_P) = s\text{-}words(S) = t\text{-}words = p\text{-}words(P)$, w^{-1} hence is represented in P as an explicit or implicit node.
$\Rightarrow$ 2 cases: 1. $\overleftarrow{w}$ is an explicit node in P;
$\Rightarrow$ $\overleftarrow{w}$ is explicit in P_S as well;
or 2. w is represented by an implicit node in P;
$\Rightarrow$ $\overrightarrow{w}$ is an explicit node in S, otherwise $\overrightarrow{w}$ would not be an explicit node in S_P;
$\Rightarrow$ $\overleftarrow{w}$ is made explicit during the S-extension of P yielding P_S.

"$\Longleftarrow$" The opposite direction can be shown in exactly the same way. □

3.2 Space Usage of Affix Trees

In this section we study the size of the extreme cases of affix trees, *aat* and *cat*. The asymptotic results given first are a simple consequence of the definition of affix trees. The discussion of the exact maximal space usage of *cat* that follows is somewhat more elaborate. The section finishes with a few remarks on the expected size of affix trees over random texts.

3.2.1 Asymptotic Space Usage

In the reverse union of two $\mathcal{A}^+$-trees the number of nodes is at most added. Hence, for a text of size $n = |t|$ we get the following asymptotic bounds for the size of affix trees:

$\boldsymbol{aat(t)}$ is the reverse union of $ast(t)$ and $ast(t^{-1})$ and hence contains $\mathcal{O}(n^2)$ nodes,

$\boldsymbol{cat(t)}$ is the reverse union of $cst(t)$ and $cst(t^{-1})$ and hence contains $\mathcal{O}(n)$ nodes.

[2]Motivated by this one might consider to completely abstract from the terms prefix and suffix in favor of a more general terminology (expressible, for example, by the terms *fix* and *antifix*). This, however, would reduce clarity and in examples one would have to instantiate these terms as *fix* = *suffix* and *antifix* = *prefix* (or vice versa) anyway.

In general, bi-trees may contain nodes that are only reachable on one path. Each node in an affix tree, however, is contained in the suffix- and in the prefix-subtree. Except for the root, at each node ends exactly one suffix-edge and one prefix-edge. Hence, each affix tree with p nodes contains exactly $2p-2$ edges, so the edges of $cat(t)$ require $\mathcal{O}(n)$ space as well. This includes edge labels because, as for suffix trees, edge labels can be stored in constant space as will be shown in Section 3.3.2.

In the following discussion of the exact space usage hence we can focus on the number of nodes in the affix tree.

3.2.2 Maximal Space Usage of Compact Affix Trees

While in a complexity analysis asymptotic values are of main interest, for the practical implementation of an algorithm it is of a similar importance to know the constant factor that upper-bounds the worst case. This will be determined for the more interesting case of the compact affix tree whose size, as shown above, increases only linearly with the text length.

First recall that the compact suffix tree of a text t of length $n > 1$ contains at most $n-1$ nodes. An example is the text $t = \mathtt{a}^{n-1}\$$ whose compact suffix tree contains n leaves and $n-1$ branching internal nodes.

For symmetry reasons we get the same number for the compact reverse prefix tree. By the reverse union of the two trees this value can maximally double: $4n-2$. At least two nodes, however, are merged, namely the root and the node $\overrightarrow{t} = \overleftarrow{t^{-1}}$. For $n > 1$, hence, $4n-4$ is an upper bound on the number of nodes of cat.

We can not give an example where this value is exactly reached but we will discuss a text whose relative deviation from the node number $4n$ becomes arbitrarily small as the text length increases, showing that the constant factor is 4.

Consider an alphabet $\mathcal{A}_k$ of size k,

$$\mathcal{A}_k = \{a_1, a_2, a_3, \ldots, a_k\},$$

and the text

$$t_k = a_1(a_2a_3\ldots a_{k-1})^k a_k.$$

The length of this text is $n = |t_k| = 1 + k(k-2) + 1 = k^2 - 2k + 2$.

To find the exact number of nodes of $cat(t_k)$, we have listed in Table 3.1 all six possible types of t_k-words that correspond to the classes I. – VI. of nodes in the compact affix tree $cat(t_k)$. The center column shows these strings using the example $k = 6$, $\mathcal{A}_6 = \{\mathtt{a}, \mathtt{b}, \mathtt{c}, \mathtt{d}, \mathtt{e}, \mathtt{f}\}$, $t_6 = \mathtt{a}(\mathtt{bcde})^6\mathtt{f}$. The number of words for arbitrary k is given in the right column of Table 3.1.

In total $cat(t_k)$ contains

$$\begin{aligned} |cat(t_k)| &= 2(k-3)(k-1) + 2(n-1) + (k-1) + 1 \\ &= 2(k^2-4k+3) + 2(k^2-2k+2) - 2 + (k-1) + 1 \\ &= 4k^2 - 11k + 8 \end{aligned}$$

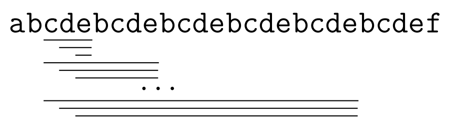
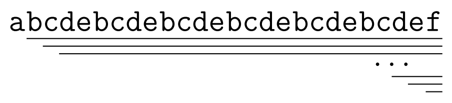
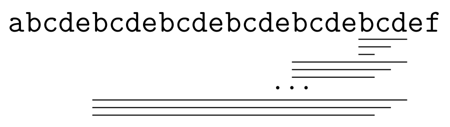
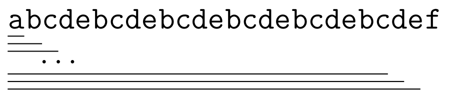
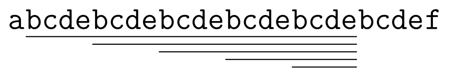
node type	$k = 6, \quad n = \lvert t_k \rvert = 26$	in general
I. internal node in the suffix-subtree (right- but not left-branching t_k-word)	 `abcdebcdebcdebcdebcdebcdef`	$(k-3)(k-1)$
II. leaf in the suffix-subtree (non-nested suffix that is not a non-nested prefix of t_k)	 `abcdebcdebcdebcdebcdebcdef`	$n-1$
III. internal node in the prefix-subtree (left- but not right-branching t_k-word)	 `abcdebcdebcdebcdebcdebcdef`	$(k-3)(k-1)$
IV. leaf in the prefix-subtree (non-nested prefix that is not a non-nested suffix of t_k)	 `abcdebcdebcdebcdebcdebcdef`	$n-1$
V. doubly branching node (both right- and left-branching t_k-word)	 `abcdebcdebcdebcdebcdebcdef`	$k-1$
VI. double-leaf (both non-nested suffix and non-nested prefix)	`abcdebcdebcdebcdebcdebcdef`	1

Table 3.1: The nodes of $cat(t_k)$ for $k = 6$ (center column) and in general (right column).

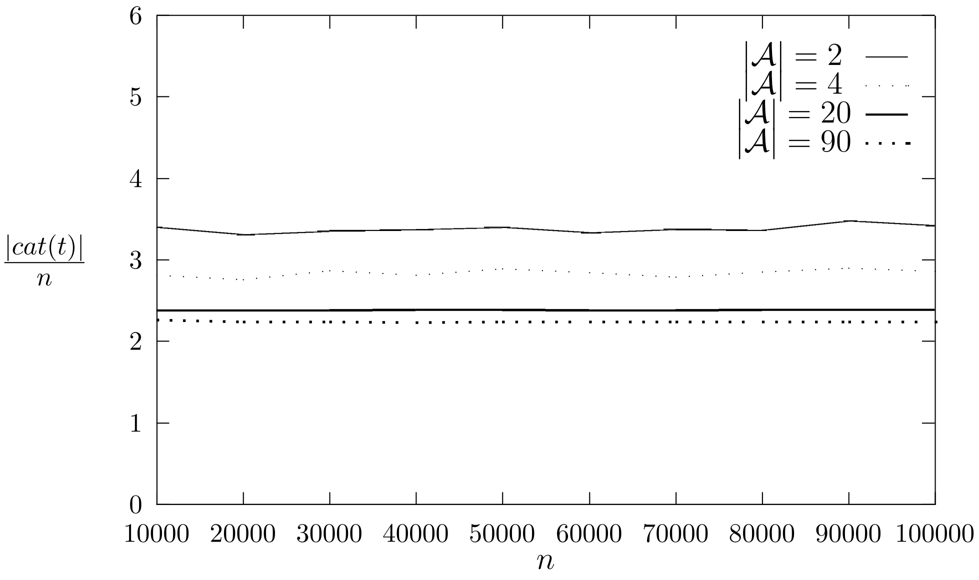


Figure 3.2: The size of $cat(t)$ for random texts t versus the text length n for alphabets of different size $|\mathcal{A}|$.

nodes.

For large k the relative deviation from $4n = 4k^2 - 8k + 8$ becomes arbitrarily small:[3]

$$\frac{|cat(t_k)| - 4n}{4n} = \frac{-3k}{4k^2 - 8k + 8} = \frac{-\frac{3}{k}}{4 - \frac{8}{k} + \frac{8}{k^2}} \underset{k \to \infty}{\longrightarrow} 0.$$

3.2.3 Expected Space Usage

In the previous section we have considered the example t_k where the fraction $\frac{|cat(t_k)|}{n}$ for large k asymptotically approaches 4.

In contrast, for random[4] texts one finds that for large alphabets the tree size decreases (see Figure 3.2). The text length does not have an influence on this phenomenon, which is not surprising.

To explain the dependence on the alphabet size that one finds similarly for suffix trees (see Figure 3.3), we make three assumptions:

1. Due to [AS92], the expected size of the longest repeat in a text $t \in \mathcal{A}^n$ is of the order $O(\log n)$. Hence we can ignore the effect of nested suffixes and assume that the number of leaves in the suffix tree is n as for texts that end with a character \$ not occurring elsewhere in t.

2. The branching structure of internal nodes depends on k, the alphabet size. In what follows we will assume a constant value $q \geq 2$. (As we will see, the exact value is irrelevant for qualitative results.)

[3]The notation $q \underset{k\to\infty}{\longrightarrow} 0$ is short for $\lim_{k\to\infty}(q) = 0$.

[4]For the Bernoulli distribution used here, see e. g. [AS92].

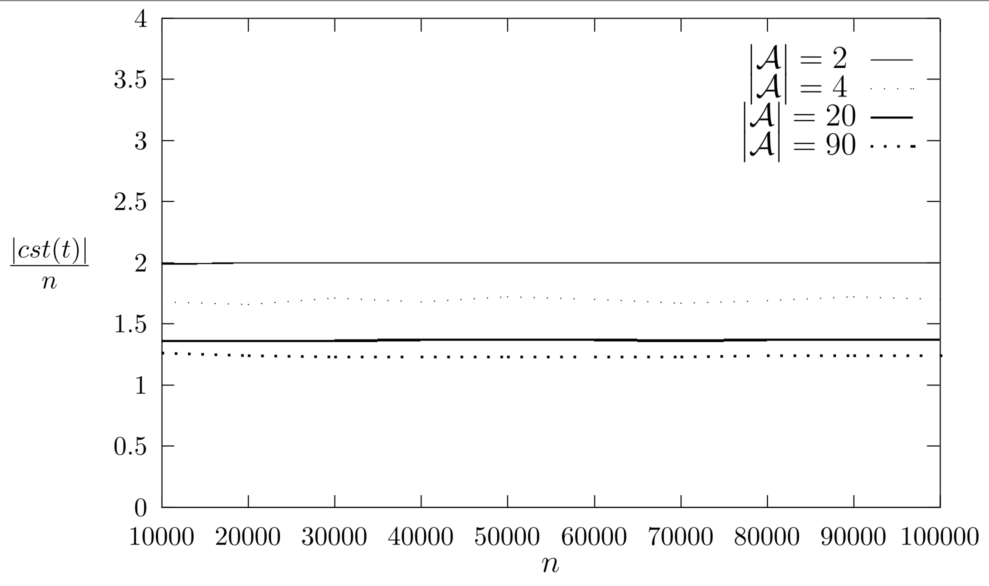


Figure 3.3: The size of $cst(t)$ for random texts t versus the text length n for alphabets of different size $|\mathcal{A}|$.

3. For random texts, the suffix tree is balanced such that all leaves have the same depth.

Under these assumptions, a compact suffix tree of depth d (i. e., with $d+1$ levels $i = 0 \dots d$) contains in level i exactly q^i nodes, and hence in total

$$\sum_{i=0}^{d-1} q^i \;=\; \frac{q^d - 1}{q - 1} \quad \text{internal nodes and } q^d \text{ leaves.}$$

This yields the following ratio for internal nodes and leaves:

$$\frac{q^d - 1}{q - 1} \cdot \frac{1}{q^d} \quad = \quad \frac{1 - \frac{1}{q^d}}{q - 1} \quad \xrightarrow[d\to\infty]{} \quad \frac{1}{q-1}.$$

For small alphabets ($q \approx 2$) one hence expects a fraction of 1 while for larger alphabets ($q \to \infty$) this fraction asymptotically approaches 0. This corresponds to a node number of $2n$ for small alphabets. For large alphabets the number of nodes comes close to $1n$, a behavior that corresponds well to the empirical results shown in Figure 3.3.[5]

Qualitatively, this behavior should carry over to affix trees because for random texts $|cat(t)|$ should be proportional to $|cst(t)|$. Comparing Figures 3.2 and 3.3 quantitatively, a factor between 1.6 and 1.8 seems appropriate.

3.3 Representation of Affix Trees

In this section we describe how affix trees can be stored efficiently. First, we compare two general representation techniques for $\mathcal{A}^+$- and bi-trees[6], and then we discuss a special con-

[5]Blumer, Ehrenfeucht and Haussler [BEH89] who performed a similar quantitative study obtain similar results.

[6]A similar comparison for suffix trees has been carried out in [CS85].

vention that is important for an efficient implementation of the online construction algorithm for compact affix trees presented in Chapter 4.

3.3.1 Edge vs. Node Labels

A standard implementation of $\mathcal{A}^+$-trees is formalized by the following recursive type definition:[7]

```
AplusTree a = Node [(Label a,AplusTree a)].
```

An $\mathcal{A}^+$-tree is represented by a root node and a list of labeled edges leading to subtrees of the same type. Leaves are nodes with an empty edge/subtree list. Here, edge labels are denoted in an abstract way as `(Label a)`. We will describe below how these labels can be represented using only constant space per label.

The corresponding type definition of bi-trees contains two lists of edges, one for suffix edges and one for prefix edges:[8]

```
BiTree a = Node [(Label a, BiTree a)] [(Label a, BiTree a)].
```

Alternatively, an $\mathcal{A}^+$-tree can be represented with node labels

```
AplusTree2 a = Node (Label a) [AplusTree2 a],
```

i. e., a tree with root node $\overrightarrow{w}$ is represented by the string w, again denoted as `(Label a)` in an abstract way, and the list of its subtrees. For bi-trees two subtree lists are necessary:

```
BiTree2 a = Node (Label a) [BiTree2 a] [BiTree2 a].
```

Comparing these two representations, one finds that the second variant is slightly more powerful: Given the labels of two neighboring nodes it is easily possible to compute the edge label of the connecting edge

$$\overrightarrow{u} \longrightarrow \overrightarrow{uv} \quad \Rightarrow \quad \bullet \xrightarrow{v} \bullet,$$

while without knowledge of the position in the tree, the opposite is not possible. On the other hand, the representation by node labels is restricted in the sense that it can not be applied to arbitrary edge-labeled graphs.

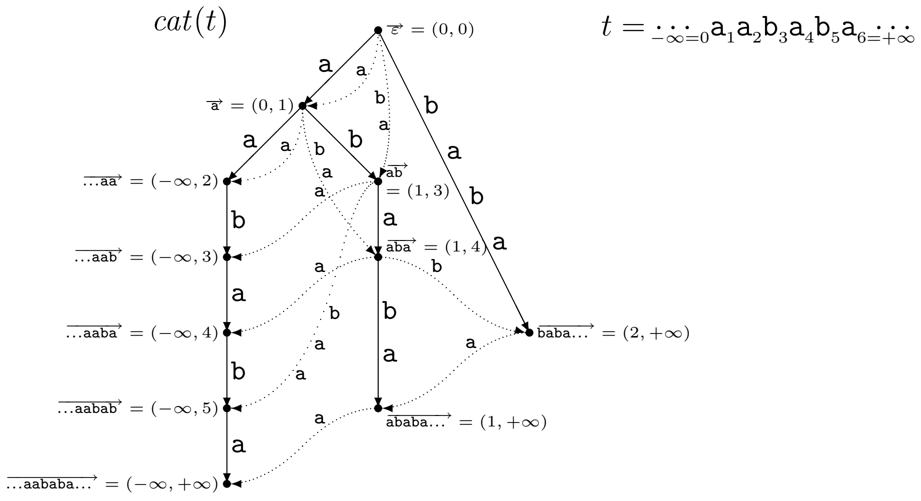
We now compare the space usage of the two representations. As noted above, we will assume that every `(Label a)` can be stored using the same, constant space.

For $\mathcal{A}^+$-trees the difference between the two representations is negligible. They contain roughly the same number of edges and nodes.

For bi-trees this is different: Here we have one or two edges ending at each node other than the root. In particular affix trees, like all dual bi-trees, contain almost twice as many edges as nodes. Hence, for affix trees node labels are the preferred representation. That does not

[7]We use the syntax of the lazy functional programming language Haskell [FHPW92].

[8]Due to the symmetry of the data structure it is not said which of the two lists represents the suffix edges and which list represents the prefix edges. We do not need to make a choice here but in an actual implementation one has to decide and fix the order, of course.



$cat(t)$
$t = \therefore_{-\infty=0}\mathtt{a}_1\mathtt{a}_2\mathtt{b}_3\mathtt{a}_4\mathtt{b}_5\mathtt{a}_{6=+\infty}\therefore$
$\overrightarrow{\varepsilon} = (0,0)$
$\overrightarrow{\mathtt{a}} = (0,1)$
$\overrightarrow{\ldots\mathtt{aa}} = (-\infty,2)$
$\overrightarrow{\mathtt{ab}} = (1,3)$
$\overrightarrow{\ldots\mathtt{aab}} = (-\infty,3)$
$\overrightarrow{\mathtt{aba}} = (1,4)$
$\overrightarrow{\ldots\mathtt{aaba}} = (-\infty,4)$
$\overrightarrow{\mathtt{baba}\ldots} = (2,+\infty)$
$\overrightarrow{\ldots\mathtt{aabab}} = (-\infty,5)$
$\overrightarrow{\mathtt{ababa}\ldots} = (1,+\infty)$
$\overrightarrow{\ldots\mathtt{aababa}\ldots} = (-\infty,+\infty)$

Figure 3.4: *cat*(`aababa`) with node labels and index pairs.

mean, though, that in total we save half of the memory: edges still have to stored, but without their label. Nevertheless, in our implementation in the programming language C [KR78] that we describe later, we gain a noticeable space reduction of about 40% (128 rather than 224 bytes per node).

3.3.2 Node Labels

One assumption in the discussion of the space usage above was the possibility to store edge and node labels of type `(Label a)` in constant space. This is possible for suffix trees and for affix trees because for any node $\overrightarrow{w} \in nodes(T)$, w is a t-word and hence the label of node $\overrightarrow{w}$ can be represented by a pair of indices `(Label a)` $= (l, r)$ if $t = {}_{-l}w_{r}{}_{-}$.

This notation is extended in [Ukk93] where edges that end at a leaf of the suffix tree are represented as so-called *open edges*, i. e., the right index $r = |t|$ remains open, which is symbolized by an infinity-sign: $(l, |t|) = (l, \infty)$.

This convention can be applied to node labels in an affix tree T as well: A leaf $\overrightarrow{w}$ in the suffix-subtree of T is represented as an *open suffix leaf* by the index pair $(l, +\infty)$ if $t = \_\_{}_l w_{|t|}$, and a leaf $\overleftarrow{w}$ in the prefix-subtree of T is represented as an *open prefix leaf* by the index pair $(-\infty, r)$ if $t = {}_0 w^{-1}{}_r\_\_$.

In such cases we extend our previous notation for node labels, $\overrightarrow{w}$ ($\overleftarrow{w}$), by three dots yielding $\overrightarrow{w\ldots}$ ($\overleftarrow{\ldots w}$). Figure 3.4 shows *cat*(`aababa`) with node labels and index pairs.

3.4 Neuralgic Points in the Affix Tree

3.4.1 Active Suffix and Active Prefix

One location in the suffix tree is of main importance during the online construction [Ukk93], the *active point*. This point moves up and down the tree, and its role is described in [GK94a] as follows:

> "If t is to be extended to the right by another letter, changes in the suffix tree (if any) will start at this point."

The active point is the location of the active suffix of t in $cst(t)$:

Definition 3.3 (Active Suffix, Active Prefix)
The *active suffix* of a text t, denoted $\alpha(t)$, is the longest nested suffix of t. The *active prefix* of a text t, denoted $\alpha^{-1}(t)$, is the longest nested prefix of t. $\Box$

A simple, frequently used property of the active suffix/prefix is expressed by the following lemma:

Lemma 3.4 (Active Suffix, Active Prefix)
If the active suffix $\alpha(t)$ is not left-branching in t, then it is a prefix of t.
If the active prefix $\alpha^{-1}(t)$ is not right-branching in t, then it is a suffix of t. $\Box$

Proof
Let $t = \_\_xu$, $u = \alpha(t)$, and let u be not left-branching in t. Because u is the active suffix of t, u must occur a second time in $t = \left\{ \begin{matrix} \_\_xu \\ \_u\_\_ \end{matrix} \right\}$. The letter to the immediate left of u in the lower row can not be an x (if so, xu would be the active suffix of t), nor can it be a different letter $y \neq x$ (if so, u would be left-branching). Hence only ε is possible to the left of u, i. e., u is a prefix of $t = \left\{ \begin{matrix} \_\_xu \\ u\_\_ \end{matrix} \right\}$, and u does not occur a second time in t.

The symmetric assertion for the active prefix can be shown similarly. $\Box$

3.4.2 Active Suffix Leaf and Active Prefix Leaf

For the construction of compact affix trees, another location in the tree turns out to be important as well, the active suffix leaf/active prefix leaf.[9]

Taking a look at a compact affix tree $T = cat(t)$ it is easy to see that the p-path from node $\overrightarrow{t}$ to the root consists of a number of atomic suffix links that one after the other connect the leaves of the suffix-subtree of T. These suffix-leaves represent shorter and shorter non-nested suffixes of t.

Definition 3.5 (Active Suffix Leaf, Active Prefix Leaf)
The leaf in the suffix-subtree of $cat(t)$ that represents the shortest non-nested suffix of t is called the *active suffix leaf* of $cat(t)$. Similarly, the leaf in the prefix-subtree of $cat(t)$ that represents the shortest non-nested prefix of t is called the *active prefix leaf* of $cat(t)$. $\Box$

The active suffix leaf in $cat(\texttt{aababa})$ (Figure 3.4) is $\overrightarrow{\texttt{baba...}}$. The active prefix leaf is $\overrightarrow{\texttt{...aa}}$.

[9]Actually, the active suffix leaf is important for the construction of suffix trees as well, but only if suffix links of the leaves are considered which usually is not the case.

Remarks on Definition 3.5

1. The suffix link of the active suffix leaf ends at the node that represents the longest nested suffix s of t for which $\overrightarrow{s}$ is an explicit node in $cat(t)$.
2. If the active suffix $\alpha(t)$ is represented by an explicit node $\overrightarrow{\alpha(t)}$ in $cat(t)$, then $s = \alpha(t)$ and the suffix link of the active suffix leaf is atomic.
3. The active suffix leaf is the only leaf in the suffix-subtree of $cat(t)$ whose suffix links may be non-atomic. This is the case if and only if the active suffix $\alpha(t)$ is not represented by an explicit node.

Clearly, symmetric properties hold for the active prefix leaf.

3.5 Construction of Compact Affix Trees

In this thesis we suggest three ways for the construction of compact affix trees. Two of these are briefly described in Sections 3.5.1 and 3.5.2, the third approach is extensively studied and developed to a concrete algorithm in Chapter 4.

3.5.1 Construction of Compact Affix Trees by Edge Contraction

The algorithm described here originally was developed to quickly have a method at hand to automatically construct affix trees, so as to avoid errors and to minimize effort for the manual construction. To this end it seemed suitable to use a functional programming language that on the one hand results in relatively slow and memory intensive programs, on the other hand allows much quicker program development.[10]

The algorithm is very simple: As noted in Section 3.1, the compact affix tree of a text t, among all affix trees of t is the normal form under edge contraction. Hence, given an arbitrary affix tree of t, for example $aat(t)$, it is possible to construct $cat(t)$ by elimination of all internal nodes that are branching neither in the suffix- nor in the prefix-subtree. The atomic affix tree $aat(t)$ as the starting point is a suitable choice because it contains the same nodes as $ast(t)$ for which simple construction methods exist.[11]

On the other hand this method in the worst case requires quadratic time and space because the intermediate structure of the atomic affix tree has to be constructed completely. Nevertheless, the requirements for a prototypic method for testing the final system are perfectly fulfilled.

[10]Indeed, the ideas described here have been implemented in Haskell during one afternoon while the development of the methods described in the next chapter and their implementation in C took about two months.

[11]See for example [GK94b].

3.5.2 Construction of Compact Affix Trees by Reverse Union of Suffix and Reverse Prefix Tree

The second approach to construct compact affix trees immediately follows Definition 3.1: First $S = cst(t)$ and $P = cst(t^{-1})$ are constructed. Then these trees are mutually extended yielding S_P and P_S. Finally $cat(t)$ is constructed by merging the edges of one of these two trees into the other one.

While it is well known that the first step only takes $\mathcal{O}(n)$ time, it seems not so easy to see how the other two steps can be implemented in linear time. At this point we are unable to finally answer the question if this approach can be formed into a method for the linear-time construction of $cat(t)$.

3.6 Applications of Affix Trees

Several applications of suffix trees[12] are described in [Apo85]. Since affix trees completely include the suffix tree structure, these methods can be carried out with affix trees as underlying data structure as well.

However, it is not obvious that also the time complexity remains the same. As shown in Section 3.1, the suffix-subtree of a compact affix tree is not a compact suffix tree and some suffix links are non-atomic. This might lead to an increase in the number of steps one has to go even if the paths traversed in the suffix tree are the same. Such a case will be discussed in detail in Section 5. However, the following remark can be made here:

In all cases where paths from the root into the affix tree are traversed spelling the characters of a pattern, this does not lead to an increase in worst case time complexity as compared to the corresponding path in the suffix tree because the path from the root to node $\overrightarrow{w}$ in the suffix tree as well as in the affix tree is of length $|w|$.

This alleviates the drawback described above because many of the applications described in [Apo85] traverse only paths top-down into the tree. These include:[13]

- recognizing if w occurs in t in $\mathcal{O}(|w|)$ time;
- finding the first/last occurrence of w in t in $\mathcal{O}(|w|)$ time;
- finding the weight of a string w in a weighted vocabulary in $\mathcal{O}(|w|)$ time;
- finding all repeats of a text in $\mathcal{O}(\text{output})$ time.

Beyond these applications that are directly inherited from suffix trees, the symmetric structure of affix trees can also be used for more flexible applications. At this point we refer to the program *bigrep* that was developed as part of this thesis and that is described in Chapter 6.

[12]To be precise, Apostolico considers the subword tree of text $t \in \mathcal{A}^*$, the suffix tree of the text appended by an additional character \$ not occurring in $\mathcal{A}$.

[13]The complexity measures given here only account for the time of the mentioned applications and not for the time of the affix tree construction. Moreover, like [Apo85] we assume the alphabet size and hence the time for finding at each internal node the edge leading to the desired successor node to be constant. In practice this can be achieved by using suitable implementation techniques (edge list as array, hashing, etc.).

This program allows to perform exact string matching in a text that can be extended *online* both to the left and to the right, an application that is impossible with methods that only use suffix trees.

Chapter 4

Online Construction of Compact Affix Trees

The method for the online construction of compact affix trees presented in this chapter can be seen as an extension of Ukkonen's algorithm for the construction of compact suffix trees [Ukk93][1], called *ukk* in the following. The affix tree, like the suffix tree in *ukk*, is constructed *online*: Starting with the affix tree of the empty text, $cat(\varepsilon)$, the text is read character by character, and with each new character a the existing affix tree $cat(t)$ is extended yielding the affix tree of the elongated text $cat(ta)$.[2]

While *ukk* is restricted to the case where t is extended to the right, here it will be possible to extend the text to the left as well and hence to construct $cat(at)$ from $cat(t)$. This way the construction can start at any position in the text and extensions are possible in both directions.

The main part of the following discussion will deal with the step from $cat(t)$ to $cat(ta)$, though. The opposite direction $cat(t) \rightarrow cat(at)$ is symmetric and hence does not need a separate treatment if certain conditions are fulfilled.

The algorithm is presented in a top-down manner. First we examine the differences between $cat(t)$ and $cat(ta)$ on an abstract level. Section 4.2 then develops the resulting consequences for a procedure that converts $cat(t)$ into $cat(ta)$. In Section 4.3 the results are summarized giving a formal algorithm. Section 4.4 finally discusses the aspects that have to be taken into consideration when the method is applied bidirectionally.

4.1 Differences between $cat(t)$ and $cat(ta)$

In this section we analyze the differences between $cat(t)$ and $cat(ta)$ statically (i. e., without operational aspects).

[1]For comparisons the reader might prefer the presentation in [Kur95] whose notation is more similar to the one used here.

[2]In this and in the next chapter, t always denotes the text read so far (for which the affix tree is already constructed) and a denotes the new character by which the text is extended. The length of t is n.

We discuss how the active suffix and the active prefix change, which nodes are added and which nodes are removed. From these changes, the necessary modifications of the edges and the active leaves are easily derived so that they do not need to be considered separately.

Some of these results are identical to corresponding ones for compact suffix trees and can be carried over from *ukk*. Yet, several properties are new and will be proven in the following.

4.1.1 Changes of the Active Suffix and the Active Prefix

We first have a look at the active suffix $\alpha(ta)$ as compared to $\alpha(t)$. As explained in detail in [GK94b], $\alpha(ta)$ is a suffix of $\alpha(t)a$. Hence, the active suffix can lengthen by at most one character. On the other hand, it can shrink by several characters, at most by its whole length which is the case if the new character a does not occur in t.

What are the possible changes of the active prefix, how can $\alpha^{-1}(ta)$ look like as compared to $v = \alpha^{-1}(t)$?

1. Obviously it is not possible that the active prefix shrinks because v is a nested prefix of ta as well.
2. The active prefix can remain unchanged as the example $t =$ `aababa`, $a =$ `b` shows: $\underline{\texttt{a}}\texttt{ababa} \rightarrow \underline{\texttt{a}}\texttt{ababab}$ (the active prefix in both cases is underlined).
3. The active prefix can also increase as the example $\underline{\texttt{a}}\texttt{ababa} \rightarrow \underline{\texttt{aa}}\texttt{babaa}$ shows. This elongation, however, can be by at most one character as is established in the following theorem.

Theorem 4.1[3] (Elongation of the Active Prefix)
Let $v = \alpha^{-1}(t)$. In the step $t \rightarrow ta$ the active prefix increases by the character a if and only if v (as a prefix of t) is followed by a and v simultaneously is a suffix of t:

$$\alpha^{-1}(ta) = va \quad \Longleftrightarrow \quad t = \left\{ \begin{array}{c} va\_\_ \\ \_\_.v \end{array} \right\}.$$

Otherwise the active prefix remains unchanged. □

Proof

"$\Longrightarrow$" Given: (i) v is the active prefix of t.
(ii) va is the active prefix of $ta = va\_\_a$.

To show: $t = \left\{ \begin{array}{c} va\_\_ \\ \_\_.v \end{array} \right\}$.

1. $t = va\_\_$ is an obvious consequence of (ii).
2. (i) $\Rightarrow$ v is the longest nested prefix of t; va does not occur nested in t;
   (ii) $\Rightarrow$ va is the longest nested prefix of ta; va occurs a second time in ta, not as a prefix. This occurrence can only be as a suffix of ta because this is the only place where t and ta differ: $ta = \_\_.va$
   $\Rightarrow t = \_\_.v$

[3]This is a formal version of the "beer theorem" by R. Giegerich and S. Kurtz, the first step towards affix trees.

"$\Longleftarrow$" Given: (i) v is the active prefix of t.

(ii) $t = \left\{\begin{array}{l} va\_\_\_ \\ \_\_\_.v \end{array}\right\}$.

To show: va is the active prefix of ta.

(ii) $\Rightarrow ta = \left\{\begin{array}{l} va\_\_\_a \\ \_\_\_.va \end{array}\right\}$

$\Rightarrow$ va is nested in ta, hence it is the active prefix if there is no longer prefix that is nested in ta as well.
Assuming there is a longer prefix vaz for some $z \in \mathcal{A}^+$ that is nested in ta implies that va is nested in t which is in contradiction to (i).

In all other cases where t is not of the form $t = \left\{\begin{array}{l} va\_\_\_ \\ \_\_\_.v \end{array}\right\}$, the active prefix remains unchanged because

1. it can not shrink (see the first remark above),
2. an extension by another character $b \neq a$ is impossible:

$$ta = \left\{\begin{array}{l} vb\_\_\_a \\ \_.vb\_\_ \end{array}\right\} \quad \Rightarrow \quad \begin{array}{ll} \text{either} & t = \left\{\begin{array}{l} vb\_\_\_ \\ \_\_\_\_v \end{array}\right\} \Rightarrow a = b \\ \text{or} & t = \left\{\begin{array}{l} vb\_\_\_ \\ \_.vb\_ \end{array}\right\} \Rightarrow \alpha^{-1}(t) = vb\_, \end{array}$$

3. and an elongation by more than one character is impossible as well: assume $\alpha^{-1}(ta) = vzb, z \in \mathcal{A}^+, b \in \mathcal{A}$

$$\Rightarrow ta = \left\{\begin{array}{l} vzb\_\_\_a \\ \_.vzb\_\_ \end{array}\right\} \Rightarrow t = \left\{\begin{array}{l} vzb\_\_\_ \\ \_.vz\_\_ \end{array}\right\} \Rightarrow \alpha^{-1}(t) = vz\_$$

which is in contradiction to the assumption $\alpha^{-1}(t) = v$.

□

This theorem implies a number of further properties of the active points. The first of the following two theorems describes a surprising relation between the active prefix and the active suffix:

Theorem 4.2 (Relation of Active Prefix and Active Suffix)
If the active prefix lengthens in the step $t \to ta$, then the new active prefix equals the new active suffix:

$$\alpha^{-1}(ta) = \alpha^{-1}(t)a \quad \Longrightarrow \quad \alpha^{-1}(ta) = \alpha(ta).$$

□

Proof
Let $v = \alpha^{-1}(t)$. Then

$$\alpha^{-1}(ta) = va \overset{\text{Theorem 4.1}}{\Longrightarrow} t = \left\{\begin{array}{l} va\_\_\_ \\ \_\_\_.v \end{array}\right\} \Longrightarrow ta = \left\{\begin{array}{l} va\_\_\_a \\ \_\_\_.va \end{array}\right\}.$$

1. The suffix va of ta is nested in ta, hence $\alpha(ta)$ has the suffix va.
2. $\alpha(ta)$ can not be longer than va, otherwise $\alpha(ta) = zva$ for some $z \in \mathcal{A}^+$ implying

$$ta = \left\{ \begin{array}{c} va\underline{\quad\;}a \\ \underline{\;}zva\underline{\;}a \\ \underline{\quad\;}.zva \end{array} \right\} \;\Rightarrow\; t = \left\{ \begin{array}{c} va\underline{\quad\;} \\ \underline{\;}zva\underline{\;} \\ \underline{\quad\;}.zv \end{array} \right\}$$

   which is in contradiction to the assumption that v (rather than va) is the active prefix of t.

$\square$

As the example $ta = \texttt{aaba}$ shows, the reverse of Theorem 4.2 does not hold because here the new active suffix equals the new active prefix ($\alpha(ta) = \alpha^{-1}(ta) = \texttt{a}$) but the active prefix was not extended in the step $t \to ta$ ($\alpha^{-1}(ta) = \alpha^{-1}(t) = \texttt{a}$).

What follows are two theorems that describe the relation between the active points and modifications of the strings represented in the affix tree:

Theorem 4.3 (Left-Branching Active Suffix)
If the new active suffix $u = \alpha(ta)$ is not represented by an explicit node in $cat(t)$, then it is a left-branching ta-word:

$$\overrightarrow{u} \notin nodes(cat(t)) \quad\Longrightarrow\quad ta = \left\{ \begin{array}{c} \underline{\quad}xu \\ \underline{\;}yu.\underline{\;} \end{array} \right\}, x \neq y.$$

$\square$

Proof
Let $u = \alpha(ta)$, $\overrightarrow{u} \notin nodes(cat(t))$.

Assume that u is not left-branching in ta, hence

$$\begin{array}{ll} \overset{\text{Lemma 3.4}}{\Longrightarrow} & ta = \left\{ \begin{array}{c} \underline{\quad}.u \\ u.\underline{\quad} \end{array} \right\} \text{ and no other occurrence of } u \text{ in } ta \\ \Longrightarrow & t = u\underline{\quad} \text{ and no other occurrence of } u \text{ in } t \end{array}$$

which implies that u is a non-nested prefix of t and hence is represented by a prefix leaf in $cat(t)$. This is in contradiction to the assumption that $\overrightarrow{u} \notin nodes(cat(t))$ which proves the theorem. $\square$

Theorem 4.4 (Nesting of a Prefix)
Every non-nested prefix w of t is a non-nested prefix of ta as well, except if w is the extended active prefix, $w = \alpha^{-1}(ta) = \alpha^{-1}(t)a$. $\square$

Proof
Let w be a non-nested prefix of t: $t = w\underline{\quad}$ and w does not occur a second time in t $\Rightarrow \quad ta = w\underline{\quad}a$. Then we have:

1. w is a prefix of ta as well.

2. The only possibility for w to be nested in ta is as a suffix of ta:

$$\begin{aligned} & ta = \left\{ \begin{matrix} w\_\_\_a \\ \_\_\_.w \end{matrix} \right\} \quad \Rightarrow \quad w = va \text{ for some } v \in \mathcal{A}^* \\ \Rightarrow\quad & ta = \left\{ \begin{matrix} va\_\_\_a \\ \_\_\_.va \end{matrix} \right\} \\ \Rightarrow\quad & t \;= \left\{ \begin{matrix} va\_\_\_ \\ \_\_\_.v \end{matrix} \right\} \quad \text{and } w = va \text{ does not occur a second time in } t \text{ (see above)} \\ \Rightarrow\quad & \alpha^{-1}(t) = v. \end{aligned}$$

Hence w becomes a nested prefix of ta if it is of the form $w = va$, $v = \alpha^{-1}(t)$, and $t = \left\{ \begin{matrix} va\_\_\_ \\ \_\_\_.v \end{matrix} \right\}$.

Due to Theorem 4.1 this is the case if and only if the active prefix lengthens:

$$w = va = \alpha^{-1}(t)a = \alpha^{-1}(ta).$$

□

4.1.2 Changes of the Node Set

In their description of the suffix tree nodes that are added in the step $t \to ta$, Giegerich and Kurtz [GK94b] use the following term:

Definition 4.5 (Relevant Suffix)
A suffix sa of ta is called *relevant* if and only if s is a nested suffix of t and sa is a non-nested suffix of ta. □

This allows to describe the changes of substrings of types I. – IV. as defined in Section 3.1 which correspond to the different node types in a compact affix tree:[4]

I. Let w be a right-branching t-word.
   a) Then w is a right-branching ta-word as well.
   b) In addition, s is a new right-branching ta-word if sa is a relevant suffix of ta and s was not already a right-branching t-word.

II. Let w be a non-nested suffix of t.
   a) Then wa (rather than w) is a non-nested suffix of ta.
   b) In addition, all relevant suffixes sa of ta are new non-nested suffixes of ta.

III. Let w be a left-branching t-word.
   a) Then w is a left-branching ta-word as well.

[4]The changes in the suffix tree (cases I. and II.) are established in detail in [GK94b] using a similar terminology as we use here. At those assertions of III. and IV. that are not obvious we refer to the respective theorems of the previous section.

b) In addition, the new active suffix $u = \alpha(ta)$ is a new left-branching ta-word if $\overrightarrow{u}$ was not an explicit node in $cat(t)$ (Theorem 4.3).

IV. Let w be a non-nested prefix of t.

a) Then w is a non-nested prefix of ta as well, except if w is the elongated active prefix (Theorem 4.4).

b) In addition, the complete string ta is a new non-nested prefix of ta.

Note that in contrast to the online construction of suffix trees, due to (IV.a) it is possible that $cat(t)$ contains nodes that do no longer exist in $cat(ta)$. An example is $t = \texttt{aababa}$, $a = \texttt{a}$: The node $\overrightarrow{...\texttt{aa}}$ of $cat(t)$ is removed in $cat(ta)$ because the prefix $\texttt{aa}$ that was non-nested in t becomes nested in ta.

4.2 Operational Consequences

Next we assemble the observed changes in a suitable order such that an algorithm is obtained that constructs the nodes of $cat(ta)$ from the nodes of $cat(t)$.

In doing so, we keep the following in mind:

- The elongation of the non-nested suffixes (II.a) does not need to be performed explicitly if the leaves are represented as "open leaves".
- With this "automatic elongation" also the node $\overrightarrow{ta}$ is constructed, accounting for (IV.b).
- On the other hand, the non-nested prefix t gets lost as an explicit node, such that $\overrightarrow{t}$ has to be inserted again if t is a non-nested prefix of ta as well. This is the case if and only if $ta \neq a^{n+1}$.

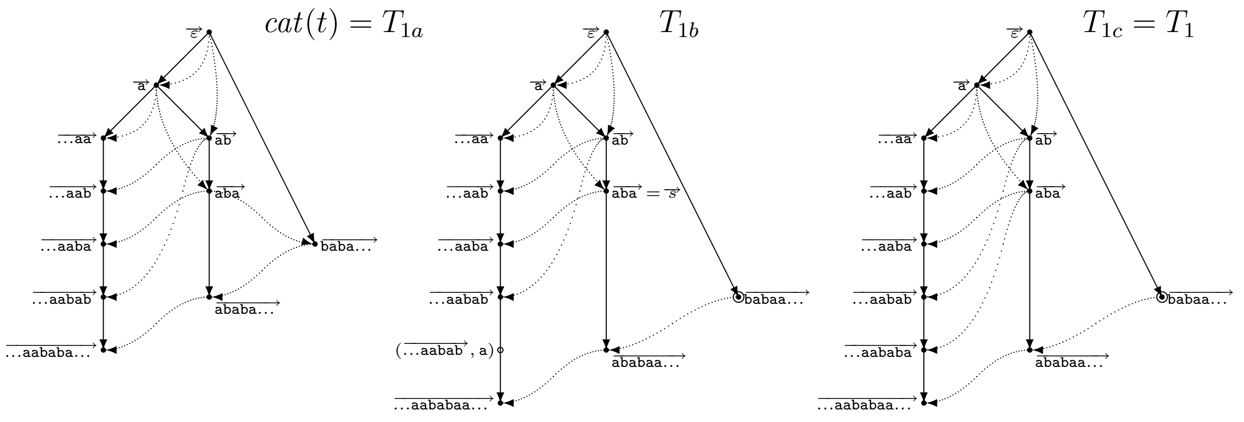
  However, for the implementation of (IV.a) the node $\overrightarrow{va}$ has to be removed anyway if the active prefix $v = \alpha^{-1}(t)$ increases to $va = \alpha^{-1}(ta)$ (which is the case for $ta = a^{n+1}$), and hence this step can be split: First, always the node $\overrightarrow{t}$ is inserted, and later the node $\overrightarrow{va}$ (in this case $\overrightarrow{t}$) is removed.

In summary, we have the following algorithm:[5]

Algorithm 4.6 ($cat(t) \rightarrow cat(ta)$)

1. Lengthen the text (and thereby implicitly lengthen every leaf in the suffix tree) and insert node $\overrightarrow{t}$ (II.a, IV.b).
2. Insert for every relevant suffix sa of ta an internal node $\overrightarrow{s}$ if such a node does not yet exist (I.b) and a leaf $\overrightarrow{sa}$ in the suffix-subtree (II.b).
3. Insert the node $\overrightarrow{\alpha(ta)}$ if it does not yet exist (III.b).

[5]Items (I.a) and (III.a) do not need to be considered explicitly because they do not describe a change in the node set.



$cat(t) = T_{1a}$
T_{1b}
$T_{1c} = T_1$
$\overrightarrow{\varepsilon}$
$\overrightarrow{\text{a}}$
$\overrightarrow{\text{...aa}}$
$\overrightarrow{\text{ab}}$
$\overrightarrow{\text{...aab}}$
$\overrightarrow{\text{aba}}$
$\overrightarrow{\text{aba}} = \overrightarrow{s}$
$\overrightarrow{\text{...aaba}}$
$\overrightarrow{\text{baba...}}$
$\overrightarrow{\text{...aabab}}$
$\overrightarrow{\text{ababa...}}$
$\overrightarrow{\text{...aababa...}}$
$\overrightarrow{\text{babaa...}}$
$(\overrightarrow{\text{...aabab}}, \text{a})$
$\overrightarrow{\text{ababaa...}}$
$\overrightarrow{\text{...aababaa...}}$
$\overrightarrow{\text{...aababa}}$

Figure 4.1: Elongation of the text ($T_{1a} \to T_{1b}$) and insertion of the node $\overrightarrow{t}$ ($T_{1b} \to T_{1c}$) for $t = \texttt{aababa}$, $a = \texttt{a}$.

4. If the active prefix lengthens, remove the node $\overrightarrow{\alpha(ta)} = \overrightarrow{\alpha^{-1}(ta)}$ (IV.a together with Theorem 4.2).

What follows is a detailed description of the individual steps. We will try, though, not to lose the global view or become overparticular.

We describe the necessary changes to the nodes of the tree and how the relevant locations are found. It will not be necessary to talk about node labels because the label (l, r) of an inserted node $\overrightarrow{w}$ always can be determined easily.

For the individual modifications of the tree we define functions that will be used in Section 4.3 for a concluding formulation of the online construction algorithm.

4.2.1 Elongation of the Text and Insertion of Node $\overrightarrow{t}$

The first step of Algorithm 4.6 consists of two parts that we consider here separately: the elongation of the text t yielding ta and the following insertion of node $\overrightarrow{t}$. These steps are illustrated in Figure 4.1 for the example $t = \texttt{aababa}$, $a = \texttt{a}$.

Elongation of the Text

Like in procedure *ukk* the automatic elongation of all leaves in the suffix tree destroys the structure of the tree. While in *ukk* an $\mathcal{A}^+$-tree remains, what we get here is not even a bi-tree. This becomes apparent at the active suffix leaf whose path-label changes in the suffix-subtree while its position in the prefix-subtree remains unchanged.

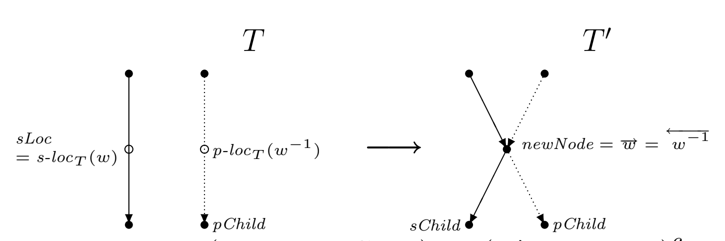
However, the structure remains a bi-tree if at the time of elongating the text all suffix-leaves are removed from the prefix subtree. This is done by deleting the suffix link of the active suffix leaf. Now, ignoring the prefix edges that connect the leaves in the suffix-subtree one again has a bi-tree that we denote by T_{1b}, see Figure 4.1. The active suffix leaf $\overrightarrow{\texttt{babaa...}}$ that was removed from the prefix-subtree is indicated by an additional circle ⦿.

In Step 3 of Algorithm 4.6 where after insertion of the node $\overrightarrow{\alpha(ta)}$ the active suffix is represented by an explicit node, the leaves of the suffix-subtree are re-attached by inserting an atomic prefix edge from node $\overrightarrow{\alpha(ta)}$ to the node that then is the active suffix leaf.

Insertion of the Node $\overrightarrow{t}$

In general, when inserting a node $\overrightarrow{w} = \overleftarrow{w^{-1}}$ into a bi-tree T, one first has to test if w is already an s-word and if w^{-1} is already a p-word in T. If that is the case, then w is already represented by implicit nodes in T. To make it explicit, one has to split the edges that "contain" w and/or w^{-1}, respectively.

For the case where both w is an s-word and w^{-1} is a p-word in T, we define the function *insert_iNode* (see box). This function splits the edge "containing" the implicit node *sLoc* and the prefix edge that ends at node *pChild* and then inserts the new node $\overrightarrow{w} = \overleftarrow{w^{-1}}$. Moreover, we assume that function *insert_iNode* is also defined if *sLoc* is already an explicit node, in which case nothing happens: $T' = T$ and $newNode = \overrightarrow{w}$; *pChild* is ignored.



$insert\_iNode(T, sLoc, pChild) \Rightarrow (T', newNode)$[6]

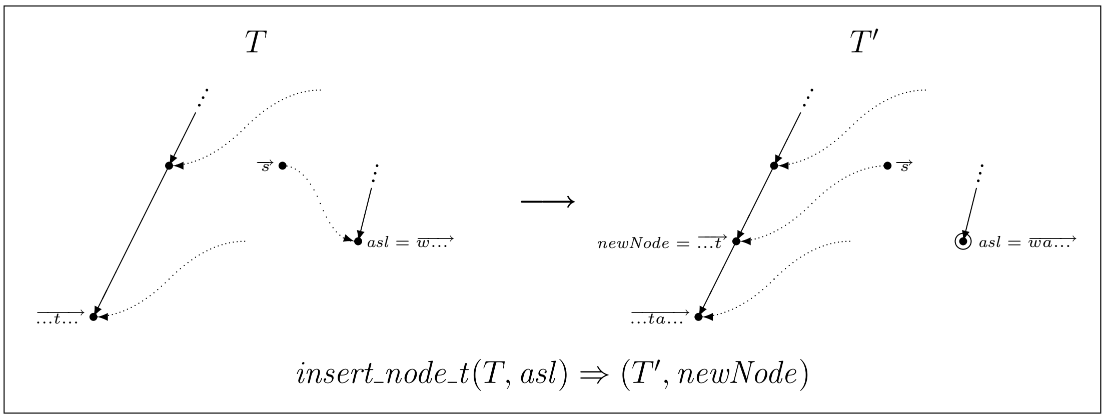
If w is not an s-word or w^{-1} is not a p-word in T, a new leaf has to be created in the respective subtree. Therefore one has to find the node that represents the longest prefix resp. suffix of w because that is where the edge leading to the new node starts.

In the special case of node $\overrightarrow{t}$ in the step $cat(t) \to cat(ta)$ we have:

- The previously explicit node $\overrightarrow{t}$ has become $\overrightarrow{ta}$ by the automatic elongation so that $s\text{-}loc_{T_{1b}}(t)$ is implicit. This location is "within" the suffix-edge that ends at node $\overrightarrow{ta}$.
- As a p-word t does not occur in T_{1b} any more because the edge that was removed in the step $cat(t) \to T_{1b}$ was part of the prefix-path to the former node $\overrightarrow{t}$. Hence, the new node $\overrightarrow{t}$ needs to be newly inserted as a new prefix leaf. As already mentioned in 3.4.2, node $\overrightarrow{s}$ where the (now removed) prefix edge leading to the active suffix leaf started is the longest nested suffix s of t that is represented by a node. Moreover, all nodes $\overrightarrow{u}$ that represent longer (non-nested) suffixes of t have been elongated to $\overrightarrow{ua}$. Hence, the prefix edge leading to $\overrightarrow{t}$ starts at node $\overrightarrow{s}$ which is the same location where the edge to the active suffix leaf was previously removed.

Because t is not a p-word in T_{1b}, we can not use function *insert_iNode* here. Instead we define a special function *insert_node_t* that performs the complete procedure in one step: elongation of the text, removal of the prefix edge that ends at the active suffix leaf, and insertion of the node $\overrightarrow{t}$ (more correctly the node $\overrightarrow{...t}$ because it is a prefix leaf), see the following box:

[6]The double arrow "⇒" in the function definition can be read as "yields".


T
T'
$\overrightarrow{s}$
$asl = \overrightarrow{w...}$
$\overrightarrow{...t...}$
$newNode = \overrightarrow{...t}$
$\overrightarrow{s}$
$asl = \overrightarrow{wa...}$
$\overrightarrow{...ta...}$
$insert\_node\_t(T, asl) \Rightarrow (T', newNode)$

Because suffix leaves are represented with open labels, function *insert_node_t* needs not be parameterized with the new character a.

4.2.2 Insertion of the Relevant Suffixes

Step 2 of Algorithm 4.6 describes an iteration over all relevant suffixes of ta and can be realized like function *update*[7] in *ukk*: Starting with the longest relevant suffix ua ($u = \alpha(t)$), in each step an internal node $\overrightarrow{u}$ is created if it does not already exist, and a suffix-leaf $\overrightarrow{ua...}$ is inserted. Between the steps u shrinks by its leftmost character which is achieved by traversing a suffix link followed by a "canonization". The procedure stops if u is empty or if for a (possibly shrunk) u there already exists a continuation a. In this case the new active suffix is $\alpha(ta) = ua$.

In contrast to *ukk*, however, here the underlying data structure is an affix tree, causing two serious differences:

1. Not only internal nodes, but also newly introduced leaves in the suffix-subtree have to be inserted in the corresponding position in the prefix-subtree. (In the suffix tree view: leaves need suffix links as well.)

2. Since prefix edges leading to internal nodes might be non-atomic, it is possible that by traversing a suffix link the active suffix is shortened by more than one character.

We now discuss in detail how these two problems can be solved.

Insertion of New Internal Nodes and Leaves

As in *ukk*, the s-location of the active suffix is an invariant during the whole construction, and hence when node $\overrightarrow{u}$ is inserted, its position in the suffix-subtree is known.

The corresponding p-location can also be found similar as it is done in algorithm *ukk*: A newly inserted node $\overrightarrow{u}$ splits the suffix link of the node that was inserted[8] previously, $\overrightarrow{.u}$. When inserting the first relevant suffix, i. e., when node $\overrightarrow{\alpha(t)}$ is inserted, the edge to be split

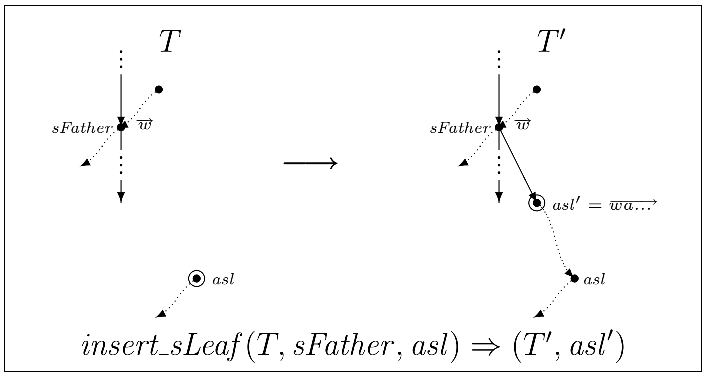
[7]To be precise, one has to use this function as it is described in [GK94b] or function *ukkstep* in [Kur95] while the functionality of *update* in [Ukk93] differs slightly.

[8]It is possible that node $\overrightarrow{.u}$ already exists in $cat(t)$. Then this node is not inserted but "passed".

is the suffix link of node $\overrightarrow{...t}$ because $\alpha(t)$ is the longest suffix of t that is represented by an explicit node in $cat(ta)$.

Similar for the leaves: The prefix edge starting at a newly inserted leaf $\overleftarrow{ua...}$ (the new active suffix leaf) always ends at the leaf $\overleftarrow{.ua...}$ inserted in the previous step (the previous active suffix leaf). This way all the leaves of the suffix-subtree keep being connected by atomic prefix edges, sorted according to their length. After this step there is no prefix edge ending at the leaf that was inserted last. As noted above, this edge is inserted not before Step 3.

The new internal nodes are inserted using function *insert_iNode* described above. The leaves in the suffix-subtree are inserted using a new function *insert_sLeaf* (see box). This function is parameterized with the tree T, the node *sFather* where the newly inserted suffix edge starts, and the node *asl* above which the new node is inserted into the prefix-subtree. The result is the modified tree T' and the newly inserted suffix leaf asl'.



T
T'
sFather
$\overleftarrow{w}$
sFather
$\overleftarrow{w}$
$asl' = \overleftarrow{wa...}$
asl
asl
$insert\_sLeaf(T, sFather, asl) \Rightarrow (T', asl')$

The complete procedure of inserting a relevant suffix ua into tree T is realized by function *insert_relevant_suffix*; *uLoc* is the location of u in T, *pChild* represents the node $\overrightarrow{...t}$ or the last inserted or "passed" internal node, and *asl* is the active suffix leaf:

$$\begin{aligned} & insert\_relevant\_suffix(T, uLoc, pChild, asl) \\ & \quad = (T'', new\_iNode, new\_sLeaf) \\ & \qquad \underline{\text{where}} \quad (T'', new\_sLeaf) = insert\_sLeaf(T', new\_iNode, asl) \\ & \qquad\qquad\quad\;\; (T', new\_iNode) = insert\_iNode(T, uLoc, pChild). \end{aligned}$$

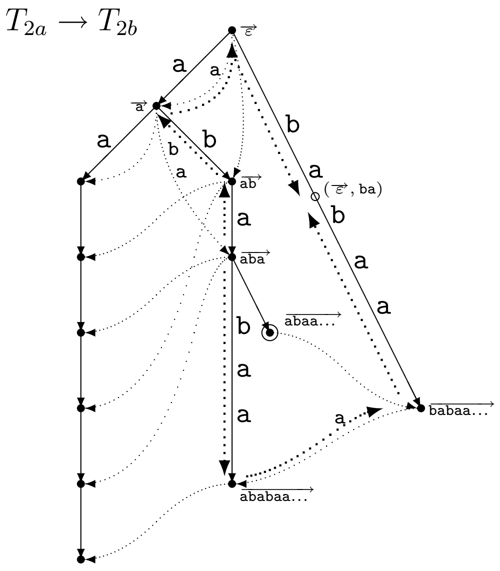
Shortening the Active Suffix

To shorten the active suffix, similar as in [Kur95] we define a function *linkloc*:

$$linkloc_T(s\text{-}loc_T(.w)) \Rightarrow s\text{-}loc_T(w).$$

"Naively" this function can be implemented by traversing the suffix-edges from the root while spelling the string w. From similar considerations on the suffix tree it is known, however, that this approach will lead to a quadratic-time algorithm (as an example, consider the text $t = \mathtt{a}^{n-1}\$$), prohibiting our desired goal of an online construction of $cat(t)$ in linear time. That is why in *ukk* suffix links are used to reach the location of the shortened active suffix in amortized $\mathcal{O}(1)$ time.

In affix trees due to the non-atomic suffix links, this is not always possible in the direct way. Sometimes one has to traverse a "detour" for which we suggest two possible ways, illustrated in Figure 4.2 ($t = \mathtt{aababa}$, $u = \mathtt{aba}$, $a = \mathtt{a}$; see also Figure 4.3) where the non-atomic suffix link $\overrightarrow{\mathtt{aba}} \xleftarrow{\mathtt{ba}} \overrightarrow{\mathtt{a}}$ "misses" the (implicit) node $\overrightarrow{\mathtt{ba}}$:



$T_{2a} \longrightarrow T_{2b}$
a
a
b
a
b
a
b
a
b
a
a
a
b
a
a
a
$\overrightarrow{\varepsilon}$
$\overrightarrow{\mathtt{a}}$
$\overrightarrow{\mathtt{ab}}$
$\overrightarrow{\mathtt{aba}}$
$\overrightarrow{\mathtt{abaa...}}$
$\overrightarrow{\mathtt{babaa...}}$
$\overrightarrow{\mathtt{ababaa...}}$
$(\overrightarrow{\varepsilon}, \mathtt{ba})$

Figure 4.2: The two possibilities to shorten the active suffix ($T_{2a} \longrightarrow T_{2b}$) for $t = \mathtt{aababa}$, $u = \mathtt{aba}$, $a = \mathtt{a}$ (see also Figure 4.3).

a) One starts at the active point and traverses "upwards" in the suffix-subtree until one reaches a node with an atomic suffix link:

$$(\overrightarrow{\mathtt{aba}}, \varepsilon) \xleftarrow{\mathtt{a}} (\overrightarrow{\mathtt{ab}}, \mathtt{a}) \xleftarrow{\mathtt{b}} (\overrightarrow{\mathtt{a}}, \mathtt{ba}).$$

Then one follows this suffix link

$$(\overrightarrow{\mathtt{a}}, \mathtt{ba}) \overset{\mathtt{a}}{\prec\cdots\cdots} (\overrightarrow{\varepsilon}, \mathtt{ba}),$$

which shortens the active suffix. Finally one canonizes as in *ukk* (this is not necessary in the example of Figure 4.2 because $(\overrightarrow{\varepsilon}, \mathtt{ba})$ is already a canonical reference pair).

The upwards-traversal at the latest terminates at the root where in general we have a reference pair $(\overrightarrow{\varepsilon}, cu)$. Here, the active suffix can be shortened simply by removing the first character yielding $(\overrightarrow{\varepsilon}, u)$.

b) The second possibility is similar but traverses "downwards": One follows the suffix edges (but not the just inserted edge to the active suffix leaf) until one reaches a node with an atomic suffix link

$$(\overrightarrow{\mathtt{aba}}, \varepsilon) \xrightarrow{\mathtt{baa}} (\overrightarrow{\mathtt{ababaa...}}, -\mathtt{baa}),^{9}$$

then one follows this link

$$(\overrightarrow{\mathtt{ababaa...}}, -\mathtt{baa}) \overset{\mathtt{a}}{\prec\cdots\cdots} (\overrightarrow{\mathtt{babaa...}}, -\mathtt{baa})$$

[9]The notation "$-u$" denotes a negative offset: $(\overrightarrow{v}, -u)$ is an s-reference pair of w if $\overrightarrow{v}$ is a node in T and $v = wu$.

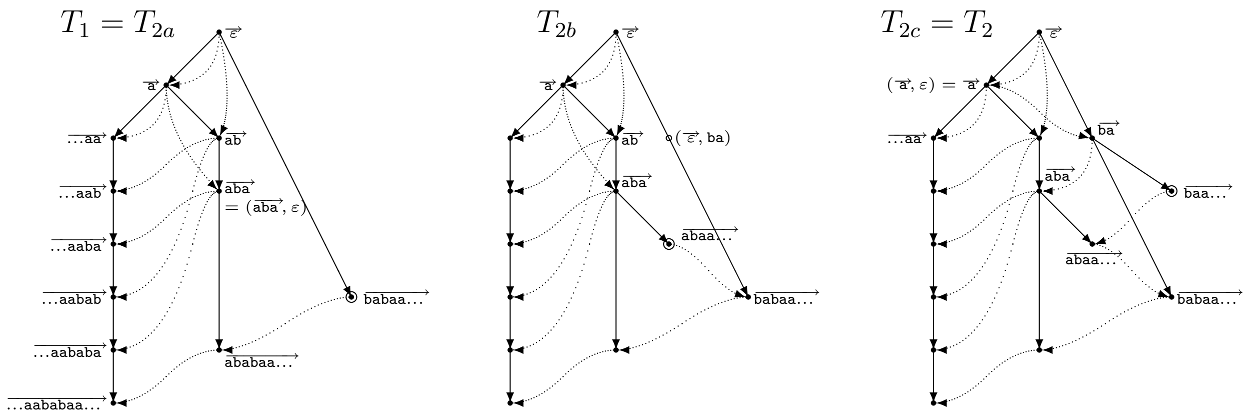

$T_1 = T_{2a}$
$\overrightarrow{\varepsilon}$
$\overrightarrow{\texttt{a}}$
$\overrightarrow{\texttt{...aa}}$
$\overrightarrow{\texttt{ab}}$
$\overrightarrow{\texttt{...aab}}$
$\overrightarrow{\texttt{aba}} = (\overrightarrow{\texttt{aba}}, \varepsilon)$
$\overrightarrow{\texttt{...aaba}}$
$\overrightarrow{\texttt{...aabab}}$
$\overrightarrow{\texttt{babaa...}}$
$\overrightarrow{\texttt{...aababa}}$
$\overrightarrow{\texttt{ababaa...}}$
$\overrightarrow{\texttt{...aababaa...}}$
T_{2b}
$\overrightarrow{\varepsilon}$
$\overrightarrow{\texttt{a}}$
$\overrightarrow{\texttt{ab}}$
$(\overrightarrow{\varepsilon}, \texttt{ba})$
$\overrightarrow{\texttt{aba}}$
$\overrightarrow{\texttt{abaa...}}$
$\overrightarrow{\texttt{babaa...}}$
$T_{2c} = T_2$
$\overrightarrow{\varepsilon}$
$(\overrightarrow{\texttt{a}}, \varepsilon) = \overrightarrow{\texttt{a}}$
$\overrightarrow{\texttt{...aa}}$
$\overrightarrow{\texttt{ba}}$
$\overrightarrow{\texttt{aba}}$
$\overrightarrow{\texttt{baa...}}$
$\overrightarrow{\texttt{abaa...}}$
$\overrightarrow{\texttt{babaa...}}$

Figure 4.3: Insertion of the relevant suffixes for $t = \texttt{aababa}$ and $a = \texttt{a}$.

and canonizes "upwards" by the appropriate amount:

$$(\overrightarrow{\texttt{babaa...}}, -\texttt{baa}) \overset{\texttt{babaa}}{\longleftarrow} (\overrightarrow{\varepsilon}, \texttt{ba}).$$

There is a simple reason why following the path "downwards" always is unique: None of the nodes with a non-atomic suffix link is branching in the suffix-subtree (see Remark 3 on Definition 3.1).

The downwards-traversal surely terminates because at a suffix-leaf there is always an atomic suffix link. It is not possible that this is the active suffix leaf (whose suffix link was deleted previously) because this case was explicitly excluded above.

The insertion of all relevant suffixes of ta is described by Function *ukkstep*[10]:

$$\begin{aligned}
&\mathit{ukkstep}(T, a, \mathit{actS}, \mathit{pChild}, \mathit{asl}) \\
&\quad = \begin{cases} (T, \mathit{down}_T(\mathit{actS}, a), \mathit{asl}), & \underline{\text{if}}\ \mathit{occurs}(\mathit{actS}, a) \\ (T', \mathit{actS}, \mathit{asl}'), & \underline{\text{else if}}\ \mathit{actS} = (\mathit{root}(T), \varepsilon) \\ \mathit{ukkstep}(T', a, \mathit{linkloc}_{T'}(\mathit{actS}), \mathit{pChild}', \mathit{asl}'), & \underline{\text{otherwise}} \end{cases} \\
&\qquad \underline{\text{where}}\ \ (T', \mathit{pChild}', \mathit{asl}') = \mathit{insert\_relevant\_suffix}(T, \mathit{actS}, \mathit{pChild}, \mathit{asl}).
\end{aligned}$$

Throughout the procedure, *actS* denotes the current location of the active suffix. The predicate $\mathit{occurs}(\mathit{actS}, a)$ (see [Kur95]) tests if there exists a continuation a at location *actS*, and $\mathit{down}_T(\mathit{s\text{-}loc}_T(w), a)$ yields $\mathit{s\text{-}loc}_T(wa)$.

In summary, we have:

$$\mathit{ukkstep}(T_1, a, \mathit{actS}_1, \mathit{node\_t}, \mathit{asl}_1) \Rightarrow (T_2, \mathit{actS}_2, \mathit{asl}_2),$$

where T_i, actS_i and asl_i denote the bi-tree, the location of the active suffix and the active suffix leaf after Step i of Algorithm 4.6, and *node_t* is the node $\overrightarrow{...t}$ that was introduced in Step 1.

Figure 4.3 shows the two iterations for $t = \texttt{aababa}$, $a = \texttt{a}$.

[10]For an explanation see also [Kur95].

4.2.3 Making the Active Suffix Explicit

In the next step, the insertion of node $\overrightarrow{\alpha(ta)}$, we will encounter a similar difficulty as above in function *linkloc*. The following fact describes a property of the tree T_2 that will be helpful:

Theorem 4.7 (Active Suffix after Step 2)
If the new active suffix $ua = \alpha(ta)$ in T_2 is not represented by an explicit node, then it is of the form $s\text{-}loc_{T_2}(\alpha(ta)) = (\overrightarrow{u}, a)$. □

Proof
Let $ua = \alpha(ta)$ after Step 2 of Algorithm 4.6 be not explicit. We show that $s\text{-}loc_{T_2}(\alpha(ta)) = (\overrightarrow{u}, a)$ which is clearly the case if $\overrightarrow{u}$ is an explicit node.

There are two possibilities for Step 2:

a) Before encountering an a-continuation at u, the active suffix has already been shortened by one or more characters. Then there surely exists a different continuation $b \neq a$ of $\alpha(t) = \_.u$ in $t = \left\{ \begin{array}{c} \_\_.u \\ \_ub\_\_ \end{array} \right\}$, and hence u is a right-branching ta-word.

b) ta does not have any relevant suffixes, the active suffix immediately is elongated: $\alpha(ta) = \alpha(t)a = ua$. In this case $\overrightarrow{u}$ was already an explicit node in $cat(t)$:

   Assume that $\overrightarrow{u} = \overrightarrow{\alpha(t)}$ was not an explicit node in $cat(t)$. Then it was removed in Step 4 of the previous iteration of the construction because the active prefix has been elongated there. By Theorem 4.1 in combination with Theorem 4.2 then we have that $t = \left\{ \begin{array}{c} ua\_\_ \\ \_\_.u \end{array} \right\}$ and there is no other occurrence of ua in t.

   This means, however, that ua is a non-nested prefix of t and hence was represented as an explicit node $\overrightarrow{ua}$ in $cat(t)$. This is in contradiction to the assumption of the theorem.

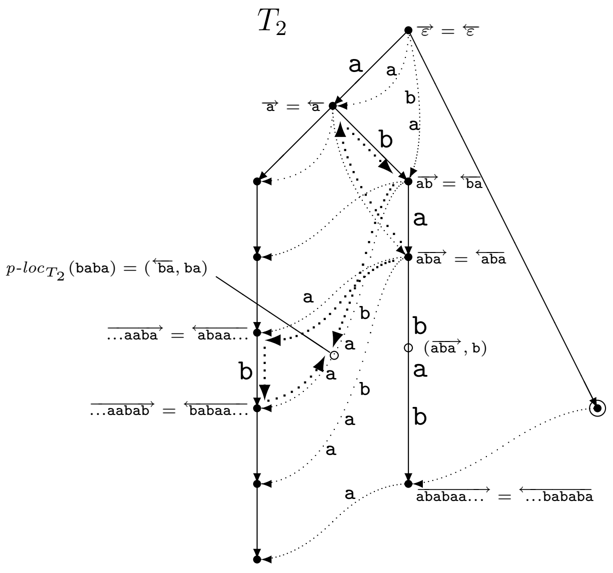
□

Hence, whenever at the beginning of Step 3 the active suffix is not explicit, we have the following situation: The implicit node $s\text{-}loc_{T_2}(\alpha(ta)) = (\overrightarrow{u}, a)$ needs to be made explicit. In order to find the corresponding location in the prefix-subtree, $p\text{-}loc_{T_2}(au^{-1})$, we define function *find_pLoc*:

$$find\_pLoc_T(s\text{-}loc_T(w)) \Rightarrow p\text{-}loc_T(w^{-1}).$$

Similar as above we suggest two possible ways for finding the desired p-location[11], see Figure 4.4 ($t =$ `aababa`, $u =$ `aba`, $a =$ `b`):

[11] The "naive" solution to always walk into the tree starting at the root here requires quadratic time in the worst case as well (for an example consider $t = \mathtt{a}(\mathtt{ab})^i$ as in Figure 5.1). This already caused Weiner in the development of his "classic" suffix tree construction algorithm [Wei73] to traverse a detour "inside" the tree which is very similar to the "upward" way we describe here under a).



T_2
$\overrightarrow{\varepsilon} = \overleftarrow{\varepsilon}$
a
a
b
a
$\overrightarrow{\mathtt{a}} = \overleftarrow{\mathtt{a}}$
b
$\overrightarrow{\mathtt{ab}} = \overleftarrow{\mathtt{ba}}$
a
$\overrightarrow{\mathtt{aba}} = \overleftarrow{\mathtt{aba}}$
$p\text{-}loc_{T_2}(\mathtt{baba}) = (\overleftarrow{\mathtt{ba}}, \mathtt{ba})$
a
b
$\overrightarrow{\mathtt{...aaba}} = \overleftarrow{\mathtt{abaa...}}$
a
b
$(\overrightarrow{\mathtt{aba}}, \mathtt{b})$
b
a
a
b
$\overrightarrow{\mathtt{...aabab}} = \overleftarrow{\mathtt{babaa...}}$
b
a
b
a
a
$\overrightarrow{\mathtt{ababaa...}} = \overleftarrow{\mathtt{...bababa}}$

Figure 4.4: The two possibilities to find the corresponding location of the active suffix in the prefix-subtree for $t = \mathtt{aababa}$, $u = \mathtt{aba}$, $a = \mathtt{b}$.

a) One starts at the active point and walks "upwards" until one reaches a node where an atomic a-suffix edge starts:

$$(\overleftarrow{\mathtt{aba}}, \varepsilon) \overset{\mathtt{ab}}{\dashleftarrow} (\overleftarrow{\mathtt{a}}, \mathtt{ba}).$$

Then one follows this edge

$$(\overleftarrow{\mathtt{a}}, \mathtt{ba}) \xrightarrow{\mathtt{b}} (\overleftarrow{\mathtt{ba}}, \mathtt{ba})$$

and finally one "canonizes" by the appropriate amount "downwards". (In the example this is not necessary: $(\overleftarrow{\mathtt{ba}}, \mathtt{ba})$ is already a canonical p-reference pair.)

Again, the way "upwards" at the latest terminates at the root of the tree where $(\overleftarrow{\varepsilon}, u^{-1})$ can be elongated manually yielding $(\overleftarrow{\varepsilon}, au^{-1})$.

b) The second possibility is similar but traverses "downwards" in the tree: One follows the prefix edges[12] until one reaches a node where an atomic a-suffix edge starts (this is definitely the case when a leaf is reached)

$$(\overleftarrow{\mathtt{aba}}, \varepsilon) \overset{\mathtt{a}}{\dashrightarrow} (\overleftarrow{\mathtt{abaa...}}, -\mathtt{a}),$$

then one follows this edge

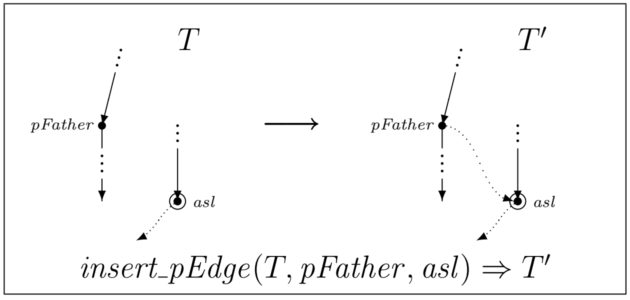
$$(\overleftarrow{\mathtt{abaa...}}, -\mathtt{a}) \xrightarrow{\mathtt{b}} (\overleftarrow{\mathtt{babaa...}}, -\mathtt{a}),$$

and finally one "canonizes" by the appropriate amount "upwards":

$$(\overleftarrow{\mathtt{babaa...}}, -\mathtt{a}) \overset{\mathtt{aab}}{\dashleftarrow} (\overleftarrow{\mathtt{ba}}, \mathtt{ba}).$$

[12]During this walk there might be more than one prefix edge yielding "downwards". In such a case one has to look at the successor node along the a-suffix edge. This node is not branching in the prefix-subtree (otherwise it had an atomic prefix link) and hence the prefix edge starting there begins with the same character as the edge one has to follow because it leads to a node that lies "below" the location $p\text{-}loc_{T_2}(au^{-1})$ we are aiming at.

After this step, node $\overrightarrow{\alpha(ta)}$ will be explicit and as mentioned above the leaves in the suffix-subtree can be re-inserted into the tree by a prefix edge that connects $\overrightarrow{\alpha(ta)}$ and the active suffix leaf *asl*. To this end, we define function *insert_pEdge* (see box).



$insert\_pEdge(T, pFather, asl) \Rightarrow T'$

The complete Step 3 of Algorithm 4.6 is implemented by function *insert_asl*:

$$\begin{aligned} insert\_asl(T, actS, asl) \;=\; & (insert\_pEdge(T', actSnode, asl), actSnode) \\ & \underline{\text{where}} \quad (T', actSnode) = insert\_iNode(T, actS, pChild) \\ & \qquad\quad (\bullet, \_, \_, pChild) = find\_pLoc_T(actS). \end{aligned}$$

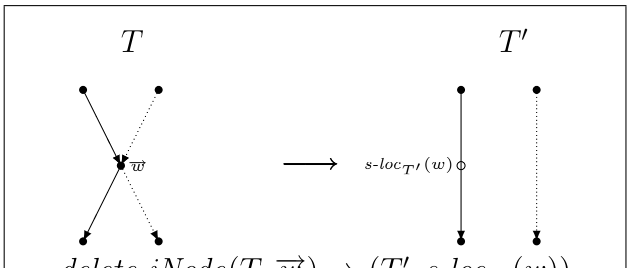
Note that *find_pLoc* is only evaluated if *actS* is an implicit node because only in this case *insert_iNode* uses its argument *pChild*.

4.2.4 Elongation of the Active Prefix

The following lemma follows from a combination of Theorems 4.2 and 4.4 and allows to efficiently test for the condition "if the active prefix lengthens":

Lemma 4.8 (Elongation of the Active Prefix)
The new active suffix $\alpha(ta)$ is represented in $cat(t)$ by a prefix leaf if and only if the active prefix lengthens in the step $t \to ta$. $\square$

Then removing the node $\overrightarrow{\alpha(ta)}$ is straightforward: One just connects the suffix edges leading upwards and downwards as well as the prefix edges leading upwards and downwards, which is carried out by function *delete_iNode* (see box).



$delete\_iNode(T, \overrightarrow{w}) \Rightarrow (T', s\text{-}loc_{T'}(w))$

The complete Step 4 of Algorithm 4.6 is implemented by function *lengthen_actP*:

$$\begin{aligned} lengthen\_actP(T, actSnode) \;=\; & delete\_iNode(T, actSnode), && \underline{\text{if}}\ actSnode = \overrightarrow{...u} \\ =\; & (T, (actSnode, \varepsilon)), && \underline{\text{otherwise}}\ . \end{aligned}$$

Figure 4.5 continues our example $t = \texttt{aababa}$, $a = \texttt{a}$.

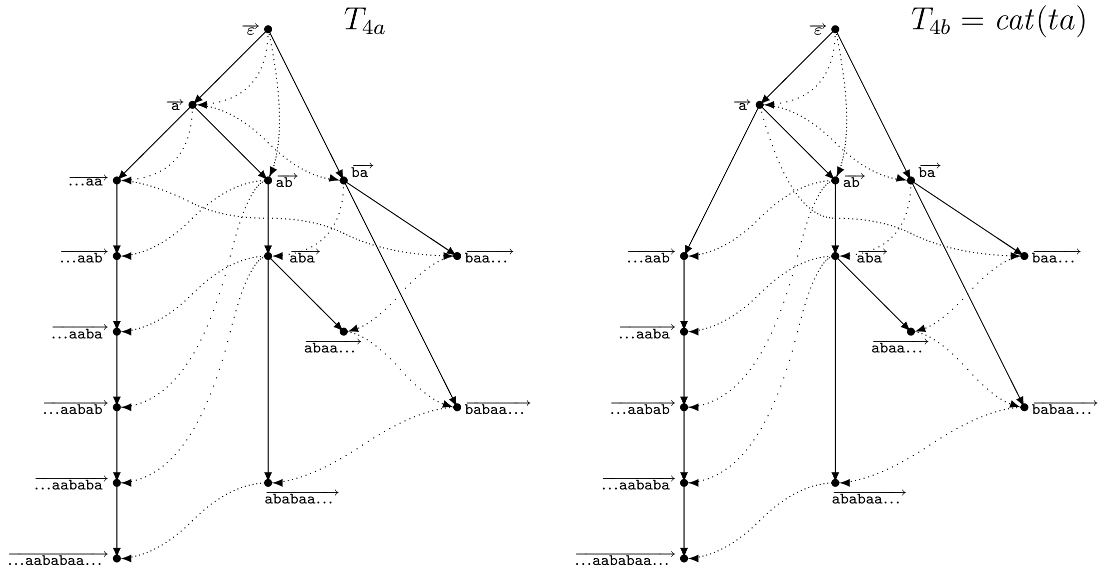

T_{4a}
$\overrightarrow{\varepsilon}$
$\overrightarrow{\texttt{a}}$
$\overrightarrow{\texttt{...aa}}$
$\overrightarrow{\texttt{ab}}$
$\overrightarrow{\texttt{ba}}$
$\overrightarrow{\texttt{...aab}}$
$\overrightarrow{\texttt{aba}}$
$\overrightarrow{\texttt{baa...}}$
$\overrightarrow{\texttt{...aaba}}$
$\overrightarrow{\texttt{abaa...}}$
$\overrightarrow{\texttt{...aabab}}$
$\overrightarrow{\texttt{babaa...}}$
$\overrightarrow{\texttt{...aababa}}$
$\overrightarrow{\texttt{ababaa...}}$
$\overrightarrow{\texttt{...aababaa...}}$
$T_{4b} = cat(ta)$
$\overrightarrow{\varepsilon}$
$\overrightarrow{\texttt{a}}$
$\overrightarrow{\texttt{ab}}$
$\overrightarrow{\texttt{ba}}$
$\overrightarrow{\texttt{...aab}}$
$\overrightarrow{\texttt{aba}}$
$\overrightarrow{\texttt{baa...}}$
$\overrightarrow{\texttt{...aaba}}$
$\overrightarrow{\texttt{abaa...}}$
$\overrightarrow{\texttt{...aabab}}$
$\overrightarrow{\texttt{babaa...}}$
$\overrightarrow{\texttt{...aababa}}$
$\overrightarrow{\texttt{ababaa...}}$
$\overrightarrow{\texttt{...aababaa...}}$

Figure 4.5: Removal of the active node for $t = \texttt{aababa}$, $a = \texttt{a}$, $actS = (\overrightarrow{\texttt{...aa}}, \varepsilon)$.

4.3 The Online-Algorithm (Unidirectional)

By composition of the four functions that implement Steps 1 – 4 of Algorithm 4.6, we get *catstep*:

$$
\begin{aligned}
\mathit{catstep}(T, a, \mathit{actS}, \mathit{asl}) = (T_4, \mathit{actS}_4, \mathit{asl}_2)\quad & \\
\underline{\text{where}}\quad (T_4, \mathit{actS}_4) &= \mathit{lengthen\_actP}(T_3, \mathit{actSnode}_3)\\
(T_3, \mathit{actSnode}_3) &= \mathit{insert\_asl}(T_2, \mathit{actS}_2, \mathit{asl}_2)\\
(T_2, \mathit{actS}_2, \mathit{asl}_2) &= \mathit{ukkstep}(T_1, a, \mathit{actS}, \mathit{node\_t}, \mathit{asl})\\
(T_1, \mathit{node\_t}) &= \mathit{insert\_node\_t}(T, \mathit{asl}).
\end{aligned}
$$

Function *cat* iteratively applies *catstep* to all characters of a string:

$$
\begin{aligned}
\mathit{cat}(T, \varepsilon, \mathit{actS}, \mathit{asl}) &= T\\
\mathit{cat}(T, aw, \mathit{actS}, \mathit{asl}) &= \mathit{cat}(T', w, \mathit{actS}', \mathit{asl}')\\
&\qquad \underline{\text{where}}\quad (T', \mathit{actS}', \mathit{asl}') = \mathit{catstep}(T, a, \mathit{actS}, \mathit{asl}).
\end{aligned}
$$

Summarizing this chapter, we have:

Theorem 4.9 (Online Construction of Compact Affix Trees)
Let $T_0 = \mathit{cat}(\varepsilon)$. Then $\mathit{cat}(T_0, t, (\mathit{root}(T_0), \varepsilon), \mathit{root}(T_0))$ constructs the compact affix tree of text t.[13] □

Above we have reformulated Algorithm 4.6 in a functional way. Nevertheless, function *cat* can not easily be implemented in a purely functional programming language because the

[13]For a proper applicability of $\mathit{insert\_node\_t}(T_0, \mathit{root}(T_0))$, the root of $T_0 = \mathit{cat}(\varepsilon)$ needs a suffix link and a prefix link pointing at itself. These links, however, are irrelevant for the remaining construction and therefore have not been mentioned so far. Alternatively, one could also directly start with $\mathit{cat}(a)$.

algorithm locally modifies the global structure of the affix tree which is not possible with the usual data types of traditional lazy functional programming languages.[14] However, because the demand for *single-threadedness* is fulfilled (i. e., no copies of parts of the global data structure are made), an implementation by more modern techniques as provided by present-day languages like Haskell or Clean [Pla95] might be possible. Such an implementation of *ukk* is described in [Kur95] and [Kra95]. Here we have, however, refrained from a similar functional implementation of *cat* because such an undertaking seemed to be too far away from the original problem definition.

4.4 Comments on the Bidirectional Application

Affix trees are completely symmetrical with respect to an exchange of suffix and prefix view. Hence, without any modification Algorithm 4.6 can be applied in the opposite direction $cat(t) \rightarrow cat(at)$ as well.

The two neuralgic points where all modifications in the tree are performed then are the location of the active prefix and the active prefix leaf. As for the active suffix and the active suffix leaf these are invariants of the construction, i. e., before the step $cat(t) \rightarrow cat(at)$ the p-location of $\alpha^{-1}(t)$ and the active prefix leaf in $cat(t)$ need to be known, and after this step the corresponding locations in $cat(at)$ are available.

To be able to extend t in both right and left direction, though, one has to ensure that during the extension in one direction, the invariants needed for the other direction keep up-to-date.

To see how this can be done, we again have a look at the step $cat(t) \rightarrow cat(ta)$:

1. In Section 4.1.1 we have discussed the possible changes of the active prefix. Because in *lengthen_actP* it is tested anyway if the active prefix lengthens, it is easy to perform the necessary changes of the reference pair of the active prefix there as well.

   Moreover, it is possible that the active prefix is made explicit during the insertion of an internal node in function *insert_relevant_suffix*. Hence, in such a case the reference pair of the active prefix needs to be updated as well.

2. The active prefix leaf might change in two situations:

   (a) If before inserting the node $\overrightarrow{t}$ the (elongated) node $\overrightarrow{ta}$ was the active prefix leaf, then $\overrightarrow{t}$ becomes the active prefix leaf.

   (b) It is easy to see that the node that is deleted in function *lengthen_actP* always is the active prefix leaf. The new active prefix leaf then is the prefix leaf that is one character longer and was connected to the old one by an atomic suffix edge.

It is possible during the online affix tree construction in one direction to keep the invariants of the other direction up-to-date. Hence the method is bidirectional.

[14]A detailed discussion of these matters for algorithm *ukk* can be found in [GK94b].

Chapter 5

Complexity Analysis

The goal of this chapter is to show that the online construction algorithm presented in Chapter 4 runs in practically linear time and in linear space. While the linear space requirements are obvious[1], the worst case time analysis turns out to be unexpectedly difficult.

Indeed, we can not finally answer the question about the time complexity of our algorithm, although a few hints in favor of linearity exist. Additionally, the time complexity also depends on the alphabet size. This, as in *ukk*, is implied by the fact that the branching degree of the tree increases proportionally to the alphabet size. Here, however, we will not consider such effects and rather assume a constant alphabet size.[2]

5.1 Overview

We separately discuss the functions *insert_node_t*, *ukkstep*, *insert_asl*, and *lengthen_actP* that implement the four steps of Algorithm 4.6. Because already the unidirectional analysis is rather difficult, we refrain from discussing cases where the direction of the construction changes in an unfavorable situation.

Step 1: *insert_node_t*

Each of the n steps for the construction of $cat(t)$ calls *insert_node_t* exactly once. The active suffix leaf is known at that time and *insert_node_t* does not contain any loops or recursions. Hence the total time usage is $\mathcal{O}(n)$.

Step 2: *ukkstep*

While it is known from [Ukk93] that the online construction of compact suffix trees is possible in $\mathcal{O}(n)$ time, the proof can not without further consideration be adopted to the modified

[1]Apart from the text and the tree which are both of size $\mathcal{O}(n)$ no additional space is used. Although during the construction nodes are removed from the tree, the size of the tree increases monotonically (in each step at most one node is removed and at least one node is inserted), and hence we can exclude the possibility that intermediate structures are larger than the final affix tree.

[2]For a discussion of the effect of the alphabet size on the construction time of suffix trees see [Kur95] who compares the different possibilities to find the correct successor at a branching node in the tree.

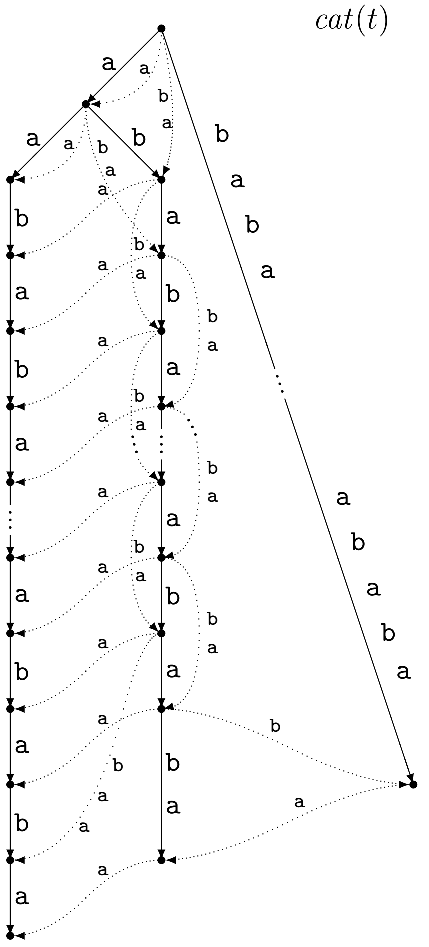


Figure 5.1: An example for quadratic behavior of *ukkstep* if *linkloc* always chooses the "upwards" way: $t = \mathtt{aa}(\mathtt{ba})^i$, $a = \mathtt{a}$.

data structure of the suffix-subtree in the affix tree.

While the number of new nodes and leaves inserted into the tree is the same as in *ukk*, here it is more expensive to reach the corresponding locations in the tree by function *linkloc*. If in the search for an atomic suffix link one always chooses the "upwards" way (which is most similar to the corresponding step in *ukk*), the algorithm requires quadratic time in the worst case as one can easily see for $t = \mathtt{aa}(\mathtt{ba})^i$, $a = \mathtt{a}$ (Figure 5.1).

This is why in the following we assume that always the shorter of the two possible ways ("upwards" or "downwards") described in Section 4.2.2 is chosen. This can be realized, for example, by traversing both directions in parallel, and as soon as one of the two ways encounters an atomic suffix link, the other one is stopped.

For the detailed analysis, as in [Ukk93] we split *linkloc* in three sub-functions: *find_aLink*, *canonize_down*, and *canonize_up*. See Sections 5.2.1 – 5.2.3.

Step 3: *insert_asl*

While the insertion of the new node is possible in constant time and the parameters *actS* and *asl* are known because they are invariants of the construction, determining the *p*-location of the new active suffix using function *find_pLoc* might be expensive. Due to our experiences with *linkloc*, here we also always choose the shorter of the two possible ways. Section 5.2.4 contains the details of the analysis.

Step 4: *lengthen_actP*

Following Lemma 4.8, the condition for an elongation of the active prefix was: "If the new active suffix is represented by a prefix leaf". This can be tested in $\mathcal{O}(1)$ time, and removing the node $\overrightarrow{\alpha(ta)}$ is possible in constant time as well. Hence for the complete text all calls to *lengthen_actP* together require $\mathcal{O}(n)$ time.

5.2 Discussion of the Difficult Cases

5.2.1 *find_aLink*

The following theorem shows that the search for an atomic suffix link is necessary only if the canonic reference pair of the (possibly already shortened) active suffix *actS* is an explicit, only left-branching node:

Theorem 5.1 (Atomic Suffix Link of the Implicit Active Suffix)
If *actS* is an implicit node at the time of calling *linkloc* in *ukkstep* and it is not the root, hence $actS = (\overrightarrow{bu}, cv)$, then it always has an atomic suffix link $\overrightarrow{bu} \overset{b}{\dashleftarrow} \overrightarrow{u}$. □

Proof
Let $actS = (\overrightarrow{bu}, cv)$ be the canonical reference pair of the (possibly shortened) active suffix of $t = \left\{ \begin{matrix} \_\_.bucv \\ \_bucv.\_ \end{matrix} \right\}$. We show that $\overrightarrow{bu}$ has an atomic suffix link, i. e., $\overrightarrow{bu} \overset{b}{\dashleftarrow} \overrightarrow{u} \in$ *p-edges*$(cat(t))$.[3] Following Remark 3 on Definition 3.1 this is certainly the case if $\overrightarrow{bu}$ is branching in the suffix-subtree of $cat(t)$.

Suppose for a contradiction that $\overrightarrow{bu}$ is non-branching in the suffix-subtree of $cat(t)$, which is equivalent to *bu* being a left- but not a right-branching *t*-word. Two cases are possible:

1. $t = \left\{ \begin{matrix} \_\_xbucv \\ \_ybuc\_\_ \end{matrix} \right\}, x \neq y$, hence *buc* is a left-branching *t*-word as well, which is in contradiction to the assumption that $(\overrightarrow{bu}, cv)$ is a canonical reference pair.

[3]While at this point of the algorithm the tree has already been modified, the location considered here still is the same as in $cat(t)$.

2. $t = \left\{ \begin{array}{c} \_\_xbucv \\ \_xbucv.\_ \\ \_\_\_\_ybu \end{array} \right\}, x \neq y \quad \Rightarrow \quad bucv = \_ybu$

   $\Rightarrow \quad t = \left\{ \begin{array}{c} \_\_xbucv \\ \_x\_ybu.\_ \\ \_\_\_\_ybu \end{array} \right\} \quad \Rightarrow \quad . = c$, because bu by the assumption is not right-branching

   $\Rightarrow \quad t = \left\{ \begin{array}{c} \_\_xbucv \\ \_x\_ybuc\_ \\ \_\_\_\_ybu \end{array} \right\}, x \neq y,$

   i. e., buc is a left-branching t-word which, as above, is a contradiction to the assumption that $(\overrightarrow{bu}, cv)$ is a canonical reference pair.

□

Find_aLink hence requires more than one step only if *actS* is an explicit node that is branching only in the prefix-subtree.

What follows are some remarks on this class of nodes which represent only left-branching t-words:

1. Such nodes are created in *cat* only by function *insert_asl*. Hence there can exist maximally n of them during the whole construction.

2. If in a series of two or more steps the set of relevant suffixes is empty (and hence the active suffix several times in a row is only elongated without being shortened in between), then a “chain” of these nodes is formed.

   Such a “chain” causes the quadratic behavior of function *find_aLink* if one always chooses the “upwards” direction (Figure 5.1).

3. Each call to *find_aLink*, however, “disarms” such a node by inserting a new atomic suffix link that leads to the node that will be inserted in the next step.

Due to Items 1 and 3, hence, function *find_aLink* takes at most $\mathcal{O}(n \log n)$ time if always the shorter of the two ways is chosen: The maximal total length of all “chains” is n. Each call to *find_aLink* splits a chain in two sub-chains which in later steps are further split, and so on. In the worst case, each time one has to walk the maximal number of steps, which is the case if each time a chain is split in two halves of equal length. This gives the logarithmic behavior.

This result, however, by no means includes all constraints that diminish the number of steps needed by *find_aLink*. For example, not every sequence of active suffixes is possible. In the described scenario the active suffix has to be located two or more times in the same “chain”. To achieve this, however, there have to be certain steps in between which might amortize the additional effort for walking the same chain several times.

The existence of this and other constraints and the empirical results presented in Section 5.3 suggest a linear worst-case behavior of *find_aLink*.

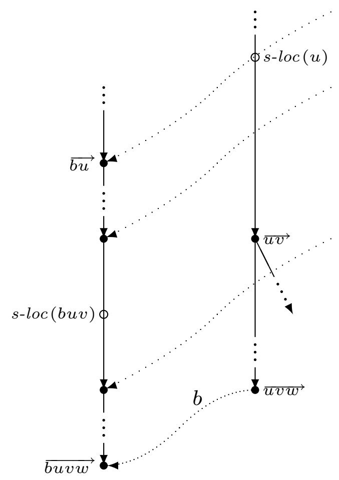


Figure 5.2: On the discussion of *canonize_up*.

5.2.2 *canonize_down*

The proof for the amortized linear time behavior of the "downwards canonization" can be taken over from [Ukk93]:

The second element of the reference pair of the active suffix is elongated during the whole construction at most n times. Because each step of *canonize_down* shortens this element by at least one character, in total there are at most n steps possible for *canonize_down*.

5.2.3 *canonize_up*

The problem here is the following (see Figure 5.2): Let $actS = (\overrightarrow{bu}, \varepsilon)$ be the active suffix that is going to be shortened, and let $\overrightarrow{buvw}$ be the first node "below" $\overrightarrow{bu}$ with an atomic suffix link. How many nodes $\overrightarrow{uv}$ on the "way back" are maximally possible?

The following holds for node $\overrightarrow{uv}$:

1. $\overrightarrow{buv}$ is not an explicit node, otherwise *find_aLink* would have come across an atomic suffix link $\overrightarrow{buv} \overset{b}{\leftarrow\cdots} \overrightarrow{uv}$ already there.

2. $\overrightarrow{uv}$ is not branching in the prefix-subtree because not even $\overrightarrow{u}$ is branching in the prefix subtree. Hence $\overrightarrow{uv}$ is branching only in the suffix-subtree.

Unfortunately these observations do not give a criterion for the maximal number of nodes traversed by *canonize_up*. The empirical results given in Section 5.3, however, hint at a particularly harmless linear behavior with a constant factor of $\frac{1}{2}$ in the worst case.

5.2.4 *find_pLoc*

Function *find_pLoc* seems to be the sub-function of *cat* with the most delicate behavior. Nevertheless, two interesting observations can be made:

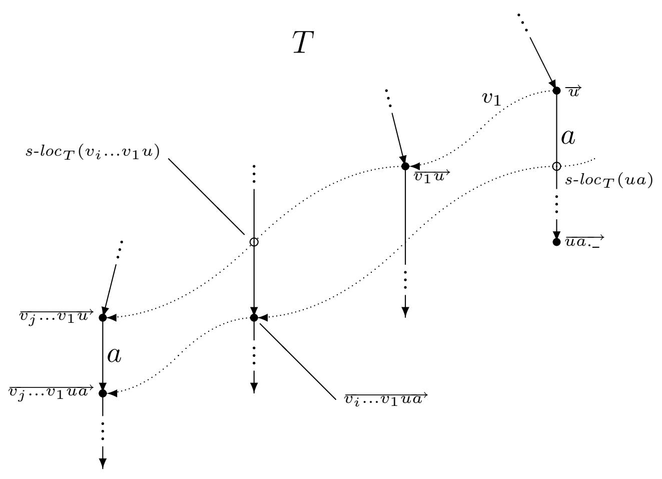

T
$\overrightarrow{u}$
v_1
a
$s\text{-}loc_T(v_i...v_1u)$
$\overrightarrow{v_1u}$
$s\text{-}loc_T(ua)$
$\overrightarrow{ua\_}$
$\overrightarrow{v_j...v_1u}$
a
$\overrightarrow{v_j...v_1ua}$
$\overrightarrow{v_i...v_1ua}$

Figure 5.3: A node $\overrightarrow{v_i \dots v_1ua}$ can not exist in T.

First, it is possible to show that a "canonization" is not necessary if for finding the corresponding p-location one chooses the "downwards" direction.

The scenario has already been described in Section 4.2.3 (see also Figure 5.3): An "upward canonization" is only necessary if there exists an (internal) node of the form $\overrightarrow{v_i \dots v_1ua}$. Such a node, however, can not exist because $v_i \dots v_1ua$ is neither left-branching (otherwise $v_i \dots v_1u$ would be left-branching as well and be represented by an explicit node) nor right-branching (with the same reasoning for ua).

Moreover, we have observed that if one always chooses the shorter way, also in cases where the upwards direction is chosen, no canonization is necessary. This motivates the following claim that, however, can not be proven here, but was empirically verified on all strings over $\{\mathtt{a}, \mathtt{b}\}$ up to a length of $n = 20$:

Proposition 5.2 (Downwards-Canonization is Not Necessary)
Whenever in function *find_pLoc* the "upwards" walk in its canonization phase traverses nodes, then the "downwards" walk takes only one step. More precisely, there exists an edge $\overrightarrow{u} \dashrightarrow \overrightarrow{...v_1u}$ that ends at a prefix leaf. □

Nevertheless, the main question remains unanswered, namely how far it is to reach the next atomic a-edge from node $\overrightarrow{u}$.

As long as this is unsolved, one has to assume the complexity $\mathcal{O}(n^2)$ of the naive solution, although the empirical results of the next section do not exclude even a linear time behavior.

5.3 Empirical Results

Because we were unable to derive satisfactory analytic results for the complexity of functions *find_aLink*, *canonize_up*, and *find_pLoc*, we have empirically counted the number of steps performed in these functions. The goal of these investigations was

- either to find examples where the method behaves worse than linear,[4]
- or to get hints for further analytic studies, for example by determination of the constant factor that upper-bounds the number of steps in the worst case.

Tables 5.1, 5.2 and 5.3 summarize the results. For increasing text lengths n we have created all strings $t \in \{\mathtt{a}, \mathtt{b}\}^n$. (Strings that are equal except for the exchange of `a` and `b`, however, are considered only once.) The text length n is given in the left column, the center column contains the maximal number of steps of the respective function, and the right column, in parentheses, shows the number of occurrences of this worst case (again, strings that are equal upon exchanging `a` and `b` are counted only once) as well as the "worst case texts" themselves.

While *find_aLink* and *canonize_up* show a clear linear behavior (and hence it surprises that the analytic proof seems so difficult), the interpretation of the results of *find_pLoc* is more difficult:

In the beginning, the pattern $t = \mathtt{ab}^i$ seems to describe the worst case (with a linear behavior and a constant factor of 1). Starting with $n = 13$, however, a second pattern joins in, $t = \mathtt{aab}^{2i}\mathtt{ab}^i\mathtt{a}$, which for $n = 16$ even surpasses the first pattern. This second pattern (if generalized for longer sequences) again describes a linear behavior with a constant factor of 1.33. Beginning with $n = 26$, another pattern is found with a worse behavior ($t = \mathtt{aab}^{4i}\mathtt{ab}^{2i}\mathtt{ab}^i\mathtt{a}$ with a constant factor of 1.43), and so on.

As a consequence of these observations, one might ask if this series of patterns converges against a sequence of worst case patterns. And does this generalized pattern still show a linear behavior or does the constant factor constantly increase such that the general behavior is worse than linear?

One possible generalization of the above worst case patterns is the sequence

$$t = \mathtt{aab}^{2^k i}\mathtt{ab}^{2^{k-1} i} \ldots \mathtt{ab}^{2i}\mathtt{ab}^i\mathtt{a}, \quad n = (2^{k+1} - 1)i + (k + 3).$$

The following table shows the number of steps of *find_pLoc* for some values of k and i:

i \ k	1	2	3	5	10	15
2	6	18	42	186	6138	196602
3	10	28	64	280	9208	294904
5	18	48	108	468	15348	491508
10	38	98	218	938	30698	983018
20	78	198	438	1878	61398	?
100	398	998	2198	9398	306998	?
in general	$4i - 2$	$10i - 2$	$22i - 2$	$94i - 2$	$3070i - 2$	$98302i - 2$
n	$3i + 4$	$7i + 5$	$15i + 6$	$63i + 8$	$2047i + 13$	$65535i + 18$
$\frac{\text{\# of steps}}{n}$	$\sim \frac{4}{3}$ ≈ 1.33	$\sim \frac{10}{7}$ ≈ 1.43	$\sim \frac{22}{15}$ ≈ 1.467	$\sim \frac{94}{63}$ ≈ 1.492	$\sim \frac{3070}{2047}$ ≈ 1.4998	$\sim \frac{98302}{65535}$ ≈ 1.499992

[4]Indeed, this way we found the text $ta = \mathtt{aa}(\mathtt{ba})^i\mathtt{a}$ which provides an example for the quadratic complexity of *find_aLink* if always the "upwards" direction is chosen.

text length n	# of steps in the worst case	(#)	texts
0	0		—
1	0		—
2	0		—
3	0		—
4	0		—
5	0		—
6	1	(1)	$\mathtt{a(ab)^2b}$
7	1	(4)	$\mathtt{a(ab)^2aa}, \mathtt{a(ab)^2bb}, \mathtt{aa(ab)^2b}, \mathtt{ba(ab)^2b}$
8	2	(1)	$\mathtt{a(ab)^3b}$
9	3	(1)	$\mathtt{a(ab)^3aa}$
10	4	(1)	$\mathtt{a(ab)^4b}$
11	5	(1)	$\mathtt{a(ab)^4aa}$
12	6	(1)	$\mathtt{a(ab)^5b}$
13	7	(1)	$\mathtt{a(ab)^5aa}$
14	8	(1)	$\mathtt{a(ab)^6b}$
15	9	(1)	$\mathtt{a(ab)^6aa}$
16	10	(1)	$\mathtt{a(ab)^7b}$
17	11	(1)	$\mathtt{a(ab)^7aa}$
18	12	(1)	$\mathtt{a(ab)^8b}$
19	13	(1)	$\mathtt{a(ab)^8aa}$
20	14	(1)	$\mathtt{a(ab)^9b}$
21	15	(1)	$\mathtt{a(ab)^9aa}$
22	16	(1)	$\mathtt{a(ab)^{10}b}$
23	17	(1)	$\mathtt{a(ab)^{10}aa}$
24	18	(1)	$\mathtt{a(ab)^{11}b}$
25	19	(1)	$\mathtt{a(ab)^{11}aa}$
26	20	(1)	$\mathtt{a(ab)^{12}b}$
27	21	(1)	$\mathtt{a(ab)^{12}aa}$
28	22	(1)	$\mathtt{a(ab)^{13}b}$
29	23	(1)	$\mathtt{a(ab)^{13}aa}$
30	24	(1)	$\mathtt{a(ab)^{14}b}$
31	25	(1)	$\mathtt{a(ab)^{14}aa}$
32	26	(1)	$\mathtt{a(ab)^{15}b}$
33	27	(1)	$\mathtt{a(ab)^{15}aa}$
34	28	(1)	$\mathtt{a(ab)^{16}b}$

Table 5.1: Worst cases of *find_aLink* for texts up to a length of 34 over the alphabet $\{\mathtt{a}, \mathtt{b}\}$.

text length n	# of steps in the worst case	(#)	texts
0	0		—
1	0		—
2	0		—
3	0		—
4	0		—
5	0		—
6	0		—
7	1	(1)	$\mathtt{a(ab)^2aa}$
8	1	(4)	$\mathtt{a(ab)^3b}, \ldots$
9	1	(13)	...
10	2	(1)	$\mathtt{a(ab)^4b}$
11	2	(6)	...
12	3	(1)	$\mathtt{a(ab)^5b}$
13	3	(4)	$\mathtt{a(ab)^5bb}, \mathtt{a(ab)^5aa}, \mathtt{aa(ab)^5b}, \mathtt{ba(ab)^5b}$
14	4	(1)	$\mathtt{a(ab)^6b}$
15	4	(4)	...
16	5	(1)	$\mathtt{a(ab)^7b}$
17	5	(4)	...
18	6	(1)	$\mathtt{a(ab)^8b}$
19	6	(4)	...
20	7	(1)	$\mathtt{a(ab)^9b}$
21	7	(4)	...
22	8	(1)	$\mathtt{a(ab)^{10}b}$
23	8	(4)	...
24	9	(1)	$\mathtt{a(ab)^{11}b}$
25	9	(4)	...
26	10	(1)	$\mathtt{a(ab)^{12}b}$
27	10	(4)	...
28	11	(1)	$\mathtt{a(ab)^{13}b}$
29	11	(4)	...
30	12	(1)	$\mathtt{a(ab)^{14}b}$
31	12	(4)	...
32	13	(1)	$\mathtt{a(ab)^{15}b}$
33	13	(4)	...
34	14	(1)	$\mathtt{a(ab)^{16}b}$

Table 5.2: Worst cases of *canonize_up* for texts up to a length of 34 over the alphabet $\{\mathtt{a}, \mathtt{b}\}$.

text length n	# of steps in the worst case		(#)	texts
0	0			—
1	0			—
2	0			—
3	0			—
4	1	(+1)	(1)	$\texttt{ab}^{3}$
5	2	(+1)	(1)	$\texttt{ab}^{4}$
6	3	(+1)	(1)	$\texttt{ab}^{5}$
7	4	(+1)	(1)	$\texttt{ab}^{6}$
8	5	(+1)	(1)	$\texttt{ab}^{7}$
9	6	(+1)	(1)	$\texttt{ab}^{8}$
10	7	(+1)	(1)	$\texttt{ab}^{9}$
11	8	(+1)	(1)	$\texttt{ab}^{10}$
12	9	(+1)	(1)	$\texttt{ab}^{11}$
13	10	(+1)	(2)	$\texttt{ab}^{12}$, $\texttt{aab}^{6}\texttt{ab}^{3}\texttt{a}$
14	11	(+1)	(3)	$\texttt{ab}^{13}$, $\texttt{aab}^{7}\texttt{ab}^{3}\texttt{a}$, $\texttt{aab}^{6}\texttt{ab}^{3}\texttt{ab}$
15	12	(+1)	(6)	$\texttt{ab}^{14}$, $\texttt{aab}^{8}\texttt{ab}^{3}\texttt{a}$, $\texttt{aab}^{7}\texttt{ab}^{4}\texttt{a}$, ...
16	14	(+2)	(1)	$\texttt{aab}^{8}\texttt{ab}^{4}\texttt{a}$
17	15	(+1)	(2)	$\texttt{aab}^{9}\texttt{ab}^{4}\texttt{a}$, $\texttt{aab}^{8}\texttt{ab}^{4}\texttt{ab}$
18	16	(+1)	(6)	...
19	18	(+2)	(2)	$\texttt{aab}^{10}\texttt{ab}^{5}\texttt{a}$, $\texttt{aab}^{8}\texttt{ab}^{4}\texttt{ab}^{2}\texttt{a}$
20	19	(+1)	(4)	...
21	20	(+1)	(10)	...
22	22	(+2)	(3)	$\texttt{ab}^{12}\texttt{ab}^{6}\texttt{a}$, $\texttt{aab}^{10}\texttt{ab}^{5}\texttt{ab}^{2}\texttt{a}$, $\texttt{aab}^{8}\texttt{ab}^{4}\texttt{ab}^{2}\texttt{ab}^{2}\texttt{a}$
23	23	(+1)	(7)	...
24	24	(+1)	(18)	...
25	26	(+2)	(5)	$\texttt{aab}^{14}\texttt{ab}^{7}\texttt{a}$, $\texttt{aab}^{11}\texttt{ab}^{6}\texttt{ab}^{3}\texttt{a}$, ...
26	28	(+2)	(1)	$\texttt{aab}^{12}\texttt{ab}^{6}\texttt{ab}^{3}\texttt{a}$
27	29	(+1)	(2)	$\texttt{aab}^{13}\texttt{ab}^{6}\texttt{ab}^{3}\texttt{a}$, $\texttt{aab}^{12}\texttt{ab}^{6}\texttt{ab}^{3}\texttt{ab}$
28	30	(+1)	(10)	...
29	32	(+2)	(2)	$\texttt{aab}^{14}\texttt{ab}^{7}\texttt{ab}^{3}\texttt{a}$, $\texttt{aab}^{11}\texttt{ab}^{6}\texttt{ab}^{3}\texttt{ab}^{3}\texttt{a}$
30	34	(+2)	(1)	$\texttt{aab}^{12}\texttt{ab}^{6}\texttt{ab}^{3}\texttt{ab}^{3}\texttt{a}$
31	35	(+1)	(2)	$\texttt{aab}^{13}\texttt{ab}^{6}\texttt{ab}^{3}\texttt{ab}^{3}\texttt{a}$, $\texttt{aab}^{12}\texttt{ab}^{6}\texttt{ab}^{3}\texttt{ab}^{3}\texttt{ab}$
32	36	(+1)	(8)	...
33	38	(+2)	(3)	$\texttt{aab}^{16}\texttt{ab}^{8}\texttt{ab}^{4}$, $\texttt{aab}^{14}\texttt{ab}^{7}\texttt{ab}^{3}\texttt{ab}^{3}$, $\texttt{aab}^{11}\texttt{ab}^{6}\texttt{ab}^{3}\texttt{ab}^{3}\texttt{ab}^{4}$
34	40	(+2)	(1)	$\texttt{aab}^{12}\texttt{ab}^{6}\texttt{ab}^{3}\texttt{ab}^{3}\texttt{ab}^{4}$

Table 5.3: Worst cases of *find_pLoc* for texts up to a length of 34 over the alphabet $\{\texttt{a}, \texttt{b}\}$.

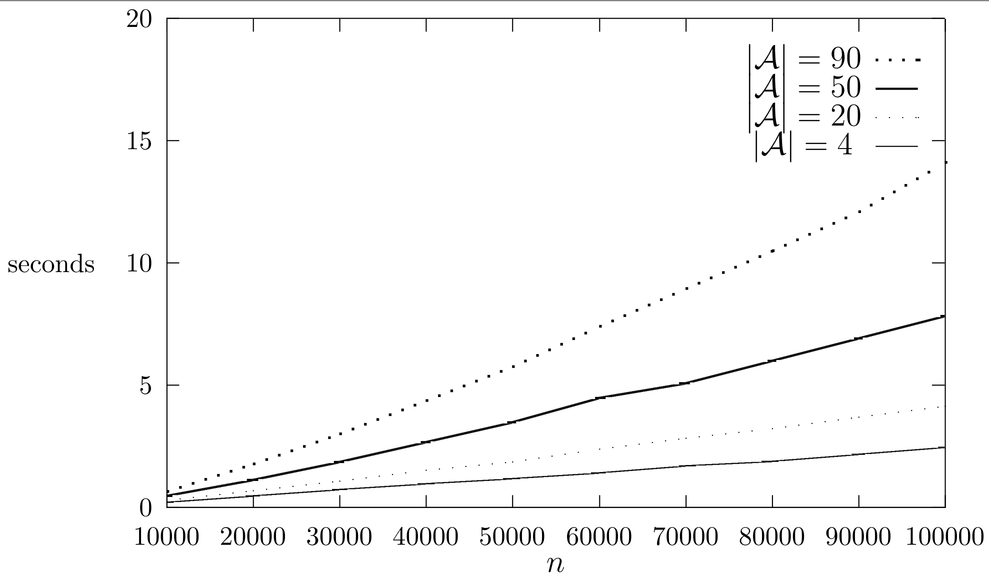


Figure 5.4: Running times of *bigrep* for random texts versus the text length n for alphabets of different size $|\mathcal{A}|$.

For strings of this type hence the ratio of the number of steps of *find_pLoc* divided by the string length n seems to stay below 1.5. The first derivative of the number of steps in the worst case (the numbers in parentheses in the center column of Table 5.3) suggest a linear behavior with the same factor: For larger n, two steps (+1) and two steps (+2) alternate.

Unfortunately, due to the exponentially growing number of sequences to be examined these empirical measurements could not be carried on further. In particular, it was almost impossible to perform similar measurements for larger alphabets. Because the measurements performed ($|\mathcal{A}| = 3$ up to $n = 20$, $|\mathcal{A}| = 4$ up to $n = 15$) in the worst case always yielded exactly the same results as were obtained for $|\mathcal{A}| = 2$, one can assume, though, that the alphabet $\{\mathtt{a}, \mathtt{b}\}$ already allows to form all "bad" sequences.

5.4 Computation Times

Finally we validated the above results for longer texts as they are more relevant in practice. We measured the computation times for random texts over alphabets of different size (again using a Bernoulli distribution). The results are shown in Figure 5.4.[5]

It is easy to see the increase of the running time with the alphabet size due to the larger branching degree, a behavior that is well known from *ukk*. The curves also suggest a linear increase with respect to the text length, although a slight "sagging" can be observed. However, this could also result from the memory management of the computer and hence needs not be overemphasized. At least in the average case the assumption of a time complexity of $\mathcal{O}(|\mathcal{A}|n)$ seems to be supported by the data.

[5]The runs were measured on a SPARC station 20 with 32 MB of main memory. The memory, however was never completely used (maximal process size: 13 MB). The operating system was Solaris 2.4, the times were measured using the Unix tool *rusage*. All values are averages over ten independent runs.

By a quantitative comparison with the running times of *ukk*[6] we found that the construction of *cat* takes about three times as long, see the following table:[7]

type of text ($n = 100000$)	*ukk*	*cat*
random sequence, $\vert\mathcal{A}\vert = 4$	0.7	2.2
random sequence, $\vert\mathcal{A}\vert = 20$	1.6	4.3
random sequence, $\vert\mathcal{A}\vert = 50$	2.8	7.9
random sequence, $\vert\mathcal{A}\vert = 90$	4.8	14.6
genetic data[8]	0.8	2.3
manual page	1.2	2.9
Fibonacci string	0.2	1.3

Moreover, the memory used is about four times as high. This is due to the larger size of the data structure. To store the affix tree not only one needs to store two suffix trees; it is also necessary to represent all edges bidirectionally and to store leaves as complete nodes because they simultaneously are internal nodes of the other subtree.

Nevertheless, the resources of the computer described above sufficed to construct affix trees of texts up to a length of 250 000 characters within the available main memory.

[6] Here we used the C implementation described in [GK95].

[7] The time measurements are in seconds, again averaged over ten runs.

[8] Excerpt from the nucleotide sequence of chromosome XI of *Saccharomyces cerevisiae* over the DNA alphabet $\mathcal{A} = \{C, G, T, A\}$.

Chapter 6

The Program *bigrep*

Here we briefly describe a simple application of affix trees that was developed as part of this thesis. The program *bigrep* can be used to find (short) patterns in (long) texts similar to the well known Unix tool *grep*.

The distinctive feature of *bigrep* is that in contrast to *grep* and all of its known variants both text and pattern are treated bidirectionally. Moreover, it is possible to search the same text for the occurrence of several patterns one after the other, and the text can be extended interactively in both directions.

The following commands are recognized by *bigrep*:

`r` <*text*>	appends <*text*> to the right of the current text,
`l` <*text*>	appends <*text*>$^{-1}$ to the left of the current text,
`f` <*pattern*>	searches for <*pattern*> in the current text,
`b` <*pattern*>	searches for <*pattern*>$^{-1}$ in the current text,
`t`	prints the current text,
`a`	prints the compact affix tree of the current text,
`c`	deletes the current text,
`q`	quits *bigrep*,
`?`	displays a short help message.

Internally, upon entering or extending a text by the commands `r` and `l`, the compact affix tree for this text is created. The commands `f` and `b`, respectively, initiate a search for <*pattern*> following the suffix- resp. prefix-edges of this tree. If this way an occurrence of <*pattern*> resp. <*pattern*>$^{-1}$ in the text is found, the program outputs `SUCCESS`; if at some point there is no continuation in the tree, the program outputs `FAIL`.

The program *bigrep* is a direct implementation of the online construction algorithm as described in Chapter 4 of this thesis. Hence, no further comments are necessary here.

Chapter 7

Summary

This thesis was devoted to a new data structure called affix trees. Affix trees generalize suffix trees for bidirectional applications, supporting exact string matching in a flexible manner.

The presentation started with the introduction of the basic data structures. We defined a general notion of duality for $\mathcal{A}^+$-trees, and we developed the data structure of a bi-tree, on the basis of which the affix tree was introduced. We have shown the duality of affix trees, the linear space requirement of their compact variant, we have determined their maximal size, discussed different possibilities for their representation, and have chosen the representation which is best suited for affix trees. Because affix trees completely subsume the structure of suffix trees, it was easy to see that all applications of suffix trees can be carried over to affix trees. We have suggested three ways to construct affix trees. One of these possibilities, the bidirectional online construction was detailed in the fourth chapter. In doing so, we found out that the algorithm is not a simple generalization of the corresponding method by Ukkonen for compact suffix trees. Several larger extensions were necessary. The following complexity analysis could not, as was initially hoped, show the linearity of the method, although for most of its parts this was possible. Three steps, however, could not be answered to satisfaction. For this reason we studied these sub-procedures empirically which at least for two of the three problematic cases clearly showed a linear behavior. The third case could not be classified uniquely, though. A generalization of the worst cases we found, as well as measurements of the actual running times using random sequences hinted at a linear time behavior as well.

We have implemented two methods for the construction of compact affix trees. One inefficient program in the lazy functional programming language Haskell as a rapid prototype, and an efficient implementation of the online construction algorithm in the programming language C. The latter one is by a factor of three slower than a corresponding implementation for suffix trees.

On the other hand, the larger data structure also has its advantages: text and pattern are handled bidirectionally which enables new pattern matching methods as they might be applied in bioinformatics and other disciplines with similar search problems. Moreover, one could examine if for genetic applications instead of the reverse union the definition of a “reverse complementary union” would be advantageous where the two complementary strands of a DNA double helix are represented and can be studied in the same tree. An extension of the matching procedure for approximate matching or for use in data compression methods seems possible as well.

We have seen: Affix trees support everything that is supported by suffix trees, and even more; or in Apostolico's words:

> Affix trees seem to outperform subword trees in versatility and elegance.

Bibliography

[AHU74] A. V. Aho, J. E. Hopcroft, and J. D. Ullman. *The Design and Analysis of Computer Algorithms.* Addison-Wesley, Reading, MA, 1974.

[Apo85] A. Apostolico. The myriad virtues of subword trees. In A. Apostolico and Z. Galil, editors, *Combinatorial Algorithms on Words*, volume F12 of *NATO ASI Series*, pages 85–96. Springer Verlag, Berlin, 1985.

[AS92] A. Apostolico and W. Szpankowski. Self-alignments in words and their applications. *J. Algorithms*, 13:446–467, 1992.

[BEH89] A. Blumer, A. Ehrenfeucht, and D. Haussler. Average sizes of suffix trees and DAWGS. *Discrete Applied Mathematics*, 24:37–45, 1989.

[BM77] R. S. Boyer and J. S. Moore. A fast string searching algorithm. *Commun. ACM*, 20(10):762–772, 1977.

[CL94] W. I. Chang and E. L. Lawler. Sublinear approximate string matching and biological applications. *Algorithmica*, 12(4/5):327–344, 1994.

[CS85] M. T. Chen and J. Seiferas. Efficient and elegant subword tree construction. In A. Apostolico and Z. Galil, editors, *Combinatorial Algorithms on Words.* Springer Verlag, 1985.

[FHPW92] J. H. Fasel, P. Hudak, S. Peyton-Jones, and P. Wadler. *Special Issue on the Functional Programming Language Haskell.* ACM SIGPLAN Notices 27(5), 1992.

[GK94a] R. Giegerich and S. Kurtz. From Ukkonen to McCreight and Weiner: A unifying view of linear-time suffix tree construction. Report Nr. 94-03, Technische Fakultät der Universität Bielefeld, 1994. Journal version appeared in *Algorithmica*, 19:331-353, 1997.

[GK94b] R. Giegerich and S. Kurtz. Suffix trees in the functional programming paradigm. In *Proceedings of the European Symposium on Programming (ESOP'94)*, volume 788 of *LNCS*, pages 225–240. Springer Verlag, 1994.

[GK95] R. Giegerich and S. Kurtz. A comparison of imperative and purely functional suffix tree constructions. *Science of Computer Programming*, 25:187–218, 1995.

[KR78] B. W. Kernighan and D. M. Ritchie. *The C Programming Language.* Prentice-Hall, Englewood Cliffs, NJ, 1978.

[Kra95] A. Krause. Realisierung von Zustandskonzepten in funktionalen Programmiersprachen am Beispiel linearer Suffixbaum-Konstruktionen. Diplomarbeit, Technische Fakultät der Universität Bielefeld, 1995.

[Kur95] S. Kurtz. *Fundamental Algorithms for a Declarative Pattern Matching System.* Report Nr. 95-03, Technische Fakultät der Universität Bielefeld, 1995. Dissertation Thesis.

[McC76] E. M. McCreight. A space-economical suffix tree construction algorithm. *Journal of the ACM*, 23(2):262–272, 1976.

[Mei86] H. Meijer. *Programmar: A Translator Generator.* Ph. D. thesis, University of Nijmegen, 1986.

[Pla95] R. Plasmeijer. *Clean User's Manual.* Computing Science Department, University of Nijmegen, 1995.

[Ukk93] E. Ukkonen. On-line construction of suffix-trees. Report, A-1993-1, Dep. of Computer Science, University of Helsinki, Finland, 1993. Journal version appeared in *Algorithmica*, 14:249-260, 1995.

[Wei73] P. Weiner. Linear pattern matching algorithms. In *IEEE 14th Annual Symposium on Switching and Automata Theory*, pages 1–11. IEEE Press, 1973.

99–02 Transforming Conditional Rewrite Systems with Extra Variables into Unconditional Systems
(Enno Ohlebusch)

99–03 A Framework for Evaluating Approaches to Fuzzy Quantification
(Ingo Glöckner)

99–04 Towards Evaluation of Docking Hypotheses using elastic Matching
(Steffen Neumann, Stefan Posch und Gerhard Sagerer)

99–05 A Systematic Approach to Dynamic Programming in Bioinformatics. Part 1 and 2: Sequence Comparison and RNA Folding
(Robert Giegerich)

99–06 Autonomie für situierte Robotersysteme – Stand und Entwicklungslinien
(Alois Knoll)

2000–01 Advances in DFS Theory
(Ingo Glöckner)

2000–02 A Broad Class of DFS Models
(Ingo Glöckner)

2000–03 An Axiomatic Theory of Fuzzy Quantifiers in Natural Languages
(Ingo Glöckner)

2000–04 Affix Trees
(Jens Stoye)