Affixbäume

Jens Stoye
Technische Fakultät der Universität Bielefeld
Lehrstuhl: Praktische Informatik
Betreuer: Robert Giegerich

12. Mai 1995

Inhaltsverzeichnis

Kapitel 1

Einleitung

Eine der klassischen Disziplinen in der praktischen Informatik beschäftigt sich mit Suchalgorithmen. Neben anderen fallen darunter Verfahren zur Erkennung von (kurzen) Mustern in (langen) Texten, wie sie beispielsweise in der Linguistik oder der Bioinformatik Einsatz finden.

Man unterscheidet zwischen exakter und approximativer Textsuche. Bei der approximativen Suche ist eine Abweichung des Musters von seinem Auftreten im Text bis zu einer gewissen Schranke zugelassen, wogegen bei der exakten Suche davon ausgegangen wird, daß das Muster und die entsprechende Stelle im Text präzise übereinstimmen müssen.

Innerhalb der exakten Suche bedient man sich in Abhängigkeit von der jeweiligen Situation verschiedener Techniken: Sind Muster und Text im voraus nicht bekannt, werden auf dem Boyer-Moore-Algorithmus [BM77] basierende Verfahren eingesetzt, bei denen das Muster auf einfache Weise vorverarbeitet wird, um eine effizientere Suche zu ermöglichen. Bei im voraus bekannten Mustern, nach denen in verschiedenen Texten gesucht werden soll, kommen Techniken mit endlichen Automaten zum Einsatz. Falls der umgekehrte Fall vorliegt, daß der Text statisch bekannt ist und das Muster von Suche zu Suche variiert, so bietet sich eine Vorverarbeitung des Textes an.

Eine häufig dabei erzeugte Datenstruktur sind (kompakte) Suffixbäume. In diesen sind die Teilworte eines Textes auf besonders nützliche Art und Weise aufbereitet (deshalb wird auch die Bezeichnung *Subwort-Baum/subword tree* verwendet). Die Vielseitigkeit dieser Datenstruktur beschreibt Apostolico folgendermaßen:

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

Es sind verschiedene Verfahren zur effizienten Konstruktion von kompakten Suffixbäumen bekannt, angefangen mit dem „Klassiker“ von Weiner [Wei73], über die Verfahren von McCreight [McC76] und Chen und Seiferas [CS85] bis zu der *online*-Konstruktion von Ukkonen [Ukk93], bei der der Text zeichenweise von links nach rechts gelesen wird und nach dem i-ten Schritt der Suffixbaum für die ersten i Zeichen fertig vorliegt.

Bei der Betrachtung dieses Verfahrens stellt sich die Frage, ob das Einlesen des Textes auch in entgegengesetzter Richtung oder sogar ein bidirektionales Vorgehen möglich ist, so daß an einer beliebigen Stelle im Text begonnen werden und die Konstruktion in beiden Richtungen erfolgen kann. In der vorliegenden Arbeit wird sich herausstellen, daß hierfür eine Erweiterung der Datenstruktur von Vorteil ist. Der so entstehende Graph wird als Affixbaum bezeichnet und besitzt die in [GK94a] beschriebene Dualitätseigenschaft, die kompakte Suffixbäume nicht aufweisen. Da die Größe der Affixbäume sich nicht signifikant von der der Suffixbäume unterscheidet, ist zu untersuchen, ob sie auch wie diese in bzgl. der Textlänge linearer Zeit konstruiert werden können.

Den Ausgangspunkt dieser Arbeit bilden [GK94a] und [Kur95]. Auch bei der Weiterentwicklung des ursprünglich von Ukkonen stammenden Algorithmus wird die in [Kur95] verwendete Formulierung übernommen.

Die Arbeit ist wie folgt aufgebaut: Zunächst werden grundlegende Datenstrukturen eingeführt, auf deren Basis in dem darauffolgenden Kapitel Affixbäume definiert und einige ihrer Eigenschaften diskutiert werden. Im vierten Kapitel wird die oben angedeutete *online*-Konstruktion von Affixbäumen ausführlich besprochen, im fünften Kapitel deren Komplexität analysiert, was sich aber als schwierig herausstellen wird, so daß kein endgültiges Ergebnis präsentiert werden kann. Schließlich wird kurz eine Anwendung von Affixbäumen vorgestellt.

Der Code der im Rahmen dieser Arbeit erstellten Computerprogramme ist bei Bedarf beim Autor zu erhalten.

Kapitel 2

Grundlegende Datenstrukturen

In [GK94a] wird eine Dualität von atomaren Suffixbäumen und den zugehörigen Suffixlink-Bäumen beschrieben. Um zu zeigen, daß diese Dualität nicht allein eine Eigenschaft von Suffixbäumen ist, wird in dieser Arbeit ein allgemeinerer Zugang als über Suffixlinks gewählt. Auf diese Weise werden von vornherein gewisse Symmetrien von der resultierenden Datenstruktur gefordert, die später von großer Bedeutung sind.

Die in diesem Kapitel eingeführte Notation und Terminologie folgt in Anlehnung an [GK94a] und [Kur95].

2.1 Grundbegriffe

Sei $\mathcal{A}$ eine endliche Menge von Zeichen, das *Alphabet*. Eine Sequenz von Zeichen aus $\mathcal{A}$ heißt *Wort* oder *Text* über $\mathcal{A}$, $|t|$ bezeichnet die *Länge* des Textes t, d. h. die Anzahl Zeichen, aus denen t besteht. Das leere Wort der Länge null wird mit ε bezeichnet. $\mathcal{A}^m$ ist die Menge aller Worte über $\mathcal{A}$ der Länge m, $\mathcal{A}^*$ die Menge aller Worte über $\mathcal{A}$, $\mathcal{A}^+ = \mathcal{A}^* \setminus \{\varepsilon\}$. In dieser Arbeit sind a, b, c, x, y Bezeichner für Zeichen aus $\mathcal{A}$; s, t, u, v, w sind Bezeichner für Worte über $\mathcal{A}$. Die Zeichen selbst werden in Schreibmaschinenschrift angegeben: `a`, `b`, `c`, ...

Die m-fache Wiederholung eines Textes t wird mit t^m, der reverse Text $a_n \ldots a_1$ von $t = a_1 \ldots a_n$ mit t^{-1} bezeichnet. Für eine Aufspaltung von $t = vwu$ in (ggf. leere) v, w und u heißt v *Präfix* von t, w t-*Wort* und u *Suffix* von t. Falls $v \neq t$ ($u \neq t$), wird v (u) als *echtes Präfix* (*echtes Suffix*) von t bezeichnet, t-*words* ist die Menge aller t-Worte.

Falls die Bezeichner für Teilworte von t irrelevant sind, können diese auch durch einen Unterstrich gekennzeichnet werden:

$$\begin{aligned} t = v\_\_ &\iff v \text{ ist Präfix von } t, \\ t = \_w\_ &\iff w \text{ ist } t\text{-Wort}, \\ t = \_\_u &\iff u \text{ ist Suffix von } t. \end{aligned}$$

Ein Punkt bezeichnet in dieser Notation ein einzelnes Zeichen aus $\mathcal{A}$:

$$\begin{aligned} t = v.\_\_ &\iff v \text{ ist echtes Präfix von } t, \\ t = \_\_.u &\iff u \text{ ist echtes Suffix von } t. \end{aligned}$$

Zur Bezeichnung des Beginns/Endes eines Teilwortes $w = a_{l+1} \dots a_r$ in $t = a_1 \dots a_n$ wird die Notation $t = \_\__l w_r\_\_$ für $0 \le l \le r \le n$ verwendet[1].

Ein Präfix oder Suffix von t heißt *eingebettet*, wenn es noch an irgendeiner weiteren Stelle in t auftritt[2]:

$$\begin{aligned} t = \left\{ \begin{array}{l} v\_\_ \\ \_.v\_ \end{array} \right\} &\iff v \text{ ist eingebettetes Präfix von } t, \\ t = \left\{ \begin{array}{l} \_\_u \\ \_u.\_ \end{array} \right\} &\iff u \text{ ist eingebettetes Suffix von } t. \end{aligned}$$

Ein in t eingebettetes Präfix (Suffix) ist also immer echtes Präfix (Suffix) von t. Unter einem *nicht-eingebetteten Präfix* (*Suffix*) von t wird ein Präfix (Suffix) von t verstanden, das nicht in t eingebettet ist[3].

Ein t-Wort w heißt *rechtsverzweigend* bzw. *linksverzweigend*, wenn t von der Form

$$t = \left\{ \begin{array}{l} \_wx\_\_ \\ \_\_wy\_ \end{array} \right\},\ x \neq y \quad \text{bzw.} \quad t = \left\{ \begin{array}{l} \_xw\_\_ \\ \_\_yw\_ \end{array} \right\},\ x \neq y$$

ist. Per definitionem sei das leere Wort ε in jedem Text sowohl rechts- als auch linksverzweigend.

---

[1] Die Indizes l und r bezeichnen hier nicht Zeichen von t, sondern die Grenzen zwischen den Zeichen. Zur Motivation dieser Indizierungsweise s. [Mei86].

[2] Die Notation $t = \left\{ \begin{array}{l} w_1 \\ w_2 \end{array} \right\}$ kennzeichnet, daß t sowohl von der Form $t = w_1$ als auch von der Form $t = w_2$ ist.

[3] Nicht aber ein beliebiges Wort $w \in \mathcal{A}^*$, das nicht eingebettetes Präfix (Suffix) von t ist!

2.2 $\mathcal{A}^+$-Bäume und verwandte Datenstrukturen

2.2.1 $\mathcal{A}^+$-Bäume

In Anlehnung an [GK94a] wird der überwiegende Teil der später für Suffixbäume verwendeten Terminologie für eine allgemeinere Datenstruktur, die $\mathcal{A}^+$-Bäume, definiert.

Definition 2.1 ($\mathcal{A}^+$-Baum)
Ein $\mathcal{A}^+$-Baum T ist ein Baum mit Wurzel und Kantenmarkierungen aus $\mathcal{A}^+$. Dabei darf von jedem Knoten k in T für jedes $a \in \mathcal{A}$ höchstens eine a-Kante $k \xrightarrow{a} \bullet$ ausgehen[4]. □

Terminologie:

- Die Menge aller Kanten eines $\mathcal{A}^+$-Baumes T wird mit $edges(T)$ bezeichnet, die Menge aller Knoten mit $nodes(T)$ und die Wurzel mit $root(T)$. Ein Knoten aus $nodes(T)$ heißt *Blatt* von T, wenn an ihm keine Kante beginnt. Alle anderen Knoten heißen *innere Knoten* von T.

- Die Anzahl Knoten $|nodes(T)|$ eines $\mathcal{A}^+$-Baumes T wird als *Größe* von T, kurz $|T|$, bezeichnet.

- Ein innerer Knoten heißt *verzweigend*, wenn an ihm mindestens zwei Kanten beginnen, sonst heißt er *nicht-verzweigend*.

- Für einen gegebenen $\mathcal{A}^+$-Baum T bezeichnet $s\text{-}path(k)$ die Hintereinanderhängung (Konkatenation) der Kantenmarkierungen auf dem (eindeutigen) Weg von der Wurzel zum Knoten k. Mit $p\text{-}path(k)$ wird die Konkatenation der Markierungen auf dem Weg vom Knoten k zur Wurzel von T bezeichnet[5].

  Offensichtlich gilt: $s\text{-}path(k) = (p\text{-}path(k))^{-1}$.

[4] Als Bezeichner für Knoten wird in dieser Arbeit in der Regel die Variable k verwendet, in Ausnahmefällen treten aber auch längere Namen auf. Falls die Bezeichnung eines Knotens irrelevant ist, kann dieser durch einen Punkt $\bullet$ gekennzeichnet werden. Hier ist das der Knoten, an dem die a-Kante endet.

[5] Die Begriffe $s\text{-}path(k)$ und $p\text{-}path(k)$ sind ein Vorgriff auf die in Abschnitt 2.2.2 einzuführenden Suffix- und reversen Präfixbäume. Üblicherweise finden Pfade der s-Art bei der Betrachtung von Suffixbäumen Verwendung, wogegen in reversen Präfixbäumen die p-Sichtweise überwiegt.

- Wegen der geforderten Eindeutigkeit der a-Kanten im $\mathcal{A}^+$-Baum T gibt es umgekehrt zu einem vorgegebenen Wort w auch nur höchstens einen Knoten k in T mit $s\text{-}path(k) = w$ bzw. $p\text{-}path(k) = w$. Aufgrund dieser eineindeutigen Beziehung kann jeder Knoten k mit dem zu ihm führenden Pfad identifiziert werden. Zwei äquivalente Bezeichnungen sind möglich:

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

- Es läßt sich leicht einsehen, daß für alle Knoten $\overrightarrow{v}$ auf dem Weg von der Wurzel zu einem Knoten $\overrightarrow{w}$ v Präfix von w ist. Darüber hinaus enthält T wegen der Eindeutigkeitsforderung der a-Kanten keine weiteren Knoten, die Präfixe von w repräsentieren.
  Analoges gilt für $\overleftarrow{w}$: Auf dem Weg von der Wurzel zum Knoten $\overleftarrow{w}$ liegen genau die Knoten $\overleftarrow{v}$ in T mit $w = \_v$.

- Ein Wort w heißt *als S-Wort* im $\mathcal{A}^+$-Baum T *repräsentiert*, falls es einen Knoten $\overrightarrow{w\_} \in nodes(T)$ gibt. Die Menge aller in T als S-Wort repräsentierten Worte wird mit $s\text{-}words(T)$ bezeichnet.
  Analog ist $p\text{-}words(T) = \{w \in \mathcal{A}^* | \overleftarrow{\_w} \in nodes(T)\}$.

  Offensichtlich gilt: $w \in s\text{-}words(T)$ gdw. $w^{-1} \in p\text{-}words(T)$.

- Für ein Wort $w \in s\text{-}words(T)$ heißt in Anlehnung an [Ukk93] $(\overrightarrow{v}, u)$ *Referenzpaar* von w bezüglich T, falls $\overrightarrow{v}$ Knoten in T und $w = vu$ ist[6]. Falls v das längste solche Präfix von w ist, heißt $loc_T(w) = (\overrightarrow{v}, u)$ *kanonisches Referenzpaar* oder *Lokation*[7] von w bezüglich T.

- Ein kanonisches Referenzpaar der Form $(\overrightarrow{v}, \varepsilon)$ heißt *expliziter Knoten*; ein kanonisches Referenzpaar der Form $(\overrightarrow{v}, au)$ heißt *impliziter Knoten*, weil der Knoten $\overrightarrow{vau}$ nicht explizit in T existiert, er aber implizit „innerhalb“ der Kante $\overrightarrow{v} \xrightarrow{au} \overrightarrow{vau\_}$ vorhanden ist.

  An einigen Stellen wird es von Nutzen sein, die Notation der kanonischen Referenzpaare impliziter Knoten zu Quadrupeln zu erweitern:

  $$loc_T(w) = (\overrightarrow{v}, au, s, \overrightarrow{vaus}),$$

  falls $w = vau$ und $(\overrightarrow{v}, au)$ impliziter Knoten „innerhalb“ der Kante $\overrightarrow{v} \xrightarrow{aus} \overrightarrow{vaus}$ von T ist.

[6] Analog ließen sich Referenzpaare auch über $w \in p\text{-}words(T)$ definieren. Das Ergebnis unterscheidet sich aber nicht signifikant von dem oben formulierten: $(\overleftarrow{u}, v)$, falls $\overleftarrow{u}$ Knoten in T und $w = vu$ ist.

[7] „Lokation“ ist die deutsche Übersetzung des englischen *location* aus [Kur95].

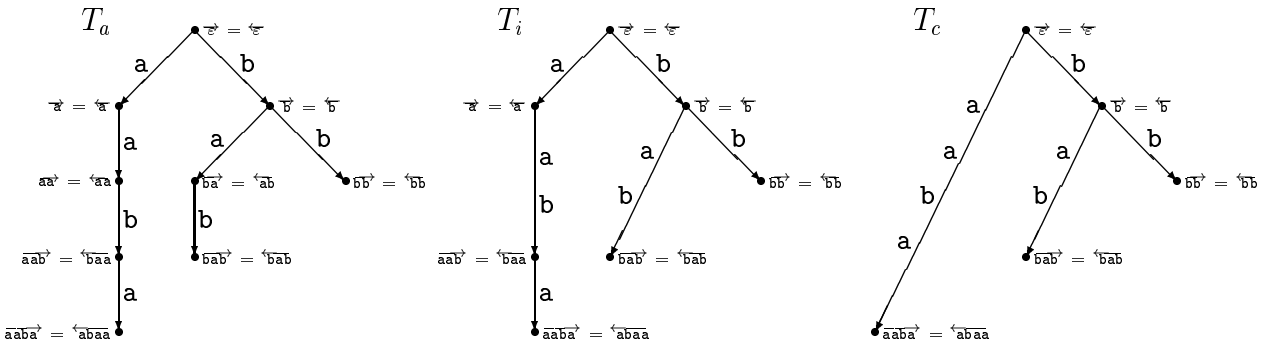


Abbildung 2.1: Drei $\mathcal{A}^+$-Bäume, die die gleichen Wortmengen repräsentieren: links der atomare Baum T_a, rechts der kompakte Baum T_c und in der Mitte ein Zwischenstadium T_i.

- Wie man leicht an den drei Bäumen T_a, T_i und T_c in Abb. 2.1 sieht, ist durch die Menge der in einem $\mathcal{A}^+$-Baum T repräsentierten Worte nicht der Aufbau von T eindeutig bestimmt. T_a, T_i und T_c repräsentieren nämlich die gleichen Wortmengen

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

  bzw.

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

- Der Baum T_a zeichnet sich dadurch aus, daß er ausschließlich Kanten mit Markierungen der Länge 1 enthält. Solche Kanten werden als *atomare Kanten* bezeichnet. Ein $\mathcal{A}^+$-Baum, der ausschließlich atomare Kanten enthält, heißt *atomarer $\mathcal{A}^+$-Baum*.

  Im Unterschied dazu wird ein $\mathcal{A}^+$-Baum, der abgesehen von der Wurzel keine nicht-verzweigenden inneren Knoten enthält, als *kompakter $\mathcal{A}^+$-Baum* bezeichnet. Der in Abb. 2.1 rechts abgebildete Baum T_c ist kompakt.

- Wie in [GK95] ausgeführt, kann man zeigen, daß unter allen $\mathcal{A}^+$-Bäumen, die eine vorgegebene Wortmenge repräsentieren, der atomare Baum die maximale und der kompakte Baum die minimale Anzahl expliziter Knoten enthält. Der atomare $\mathcal{A}^+$-Baum stellt die Normalform unter Kantenaufspaltung und der kompakte Baum die Normalform unter Kantenkontraktion dar.

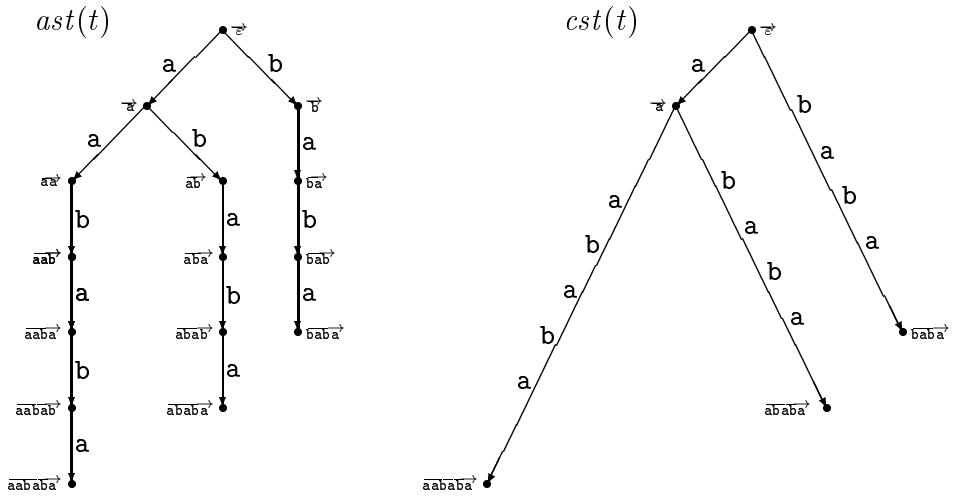


Abbildung 2.2: Atomarer und kompakter Suffixbaum von $t =$ `aababa`.

2.2.2 Suffixbäume

Suffixbäume werden nun als spezielle $\mathcal{A}^+$-Bäume definiert, so daß sich die oben eingeführte Terminologie automatisch auf sie überträgt.

Definition 2.2 (Suffixbaum)
Ein *Suffixbaum* eines Textes t ist ein $\mathcal{A}^+$-Baum, der als S-Worte genau alle t-Worte repräsentiert. □

Bemerkungen zu Definition 2.2

1. Wie alle $\mathcal{A}^+$-Bäume können auch Suffixbäume verschiedene Ausprägungen haben. In Abb. 2.2 sind für den Text $t =$ `aababa` der atomare Suffixbaum $ast(t)$ und der kompakte Suffixbaum $cst(t)$ angegeben.

2. Wie in [AHU74] dargestellt, ergibt sich für Texte t der Länge $n = |t|$ bei geeigneter Repräsentation der Kantenmarkierungen folgender Platzbedarf:

   - $\mathcal{O}(n^2)$ für den atomaren Suffixbaum $ast(t)$,
   - $\mathcal{O}(n)$ für den kompakten Suffixbaum $cst(t)$.

   Aus diesem Grunde ist für relevante Anwendungen, insbesondere bei langen Texten, die kompakte Variante vorzuziehen.

3. Eine sehr wichtige Eigenschaft des kompakten Suffixbaumes, auf die auch in [CL94] hingewiesen wird, lautet:

   I. Jeder innere Knoten $\overrightarrow{w}$ von $cst(t)$ korrespondiert mit einem rechtsverzweigenden[8] t-Wort w.

   II. Jedes Blatt $\overrightarrow{w}$ von $cst(t)$ korrespondiert mit einem nicht-eingebetteten[9] Suffix w von t.

Analog zum Suffixbaum läßt sich definieren:

Definition 2.3 (Reverser Präfixbaum)
Ein *reverser Präfixbaum* eines Textes t ist ein $\mathcal{A}^+$-Baum, der als P-Worte genau alle t-Worte repräsentiert. □

Diese Definition ist äquivalent zu der in [GK94a] gegebenen:

> Ein reverser Präfixbaum eines Textes t ist ein Suffixbaum des reversen Textes t^{-1},

da ein Wort $w \in \mathcal{A}^*$ genau dann t-Wort ist, wenn w^{-1} t^{-1}-Wort ist.

Deshalb muß für reverse Präfixbäume auch keine eigene Notation eingeführt werden: Der atomare reverse Präfixbaum eines Textes t ist gleich $ast(t^{-1})$, der kompakte reverse Präfixbaum von t ist gleich $cst(t^{-1})$.

2.2.3 Suffixlinks

Zur effizienten Konstruktion von Suffixbäumen und für viele Anwendungen ist eine zusätzliche Art von Kanten nützlich, die quasi orthogonal zur eigentlichen Baumstruktur verlaufen. Obwohl diese „Suffixlinks“ ursprünglich erstmals in [McC76] nur für Suffixbäume eingeführt wurden, werden sie hier in Anlehnung an [GK94a] allgemein für $\mathcal{A}^+$-Bäume definiert:

[8] Da ε in dieser Arbeit als in jedem Text rechtsverzweigend definiert wurde, ist hier (im Gegensatz zu [CL94], wo dafür eine spezielle Fallunterscheidung notwendig wird) $root(cst(t))$ als Repräsentant von ε in Fall I. mit eingeschlossen.

[9] In [CL94] werden ausschließlich Texte t betrachtet, die mit einem sonst nicht in t auftretenden Zeichen \$ enden, was bewirkt, daß kein Suffix von t eingebettet ist. Aus diesem Grunde kann dort von einer allgemeinen Korrespondenz zwischen Suffixen und Blättern des Suffixbaumes gesprochen werden.

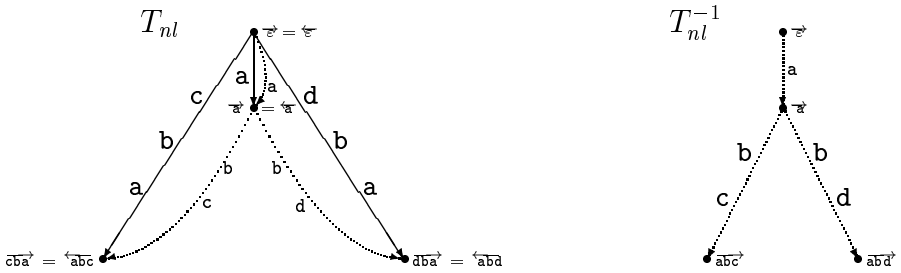


Abbildung 2.3: Ein $\mathcal{A}^+$-Baum T_{nl} mit Suffixlinks und sein Suffixlink-Baum T_{nl}^{-1}, der kein $\mathcal{A}^+$-Baum ist.

Definition 2.4 (Suffixlinks)
Sei $\overrightarrow{aw}$ ein Knoten im $\mathcal{A}^+$-Baum T. Der *Suffixlink* von $\overrightarrow{aw}$ ist die (zur Unterscheidung von den Baum-Kanten gepunktet gezeichnete) Kante $\overrightarrow{aw} \dashleftarrow \overrightarrow{v}$, wobei v das längste Suffix von w ist, das durch einen expliziten Knoten $\overrightarrow{v}$ in T repräsentiert wird[10].

Suffixlinks können markiert werden mit den vom Knoten $\overrightarrow{aw} = \overrightarrow{uv}$ zu Knoten $\overrightarrow{v}$ wegfallenden Zeichen: $\overrightarrow{uv} \overset{u}{\dashleftarrow} \overrightarrow{v}$. □

Bemerkungen[11] zu Definition 2.4

1. Suffixlinks spielen eine wichtige Rolle bei der effizienten Konstruktion von Suffixbäumen, deren effizienteste Vertreter in [McC76] und [Ukk93] beschrieben sind:

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

2. Die Suffixlinks eines $\mathcal{A}^+$-Baumes T bilden für sich genommen ebenfalls einen Baum, den *Suffixlink-Baum* von T, kurz T^{-1}. Dieser Baum ist im allgemeinen kein $\mathcal{A}^+$-Baum, wie man an dem Beispiel T_{nl}^{-1} in Abb. 2.3 sieht.

[10] Verglichen mit der Definition in [GK94a] haben hier Suffixlinks die umgekehrte Richtung, so daß sie, ohne daß ein Umdrehen nötig ist, einen Baum bilden. Dennoch „beginnt" der Suffixlink bei $\overrightarrow{aw}$ und „endet" bei $\overrightarrow{v}$.

[11] Entnommen aus [GK94a].

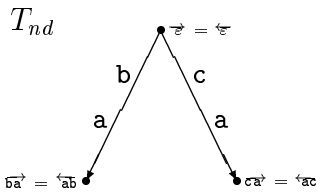


Abbildung 2.4: Ein $\mathcal{A}^+$-Baum T_{nd}, zu dem kein dualer $\mathcal{A}^+$-Baum existiert.

3. In einem atomaren Suffixbaum sind auch alle Suffixlinks atomar, in einem kompakten Suffixbaum zumindest die Suffixlinks innerer Knoten.

2.2.4 Dualität von $\mathcal{A}^+$-Bäumen

In diesem Abschnitt wird eine allgemeine Definition für die Dualität zweier $\mathcal{A}^+$-Bäume eingeführt:

Definition 2.5 (Dualität zweier $\mathcal{A}^+$-Bäume)
Zwei $\mathcal{A}^+$-Bäume T_1 und T_2 heißen *dual*, falls für jeden in T_1 enthaltenen Knoten $\overrightarrow{w}$ ein Knoten $\overleftarrow{w}$ in T_2 existiert und umgekehrt. □

Bemerkungen zu Definition 2.5

1. Nicht zu jedem $\mathcal{A}^+$-Baum existiert ein dualer Baum. Beispielsweise müßte der zu dem $\mathcal{A}^+$-Baum T_{nd} (Abb. 2.4) duale Baum die Knoten $\overrightarrow{\mathtt{ab}}$ und $\overrightarrow{\mathtt{ac}}$ enthalten. Ein $\mathcal{A}^+$-Baum mit diesen beiden Knoten enthält aber notwendigerweise auch den Knoten $\overrightarrow{\mathtt{a}}$, mit dem Resultat, daß er nicht dual zu T_{nd} sein kann, da $\overleftarrow{\mathtt{a}} \notin \mathit{nodes}(T_{nd})$.

2. Falls der zu einem $\mathcal{A}^+$-Baum T duale Baum existiert, ist dies der Suffixlink-Baum von T.

Der Beweis der zweiten Bemerkung wird erst in Abschnitt 2.3.2 (Lemma 2.12) durchgeführt, da mit der bis dahin eingeführten Notation eine einfachere Argumentation möglich ist.

2.2.5 Dualitätseigenschaften von Suffixbäumen

Die im folgenden beschriebenen Eigenschaften von Suffixbäumen sind aus [GK94a] entnommen, wo auch der Beweis von Satz 2.7 zu finden ist.

Satz 2.6 (Dualität atomarer Suffixbäume)
Der zu einem atomaren Suffixbaum $ast(t)$ duale $\mathcal{A}^+$-Baum existiert immer und ist gleich dem atomaren reversen Präfixbaum $ast(t^{-1})$. □

Nach den bis hierher erfolgten Überlegungen ist diese Behauptung offensichtlich, da genau alle t-Worte in $ast(t)$ durch explizite Knoten $\overrightarrow{w}$ und in $ast(t^{-1})$ durch explizite Knoten $\overleftarrow{w}$ repräsentiert sind.

Da aber im Falle des kompakten Suffixbaumes $cst(t)$ nicht immer zu jedem Knoten $\overrightarrow{w}$ ein korrespondierender Knoten $\overleftarrow{w}$ in $cst(t^{-1})$ existiert, gilt für diesen nur eine schwächere Variante des obigen Satzes:

Satz 2.7 (Schwache Dualität kompakter Suffixbäume)
Der Suffixlink-Baum $(cst(t))^{-1}$ eines kompakten Suffixbaumes $cst(t)$ ist ein $\mathcal{A}^+$-Baum, dessen Knotenmenge eine Teilmenge der Knotenmenge von $cst(t^{-1})$ ist. Darüber hinaus gilt: $((cst(t))^{-1})^{-1} = cst(t)$. □

Hier stellt sich nun die Frage, ob es eine Möglichkeit gibt, die Effizienz der Darstellung von kompakten Suffixbäumen mit den Symmetrieeigenschaften dualer Bäume zu verbinden. Wie man in Kapitel 3 sehen wird, stellen Affixbäume eine elegante Lösung dar.

2.3 Bi-Bäume

Trotz der Bezeichnung „Affixbaum“ handelt es sich bei diesem im strengeren Sinne gar nicht um einen Baum, sondern um den Spezialfall eines Bi-Baumes, einen aus zwei Bäumen zusammengesetzten Graphen. Dieser soll nun als letzte der grundlegenden Datenstrukturen eingeführt werden.

2.3.1 Definition der Datenstruktur

Definition 2.8 (Erweiterung von $\mathcal{A}^+$-Bäumen)
Seien T und U zwei $\mathcal{A}^+$-Bäume. Die *U-Erweiterung* von T, kurz T_U, ist der $\mathcal{A}^+$-Baum, der entsteht, wenn man alle diejenigen impliziten Knoten $(\overrightarrow{w}, au)$ von T, die als explizite Knoten $\overleftarrow{wau}$ in U enthalten sind, zu expliziten Knoten $\overrightarrow{wau}$ in T_U macht. □

In Abb. 2.5 sind zwei $\mathcal{A}^+$-Bäume S und P sowie deren gegenseitige Erweiterungen S_P und P_S abgebildet.

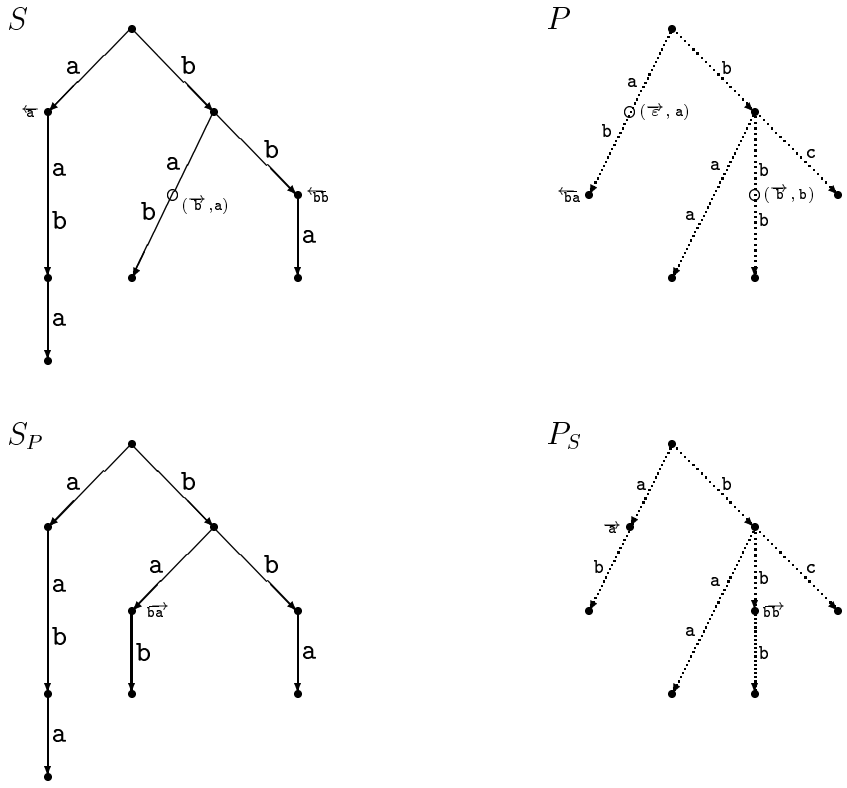


Abbildung 2.5: Zwei $\mathcal{A}^+$-Bäume S und P sowie deren gegenseitige Erweiterungen S_P und P_S.

Bemerkungen zu Definition 2.8

1. Die Menge der von einem $\mathcal{A}^+$-Baum repräsentierten Worte bleibt bei Erweiterungen unverändert.

2. Atomare $\mathcal{A}^+$-Bäume sind invariant unter Erweiterungen, da sie keine impliziten Knoten enthalten.

Definition 2.9 (Reverse Vereinigung zweier $\mathcal{A}^+$-Bäume, Bi-Baum)
Seien S und P zwei $\mathcal{A}^+$-Bäume. Die *reverse Vereinigung* von S und P, kurz ${}_S\widetilde{\bigcup}_P$, ist folgendermaßen definiert:

1. ${}_S\widetilde{\bigcup}_P$ enthält alle Knoten von S_P und P_S, wobei Knoten $\overrightarrow{w} \in nodes(S_P)$ mit Knoten $\overleftarrow{w} \in nodes(P_S)$ zu $\overrightarrow{w} = \overleftarrow{w^{-1}}$ identifiziert werden[12]:

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

   mit

$$\begin{aligned} A \,\widetilde{\cup}\, B \;=\; & \{\overrightarrow{w} = \overleftarrow{w^{-1}} \mid \overrightarrow{w} \in A \wedge \overleftarrow{w} \in B\} \\ & \cup \{\overrightarrow{w} \mid \overrightarrow{w} \in A \wedge \overleftarrow{w} \notin B\} \cup \{\overleftarrow{w^{-1}} \mid \overleftarrow{w} \in B \wedge \overrightarrow{w} \notin A\}. \end{aligned}$$

2. ${}_S\widetilde{\bigcup}_P$ erbt alle Kanten von S_P und P_S:

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

□

In Abb. 2.6 ist die reverse Vereinigung ${}_S\widetilde{\bigcup}_P$ der $\mathcal{A}^+$-Bäume S und P aus Abb. 2.5 dargestellt. S-Kanten werden dort mit durchgezogenen Pfeilen, P-Kanten mit gepunkteten Pfeilen gekennzeichnet.

Terminologie:

- Der so entstehende zweigefärbte Multigraph mit Wurzel und Kantenmarkierungen aus $\mathcal{A}^+$ wird als *Bi-Baum*[13] bezeichnet, der Knoten $\overrightarrow{\varepsilon} = \overleftarrow{\varepsilon}$ als *Wurzel* von ${}_S\widetilde{\bigcup}_P$ oder $root({}_S\widetilde{\bigcup}_P)$ und $|nodes({}_S\widetilde{\bigcup}_P)|$ als *Größe* von ${}_S\widetilde{\bigcup}_P$, kurz $|{}_S\widetilde{\bigcup}_P|$.

[12] An dieser Stelle schleicht sich eine kleine Unsymmetrie bzgl. S und P in die Notation ein: Knoten $\overrightarrow{w} \in nodes(S_P)$ werden als $\overrightarrow{w}$ in ${}_S\widetilde{\bigcup}_P$ übernommen, wogegen die Bezeichnung von $\overleftarrow{w} \in nodes(P_S)$ zu $\overleftarrow{w^{-1}}$ umgedreht wird. Hierbei handelt es sich aber ausschließlich um einen Effekt der Notation. Die Struktur von ${}_S\widetilde{\bigcup}_P$ ist völlig symmetrisch in S und P.

[13] Diese Bezeichnung, nicht zu verwechseln mit dem Binärbaum, stammt aus [Wei73], wo allerdings ausschließlich die duale atomare Variante betrachtet und als *bi-tree* bezeichnet wird.

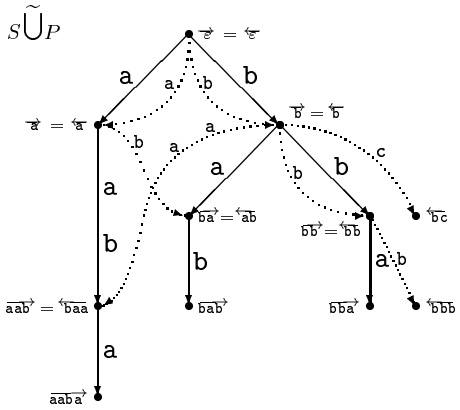


Abbildung 2.6: Die reverse Vereingung $_S\widetilde{\bigcup}_P$ der $\mathcal{A}^+$-Bäume S und P aus Abb. 2.5.

- Der von den S-Kanten gebildete Teilgraph von $_S\widetilde{\bigcup}_P$ ist identisch mit S_P und wird als *S-Baum* (in $_S\widetilde{\bigcup}_P$) bezeichnet. Er repräsentiert dieselben Worte wie S:

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

  Analog ist der von den P-Kanten gebildete Teilgraph gleich P_S und wird als *P-Baum* (in $_S\widetilde{\bigcup}_P$) bezeichnet. Er repräsentiert

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

  Die Eigenschaft $w \in s\text{-}words(T) \iff w^{-1} \in p\text{-}words(T)$ der $\mathcal{A}^+$-Bäume gilt also nicht mehr allgemein für Bi-Bäume (wohl aber für die im nächsten Abschnitt betrachteten dualen Bi-Bäume).

- Anhand der Struktur dieser beiden in $_S\widetilde{\bigcup}_P$ enthaltenen $\mathcal{A}^+$-Bäume werden die Begriffe *Blatt im S-/P-Baum* und *innerer verzweigender/nichtverzweigender Knoten im S-/P-Baum* für Bi-Bäume definiert.

- Jetzt gibt es zwei unterschiedliche Arten von Referenzpaaren in $T = {}_S\widetilde{\bigcup}_P$:

  - Für $w \in s\text{-}words(T)$ heißt $(\overrightarrow{v}, u)$ *S-Referenzpaar* von w, falls $\overrightarrow{v} \in nodes(T)$ und $w = vu$.
  - Für $w^{-1} \in p\text{-}words(T)$ heißt $(\overleftarrow{u}, v)$ *P-Referenzpaar* von w, falls $\overleftarrow{u} \in nodes(T)$ und $w = vu$.

Kanonisches S-/P-Referenzpaar bzw. S-/P-Lokation ($s\text{-}loc_T(w)$, $p\text{-}loc_T(w)$) ergeben sich daraus analog zu den entsprechenden Definitionen für $\mathcal{A}^+$-Bäume.

- Wie für $\mathcal{A}^+$-Bäume gilt auch für Bi-Bäume: Der Weg von der Wurzel zu einem Knoten $\overrightarrow{w}$ entlang den S-Kanten durchläuft genau alle in ${}_S\widetilde{\bigcup}_P$ enthaltenen Knoten $\overrightarrow{v}$, die Präfixe v von w repräsentieren. Ebenso durchläuft der Weg entlang den P-Kanten zum Knoten $\overleftarrow{w}$ genau die in ${}_S\widetilde{\bigcup}_P$ enthaltenen Knoten $\overleftarrow{u}$ mit $w = \_u$.
- Ähnlich wie für $\mathcal{A}^+$-Bäume sind atomarer und kompakter Bi-Baum definiert: Ein Bi-Baum, der ausschließlich atomare Kanten enthält, wird als *atomarer Bi-Baum* bezeichnet, ein Bi-Baum, der (abgesehen von der Wurzel) keine inneren Knoten enthält, die weder im S- noch im P-Baum verzweigen, als *kompakter Bi-Baum.*

2.3.2 Dualität von Bi-Bäumen

Definition 2.10 (Dualer Bi-Baum)
Ein Bi-Baum ${}_S\widetilde{\bigcup}_P$ heißt *dual*, falls jeder Knoten von ${}_S\widetilde{\bigcup}_P$ sowohl Knoten im S-Baum als auch Knoten im P-Baum in ${}_S\widetilde{\bigcup}_P$ ist. □

Den Zusammenhang zur Dualität zweier $\mathcal{A}^+$-Bäume beschreibt der folgende Satz:

Satz 2.11 (Äquivalenz der Dualitätsbegriffe)

$${}_S\widetilde{\bigcup}_P \text{ ist dual} \iff S_P \text{ und } P_S \text{ sind duale } \mathcal{A}^+\text{-Bäume.}$$

□

Beweis

$$\begin{aligned}
{}_S\widetilde{\bigcup}_P \text{ ist dual} &\iff \text{jeder Knoten } \overrightarrow{w} = \overleftarrow{w^{-1}} \in nodes({}_S\widetilde{\bigcup}_P) \text{ liegt sowohl im} \\
&\quad S\text{- als auch im } P\text{-Baum von } {}_S\widetilde{\bigcup}_P \\
&\iff \overrightarrow{w} \in nodes(S_P) \text{ gdw. } \overleftarrow{w} \in nodes(P_S) \\
&\iff S_P \text{ und } P_S \text{ sind dual.}
\end{aligned}$$

□

Jetzt haben wir das nötige Handwerkszeug beisammen, um die oben unbewiesene zweite Bemerkung zu Definition 2.5 zeigen zu können, nämlich daß

der zu T duale $\mathcal{A}^+$-Baum gleich dem Suffixlink-Baum von T ist; oder in der Terminologie der Bi-Bäume:

Lemma 2.12 (Der duale Baum ist der Suffixlink-Baum)
Die P-Kanten eines dualen Bi-Baumes ${}_S\widetilde{\bigcup}_P$ sind die Suffixlinks des S-Baumes in ${}_S\widetilde{\bigcup}_P$ und umgekehrt. □

Beweis
Gegeben ein beliebiger Knoten $\overrightarrow{\overline{aw}} \in \mathit{nodes}({}_S\widetilde{\bigcup}_P)$. Sei u das längste Suffix von w, so daß $\overrightarrow{\overline{u}}$ ebenfalls Knoten in ${}_S\widetilde{\bigcup}_P$ ist. Es ist zu zeigen, daß an $\overrightarrow{\overline{aw}}$ eine P-Kante endet, die bei $\overrightarrow{\overline{u}}$ beginnt: $\overrightarrow{\overline{u}} \overset{-a}{\cdots\!\!\triangleright} \overrightarrow{\overline{aw}} \in \mathit{p\text{-}edges}({}_S\widetilde{\bigcup}_P)$.

1. Aufgrund der Dualität von ${}_S\widetilde{\bigcup}_P$ ist $\overrightarrow{\overline{aw}}$ auch Knoten im P-Baum in ${}_S\widetilde{\bigcup}_P$. Es muß also einen Weg von der Wurzel entlang den P-Kanten zu $\overrightarrow{\overline{aw}}$ geben.

2. Offensichtlich muß auf diesem Weg der Knoten $\overrightarrow{\overline{u}}$ liegen, da u Suffix von aw ist.

3. Bleibt also zu zeigen, daß zwischen $\overrightarrow{\overline{u}}$ und $\overrightarrow{\overline{aw}}$ kein weiterer Knoten liegen kann. Wäre dies aber der Fall, so hätte er die Bezeichnung $\overrightarrow{\overline{zu}}$ mit $w = \_zu, z \in \mathcal{A}^+$, was im Widerspruch zu der Annahme steht, daß u das längste Suffix von w ist, das durch einen expliziten Knoten in ${}_S\widetilde{\bigcup}_P$ repräsentiert wird.

Da wegen der Dualität von ${}_S\widetilde{\bigcup}_P$ kein Knoten nur im P-Baum in ${}_S\widetilde{\bigcup}_P$ vorhanden ist, ist damit auch gezeigt, daß es keine weiteren P-Kanten geben kann, die nicht Suffixlinks irgendeines Knotens $\overrightarrow{\overline{aw}}$ sind.

Wegen der Symmetrie des Bi-Baumes gilt dieser Beweis analog auch, wenn man S- und P-Baum vertauscht. □

Ein Bi-Baum kann also auch als $\mathcal{A}^+$-Baum mit markierten Suffixlinks betrachtet werden. In dieser „Suffixbaum-Sichtweise“ wird eine P-Kante $\bullet \cdots\!\!\triangleright k$ synonym auch als Suffixlink von k bezeichnet.

Kapitel 3

Affixbäume

3.1 Definition der Datenstruktur

Die in Kapitel 2 vorgestellten Begriffe und Konzepte bilden die Basis für die nun einzuführende Datenstruktur, der das eigentliche Interesse dieser Arbeit gilt:

Definition 3.1 (Affixbaum)
Die reverse Vereinigung eines Suffixbaumes und eines reversen Präfixbaumes desselben Textes t heißt *Affixbaum*[1] von t. □

Der S-Baum in einem Affixbaum ist also ein Suffixbaum, der P-Baum ein reverser Präfixbaum von t. Die S-Kanten von Affixbäumen werden dementsprechend auch als *Suffix(baum)kanten* und die P-Kanten als *Präfix(baum)kanten* bezeichnet.

Bemerkungen zu Definition 3.1

1. Für Affixbäume ergeben sich als Normalformen unter Kantenaufspaltung/Kantenkontraktion:

   - der atomare Affixbaum von t, kurz $aat(t)$, der durch die reverse Vereinigung von $ast(t)$ und $ast(t^{-1})$ entsteht. Er enthält die gleichen Knoten wie der atomare Suffixbaum von t.

[1] In der Linguistik ist „Affix“ eine Sammelbezeichnung für Wortbildungselemente wie Suffixe und Präfixe.

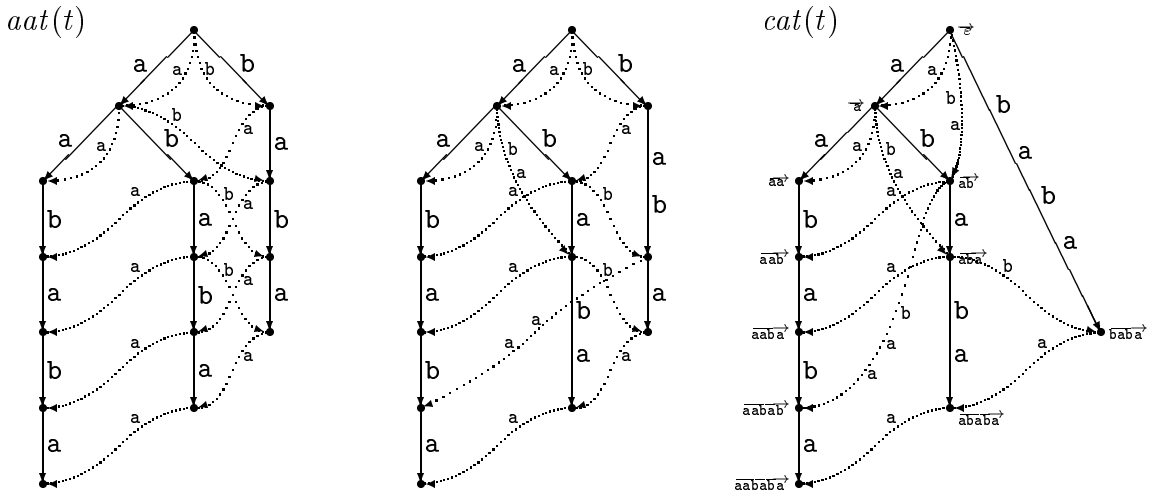


Abbildung 3.1: Drei Affixbäume für den Text $t =$ `aababa`.

- der kompakte Affixbaum $cat(t)$, der die reverse Vereinigung von $cst(t)$ und $cst(t^{-1})$ darstellt. Er enthält bis auf die Wurzel keine inneren Knoten, die weder im Suffix- noch im Präfixbaum verzweigen.

In Abb. 3.1 sind der atomare (links) und der kompakte Affixbaum (rechts), sowie ein Zwischenstadium (Mitte) für den Text $t =$ `aababa` abgebildet.

2. Ähnlich wie für den kompakten Suffixbaum läßt sich auch für den kompakten Affixbaum eine sehr wichtige Korrespondenzeigenschaft der in ihm enthaltenen Knoten angeben:

   Der kompakte Affixbaum $cat(t)$ enthält

   I. einen inneren verzweigenden Knoten im Suffixbaum für jedes rechtsverzweigende t-Wort,

   II. ein Blatt im Suffixbaum für jedes nicht-eingebettete Suffix von t,

   III. einen inneren verzweigenden Knoten im Präfixbaum für jedes linksverzweigende t-Wort und

   IV. ein Blatt im Präfixbaum für jedes nicht-eingebettete Präfix von t.

   Es können aber auch *doppelt verzweigende Knoten* (Knoten der Form I. und III.) sowie *Doppelblätter* (II. und IV.) auftreten, wie man am

kompakten Affixbaum in Abb. 3.1 sieht:

Knoten in $cat(\texttt{aababa})$	$\overrightarrow{\varepsilon}$	$\overrightarrow{\texttt{a}}$	$\overrightarrow{\texttt{aa}}$	$\overrightarrow{\texttt{aab}}$	$\overrightarrow{\texttt{aaba}}$	$\overrightarrow{\texttt{aabab}}$
Art des Knotens	I./III.	I./III.	IV.	IV.	IV.	IV.

$\overrightarrow{\texttt{aababa}}$	$\overrightarrow{\texttt{ab}}$	$\overrightarrow{\texttt{aba}}$	$\overrightarrow{\texttt{ababa}}$	$\overrightarrow{\texttt{baba}}$
II./IV.	III.	III.	II.	II.

Dagegen ist es nicht möglich, daß ein innerer verzweigender Knoten und ein Blatt aufeinanderfallen, da das einem rechts- oder linksverzweigenden t-Wort entspräche, das gleichzeitig nicht-eingebettetes Suffix/Präfix ist, was sich offensichtlich gegenseitig ausschließt.

Um also disjunkte Klassen zu erhalten, müssen noch zusätzlich folgende Fälle betrachtet werden:

V. doppelt verzweigende Knoten für sowohl rechts- als auch linksverzweigende t-Worte;

VI. Doppelblätter für t-Worte, die sowohl nicht-eingebettete Suffixe als auch nicht-eingebettete Präfixe von t sind.

Offenbar kommt für VI. nur t selber in Frage.

3. Ein kompakter Affixbaum kann nicht-atomare Kanten enthalten, die zu inneren Knoten führen, ein Beispiel ist die Kante $\overrightarrow{\texttt{a}} \overset{\texttt{ba}}{\cdots\!\succ} \overrightarrow{\texttt{aba}}$ in $cat(\texttt{aababa})$, Abb. 3.1 rechts. Bezogen auf die Suffixbaum-Sichtweise bedeutet dies, daß es im Suffixbaum in $cat(t)$ innere Knoten mit nicht-atomaren Suffixlinks geben kann. Es ist aber leicht ersichtlich, daß ein solcher Knoten $\overrightarrow{aw}$ im Suffixbaum in $cat(t)$ nicht verzweigt: Denn sonst wäre (gemäß I.) aw rechtsverzweigendes t-Wort, ebenso w, und da w längstes als Knoten in $cat(t)$ repräsentiertes Suffix von aw ist, wäre der Suffixlink von $\overrightarrow{aw}$ atomar: $\overrightarrow{aw} \overset{a}{\prec\!\cdots} \overrightarrow{w}$.

Eine schöne Eigenschaft von Affixbäumen, die deren symmetrische Struktur unterstreicht, wird durch den folgenden Satz ausgedrückt[2]:

Satz 3.2 (Dualität von Affixbäumen)
Affixbäume sind dual. □

[2]Von der Überlegung, aus diesem Grunde von Prä- und Suffixen gänzlich zu abstrahieren (beispielsweise durch eine Terminologie mit *Fix* und *Antifix*) wurde wieder Abstand genommen, da dies die Anschaulichkeit sehr stark gemindert hätte. Für Beispiele hätte ohnehin stets eine Instantiierung der Art *Fix = Suffix* und *Antifix = Präfix* (oder umgekehrt) wieder eingeführt werden müssen.

Beweis

Seien S ein Suffixbaum und P ein reverser Präfixbaum ein- und desselben Textes t. Dann ist zu zeigen, daß S_P und P_S dual sind, d. h. $\overrightarrow{w} \in nodes(S_P) \iff \overleftarrow{w} \in nodes(P_S)$.

"$\Longrightarrow$" Sei $\overrightarrow{w} \in nodes(S_P)$.
$\Rightarrow$ $w \in s\text{-}words(S_P) = s\text{-}words(S) = t\text{-}words = p\text{-}words(P)$, w muß also als expliziter oder impliziter Knoten in P repräsentiert sein.
$\Rightarrow$ 2 Fälle: 1. $\overleftarrow{w}$ ist expliziter Knoten in P;
$\Rightarrow$ $\overleftarrow{w}$ ist auch in P_S explizit;
oder 2. w wird durch einen impliziten Knoten in P repräsentiert;
$\Rightarrow$ $\overrightarrow{w}$ ist expliziter Knoten in S, denn sonst wäre $\overrightarrow{w}$ nicht explizit in S_P;
$\Rightarrow$ $\overleftarrow{w}$ wird also bei der S-Erweiterung von P zu P_S explizit gemacht.

"$\Longleftarrow$" Die andere Richtung läßt sich analog beweisen. □

3.2 Platzbedarf von Affixbäumen

Exemplarisch soll in diesem Abschnitt an den Extremfällen *aat* und *cat* die Größe von Affixbäumen untersucht werden. Die zunächst betrachteten asymptotischen Abschätzungen folgen unmittelbar aus der Affixbaum-Definition, wogegen zu der darauf folgenden Ermittlung des exakten maximalen Platzbedarfs von *cat* aufwendigere Überlegungen notwendig sind. Abschließend sollen noch einige Bemerkungen über die erwartete Größe von Affixbäumen zufälliger Texte gemacht werden.

3.2.1 Asymptotischer Platzbedarf

Da durch die reverse Vereinigung zweier $\mathcal{A}^+$-Bäume sich deren Knotenzahl maximal addiert, ergibt sich asymptotisch für Texte t der Länge $n = |t|$ folgende Größe von Affixbäumen:

***aat*(*t*)** enthält als reverse Vereinigung von $ast(t)$ und $ast(t^{-1})$ maximal $\mathcal{O}(n^2)$ Knoten,

***cat*(*t*)** hat analog als reverse Vereinigung von $cst(t)$ und $cst(t^{-1})$ mit je $\mathcal{O}(n)$ Knoten im *worst case* ebenfalls $\mathcal{O}(n)$ Knoten.

Während im allgemeinen Fall Bi-Bäume auch Knoten enthalten können, die nur auf einem Weg erreichbar sind, sind im Affixbaum alle Knoten sowohl im Suffix- als auch im Präfixbaum enthalten. An jedem Knoten außer der Wurzel endet also genau eine Suffix- und genau eine Präfixkante. Somit hat jeder Affixbaum mit p Knoten genau $2p-2$ Kanten, womit der Platzbedarf für die Kanten von $cat(t)$ ebenfalls in die Klasse $\mathcal{O}(n)$ fällt. Diese Angaben gelten ebenfalls für die Markierungen, da jede Markierung, ähnlich wie bei Suffixbäumen, mit konstantem Platzbedarf gespeichert werden kann, wie in Abschnitt 3.3.2 gezeigt wird.

Die weitergehende Diskussion des exakten Platzbedarfs im nächsten Abschnitt kann sich damit auf die Knotenzahl von Affixbäumen beschränken.

3.2.2 Maximaler Platzbedarf von kompakten Affixbäumen

Obwohl für Komplexitätsüberlegungen häufig die asymptotischen Angaben ausreichen, ist es für die praktische Implementierung eines Algorithmus ebenso wichtig, den konstanten Faktor zu bestimmen, der den *worst case* nach oben beschränkt. Dies wird nun für den interessanteren Fall des kompakten Affixbaumes durchgeführt, dessen Größe, wie oben gezeigt, proportional zur Länge des Textes ist.

Zu diesem Zweck soll zunächst daran erinnert werden, daß die exakte Größe des kompakten Suffixbaumes eines Textes t der Länge $n > 1$ maximal $2n-1$ Knoten beträgt. Ein Beispiel hierfür ist der Text $t = \mathtt{a}^{n-1}\$$ mit n Bättern und $n-1$ verzweigenden inneren Knoten.

Aus Symmetriegründen ergibt sich dieselbe Zahl natürlich auch für den kompakten reversen Präfixbaum und man erhält für die reverse Vereinigung der beiden Bäume im Höchstfall den doppelten Wert: $4n-2$. Da bei der reversen Vereinigung von Suffix- und reversem Präfixbaum jedoch mindestens zwei Knoten aufeinanderfallen, nämlich die Wurzel und der Knoten $\overrightarrow{t} = \overleftarrow{t^{-1}}$, ergibt sich schließlich für $n > 1$ als eine obere Schranke für die Knotenzahl $4n-4$.

Ein Beispiel, für das dieser Wert exakt erreicht wird, kann in dieser Arbeit leider nicht gegeben werden. Im folgenden wird aber ein Text diskutiert, für den die relative Abweichung der Knotenzahl von $4n$ beliebig klein wird, womit gezeigt ist, daß der gesuchte konstante Faktor gleich 4 ist.

Man betrachte dafür ein Alphabet $\mathcal{A}_k$ der beliebigen, aber festen Größe k

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

und den Text

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

Dieser Text hat die Länge $n = |t_k| = 1 + k(k-2) + 1 = k^2 - 2k + 2$.

Zur Ermittlung der exakten Knotenzahl von $cat(t_k)$ sind in Tabelle 3.1 alle sechs möglichen Arten von t_k-Worten angegeben, die den Klassen I. – VI. von Knoten im kompakten Affixbaum $cat(t_k)$ entsprechen. Die mittlere Spalte zeigt diese Worte am Beispiel $k = 6$, $\mathcal{A}_6 = \{\mathtt{a}, \mathtt{b}, \mathtt{c}, \mathtt{d}, \mathtt{e}, \mathtt{f}\}$, $t_6 = \mathtt{a}(\mathtt{bcde})^6\mathtt{f}$, in der rechten Spalte ist die Anzahl für allgemeines k angegeben.

Damit enthält $cat(t_k)$ also insgesamt

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

Knoten.

Die relative Abweichung von $4n = 4k^2 - 8k + 8$ wird damit für große k beliebig klein[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 Erwarteter Platzbedarf

In dem im vorigen Abschnitt beschriebenen Beispieltext t_k steigt das Verhältnis $\frac{|cat(t_k)|}{n}$ mit der Alphabetgröße asymptotisch gegen 4 an.

Im Gegensatz dazu findet man aber bei zufällig erzeugten Texten[4], daß für größere Alphabete kleinere Bäume entstehen (s. Abb. 3.2). Daß die Textlänge n auf dieses Phänomen keinen Einfluß hat, überrascht im übrigen nicht.

Um die Alphabetgrößenabhängigkeit zu erklären, die genauso auch in Suffixbäumen auftritt (s. Abb. 3.3), werden, zunächst für den Suffixbaum, drei Annahmen gemacht:

1. Da die erwartete Länge des längsten *repeats* innerhalb eines Textes $t \in \mathcal{A}^n$ nach [AS92] von der Ordnung $\log n$ ist, können für lange Texte die Effekte eingebetteter Suffixe vernachlässigt werden. Es kann also wie für Texte, die mit einem sonst nicht in t auftretenden Zeichen \$ enden, mit n Blättern im Suffixbaum gerechnet werden.

[3] Die Notation $q \underset{k\to\infty}{\longrightarrow} 0$ ist eine abkürzende Schreibweise für $\lim_{k\to\infty}(q) = 0$.

[4] Zur hier verwendeten Bernoulli-Verteilung s. [AS92].

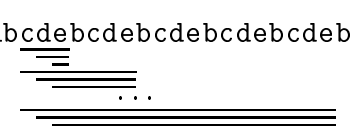
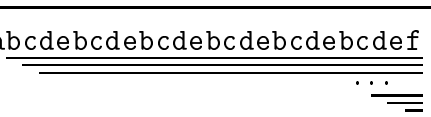
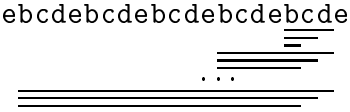
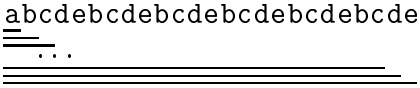
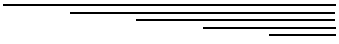
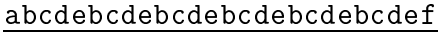
Knotenart	$k = 6, \quad n = \lvert t_k \rvert = 26$	allgemein
I. innerer Knoten im Suffixbaum (rechts-, aber nicht linksverzweigendes t_k-Wort)		$(k-3)(k-1)$
II. Blatt im Suffixbaum (nicht-eingebettetes Suffix, das aber nicht nicht-eingebettetes Präfix von t_k ist)		$n-1$
III. innerer Knoten im Präfixbaum (links-, aber nicht rechtsverzweigendes t_k-Wort)		$(k-3)(k-1)$
IV. Blatt im Präfixbaum (nicht-eingebettetes Präfix, das aber nicht nicht-eingebettetes Suffix von t_k ist)		$n-1$
V. doppelt verzweigender Knoten (sowohl rechts- als auch linksverzweigendes t_k-Wort)		$k-1$
VI. Doppelblatt (sowohl nicht-eingebettetes Suffix als auch nicht-eingebettetes Präfix)		1

Tabelle 3.1: Aufteilung der Knoten von $cat(t_k)$ am Beispiel $k = 6$ und allgemein.

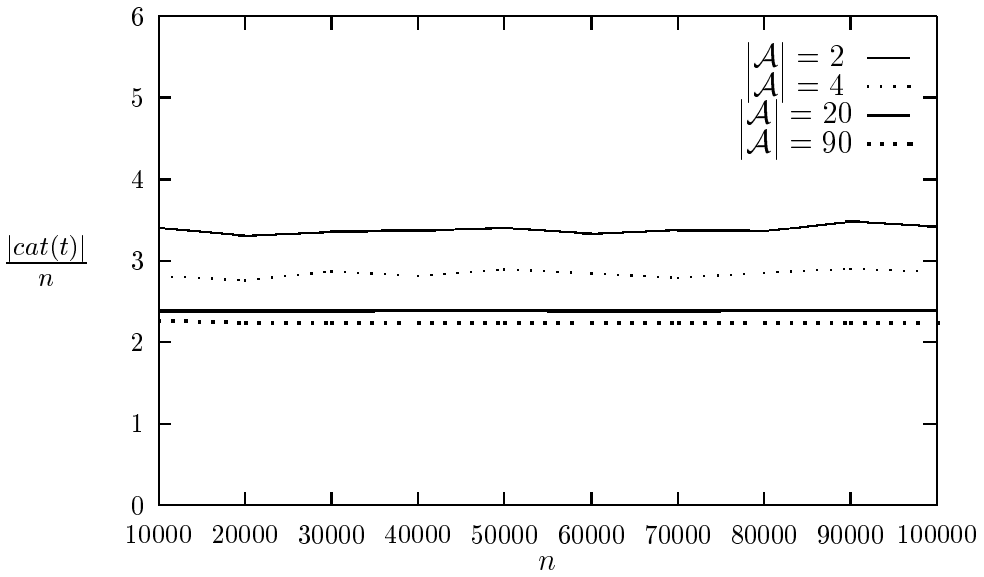


Abbildung 3.2: Die Größe von $cat(t)$ für zufällige Texte t, abhängig von der Textlänge n und der Alphabetgröße $|\mathcal{A}|$.

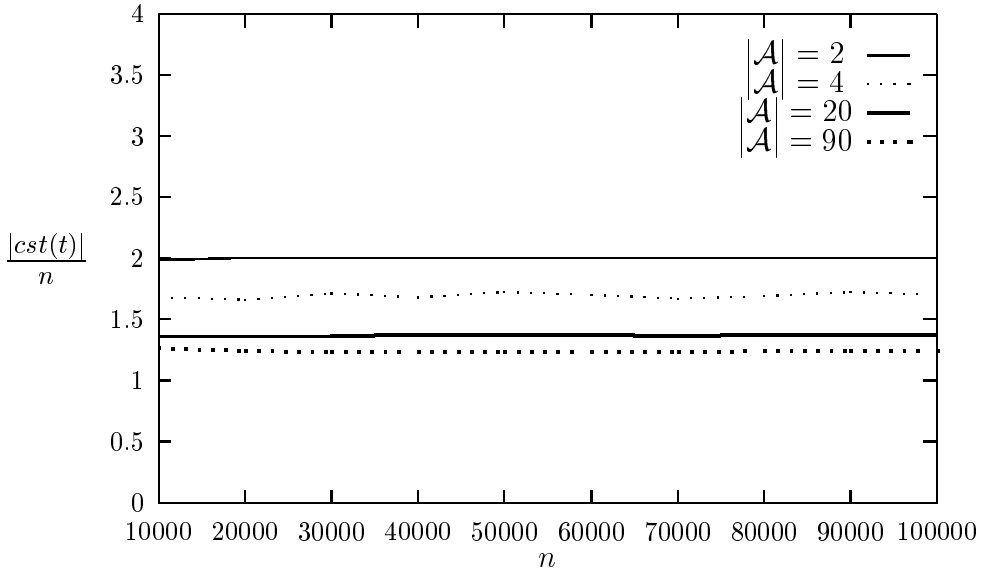


Abbildung 3.3: Die Größe von $cst(t)$ für zufällige Texte t, abhängig von der Textlänge n und der Alphabetgröße $|\mathcal{A}|$.

2. Bei Alphabeten der Größe k ist der Verzweigungsgrad innerer Knoten von k abhängig. Im folgenden wird ein konstanter Wert $q \geq 2$ angenommen. (Es wird sich zeigen, daß der exakte Wert für qualitative Ergebnisse irrelevant ist.)

3. Für zufällige Texte t ist der Suffixbaum ausbalanciert, so daß alle Blätter in einer Ebene liegen.

Mit diesen Annahmen enthält ein kompakter Suffixbaum der Tiefe d (d. h. mit $d+1$ Ebenen $i = 0 \ldots d$) in Ebene i genau q^i Knoten, insgesamt also

$$\sum_{i=0}^{d-1} q^i \;=\; \frac{q^d - 1}{q - 1} \text{ innere Knoten und } q^d \text{ Blätter.}$$

Das Verhältnis von inneren Knoten zu Blättern beträgt damit

$$\frac{q^d - 1}{q - 1} \cdot \frac{1}{q^d} \quad = \quad \frac{1 - \frac{1}{q^d}}{q - 1} \quad \underset{d \to \infty}{\longrightarrow} \quad \frac{1}{q - 1}.$$

Für kleine Alphabete ($q \approx 2$) erwartet man also ein Verhältnis von 1, wogegen sich das Verhältnis für große Alphabete ($q \to \infty$) asymptotisch von oben gegen 0 annähert. Dies entspricht einer Knotenzahl von $2n$ für kleine Alphabete und einer Annäherung an $1n$ für große Alphabete, was gut mit den empirischen Messungen in Abb. 3.3 übereinstimmt[5].

Dieses Verhalten ist qualitativ auf Affixbäume übertragbar, da für zufällige Texte $|cat(t)|$ proportional zu $|cst(t)|$ sein sollte. Als Proportionalitätsfaktor ist bei quantitativem Vergleich von Abb. 3.2 mit Abb. 3.3 ein Wert zwischen 1.6 und 1.8 anzunehmen.

3.3 Repräsentation von Affixbäumen

In diesem Abschnitt soll nun erörtert werden, wie Affixbäume mit möglichst geringem Speicherbedarf repräsentiert werden können. Zunächst werden zwei generell unterschiedliche Repräsentationsmöglichkeiten für $\mathcal{A}^+$- und Bi-Bäume diskutiert[6], im zweiten Teil wird näher auf spezielle Eigenschaften der Repräsentation eingegangen, die dem in Kapitel 4 vorgestellten *online*-Konstruktionsverfahren für Affixbäume zugrundeliegen.

[5] Auch Blumer, Ehrenfeucht und Haussler [BEH89], die quantitative Untersuchungen zu dieser Fragestellung durchführten, kommen zu ähnlichen Ergebnissen.

[6] Ein ähnlicher Vergleich für Suffixbäume wurde in [CS85] durchgeführt.

3.3.1 Kanten- vs. Knotenmarkierungen

Eine in vielen Implementierungen verwendete Art, $\mathcal{A}^+$-Bäume zu repräsentieren, läßt sich in Form einer rekursiven Typ-Notation folgendermaßen darstellen[7]:

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

Ein $\mathcal{A}^+$-Baum wird also durch einen Wurzelknoten mit einer Liste von markierten Kanten repräsentiert, die zu je einem genauso aufgebauten Unterbaum führen. Blätter werden in dieser Darstellung als Knoten mit leerer Kanten-/Unterbaumliste dargestellt. Über die Art der Kantenmarkierung, hier abstrakt mit `(Label a)` bezeichnet, wird später noch zu sprechen sein.

Für Bi-Bäume enthält der entsprechende Typ zwei Listen von Kanten, die zu Unterbäumen führen, eine für Suffix- und eine für Präfixkanten[8]:

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

Als alternative Darstellungsform läßt sich ein $\mathcal{A}^+$-Baum aber auch knotenmarkiert repräsentieren

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

d. h. ein Baum mit Wurzelknoten $\overrightarrow{w}$ wird durch das Wort w, wiederum abstrakt als `(Label a)`, und die Liste seiner Unterbäume repräsentiert. Bei Bi-Bäumen werden zwei Unterbaum-Listen benötigt:

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

Bei dem Vergleich dieser beiden Darstellungen stellt man fest, daß die zweite Variante sogar die mächtigere der beiden ist: Aus der Kenntnis der Markierungen zweier durch eine Kante verbundener Knoten läßt sich leicht die Markierung der sie verbindenden Kante berechnen

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

wogegen der umgekehrte Weg ohne Information über die Position der gegebenen Kante im Baum nicht möglich ist. Allerdings ist die Darstellung mit

[7] Es wird die Syntax der *lazy*-funktionalen Programmiersprache Haskell [FHPW92] verwendet.

[8] Durch die Symmetrie des Datentyps ist nicht festgelegt, welche der beiden Listen die Suffix- und welche die Präfixkanten repräsentiert. Als Konvention sei die erste Liste stets die der Nachfolger im Suffixbaum, die zweite die der Nachfolger im Präfixbaum.

Knotenmarkierungen wiederum auch eingeschränkter in dem Sinne, daß sie sich nicht für beliebige kantenmarkierte Bäume anwenden läßt.

Nun soll der Speicherbedarf der beiden Repräsentationsformen miteinander verglichen werden. Dabei wird wiederum davon ausgegangen, daß jedes `(Label a)` mit dem gleichen konstanten Platzbedarf gespeichert werden kann.

Für $\mathcal{A}^+$-Bäume ist der Unterschied zwischen beiden Darstellungen unerheblich, da sie nahezu genauso viele Kanten wie Knoten enthalten.

Bei Bi-Bäumen verhält es sich aber anders: Hier enden außer an der Wurzel an jedem Knoten entweder eine oder zwei Kanten. Speziell im Affixbaum tritt stets der Maximalfall ein, so daß dieser annähernd doppelt so viele Kanten wie Knoten enthält. Somit sind für Affixbäume Knotenmarkierungen eindeutig vorzuziehen. Das bedeutet zwar nicht, daß insgesamt nur die Hälfte des Speicherplatzes der Variante mit Kantenmarkierungen benötigt wird, da Kanten, wenn auch ohne Markierung, dennoch repräsentiert werden müssen. In der später beschriebenen Implementierung in C [KR78] wird sich aber dennoch eine beachtliche Einsparung von ca. 40% (128 statt 224 Byte pro Knoten) ergeben.

3.3.2 Art der Knotenmarkierung

Eine Voraussetzung für die Diskussion des Speicherbedarfs im vorigen Abschnitt war, daß Kanten- wie Knotenmarkierungen vom Typ `(Label a)` mit konstantem Aufwand gespeichert werden können. Dies ist für Suffix- wie für Affixbäume möglich, da für jeden Knoten $\overrightarrow{w} \in \mathit{nodes}(T)$ w ein t-Wort ist und somit die Markierung eines Knotens $\overrightarrow{w}$ durch ein Indexpaar `(Label a)` $= (l, r)$ repräsentiert werden kann, wenn $t = \_{}_l w_r \_{}$.

In [Ukk93] wird diese Notation noch erweitert, indem Kanten, die zu einem Blatt des Suffixbaumes führen, als sog. *offene Kanten* repräsentiert werden, d. h. die rechte Markierung, die in diesem Fall gleich $|t|$ ist, wird offen gelassen, was durch das Zeichen ∞ symbolisiert wird: $(l, |t|) = (l, \infty)$.

Diese Konvention läßt sich auch auf Knotenmarkierungen im Affixbaum T übertragen: Ein Blatt $\overrightarrow{w}$ im Suffixbaum in T wird als *offenes Suffixblatt* durch das Indexpaar $(l, +\infty)$ repräsentiert, falls $t = \_{}_l w_{|t|}$, ein Blatt $\overleftarrow{w}$ im Präfixbaum in T als *offenes Präfixblatt* durch das Paar $(-\infty, r)$, falls $t = {}_0 w^{-1}_r \_{}$.

Die bekannte Notation $\overrightarrow{w}$ bzw. $\overleftarrow{w}$ wird bei offenen Knoten um drei Punkte erweitert zu $\overrightarrow{w...}$ bzw. $\overleftarrow{...w}$. In Abb. 3.4 sind am Beispiel cat(`aababa`)

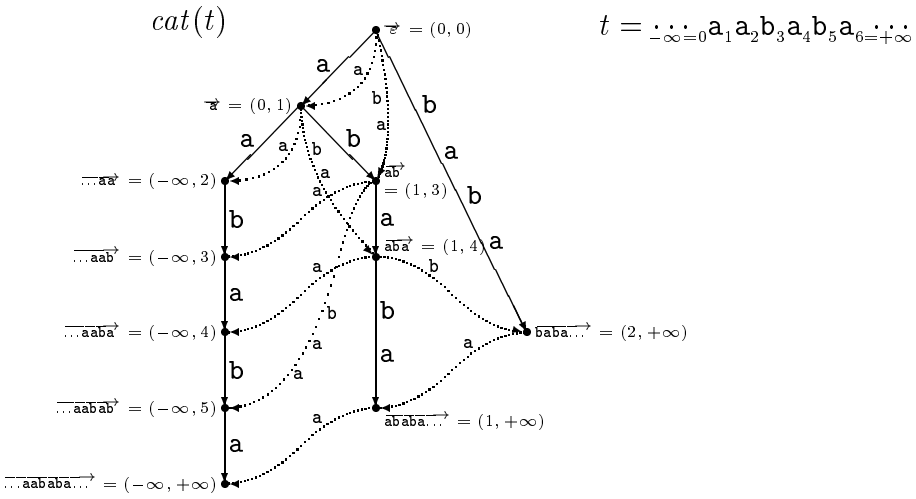


Abbildung 3.4: *cat*(aababa) mit Angabe der Knotenmarkierungen durch Indexpaare.

zusätzlich dazu die Knotenmarkierungen auch als Indexpaare angegeben.

3.4 Neuralgische Punkte im Affixbaum

3.4.1 Aktives Suffix und aktives Präfix

Bei der *online*-Konstruktion von Suffixbäumen [Ukk93] ist eine Lokation von entscheidender Bedeutung: der *aktive Punkt*. Dieser wandert während der Konstruktion im Baum auf und ab, seine Rolle wird in [GK94a] folgendermaßen beschrieben:

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

Es handelt sich hierbei um die Lokation des aktiven Suffixes von t in $cst(t)$:

Definition 3.3 (Aktives Suffix, aktives Präfix)
Das *aktive Suffix* eines Textes t ist das längste eingebettete Suffix von t und wird mit $\alpha(t)$ bezeichnet. Das *aktive Präfix* von t ist das längste eingebettete Präfix von t und wird mit $\alpha^{-1}(t)$ bezeichnet. □

Eine einfache, aber häufig benötigte Eigenschaft des aktiven Suffixes/Präfixes wird durch folgendes Lemma ausgedrückt:

Lemma 3.4 (Aktives Suffix/aktives Präfix)
Wenn das aktive Suffix $\alpha(t)$ nicht linksverzweigend ist, ist es Präfix von t.
Wenn das aktive Präfix $\alpha^{-1}(t)$ nicht rechtsverzweigend ist, ist es Suffix von t. □

Beweis
Sei $t = \_\_xu$, $u = \alpha(t)$ und u nicht linksverzweigend in t. Da u aktives Suffix von t ist, muß u ein weiteres Mal in $t = \left\{ \begin{array}{l} \_\_xu \\ \_u.\_\_ \end{array} \right\}$ auftreten. Das Zeichen links von u in der unteren Zeile kann aber weder ein x sein (weil dann xu aktives Suffix von t wäre), noch kann es ein anderes Zeichen $y \neq x$ sein (da sonst u linksverzweigendes t-Wort wäre). Also kann links von u nur ε stehen, d.h. u ist Präfix von $t = \left\{ \begin{array}{l} \_\_xu \\ u.\_\_ \end{array} \right\}$ und u tritt kein weiteres Mal in t auf.

Analog läßt sich die dazu symmetrische Aussage für das aktive Präfix zeigen. □

3.4.2 Aktives Suffix-/Präfixblatt

Neben dem aktiven Suffix/Präfix stellt sich für die Konstruktion von kompakten Affixbäumen noch ein zweiter Punkt als sehr wichtig heraus: das aktive Suffix- bzw. Präfixblatt[9].

Bei der Betrachtung des kompakten Affixbaumes $T = cat(t)$ erkennt man leicht, daß der Pfad vom Knoten $\overrightarrow{t}$ aus entlang den Präfixkanten zur Wurzel $root(T)$ mit einer Reihe atomarer Suffixlinks beginnt, die nacheinander alle Blätter des Suffixbaumes in T verbinden. Diese repräsentieren zunehmend kürzere nicht-eingebettete Suffixe von t.

Definition 3.5 (Aktives Suffix-/Präfixblatt)
Das Blatt im Suffixbaum in $cat(t)$, das das kürzeste nicht-eingebettete Suffix von t repräsentiert, heißt *aktives Suffixblatt* in $cat(t)$. Analog dazu wird das Blatt im Präfixbaum in $cat(t)$, das das kürzeste nicht-eingebettete Präfix von t repräsentiert, als *aktives Präfixblatt* in $cat(t)$ bezeichnet. □

[9] Im Grunde ist dieses auch schon für die Konstruktion von Suffixbäumen relevant, allerdings nur, wenn auch Suffixlinks der Blätter betrachtet werden, was in den üblichen Verfahren nicht der Fall ist.

In $cat(\texttt{aababa})$ (Abb. 3.4) ist $\overrightarrow{\texttt{baba...}}$ das aktive Suffixblatt und $\overrightarrow{\texttt{...aa}}$ das aktive Präfixblatt.

Bemerkungen zu Definition 3.5

1. Der Knoten, an dem der Suffixlink des aktiven Suffixblattes endet, repräsentiert das längste eingebettete Suffix s von t, für das ein expliziter Knoten $\overrightarrow{s}$ in $cat(t)$ existiert.

2. Wenn das aktive Suffix $\alpha(t)$ durch einen expliziten Knoten $\overrightarrow{\alpha(t)}$ in $cat(t)$ repräsentiert wird, ist $s = \alpha(t)$ und der Suffixlink des aktiven Suffixblattes atomar.

3. Das aktive Suffixblatt ist das einzige Blatt im Suffixbaum in $cat(t)$, das auch einen nicht-atomaren Suffixlink haben kann, nämlich genau dann, wenn das aktive Suffix $\alpha(t)$ nicht durch einen expliziten Knoten repräsentiert wird.

Symmetrisch dazu lassen sich diese Beziehungen natürlich auch für die Präfixsichtweise formulieren.

3.5 Konstruktion von kompakten Affixbäumen

In dieser Arbeit werden drei Vorschläge zur Konstruktion von kompakten Affixbäumen gemacht. Zwei dieser Ideen werden in den Abschnitten 3.5.1 und 3.5.2 kurz vorgestellt, der dritte Ansatz wird in Kapitel 4 ausführlich behandelt und zu einem konkreten Algorithmus weiterentwickelt.

3.5.1 Konstruktion von kompakten Affixbäumen durch Kantenkontraktion

Das hier beschriebene Verfahren wurde ursprünglich entwickelt, um nach kurzer Zeit eine Möglichkeit zur automatischen Erzeugung von Affixbäumen zu haben, und dadurch Fehler und Zeitaufwand bei der manuellen Konstruktion zu vermeiden. Aus diesem Grunde bot es sich an, eine funktionale Programmiersprache zu verwenden, die zwar verhältnismäßig langsame Programme

erzeugt, die zudem wesentlich mehr Speicher benötigen als vergleichbare Programme in einer imperativen Sprache, dafür aber erheblich kürzere Entwicklungszeiten ermöglicht[10].

Der Algorithmus ist sehr einfach: Wie in Abschnitt 3.1 bemerkt, handelt es sich bei dem kompakten Affixbaum eines Textes t um die Normalform aller Affixbäume von t unter Kantenkontraktion. Wenn also ein beliebiger Affixbaum von t, beispielweise $aat(t)$, gegeben ist, kann durch Eliminieren aller innerer Knoten, die weder im Suffix- noch im Präfixbaum verzweigen, $cat(t)$ erzeugt werden. Der atomare Affixbaum $aat(t)$ bietet sich als Ausgangspunkt an, da dieser dieselben Knoten wie $ast(t)$ hat, für den wiederum bekannte Konstruktionsverfahren existieren[11].

Allerdings benötigt dieses Verfahren im *worst case* quadratischen Zeit- und Platzbedarf, da die Zwischenstruktur des atomaren Affixbaumes vollständig erzeugt werden muß. Die Anforderungen an ein prototypisches Verfahren zur Überprüfung des eigentlichen Systems sind aber voll erfüllt.

3.5.2 Konstruktion von kompakten Affixbäumen durch reverse Vereinigung von Suffix- und reversem Präfixbaum

Bei der zweiten Möglichkeit zur Konstruktion kompakter Affixbäume handelt es sich um eine unmittelbare Umsetzung von Definition 3.1: Zunächst werden $S = cst(t)$ und $P = cst(t^{-1})$ konstruiert. In einem zweiten Schritt werden aus diesen durch gegenseitige Erweiterung die Bäume S_P und P_S erzeugt. Schließlich erhält man $cat(t)$, indem die Kanten eines der beiden Bäume an den entsprechenden Stellen im anderen Baum eingefügt werden.

Während der erste Schritt bekanntermaßen $\mathcal{O}(n)$ Zeit benötigt, scheint sich die Frage nicht so einfach beantworten zu lassen, ob und wie auch die beiden anderen Schritte in linearer Zeit durchführbar sind. Ob sich aus diesem Ansatz also ein Verfahren entwickeln läßt, mit dem $cat(t)$ in $\mathcal{O}(n)$ Zeit konstruierbar ist, kann an dieser Stelle nicht beantwortet werden.

[10] Tatsächlich wurde eine Haskell-Implementierung der hier beschriebenen Idee an einem Nachmittag erstellt, wogegen die Entwicklung des im nächsten Kapitel beschriebenen Verfahrens mit Implementierung in C etwa zwei Monate in Anspruch nahm.

[11] S. beispielsweise [GK94b].

3.6 Anwendungen von Affixbäumen

In [Apo85] werden zahlreiche Anwendungen für Suffixbäume[12] beschrieben. Da Affixbäume die Suffixbaum-Struktur vollständig einschließen, sind diese Verfahren auch mit Affixbäumen als zugrundeliegender Datenstruktur durchführbar.

Will man aber zeigen, daß bei der Übertragung dieser Anwendungen die Zeitkomplexität erhalten bleibt, so ist Vorsicht geboten. Denn wie in Abschnitt 3.1 gezeigt, ist der Suffixbaum im kompakten Affixbaum kein kompakter Suffixbaum, und nicht alle Suffixlinks innerer Knoten sind atomar. Dies kann dazu führen, daß man, selbst wenn man die gleichen Wege wie im kompakten Suffixbaum geht, mehr Schritte durchführen muß. In Kapitel 5 wird ein solcher Fall ausführlich diskutiert, hier aber kann die folgende Bemerkung gemacht werden:

Falls man sich lediglich von der Wurzel in einen Affixbaum hineinbewegt, führt dies zu keiner Erhöhung des Aufwandes gegenüber dem entsprechenden Weg im Suffixbaum, da die Länge des Weges von der Wurzel zum Knoten $\overrightarrow{w}$ sowohl im Suffix- wie im Affixbaum von der Ordnung $|w|$ ist.

Diese Aussage entschärft die oben dargelegte Problematik, da viele der in [Apo85] geschilderten Anwendungen lediglich den oben beschriebenen Weg zurücklegen. Darunter fallen[13]:

- Feststellen, ob w im Text t vorkommt in $\mathcal{O}(|w|)$
- Ermitteln des ersten/letzten Auftretens von w in t in $\mathcal{O}(|w|)$
- Ermitteln des Gewichts eines Wortes w in einem gewichteten Vokabular in $\mathcal{O}(|w|)$
- Ermitteln aller *repeats* eines Textes in $\mathcal{O}(\text{output})$.

[12] Apostolico betrachtet genaugenommen Subwort-Bäume eines Textes $t \in \mathcal{A}^*$, Suffixbäume des um ein nicht in $\mathcal{A}$ vorkommendes Zeichen \$ erweiterten Textes.

[13] Die Komplexitätsangaben beziehen sich ausschließlich auf die genannten Anwendungen, ohne den Aufwand für das Erzeugen des Affixbaumes zu berücksichtigen. Darüber hinaus werden (wie auch in [Apo85]) die an jeder Verzweigung im Baum anfallenden Kosten für das Finden der Kante zum gewünschten Nachfolgeknoten vernachlässigt, da die Kosten hierfür sehr stark von der Alphabetgröße, dem Verzweigungsgrad des Baumes (also der Beschaffenheit des Textes) und der jeweiligen Implementierung abhängen. Für ein bekanntes Alphabet ist aber zu erwarten, daß mit geeigneten Techniken (Kantenliste als Array, *Hashing*) nahezu ein konstanter Aufwand erreichbar ist.

Über diese unmittelbar von Suffixbäumen übernommenen Anwendungen hinaus kann die symmetrische Struktur der Affixbäume aber auch für flexiblere Verfahren genutzt werden. Hier sei nur auf das im Rahmen dieser Arbeit entworfene und in Kapitel 6 vorgestellte Programm *bigrep* verwiesen, das die Suche nach Teilworten in einem Text erlaubt, der sich *online* sowohl nach links als auch nach rechts erweitern läßt, was mit den bekannten, rein auf Suffixbäumen basierenden Verfahren nicht möglich ist.

Kapitel 4

Online-Konstruktion von kompakten Affixbäumen

Das in diesem Kapitel ausführlich vorgestellte Verfahren zur *online*-Konstruktion von kompakten Affixbäumen ist dem Algorithmus von Ukkonen [Ukk93] zur Konstruktion von kompakten Suffixbäumen (im folgenden mit *ukk* bezeichnet) nachempfunden[1]. Ebenso wie dort der Suffixbaum wird hier der Affixbaum eines Textes *online* konstruiert: Beginnend mit dem Affixbaum des leeren Textes $cat(\varepsilon)$ wird der Text zeichenweise gelesen, und mit jedem neuen Zeichen a wird der bestehende Affixbaum $cat(t)$ zum Affixbaum des verlängerten Textes $cat(ta)$ aktualisiert[2].

Während *ukk* allerdings darauf beschränkt ist, t nach rechts zu erweitern, wird es hier auch möglich sein, Zeichen vor den Text zu hängen und diesen damit nach links zu erweitern, aus $cat(t)$ also $cat(at)$ zu erzeugen. Damit kann die Konstruktion an irgendeiner Stelle im Text beginnen, und Erweiterungen sind in beliebiger Reihenfolge in beide Richtungen möglich.

Hauptbestandteil der folgenden Untersuchungen ist jedoch der Schritt von $cat(t)$ zu $cat(ta)$. Der umgekehrte Vorgang $cat(t) \rightarrow cat(at)$ läuft symmetrisch hierzu ab, so daß er nicht gesondert behandelt werden muß, wenn gewisse Nebenbedingungen beachtet werden.

Die Entwicklung des Algorithmus erfolgt in einer Art Top-Down-Verfahren: Zuerst werden abstrakt Unterschiede zwischen $cat(t)$ und $cat(ta)$ herausge-

[1] Zum Vergleich eignet sich wegen der von dort übernommenen Notation aber besser die Darstellung in [Kur95].

[2] In diesem und dem nächsten Kapitel bezeichnet t stets den bisher gelesenen Text, für den der Afffixbaum schon konstruiert ist, a ist das neu hinzukommende Zeichen. Die Länge von t sei n.

stellt. Abschnitt 4.2 erörtert dann die daraus resultierenden Konsequenzen für ein Verfahren, das $cat(t)$ in $cat(ta)$ überführt. Anschließend werden diese Ergebnisse in Abschnitt 4.3 zu einem Algorithmus zusammengefaßt. In Abschnitt 4.4 folgen Erläuterungen zu Aspekten, die zusätzlich bei einem bidirektionalen Einsatz des Verfahrens berücksichtigt werden müssen.

4.1 Unterschiede zwischen $cat(t)$ und $cat(ta)$

In diesem Abschnitt sollen zunächst statisch (d. h. ohne operationale Motive) die Unterschiede zwischen $cat(t)$ und $cat(ta)$ analysiert werden.

Es wird erörtert, wie sich aktives Suffix und Präfix verändern, welche Knoten hinzukommen und welche wegfallen. Die Veränderungen der Kanten und aktiven Blätter lassen sich daraus leicht ableiten, so daß nicht gesondert auf sie eingegangen werden muß.

Da einige der hier aufgeführten Aussagen genauso auch auf kompakte Suffixbäume zutreffen, können diese von *ukk* übernommen werden. Viele Eigenschaften sind aber neu, so daß auf deren Beweise nicht verzichtet werden kann.

4.1.1 Veränderungen von aktivem Suffix/Präfix

Betrachten wir zunächst das aktive Suffix $\alpha(ta)$, verglichen mit $\alpha(t)$. Wie in [GK94b] ausführlich erläutert, ist $\alpha(ta)$ ein Suffix von $\alpha(t)a$. Das aktive Suffix kann sich also um maximal ein Zeichen verlängern. Dagegen kann es aber um mehrere Zeichen verkürzt werden, maximal um seine gesamte Länge, falls das hinzukommende Zeichen a noch gar nicht in t auftritt.

Was für Veränderungen sind nun für das aktive Präfix möglich, wie kann also $\alpha^{-1}(ta)$ verglichen mit $v = \alpha^{-1}(t)$ aussehen?

1. Es ist offensichtlich nicht möglich, daß sich das aktive Präfix verkürzt, da v auch eingebettetes Präfix von ta ist.

2. Das aktive Präfix kann unverändert bleiben, wie das Beispiel $t =$ `aababa`, $a =$ `b` zeigt: `aababa` $\rightarrow$ `aababab` (das aktive Präfix ist jeweils unterstrichen).

3. Das aktive Präfix kann sich aber auch verlängern, wie man am Beispiel `aababa` $\rightarrow$ `aababaa` erkennt. Allerdings kann diese Verlängerung nur

maximal ein Zeichen betragen, wie im nachfolgenden Satz 4.1 gezeigt wird.

Satz 4.1 (Verlängern des aktiven Präfixes[3])
Sei $v = \alpha^{-1}(t)$. Dann verlängert sich das aktive Präfix beim Übergang $t \to ta$ um das Zeichen a genau dann, wenn auf v (als Präfix in t) ein a folgt, und wenn v gleichzeitig Suffix von t ist:

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

Sonst bleibt das aktive Präfix unverändert. □

Beweis

"$\Longrightarrow$" 2 Voraussetzungen: (i) v ist aktives Präfix von t.
(ii) va ist aktives Präfix von $ta = va\_\_a$.
Zu zeigen: $t = \left\{ \begin{array}{l} va\_\_ \\ \_\_.v \end{array} \right\}$.

1. $t = va\_\_$ folgt unmittelbar aus (ii).

2. (i) $\Rightarrow$ v ist längstes eingebettetes Präfix von t; va tritt also nicht eingebettet in t auf;
   (ii) $\Rightarrow$ va ist längstes eingebettetes Präfix von ta; va tritt also noch einmal in ta auf, nicht als Präfix. Dieses Auftreten kann nur als Suffix von ta sein, da dies die einzige Stelle ist, an der sich t und ta unterscheiden: $ta = \_\_.va$
   $\Rightarrow t = \_\_.v$

"$\Longleftarrow$" 2 Voraussetzungen: (i) v ist aktives Präfix von t.
(ii) $t = \left\{ \begin{array}{l} va\_\_ \\ \_\_.v \end{array} \right\}$.
Zu zeigen: va ist aktives Präfix von ta.

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

$\Rightarrow$ va tritt eingebettet in ta auf, ist also aktives Präfix, wenn es kein längeres Präfix gibt, das ebenfalls eingebettet in ta auftritt.
Wenn aber ein längeres Präfix $vaz, z \in \mathcal{A}^+$ eingebettet in ta wäre, dann wäre va eingebettet in t, was im Widerspruch zu (i) steht.

[3] Für Eingeweihte handelt es sich hierbei um das „Biertheorem", einen der ersten Schritte auf dem Weg zu dieser Arbeit.

In allen anderen Fällen, in denen t nicht von der Form $t = \left\{ \begin{array}{l} va\_\_ \\ \_\_.v \end{array} \right\}$ ist, bleibt das aktive Präfix unverändert, da

1. es sich nach obiger Bemerkung 1 nicht verkürzen kann,
2. eine Verlängerung um genau ein anderes Zeichen $b \neq a$ ausgeschlossen ist:

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

3. und eine Verlängerung um mehr als ein Zeichen ebenfalls nicht möglich ist: Angenomen $\alpha^{-1}(ta) = vzb, z \in \mathcal{A}^+, b \in \mathcal{A}$

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

was im Widerspruch zur Voraussetzung $\alpha^{-1}(t) = v$ steht.

□

Aus diesem Satz lassen sich weitere wichtige Eigenschaften der aktiven Punkteableiten. Der erste der drei folgenden Sätze beschreibt einen verblüffenden Zusammenhang des aktiven Präfixes mit dem aktiven Suffix:

Satz 4.2 (Zusammenhang von aktivem Präfix und aktivem Suffix)
Falls sich das aktive Präfix beim Übergang $t \to ta$ verlängert, ist das neue aktive Präfix gleich dem neuen aktiven Suffix:

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

□

Beweis
Sei $v = \alpha^{-1}(t)$. Dann ist

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

1. Das Suffix va von ta tritt eingebettet in ta auf, $\alpha(ta)$ hat also va als Suffix.

2. $\alpha(ta)$ kann nicht länger sein als va , da sonst $\alpha(ta) = zva, z \in \mathcal{A}^+$

$$\Rightarrow\ ta = \left\{ \begin{array}{l} va\_\_\_a \\ \_zva\_a \\ \_\_\_.zva \end{array} \right\}\ \Rightarrow\ t = \left\{ \begin{array}{l} va\_\_\_ \\ \_zva\_ \\ \_\_\_.zv \end{array} \right\},$$

was im Widerspruch zu der Annahme steht, daß v (und nicht va) aktives Präfix von t ist.

□

Wie das Beispiel $ta =$ `aaba` zeigt, gilt die Umkehrung von Satz 4.2 aber nicht, da hier zwar das neue aktive Suffix gleich dem neuen aktiven Präfix ist ($\alpha(ta) = \alpha^{-1}(ta) =$ `a`), das aktive Präfix sich beim Übergang $t \to ta$ aber nicht verlängert hat ($\alpha^{-1}(ta) = \alpha^{-1}(t) =$ `a`).

Nun folgen noch zwei Sätze, die den Zusammenhang der aktiven Punkte mit Veränderungen der im Affixbaum repräsentierten Worte beschreiben:

Satz 4.3 (Linksverzweigendes aktives Suffix)
Wenn das neue aktive Suffix $u = \alpha(ta)$ nicht durch einen expliziten Knoten in $cat(t)$ repräsentiert ist, ist es linksverzweigendes ta-Wort:

$$\overrightarrow{u} \notin nodes(cat(t)) \quad \Longrightarrow \quad ta = \left\{ \begin{array}{l} \_\_\_xu \\ \_yu\_\_ \end{array} \right\}, x \neq y.$$

□

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

Angenommen, u wäre dann nicht linksverzweigend in ta

$$\begin{array}{ll} \overset{\text{Lemma 3.4}}{\Longrightarrow} & ta = \left\{ \begin{array}{l} \_\_\_.u \\ u.\_\_\_ \end{array} \right\} \text{ und kein weiteres } u \text{ in } ta \\ \Longrightarrow & t = u\_\_\_ \text{ und kein weiteres } u \text{ in } t, \end{array}$$

u wäre also nicht-eingebettetes Präfix von t und somit in $cat(t)$ als Präfixblatt repräsentiert. Dies steht im Widerspruch zu der Voraussetzung $\overrightarrow{u} \notin nodes(cat(t))$, womit die Behauptung bewiesen ist. □

Satz 4.4 (Einbetten eines Präfixes)
Jedes nicht-eingebettete Präfix w von t ist auch nicht-eingebettetes Präfix

von ta, außer wenn w das verlängerte aktive Präfix $w = \alpha^{-1}(ta) = \alpha^{-1}(t)a$ ist. □

Beweis
Sei w nicht-eingebettetes Präfix von t: $t = w\_\_$ und w tritt kein weiteres Mal in t auf $\Rightarrow$ $ta = w\_\_a$. Dann gilt:

1. w ist Präfix auch von ta.
2. Die einzige Möglichkeit, daß w eingebettet in ta ist, besteht darin, daß w als Suffix von ta auftritt:

$$
\begin{aligned}
& ta = \left\{ \begin{array}{l} w\_\_a \\ \_\_.w \end{array} \right\} \quad \Rightarrow \quad w = va \text{ für irgendein } v \in \mathcal{A}^* \\
\Rightarrow \quad & ta = \left\{ \begin{array}{l} va\_\_a \\ \_\_.va \end{array} \right\} \\
\Rightarrow \quad & t = \left\{ \begin{array}{l} va\_\_ \\ \_\_.v \end{array} \right\} \quad \text{und } w = va \text{ tritt kein weiteres Mal in } t \text{ auf (s.o.)} \\
\Rightarrow \quad & \alpha^{-1}(t) = v,
\end{aligned}
$$

w ist also eingebettetes Präfix von ta, falls es von der Form $w = va$ ist, $v = \alpha^{-1}(t)$ und $t = \left\{ \begin{array}{l} va\_\_ \\ \_\_.v \end{array} \right\}$.

Nach Satz 4.1 ist dies genau dann der Fall, wenn sich das aktive Präfix verlängert:

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

□

4.1.2 Veränderungen der Knotenmenge

Zur Beschreibung der im Schritt $t \to ta$ hinzukommenden Knoten im kompakten Suffixbaum wird in [GK94b] folgender Terminus eingeführt:

Definition 4.5 (Relevantes Suffix)
Ein Suffix sa von ta heißt *relevant*, gdw. s eingebettetes Suffix von t und sa nicht-eingebettetes Suffix von ta ist. □

Damit kann angegeben werden, welche Veränderungen sich für die Wortklassen I. – IV. ergeben, die nach Abschnitt 3.1 von den Knoten im kompakten Affixbaum repräsentiert werden[4]:

I. Sei w rechtsverzweigendes t-Wort.

   a) Dann ist w ebenfalls rechtsverzweigendes ta-Wort.

   b) Zusätzlich ist s neues rechtsverzweigendes ta-Wort, falls sa relevantes Suffix von ta ist und s noch kein rechtsverzweigendes t-Wort war.

II. Sei w nicht-eingebettetes Suffix von t.

   a) Dann ist (statt w) wa nicht-eingebettetes Suffix von ta.

   b) Zusätzlich sind alle relevanten Suffixe sa von ta neue nicht-eingebettete Suffixe von ta.

III. Sei w linksverzweigendes t-Wort.

   a) Dann ist w ebenfalls linksverzweigendes ta-Wort.

   b) Zusätzlich ist das neue aktive Suffix $u = \alpha(ta)$ linksverzweigendes ta-Wort, falls $\overrightarrow{u}$ nicht expliziter Knoten in $cat(t)$ war (Satz 4.3).

IV. Sei w nicht-eingebettetes Präfix von t.

   a) Dann ist w nicht-eingebettetes Präfix auch von ta, außer wenn w das verlängerte aktive Präfix ist (Satz 4.4).

   b) Zusätzlich ist ta neues nicht-eingebettetes Präfix von ta.

Im Gegensatz zur *online*-Konstruktion von kompakten Suffixbäumen ist es (wg. IV.a) hier also auch möglich, daß $cat(t)$ Knoten enthält, die in $cat(ta)$ nicht mehr vorkommen. Ein Beispiel hierfür ist $t =$ `aababa`, $a =$ `a`: Der Knoten $\overrightarrow{\ldots\texttt{aa}}$ von $cat(t)$ tritt nicht mehr in $cat(ta)$ auf, da das Präfix `aa` in ta eingebettet ist, was in t noch nicht der Fall war.

[4]Die unter I. und II. beschriebenen Veränderungen im Suffixbaum werden in einer ähnlichen Terminologie ausführlich in [GK94b] begründet. Bei denjenigen Aussagen der Punkte III. und IV., die nicht offensichtlich sind, ist ein Verweis auf den entsprechenden Satz des vorigen Abschnittes angegeben.

4.2 Operationale Konsequenzen

Nun sollen diese Veränderungen in eine sinnvolle Reihenfolge gebracht werden, so daß sich ein Algorithmus ergibt, der aus der Knotenmenge von $cat(t)$ die Knoten von $cat(ta)$ erzeugt.

Dabei ist folgendes zu beachten:

- Das Verlängern der nicht-eingebetteten Suffixe (II.a) muß nicht explizit durchgeführt werden, wenn die Blätter „offen“ markiert sind.
- Mit diesem „automatischen Verlängern“ wird auf jeden Fall auch der Knoten $\overrightarrow{ta}$ erzeugt, womit sich (IV.b) ebenfalls erübrigt.
- Allerdings geht dabei das (zumindest vorher) nicht-eingebettete Präfix t als expliziter Knoten verloren, so daß $\overrightarrow{t}$ wieder eingefügt werden muß, falls t nicht-eingebettetes Präfix auch von ta ist. Dies ist immer genau dann der Fall, wenn nicht $ta = a^{n+1}$.

  Da zur Umsetzung von (IV.a) aber ohnehin immer dann der Knoten $\overrightarrow{va}$ gelöscht werden muß, wenn sich das aktive Präfix $v = \alpha^{-1}(t)$ zu $va = \alpha^{-1}(ta)$ verlängert (also auch im Fall $ta = a^{n+1}$), kann dieser Schritt auch aufgespalten werden: Zunächst wird stets der Knoten $\overrightarrow{t}$ eingefügt, und später wird der Knoten $\overrightarrow{va}$, in diesem Fall also $\overrightarrow{t}$, wieder gelöscht.

Zusammenfassend läßt sich damit folgender Algorithmus formulieren[5]:

Algorithmus 4.6 ($cat(t) \to cat(ta)$)

1. Verlängere den Text (und damit implizit jedes Blatt im Suffixbaum), und füge dann den Knoten $\overrightarrow{t}$ ein (II.a, IV.b).
2. Füge für jedes relevante Suffix sa von ta einen inneren Knoten $\overrightarrow{s}$ ein, sofern dieser nicht schon existiert (I.b), und ein Suffixbaum-Blatt $\overrightarrow{sa}$ (II.b).
3. Füge den Knoten $\overrightarrow{\alpha(ta)}$ ein, sofern er nicht schon existiert (III.b).
4. Falls sich das aktive Präfix verlängert, lösche den Knoten $\overrightarrow{\alpha(ta)} = \overrightarrow{\alpha^{-1}(ta)}$ (IV.a in Verbindung mit Satz 4.2).

[5] Die Punkte (I.a) und (III.a) brauchen nicht berücksichtigt zu werden, da sie keine Änderung der Knotenmenge beschreiben.

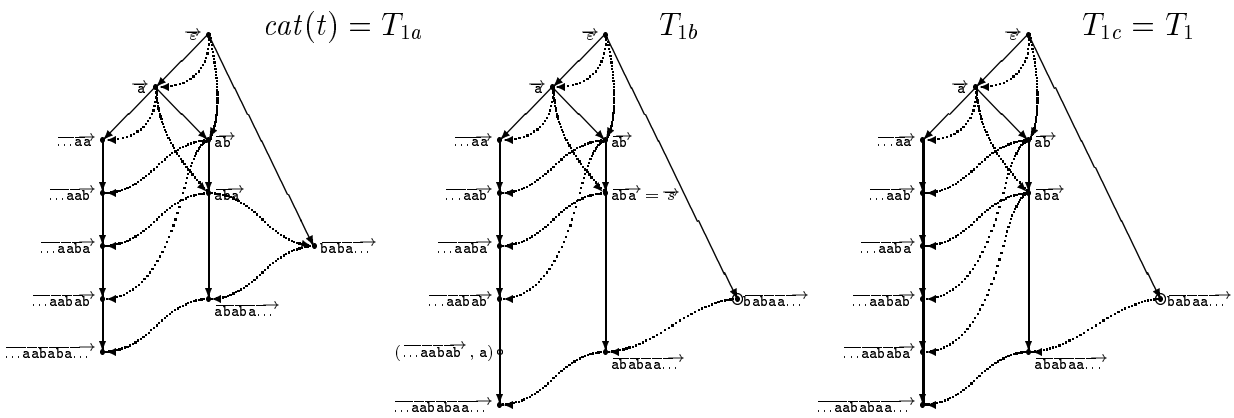


Abbildung 4.1: Verlängern des Textes ($T_{1a} \to T_{1b}$) und Einfügen des Knotens $\overrightarrow{t}$ ($T_{1b} \to T_{1c}$) am Beispiel $t = \texttt{aababa}, a = \texttt{a}$.

Im folgenden sollen nun die einzelnen Schritte ausführlich dargestellt werden. Hierbei werden nicht alle Einzelheiten formal ausdifferenziert, sondern es wird eher eine globale Darstellung verfolgt, die möglichst große Anschaulichkeit zum Ziel hat.

Es wird erörtert, wie die betreffenden Stellen im Baum gefunden werden und welche Veränderungen für die Kanten des Baumes daraus resultieren. Zu Knotenmarkierungen ist keine Erläuterung notwendig, da die Berechnung der Markierung (l, r) eines einzufügenden Knotens $\overrightarrow{w}$ stets problemlos möglich ist.

Zu den einzelnen Modifikationen des Baumes werden Funktionen definiert, mit Hilfe derer abschließend in Abschnitt 4.3 die *online*-Konstruktion noch einmal zusammenfassend formuliert wird.

4.2.1 Verlängern des Textes und Einfügen des Knotens $\overrightarrow{t}$

Der erste Schritt des Algorithmus 4.6 setzt sich aus zwei Teilen zusammen, die anschließend ausführlich betrachtet werden sollen: Dem Verlängern des Textes t zu ta und dem nachfolgenden Einfügen des Knotens $\overrightarrow{t}$. Veranschaulicht werden diese Vorgänge am Beispiel $t = \texttt{aababa}$, $a = \texttt{a}$ in Abb. 4.1.

Verlängern des Textes

Ähnlich wie im Verfahren *ukk* bewirkt auch hier das automatische Verlängern aller Blätter im Suffixbaum, daß die Struktur des Affixbaumes, wie dort die des Suffixbaumes, zerstört wird. Während bei *ukk* aber noch ein $\mathcal{A}^+$-Baum bestehen bleibt, handelt es sich bei der Datenstruktur, die aus $cat(t)$ durch Verlängern der Blätter im Suffixbaum entsteht, nicht einmal mehr um einen Bi-Baum. Deutlich wird dies an dem aktiven Suffixblatt, das im Suffixbaum verlängert wird, ohne daß sich seine Position im Präfixbaum ändert.

Ein Bi-Baum aber bleibt bestehen, wenn gleichzeitig mit der Verlängerung des Textes alle Suffixblätter aus dem Präfixbaum entfernt werden. Dies ist durch Löschen des Suffixlinks des aktiven Suffixblattes erreichbar. Wenn man nun die Präfixkanten ignoriert, die die einzelnen Suffixbaum-Blätter miteinander verbinden, erkennt man wieder einen Bi-Baum, der mit T_{1b} bezeichnet wird. Dieser ist in Abb. 4.1 in der Mitte dargestellt, wobei das auf diese Weise aus dem Präfixbaum entfernte aktive Suffixblatt $\overrightarrow{\texttt{babaa...}}$ mit einem zusätzlichen Kreis ⊙ gekennzeichnet ist.

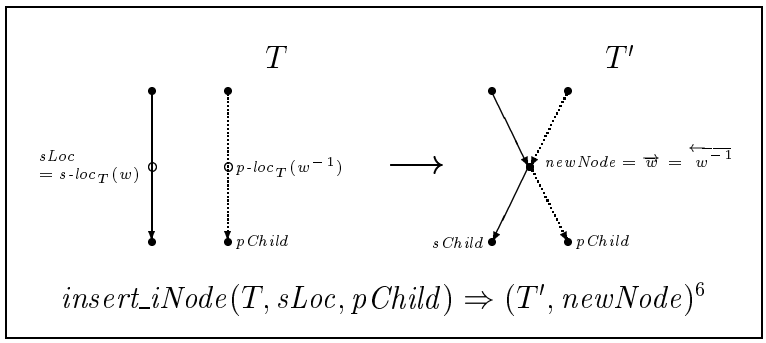
Erst in Schritt 3 des Algorithmus 4.6, wenn nach dem Einfügen des Knotens $\overrightarrow{\alpha(ta)}$ das neue aktive Suffix durch einen expliziten Knoten repräsentiert ist, werden die Suffixbaum-Blätter auch wieder in den Präfixbaum aufgenommen, indem eine atomare Präfixkante von $\overrightarrow{\alpha(ta)}$ zu dem dann aktiven Suffixblatt eingefügt wird.

Einfügen des Knotens $\overrightarrow{t}$

Generell muß man, um einen Knoten $\overrightarrow{w} = \overleftarrow{w^{-1}}$, in einen Bi-Baum T einzufügen, zunächst feststellen, ob w bereits S- und w^{-1} bereits P-Wort in T ist. Falls dies nämlich der Fall ist, existiert w schon als impliziter Knoten in T. Um diesen explizit zu machen, muß dann die Kante, „innerhalb“ derer w liegt, aufgespalten werden.

Für den Fall, daß w sowohl S- als auch w^{-1} P-Wort in T ist, wird die Funktion *insert_iNode* definiert. Diese spaltet die Kante, „innerhalb“ derer der implizite Knoten

sLoc liegt, und die am Knoten *pChild* endende Präfixkante auf und fügt dort den neuen Knoten $\overrightarrow{w} = \overleftarrow{w^{-1}}$ ein.
Darüber hinaus soll *insert_iNode* auch für den Fall definiert sein, daß *sLoc* ein expliziter Knoten $sLoc = (\overrightarrow{w}, \varepsilon)$ ist. Dann soll sich allerdings nichts ändern: $T' = T$ und $newNode = \overrightarrow{w}$; *pChild* wird ignoriert.



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

Falls w nicht S- oder w^{-1} nicht P-Wort von T ist, muß ein neues Blatt in dem entsprechenden Teilbaum erzeugt werden. Dazu ist der Knoten zu ermitteln, der das längste Präfix bzw. Suffix von w repräsentiert, da an diesem die Kante beginnt, die zu dem neuen Knoten führt.

Für den speziellen Fall des Knotens $\overrightarrow{t}$ im Schritt $cat(t) \rightarrow cat(ta)$ gilt:

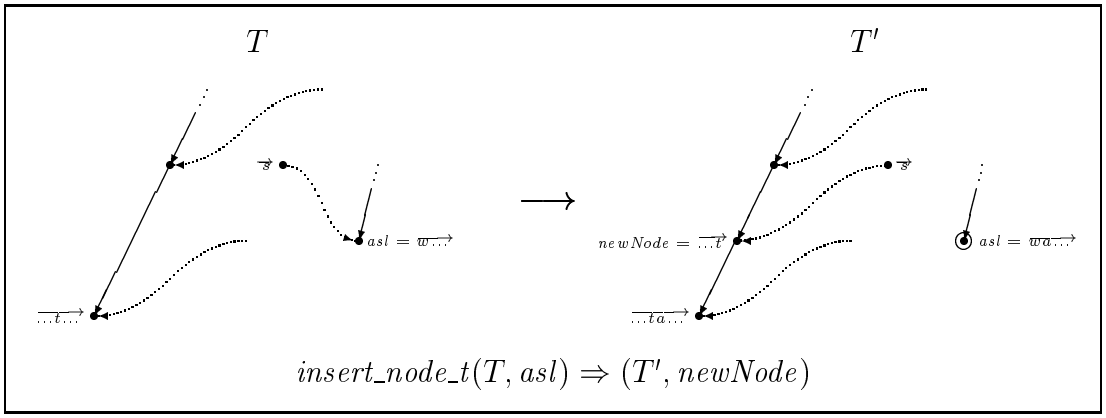
- Im Suffixbaum in T_{1b} ist der zuvor explizite Knoten $\overrightarrow{t}$ durch das automatische Verlängern zu $\overrightarrow{ta}$ implizit gemacht worden. Die aufzuspaltende Position von t liegt „innerhalb“ der am Knoten $\overrightarrow{ta}$ endenden Suffixkante.

- Als P-Wort ist t in T_{1b} sogar überhaupt nicht mehr enthalten, da die im Schritt $cat(t) \rightarrow T_{1b}$ gelöschte Kante Teil des Präfixpfades zum ehemaligen Knoten $\overrightarrow{t}$ war. Der neue Knoten $\overrightarrow{t}$ muß also als neues Präfixblatt eingefügt werden. Da, wie bereits in 3.4.2 bemerkt, der Knoten $\overrightarrow{s}$, an dem die (inzwischen gelöschte) Präfixkante zum aktiven Suffixblatt begann, das längste als Knoten repräsentierte eingebettete Suffix s von t repräsentiert und alle Knoten $\overrightarrow{u}$, die längere (nichteingebettete) Suffixe von t repräsentieren, zu $\overrightarrow{ua}$ verlängert wurden, beginnt die Präfixkante, die zu $\overrightarrow{t}$ führt, am Knoten $\overrightarrow{s}$, also genau an der Stelle, wo zuvor die Kante zum aktiven Suffixblatt gelöscht wurde.

Da es sich nicht um das Einfügen eines in beiden Teilbäumen bereits implizit vorhandenen Knotens handelt, kann nicht die Funktion *insert_iNode* verwendet werden. Stattdessen wird eine auf diesen Fall spezialisierte Funktion *insert_node_t* definiert, die den gesamten Vorgang in einem Schritt durchführt, sowohl das Verlängern des Textes mit Entfernen der Präfixkante zum aktiven

[6] Der Doppelpfeil „⇒“ in der Funktionsbeschreibung ist zu lesen als „liefert als Ergebnis“.

Suffixblatt als auch das Einfügen des Knotens $\overrightarrow{t}$ (korrekterweise $\overrightarrow{\ldots t}$, da es sich um ein Präfixblatt handelt):



Wegen der Repräsentation der Suffixblätter als offene Knoten muß das neue Zeichen a nicht als Parameter an *insert_node_t* übergeben werden.

4.2.2 Einfügen der relevanten Suffixe

Schritt 2 des Algorithmus 4.6 beschreibt eine Iteration über alle relevanten Suffixe von ta und läßt sich wie die Funktion *update* in *ukk*[7] realisieren: Beginnend mit dem längsten relevanten Suffix ua ($u = \alpha(t)$), wird in jedem Schritt ein innerer Knoten $\overrightarrow{u}$, sofern dieser nicht schon existiert, und ein Blatt $\overrightarrow{ua\ldots}$ in den Baum eingefügt. Zwischen den Schritten wird u jeweils um sein erstes Zeichen verkürzt, was durch Folgen eines Suffixlinks und anschließendes „Kanonisieren“ realisiert wird. Das Ende erkennt man daran, daß entweder u leer ist, oder für ein (evtl. schon verkürztes) u bereits eine Fortsetzung a existiert. In diesem Fall ist das neue aktive Suffix $\alpha(ta) = ua$.

Im Gegensatz zu *ukk* handelt es sich bei der hier zugrundeliegenden Datenstruktur aber um einen Affixbaum, was bei näherer Betrachtung zu zwei gravierenden Unterschieden führt:

1. Nicht nur die inneren Knoten, sondern auch die neu einzufügenden Suffixbaum-Blätter müssen an der entsprechenden Position in den Präfixbaum eingearbeitet werden (d. h. in der Suffixbaum-Sichtweise: Auch Blätter benötigen Suffixlinks).

[7] Korrekterweise in der Form, wie sie in [GK94b] oder als *ukkstep* in [Kur95] beschrieben ist, während die Funktion *update* in [Ukk93] eine etwas andere Funktionalität aufweist.

2. Da Präfixkanten zu inneren Knoten eines kompakten Affixbaumes nicht-atomar sein können, kann es passieren, daß durch Folgen eines Suffixlinks das aktive Suffix um mehr als ein Zeichen verkürzt wird.

Auf die Lösung dieser beiden Problembereiche soll nun ausführlich eingegangen werden.

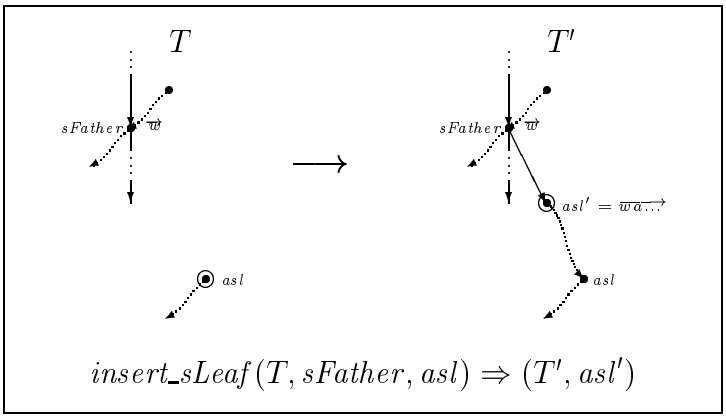
Einfügen neuer innerer Knoten und Blätter

Wie bei *ukk* stellt die S-Lokation des aktiven Suffixes während der gesamten Konstruktion eine Invariante dar, so daß die Position eines einzufügenden inneren Knotens $\overrightarrow{u}$ im Suffixbaum stets bekannt ist.

Die entsprechende P-Lokation läßt sich ebenfalls ähnlich wie bei *ukk* ermitteln: Ein neu einzufügender Knoten $\overrightarrow{u}$ spaltet den Suffixlink des unmittelbar zuvor eingefügten Knotens $\overrightarrow{.u}$ auf[8]. Beim allerersten relevanten Suffix, also beim Einfügen des Knotens $\overrightarrow{\alpha(t)}$, wo noch kein Knoten vorher eingefügt wurde, ist stattdessen der Suffixlink des Knotens $\overrightarrow{...t}$ zu wählen, da $\alpha(t)$ das neue längste Suffix von t ist, das durch einen expliziten Knoten in $cat(ta)$ repräsentiert wird.

Bei den Blättern verhält es sich ähnlich: Die an einem einzufügenden Blatt $\overrightarrow{ua...}$ (dem neuen aktiven Suffixblatt) beginnende Präfixkante führt immer zu dem im vorherigen Schritt eingefügten Blatt $\overrightarrow{.ua...}$ (dem bisherigen aktiven Suffixblatt). Alle Blätter des Suffixbaumes bleiben so in der Reihenfolge ihrer Länge durch atomare Präfixkanten verbunden. Zu dem zuletzt eingefügten Blatt führt zunächst keine Präfixkante. Diese wird, wie oben angedeutet, erst in Schritt 3 des Algorithmus eingearbeitet.

Während zum Einfügen der inneren Knoten die bereits oben beschriebene Funktion *insert_iNode* verwendet werden kann, werden die Suffixbaum-Blätter durch die Funktion *insert_sLeaf* eingebaut. Diese erhält als Argumente den Baum T, den Knoten *sFather*, an dem die einzufügende Suffixblatt-Kante beginnt, und den Knoten *asl*, oberhalb dessen der neue Knoten in



[8] Es ist auch möglich, daß ein Knoten $\overrightarrow{.u}$ schon in $cat(t)$ existierte. In diesem Fall wurde er nicht eingefügt, sondern „übersprungen“.

den Präfixbaum eingefügt werden soll. Das Ergebnis ist der veränderte Baum T' und das eingefügte Suffixblatt asl'.

Der gesamte Prozeß des Einfügens eines relevanten Suffixes ua in den Baum T wird dann durch die Funktion *insert_relevant_suffix* realisiert, wobei *uLoc* die Lokation von u in T, *pChild* den Knoten $\overrightarrow{...t}$ bzw. den zuletzt eingefügten oder „übersprungenen“ inneren Knoten und *asl* das aktive Suffixblatt repräsentiert:

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

Verkürzen des aktiven Suffixes

Um das aktive Suffix zu verkürzen, wird ähnlich wie in [Kur95] eine Funktion *linkloc* definiert:

$$\mathit{linkloc}_T(s\text{-}\mathit{loc}_T(.w)) \Rightarrow s\text{-}\mathit{loc}_T(w).$$

„Naiv“ läßt sich diese Funktion realisieren, indem von der Wurzel aus der Pfad entlang den Suffixkanten gemäß dem Wort w verfolgt wird. Aus analogen Überlegungen am Suffixbaum ist aber bekannt, daß diese Lösung zu einem Verfahren mit quadratischem Aufwand führt (man betrachte hierfür beispielsweise den Text $t = \mathtt{a}^{n-1}\$$), womit das erklärte Ziel der *online*-Konstruktion von $cat(t)$ in linearer Zeit von vornherein ausgeschlossen wäre. In *ukk* werden deshalb Suffixlinks zum Erreichen der Lokation des verkürzten aktiven Suffixes in $\mathcal{O}(1)$ verwendet.

In Affixbäumen ist dies wegen der nicht-atomaren Suffixlinks aber nicht immer auf direktem Wege möglich. Unter Umständen muß ein „Umweg“ in Kauf genommen werden, für dessen Realisierung zwei Möglichkeiten vorgeschlagen werden, veranschaulicht in Abb. 4.2 am Beispiel $T_{2a} \rightarrow T_{2b}$ ($t = \mathtt{aababa}$, $u = \mathtt{aba}$, $a = \mathtt{a}$; s. auch Abb. 4.3), wo der nicht-atomare Suffixlink $\overrightarrow{\mathtt{aba}} \overset{\mathtt{ba}}{\cdots\!\!\prec} \overrightarrow{\mathtt{a}}$ den (impliziten) Knoten $\overrightarrow{\mathtt{ba}}$ „verpaßt“:

a) Man geht vom aktiven Punkt aus zunächst so weit im Suffixbaum „nach oben“, bis man einen Knoten mit atomarem Suffixlink erreicht:

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

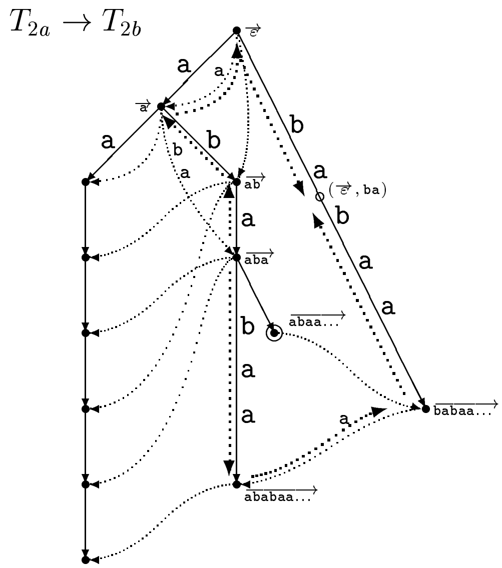


Abbildung 4.2: Die beiden Möglichkeiten, das aktive Suffix zu verkürzen, am Beispiel $T_{2a} \to T_{2b}$ ($t = \texttt{aababa}$, $u = \texttt{aba}$, $a = \texttt{a}$; s. Abb. 4.3).

Dann folgt man dem Suffixlink

$$(\overrightarrow{\texttt{a}}, \texttt{ba}) \overset{\texttt{a}}{\longleftarrow} (\overrightarrow{\varepsilon}, \texttt{ba}),$$

wodurch das Verkürzen des aktiven Suffixes realisiert wird. Anschließend kanonisiert man wie bei *ukk* (im Beispiel in Abb. 4.2 ist das nicht nötig, da $(\overrightarrow{\varepsilon}, \texttt{ba})$ schon kanonisches Referenzpaar ist).

Dieser Weg terminiert spätestens an der Wurzel, wo das Referenzpaar dann im allgemeinen Fall $(\overrightarrow{\varepsilon}, cu)$ lautet. Dort kann das aktive Suffix einfach durch Abstreichen des ersten Zeichens zu $(\overrightarrow{\varepsilon}, u)$ verkürzt werden.

b) Der zweite Vorschlag beschreibt ein ähnliches Vorgehen, aber „unten herum“: Man folgt den Suffixbaum-Kanten (nicht aber der zuletzt eingefügten Kante zum aktiven Suffixblatt) so weit „nach unten“, bis man einen Knoten mit atomarem Suffixlink erreicht[9]

$$(\overrightarrow{\texttt{aba}}, \varepsilon) \overset{\texttt{baa}}{\longrightarrow} (\overrightarrow{\texttt{ababaa...}}, -\texttt{baa}),$$

[9] Die Notation „$-u$“ bezeichnet einen negativen *offset*: $(\overrightarrow{v}, -u)$ ist S-Referenzpaar von w, falls $\overrightarrow{v}$ Knoten in T und $v = wu$ ist.

folgt diesem

$$(\overrightarrow{\texttt{ababaa...}}, -\texttt{baa}) \overset{\texttt{a}}{\dashleftarrow} (\overrightarrow{\texttt{babaa...}}, -\texttt{baa})$$

und „kanonisiert“ wieder entsprechend weit „aufwärts“:

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

Ein einfacher Umstand erklärt dabei, warum das Folgen des Weges „nach unten“ stets eindeutig ist: Keiner der Knoten mit nicht-atomarem Suffixlink verzweigt im Suffixbaum (s. Bemerkung 3 zu Definition 3.1).

Dieser Weg terminiert sicher, da spätestens ein atomarer Suffixlink existiert, wenn ein Blatt im Suffixbaum erreicht ist. Daß es sich dabei um das aktive Suffixblatt handelt (dessen Suffixlink ja gelöscht wurde), ist oben explizit ausgeschlossen worden.

Das Einfügen aller relevanten Suffixe von ta beschreibt die Funktion *ukkstep*[10]:

$$\begin{aligned}
&ukkstep(T, a, actS, pChild, asl) \\
&\quad = \begin{cases} (T, down_T(actS, a), asl), & \underline{\text{if}}\ occurs(actS, a) \\ (T', actS, asl'), & \underline{\text{else if}}\ actS = (root(T), \varepsilon) \\ ukkstep(T', a, linkloc_{T'}(actS), pChild', asl'), & \underline{\text{otherwise}} \end{cases} \\
&\quad \underline{\text{where}}\ \ (T', pChild', asl') = insert_relevant_suffix(T, actS, pChild, asl).
\end{aligned}$$

Darin bezeichnet $actS$ die jeweils aktuelle Lokation des aktiven Suffixes, $occurs(actS,\ a)$ das in [Kur95] beschriebene Prädikat zur Überprüfung des Nachfolgers a an der Lokation $actS$ und $down_T(s\text{-}loc_T(w), a) \Rightarrow s\text{-}loc_T(wa)$.

Zusammenfassend läßt sich damit formulieren:

$$ukkstep(T_1, a, actS_1, node\_t, asl_1) \Rightarrow (T_2, actS_2, asl_2),$$

wobei T_i, $actS_i$ und asl_i den Bi-Baum, die Lokation des aktiven Suffixes und das aktive Suffixblatt nach Schritt i des Algorithmus 4.6 bezeichnen und $node\_t$ der in Schritt 1 eingefügte Knoten $\overrightarrow{...t}$ ist.

In Abb. 4.3 sind die zwei Iterationen für das Beispiel $t = \texttt{aababa}$, $a = \texttt{a}$ abgebildet.

[10] Zur Erläuterung s. auch [Kur95].

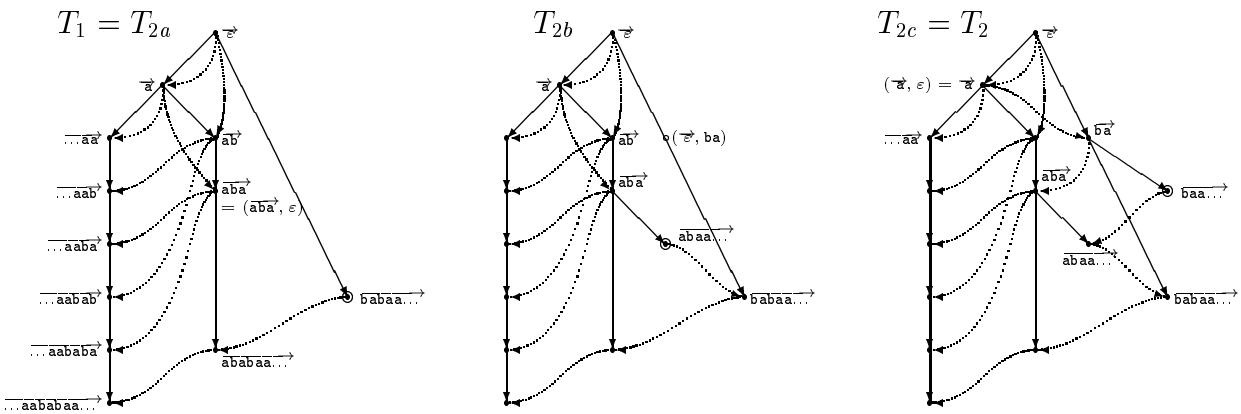


Abbildung 4.3: Einfügen der relevanten Suffixe am Beispiel $t = \texttt{aababa}, a = \texttt{a}$.

4.2.3 Explizitmachen des aktiven Suffixes

Beim Explizitmachen des aktiven Suffixes treten ähnliche Schwierigkeiten auf, wie sie oben beim Verkürzen des aktiven Suffixes zur Realisierung von *linkloc* beschrieben wurden. Als hilfreich wird sich die folgende Aussage erweisen, die eine Eigenschaft des Baumes T_2 beschreibt:

Satz 4.7 (Aktives Suffix nach Schritt 2)
Falls in T_2 das neue aktive Suffix $ua = \alpha(ta)$ nicht durch einen expliziten Knoten repräsentiert wird, ist es von der Form $s\text{-}loc_{T_2}(\alpha(ta)) = (\overrightarrow{u}, a)$. □

Beweis
Sei $ua = \alpha(ta)$ nach Schritt 2 des Algorithmus 4.6 nicht explizit. Es ist zu zeigen, daß dann $s\text{-}loc_{T_2}(\alpha(ta)) = (\overrightarrow{u}, a)$, was sicher der Fall ist, wenn $\overrightarrow{u}$ expliziter Knoten ist.

Es gibt zwei Möglichkeiten für den Ablauf von Schritt 2:

a) Vor dem Auftreten der a-Fortsetzung an u wurde das aktive Suffix bereits um ein oder mehrere Zeichen verkürzt. Dann muß es vorher eine andere Fortsetzung $b \neq a$ von $\alpha(t) = \_.u$ in $t = \left\{ \begin{array}{l} \_\_\_.u \\ \_ub\_\_ \end{array} \right\}$ gegeben haben, u ist also rechtsverzweigendes ta-Wort.

b) ta hatte keine relevanten Suffixe, das aktive Suffix hat sich unmittelbar verlängert: $\alpha(ta) = \alpha(t)a = ua$. In diesem Fall war $\overrightarrow{u}$ aber expliziter Knoten schon in $cat(t)$, wie der folgende Widerspruchsbeweis zeigt:

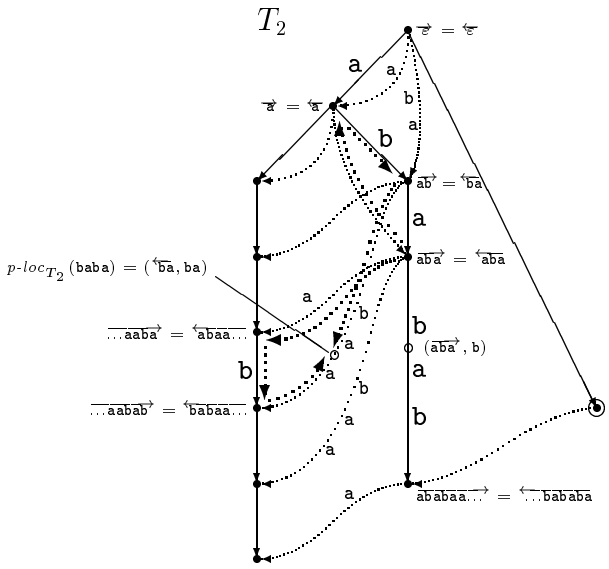


Abbildung 4.4: Die beiden Möglichkeiten, die korrespondierende Lokation des aktiven Suffixes im Präfixbaum zu finden, am Beispiel $t = \texttt{aababa}$, $u = \texttt{aba}$, $a = \texttt{b}$.

Angenommen $\overrightarrow{u} = \overrightarrow{\alpha(t)}$ wäre nicht expliziter Knoten in $cat(t)$ gewesen, dann muß er in Schritt 4 des vorherigen Konstruktionsschrittes gelöscht worden sein, weil sich dort das aktive Präfix verlängert hat. Nach Satz 4.1 in Verbindung mit Satz 4.2 wäre dann $t = \left\{ \begin{array}{l} ua\_\_\_ \\ \_\_\_.u \end{array} \right\}$ und ua tritt kein weiteres Mal in t auf.

Dies bedeutet wiederum, daß ua nicht-eingebettetes Präfix von t ist, also durch einen expliziten Knoten $\overrightarrow{ua}$ in $cat(t)$ repräsentiert war. Diese Aussage steht im Widerspruch zur Voraussetzung des Satzes.

□

Es liegt also folgende Situation vor, wenn zu Beginn von Schritt 3 das aktive Suffix nicht explizit ist: Der implizite Knoten $s\text{-}loc_{T_2}(\alpha(ta)) = (\overrightarrow{u}, a)$ soll explizit gemacht werden. Dafür muß die korrespondierende Lokation im Präfixbaum $p\text{-}loc_{T_2}(au^{-1})$ ermittelt werden, was durch die Funktion *find_pLoc* realisiert wird:

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

Zum Auffinden der gesuchten P-Lokation gibt es ähnlich wie im vorigen Abschnitt wiederum zwei Wege, die sich anhand von Abb. 4.4 ($t = \mathtt{aababa}$, $u = \mathtt{aba}$, $a = \mathtt{b}$) folgendermaßen beschreiben lassen[11]:

a) Man wandert vom aktiven Punkt aus solange im Präfixbaum „aufwärts“, bis man einen Knoten erreicht, an dem eine atomare a-Suffixkante beginnt:

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

Dann folgt man dieser Kante

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

und „kanonisiert“ wieder entsprechend weit „abwärts“ (im Beipiel ist dies nicht nötig: $(\overleftarrow{\mathtt{ba}}, \mathtt{ba})$ ist kanonisches P-Referenzpaar).

Wiederum terminiert der Weg „nach oben“ spätestens an der Wurzel des Baumes, wo ein „manuelles“ Verlängern von $(\overleftarrow{\varepsilon}, u^{-1})$ zu $(\overleftarrow{\varepsilon}, au^{-1})$ möglich ist.

b) Der zweite Vorschlag beschreibt ein ähnliches Vorgehen „nach unten“ im Baum: Man folgt so lange den Präfixkanten „abwärts“[12], bis man einen Knoten erreicht, an dem eine atomare a-Suffixkante beginnt (spätestens an einem Präfixblatt ist das immer der Fall)

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

folgt dieser Kante

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

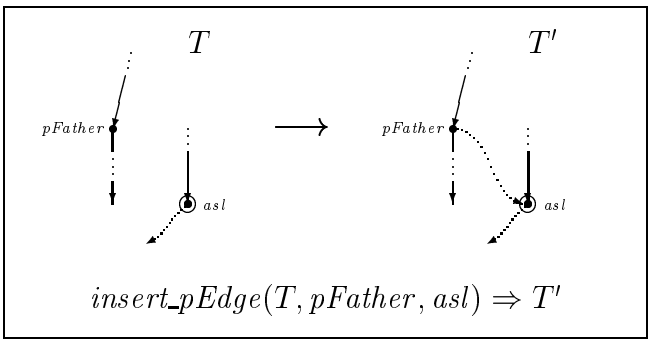
und „kanonisiert“ wieder entsprechend weit im Präfixbaum „aufwärts“:

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

[11] Die „naive“ Lösung, stets von der Wurzel aus in den Baum hinein zu laufen, ist auch hier im *worst case* von quadratischem Aufwand (Bsp.: $t = \mathtt{a}(\mathtt{ab})^i$, Abb. 5.1). Dies veranlaßte schon Weiner, der bei der Entwicklung seines „Klassikers“ unter den Verfahren zur Konstruktion von Suffixbäumen in linearer Zeit [Wei73] auf ein ähnliches Problem stieß, zu einem Umweg „innerhalb des Baumes“, der dem hier unter a) beschriebenen Weg „oben herum“ sehr ähnlich ist.

[12] Welcher Präfixkante man an Verzweigungen auf dem Weg „nach unten“ folgen muß, erkennt man an dem jeweiligen Nachfolgeknoten entlang der a-Suffixkante. Dieser verzweigt nicht im Präfixbaum (da er ja sonst einen atomaren Präfixlink hätte), und die bei ihm beginnende Präfixkante beginnt mit demselben Zeichen wie die zu verfolgende Kante, da sie zu einem Knoten führt, der „unterhalb“ der Ziel-Lokation $p\text{-}loc_{T_2}(au^{-1})$ liegt.

Da nach diesem Schritt der Knoten $\overrightarrow{\alpha(ta)}$ in jedem Fall explizit vorliegt, können nun, wie oben schon mehrfach angedeutet, die Suffixblätter durch eine Präfixkante, die $\overrightarrow{\alpha(ta)}$ mit dem aktiven Suffixblatt *asl* verbindet, wieder in den Baum aufgenommen werden. Dazu dient die Funktion *insert_pEdge*.



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

Den gesamten Schritt 3 des Algorithmus 4.6 führt die Funktion *insert_asl* durch:

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

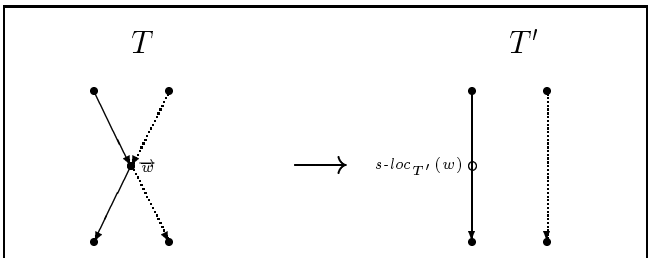
Man beachte, daß *find_pLoc* nur berechnet werden muß, wenn *actS* impliziter Knoten ist, da *insert_iNode* nur dann sein Argument *pChild* auswertet.

4.2.4 Verlängern des aktiven Präfixes

Zunächst gibt das folgende Lemma, das aus einer Kombination der Aussagen von Satz 4.2 und Satz 4.4 folgt, eine einfache Möglichkeit, die Bedingung „Falls sich das aktive Präfix verlängert“ effizient zu überprüfen:

Lemma 4.8 (Verlängern des aktiven Präfixes)
Wenn das neue aktive Suffix $\alpha(ta)$ in $cat(t)$ durch ein Präfixblatt repräsentiert wird, verlängert sich im Schritt $t \to ta$ das aktive Präfix. □

Das Löschen des Knotens $\overrightarrow{\alpha(ta)}$ selber bereitet dann keine Schwierigkeiten mehr: Es müssen lediglich die nach oben und unten führenden Suffix- und Präfixkanten miteinander verknüpft werden, was durch die Funktion *delete_iNode* realisiert wird.



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

Den gesamten Schritt 4 des Algorithmus 4.6 beschreibt die Funktion *lengt-*

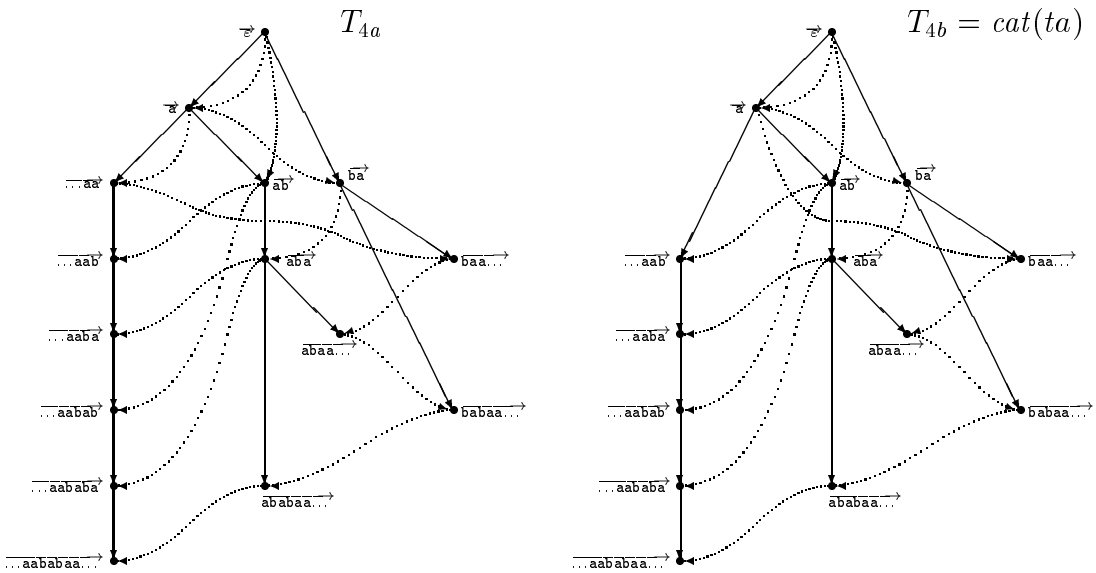


Abbildung 4.5: Löschen des aktiven Knotens am Beispiel $t = \texttt{aababa}$, $a = \texttt{a}$, $actS = (\overrightarrow{\ldots\texttt{aa}}, \varepsilon)$.

hen_actP:

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

Abb. 4.5 führt das Beispiel $t = \texttt{aababa}$, $a = \texttt{a}$ fort.

4.3 Der *online*-Algorithmus (unidirektional)

Durch Komposition der vier Funktionen, die die Schritte 1 – 4 des Algorithmus 4.6 realisieren, ergibt sich *catstep*:

$$\begin{aligned} catstep(T, a, actS, asl) = (T_4, actS_4, asl_2) \\ \underline{\text{where}}\ \ (T_4, actS_4) &= lengthen\_actP(T_3, actSnode_3) \\ (T_3, actSnode_3) &= insert\_asl(T_2, actS_2, asl_2) \\ (T_2, actS_2, asl_2) &= ukkstep(T_1, a, actS, node\_t, asl) \\ (T_1, node\_t) &= insert\_node\_t(T, asl). \end{aligned}$$

Zur Anwendung von *catstep* auf Sequenzen dient dann die Funktion *cat*:

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

Als Zusammenfassung dieses Kapitels ergibt sich damit:

Satz 4.9 (*Online*-Konstruktion von kompakten Affixbäumen)
Sei $T_0 = cat(\varepsilon)$. Dann erzeugt $cat(T_0, t, (root(T_0), \varepsilon), root(T_0))$ den kompakten Affixbaum des Textes t[13]. □

Obwohl Algorithmus 4.6 damit eine funktionale Form erhalten hat, läßt sich *cat* nicht ohne weiteres in einer rein funktionalen Programmiersprache implementieren. Auf der globalen Struktur des Affixbaumes werden nämlich lokale Änderungen vorgenommen, die mit den üblichen Datentypen herkömmlicher *lazy*-funktionaler Programmiersprachen nicht durchführbar sind[14]. Da aber die Forderung nach *single-threadedness* erfüllt ist (d. h. es werden keine Kopien von Teilen der globalen Datenstruktur gemacht), ist eine Implementierung mit geeigneten Techniken, wie sie moderne Sprachen wie Haskell oder Clean [Pla95] zur Verfügung stellen, denkbar. Eine entsprechende Implementierung von *ukk* ist in [Kur95] und [Kra95] beschrieben, eine vergleichbare funktionale Implementierung von *cat* im Rahmen dieser Arbeit wurde aber als zu weit von der ursprünglichen Fragestellung entfernt empfunden.

4.4 Anmerkungen zum bidirektionalen Verfahren

Da Affixbäume eine bzgl. Vertauschen von Suffix- und Präfixkanten absolut symmetrische Struktur haben, läßt sich Algorithmus 4.6 analog auch für den umgekehrten Schritt $cat(t) \rightarrow cat(at)$ anwenden.

Die beiden neuralgischen Punkte, an denen sich alle Veränderungen im Baum orientieren, sind dann das aktive Präfix und das aktive Präfixblatt. Ebenso

[13] Damit $insert\_node\_t(T_0, root(T_0))$ definiert ist, muß die Wurzel von $T_0 = cat(\varepsilon)$ einen Suffix- und einen Präfixlink auf sich selber haben. Da diese Links für die restliche Konstruktion aber irrelevant sind, wurden sie bisher nicht erwähnt. Als Alternative kann auch direkt mit $cat(a)$ begonnen werden, dann tritt dieses Problem gar nicht erst auf.

[14] Eine ausführliche Diskussion derselben Problematik bei *ukk* findet sich in [GK94b].

wie beim aktiven Suffix und aktiven Suffixblatt handelt es sich hier um Invarianten der Konstruktion, d. h. vor einem Schritt $cat(t) \rightarrow cat(at)$ müssen die P-Lokation von $\alpha^{-1}(t)$ und das aktive Präfixblatt in $cat(t)$ bekannt sein, und danach liegen die entsprechenden Punkte von $cat(at)$ vor.

Um nun in beliebiger Reihenfolge t nach rechts und links erweitern zu können, muß gewährleistet sein, daß bei Verlängerung des Textes in eine Richtung, die Invarianten der anderen Richtung aktuell bleiben.

Zur Erläuterung, wie dies geschehen kann, wird wiederum der Schritt $cat(t) \rightarrow cat(ta)$ betrachtet:

1. Abschnitt 4.1.1 ging ausführlich auf die möglichen Veränderungen des aktiven Präfixes ein. Da in *lengthen_actP* ohnehin explizit überprüft wird, ob sich das aktive Präfix verlängert, können dort auch die entsprechenden Änderungen im Referenzpaar des aktiven Präfixes vorgenommen werden.

   Darüber hinaus ist es möglich, daß in *insert_relevant_suffix* beim Einfügen eines inneren Knotens das aktive Präfix explizit gemacht wird. Auch an dieser Stelle muß also ggf. das Referenzpaar des aktiven Präfixes aktualisiert werden.

2. Das aktive Präfixblatt kann sich an zwei Stellen ändern:

   (a) Falls vor dem Einfügen des Knotens $\overrightarrow{t}$ der (verlängerte) Knoten $\overrightarrow{ta}$ das aktive Präfixblatt war, ist es danach der Knoten $\overrightarrow{t}$, da dieser ja ein kürzeres Präfixblatt darstellt.

   (b) Wenn in *lengthen_actP* der Knoten gelöscht wird, der das aktive Suffix repräsentiert, ist dieser auch stets das aktive Präfixblatt, wie man sich leicht überlegen kann. Das neue aktive Präfixblatt ist dann das nächstlängere Präfixblatt, das mit dem alten durch eine atomare Suffixkante verbunden war.

Damit ist gezeigt, daß es möglich ist, auch die Invarianten der jeweils anderen Konstruktionsrichtung *up-to-date* zu halten. Das Verfahren ist also bidirektional.

Kapitel 5

Komplexitätsbetrachtungen

Ziel dieses Kapitels ist es zu zeigen, daß die in Kapitel 4 beschriebene *online*-Konstruktion von kompakten Affixbäumen praktisch mit linearem Zeit- und Platzbedarf $\mathcal{O}(n)$ möglich ist. Während die Linearität des Platzbedarfs offensichtlich ist[1], erweist sich die Diskussion des Zeitbedarfs als unerwartet schwierig.

Tatsächlich kann in dieser Arbeit nicht der vollständige Beweis erbracht werden, daß die *online*-Konstruktion von Affixbäumen in linearer Zeit möglich ist, wenn auch viele Indizien dafür sprechen.

Daß die Zeitkomplexität zusätzlich dazu im allgemeinen Fall[2] auch noch linear von der Alphabetgröße abhängt, ist ein Effekt, auf den hier nicht weiter eingegangen werden soll. Er ist genauso wie in *ukk* auf den proportionalen Zusammenhang zwischen Alphabetgröße und Verzweigungsgrad des Baumes zurückzuführen.

5.1 Überblick

Es werden die Funktionen *insert_node_t*, *ukkstep*, *insert_asl* und *lengthen_actP*, die die vier Schritte des Algorithmus 4.6 realisieren, einzeln betrachtet. Da

[1] Bis auf den Text und den zu konstruierenden Baum, die beide von der Größe $\mathcal{O}(n)$ sind, wird kein weiterer Speicherplatz benötigt. Obwohl während der Konstruktion Knoten gelöscht werden, steigt die Größe des Baumes monoton an (da pro Schritt maximal ein Knoten gelöscht wird, aber auch mindestens ein Knoten hinzukommt), womit der Fall ebenfalls ausgeschlossen werden kann, daß Zwischenstrukturen größer sind als der resultierende Affixbaum.

[2] Vgl. hierzu die in [Kur95] beschriebenen verschiedenen Möglichkeiten der Suche des richtigen Nachfolgers an einer Verzweigung im Baum.

sich diese Untersuchungen schon für das unidirektionale Verfahren als sehr kompliziert erwiesen haben, wird darauf verzichtet, auch noch Fälle zu diskutieren, in denen die Richtung der Konstruktion in einem ungünstigen Augenblick wechselt.

Schritt 1: *insert_node_t*

In jedem der n Schritte zur Konstruktion von $cat(t)$ wird *insert_node_t* genau einmal aufgerufen. Da das aktive Suffixblatt zur Zeit des Aufrufs bekannt ist und *insert_node_t* keine Schleifen oder Rekursionen enthält, ergibt sich insgesamt der Aufwand $\mathcal{O}(n)$.

Schritt 2: *ukkstep*

Zwar ist aus [Ukk93] bekannt, daß die *online*-Konstruktion von kompakten Suffixbäumen in $\mathcal{O}(n)$ Zeit möglich ist, der Beweis von dort läßt sich aber nicht an allen Stellen auf die veränderte Datenstruktur des Suffixbaumes im Affixbaum übertragen.

Während das Einfügen der neuen Knoten und Blätter in den Suffixbaum genauso oft wie bei *ukk* stattfindet, ist es hier aufwendiger, mittels *linkloc* die entsprechenden Positionen im Baum zu erreichen. Für den Fall, daß bei der Suche nach dem atomaren Suffixlink immer der Weg „oben herum“ gewählt wird (was dem entsprechenden Schritt in *ukk* am nächsten kommt), ist das Verfahren sogar von quadratischem Aufwand, wie man sich leicht am Beispiel $t = \mathtt{aa}(\mathtt{ba})^i$, $a = \mathtt{a}$ (Abb. 5.1) vergegenwärtigen kann.

Aus diesem Grunde wird im folgenden davon ausgegangen, daß stets der kürzere der beiden in Abschnitt 4.2.2 beschriebenen Wege („oben herum“ oder „unten herum“) gewählt wird, was sich beispielsweise dadurch realisieren läßt, daß parallel beide Richtungen verfolgt werden, und sobald einer der beiden Wege einen Knoten mit atomarem Link erreicht hat, der andere Weg abgebrochen wird.

Für die ausführliche Komplexitätsanalyse erfolgt ähnlich wie in [Ukk93] eine Aufspaltung von *linkloc* in drei Teilfunktionen, deren Komplexität separat diskutiert wird: *find_aLink*, *canonize_down* und *canonize_up*. Die einzelnen Untersuchungen finden sich in den Abschnitten 5.2.1 – 5.2.3.

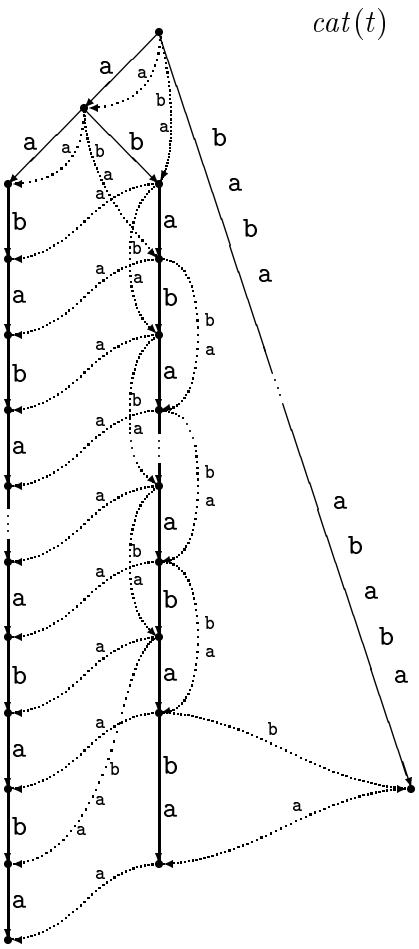


Abbildung 5.1: Ein Beispiel für quadratisches Verhalten von *ukkstep*, falls *linkloc* stets „oben herum“ ausgeführt wird: $t = \mathtt{aa}(\mathtt{ba})^i$, $a = \mathtt{a}$.

Schritt 3: *insert_asl*

Während das Einfügen des neuen Knotens in konstanter Zeit möglich ist und die Parameter *actS* und *asl* als Invarianten bekannt sind, erweist sich das Aufsuchen der P-Lokation des neuen aktiven Suffixes mit *find_pLoc* als aufwendig. Aufgrund der Erfahrungen mit *linkloc* wird auch hier stets der kürzere der beiden möglichen Wege gewählt. Abschnitt 5.2.4 behandelt diesen Schritt ausführlich.

Schritt 4: *lengthen_actP*

Die Bedingung für das Verlängern des aktiven Präfixes lautet nach Lemma 4.8: „Wenn das neue aktive Suffix durch ein Präfixblatt repräsentiert wird". Dies kann in $\mathcal{O}(1)$ überprüft werden, das Löschen des Knotens $\overrightarrow{\alpha(ta)}$ ist ebenfalls in konstanter Zeit durchführbar, so daß für den gesamten Text $\mathcal{O}(n)$ eine obere Zeitschranke bildet.

5.2 Diskussion der Problemfälle

5.2.1 *find_aLink*

Wie der folgende Satz zeigt, ist die Suche nach einem nicht-atomaren Suffixlink nur dann nötig, wenn das kanonische Referenzpaar des (ggf. schon verkürzten) aktiven Suffixes *actS* expliziter, ausschließlich linksverzweigender Knoten ist:

Satz 5.1 (Atomarer Suffixlink des impliziten aktiven Suffixes)
Wenn *actS* beim Aufruf von *linkloc* in *ukkstep* impliziter Knoten ist (und nicht an der Wurzel liegt: $actS = (\overrightarrow{bu}, cv)$), dann existiert immer ein atomarer Suffixlink $\overrightarrow{bu} \overset{b}{\dashleftarrow} \overrightarrow{u}$. □

Beweis
Sei $actS = (\overrightarrow{bu}, cv)$ das kanonische Referenzpaar des (evtl. schon verkürzten) aktiven Suffixes von $t = \left\{ \begin{array}{l} \_\_.bucv \\ \_bucv.\_\_ \end{array} \right\}$. Es ist zu zeigen, daß $\overrightarrow{bu}$ einen atomaren Suffixlink hat, d. h. $\overrightarrow{bu} \overset{b}{\dashleftarrow} \overrightarrow{u} \in p\text{-}edges(cat(t))$[3]. Nach Bemerkung 3

[3] Obwohl der Baum zu diesem Zeitpunkt schon verändert wurde, ist die hier betrachtete Stelle noch unverändert wie in $cat(t)$.

zu Definition 3.1 ist dies sicher der Fall, wenn $\overrightarrow{bu}$ im Suffixbaum in $cat(t)$ verzweigt.

Angenommen, $\overrightarrow{bu}$ verzweigte nicht im Suffixbaum in $cat(t)$, bu wäre also links- aber nicht rechtsverzweigendes t-Wort. Dann sind zwei Fälle denkbar:

1. $t = \left\{ \begin{array}{l} \_\_xbucv \\ \_ybuc\_\_ \end{array} \right\}, x \neq y$, buc ist also ebenfalls linksverzweigendes t-Wort, was im Widerspruch zu der Voraussetzung steht, daß $(\overrightarrow{bu}, cv)$ kanonisches Referenzpaar ist.

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

   $\Rightarrow \quad t = \left\{ \begin{array}{l} \_\_xbucv \\ \_x\_ybu\_\_ \\ \_\_\_\_ybu \end{array} \right\} \quad \Rightarrow \quad . = c$, da bu n. Ann. nicht rechtsverzweigt

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

   d. h. buc ist linksverzweigendes t-Wort, was wie im ersten Fall einen Widerspruch zur Voraussetzung darstellt.

□

Find_aLink benötigt also nur in denjenigen Fällen mehrere Schritte, in denen *actS* expliziter, nur im Präfixbaum verzweigender Knoten ist.

Zu dieser Klasse von Knoten, die nur linksverzweigende t-Worte repräsentieren, sollen nun einige Bemerkungen gemacht werden:

1. Solche Knoten werden in *cat* ausschließlich durch die Funktion *insert_asl* erzeugt. Es kann davon also während der Konstruktion von $cat(t)$ insgesamt höchstens n Stück geben.

2. Falls in mehreren Schritten hintereinander die Menge der relevanten Suffixe leer ist (sich das aktive Suffix also mehrmals direkt verlängert, ohne zwischendurch verkürzt zu werden), entsteht eine „Kette“ solcher nur im Präfixbaum verzweigender Knoten.

   Eine solche „Kette“ ist Ursache für die quadratische Komplexität der Funktion *find_aLink*, wenn stets der Weg „oben herum“ gewählt wird (Abb. 5.1).

3. Mit jedem *find_aLink* wird ein solcher Knoten zu einem in beiden Bäumen verzweigenden Knoten „entschärft“, da er damit einen atomaren Suffixlink zu dem im darauffolgenden Schritt eingefügten Knoten erhält.

Aufgrund der Aussage von Punkt 3 kann *find_aLink*, wenn stets der kürzere der beiden möglichen Wege gewählt wird, höchstens vom Aufwand $\mathcal{O}(n \log n)$ sein: Die maximale Gesamtlänge aller „Ketten“ ist n. Mit jedem *find_aLink* wird eine Kette in zwei Teilketten zerteilt, die in späteren Schritten wiederum in je zwei Teile aufgespalten werden können usw. Im schlimmsten Fall ergibt sich jedesmal der maximale Weg, indem die Ketten von Mal zu Mal halbiert werden, was gerade logarithmischem Verhalten entspricht.

In diesem Ergebnis sind aber noch längst nicht alle Nebenbedingungen berücksichtigt, die die Anzahl Schritte von *find_aLink* einschränken. Beispielweise sind nicht beliebige Abfolgen von aktiven Suffixen möglich. Damit das aktive Suffix zwei oder mehr Male in dieselbe „Kette“ von nur im Präfixbaum verzweigenden Knoten „geraten“ kann, müssen zwischendurch weitere Schritte stattfinden, die den erhöhten Aufwand für das mehrmalige „Entlanglaufen“ derselben „Kette“ evtl. kompensieren.

Die Existenz dieser weiteren Einschränkungen und die in Abschnitt 5.3 dargestellten empirischen Ergebnisse sind starke Indizien für ein lineares Verhalten von *find_aLink*.

5.2.2 *canonize_down*

Für das „Abwärts-Kanonisieren“ läßt sich der entsprechende Beweis aus [Ukk93] übernehmen:

Das zweite Element des Referenzpaares des aktiven Suffixes wird für den gesamten Text maximal n-Mal um je ein Zeichen verlängert. Da es mit jedem *canonize_down* um mindestens ein Zeichen verkürzt wird, sind insgesamt maximal n Schritte von *canonize_down* möglich.

5.2.3 *canonize_up*

Das hier vorliegende Problem läßt sich folgendermaßen beschreiben (s. Abb. 5.2): Sei $actS = (\overrightarrow{bu}, \varepsilon)$ das zu verkürzende aktive Suffix, $\overrightarrow{buvw}$ der erste Knoten „unterhalb“ von $\overrightarrow{bu}$ mit atomarem Suffixlink. Wie viele Knoten $\overrightarrow{uv}$ auf dem Rückweg „nach oben“ kann es maximal geben?

Für den Knoten $\overrightarrow{uv}$ gilt:

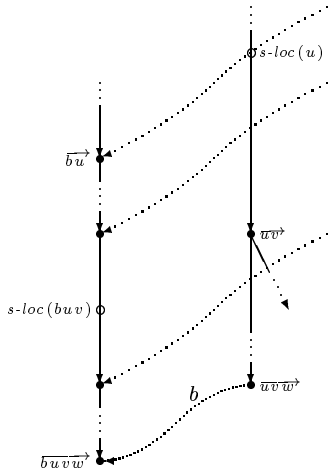


Abbildung 5.2: Erläuterung zur Diskussion von *canonize_up*.

1. $\overrightarrow{buv}$ ist kein Knoten, da *find_aLink* sonst schon dort dem atomaren Suffixlink $\overrightarrow{buv} \overset{b}{\dashrightarrow} \overrightarrow{uv}$ gefolgt wäre.
2. $\overrightarrow{uv}$ verzweigt nicht im Präfixbaum, da nicht einmal $\overrightarrow{u}$ im Präfixbaum verzweigt, $\overrightarrow{uv}$ ist also nur im Suffixbaum verzweigender Knoten.

Leider ist dadurch noch kein Kriterium gegeben, das eine Abschätzung der Maximalzahl solcher Knoten auf dem Weg von *canonize_up* erlaubt. Die empirischen Ergebnisse hierzu in Abschnitt 5.3 deuten aber auf ein besonders harmloses lineares Verhalten mit konstantem Faktor $\frac{1}{2}$ für den *worst case* hin.

5.2.4 *find_pLoc*

Bei *find_pLoc* handelt es sich wohl um die am schwierigsten zu durchschauende Unterfunktion von *cat*. Dennoch konnten zwei interessante Feststellungen gemacht werden:

Zunächst kann gezeigt werden, daß ein „Kanonisieren“ nicht nötig ist, wenn zum Finden der korrespondierenden P-Lokation der Weg „unten herum“ gewählt wird.

Das vorliegende Szenario wurde bereits in Abschnitt 4.2.3 beschrieben (s. auch Abb. 5.3): Ein „Aufwärts-Kanonisieren“ wäre nur nötig, wenn es (innere) Knoten von der Art $\overrightarrow{v_i \dots v_1 ua}$ gäbe. Ein solcher Knoten kann aber nicht existieren, da $v_i \dots v_1 ua$ weder linksverzweigend sein kann (weil dann auch

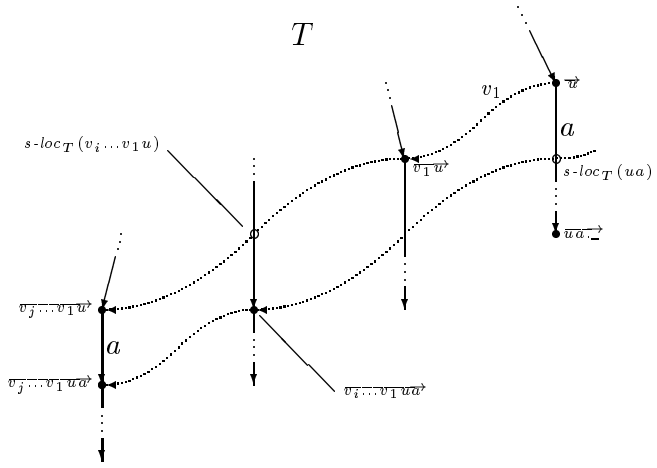


Abbildung 5.3: Ein Knoten $\overrightarrow{v_i \dots v_1 ua}$ kann in T nicht existieren.

$v_i \dots v_1 u$ linksverzweigend und damit expliziter Knoten wäre) noch rechtsverzweigend (mit derselben Begründung für ua).

Zusätzlich hat sich gezeigt, daß bei der Wahl stets des kürzeren Weges auch für die Fälle, wo der Weg „oben herum“ gewählt wird, kein Kanonisieren nötig ist, was Anlaß gab, folgende Behauptung zu formulieren, die im Rahmen dieser Arbeit allerdings nicht formal gezeigt, wohl aber empirisch an allen Sequenzen über $\{\mathtt{a}, \mathtt{b}\}$ bis zur Textlänge $n = 20$ bestätigt werden konnte:

Behauptung 5.2 (Abwärts-kanonisieren ist nicht nötig)
Jedesmal, wenn bei *find_pLoc* der Umweg „oben herum“ auf dem Rückweg über Knoten läuft, ist der Weg „unten herum“ nur einen Schritt lang, und zwar existiert eine Präfixblatt-Kante $\overrightarrow{u} \dashrightarrow \overrightarrow{\dots v_1 u}$. □

Selbst damit bleibt aber die entscheidende Frage offen, wie weit die jeweils nächste atomare a-Kante vom Knoten $\overrightarrow{u}$ entfernt ist.

Solange dies nicht geklärt ist, muß von der Komplexität $\mathcal{O}(n^2)$ der „naiven“ Lösung ausgegangen werden, auch wenn die im folgenden Abschnitt dargestellten empirischen Ergebnisse lineares Verhalten zumindest nicht ausschließen.

5.3 Empirische Untersuchungen

Da bei den analytischen Untersuchungen die Komplexität von *find_aLink*, *canonize_up* und *find_pLoc* nicht befriedigend geklärt werden konnte, wurden empirische Untersuchungen angefertigt, in denen abstrakt die Anzahl Schritte der betreffenden Funktionen gezählt wurden. Auf diese Weise sollten

- entweder Beispieltexte gefunden werden, für die sich das Verfahren schlechter als linear verhält[4],
- oder Hinweise für weitere analytische Untersuchungen ausfindig gemacht werden, etwa durch Ermittlung des konstanten Faktors, der den schlechtesten linearen Fall nach oben beschränkt.

In den Tabellen 5.1, 5.2 und 5.3 sind die Ergebnisse zusammengefaßt. Es wurden für zunehmende Textlängen n jeweils alle Texte $t \in \{\mathtt{a},\mathtt{b}\}^n$ untersucht, wobei Vertauschungen der Zeichen `a` und `b` unberücksichtigt blieben. In der linken Spalte ist jeweils die Textlänge n angegeben, in der mittleren Spalte die maximale Anzahl Schritte der entsprechenden Funktion und in der rechten Spalte in Klammern die Häufigkeit, wie oft dieser *worst case* eingetreten ist (wiederum ohne Berücksichtigung der Vertauschungen), sowie die Texte, die diese Anzahl Schritte erzeugten.

Während *find_aLink* und *canonize_up* ein recht eindeutiges lineares Verhalten zeigen (und es deshalb verwundert, warum der analytische Beweis sich als so schwierig erweist), sind die Ergebnisse von *find_pLoc* schwieriger zu deuten:

Zunächst scheint mit dem Muster $t = \mathtt{ab}^i$ der *worst case* (mit linearem Verhalten, Faktor 1) beschrieben zu sein. Ab der Textlänge $n = 13$ schleicht sich jedoch noch ein zweites Muster $t = \mathtt{aab}^{2i}\mathtt{ab}^i\mathtt{a}$ ein, das das erste Muster ab $n = 16$ sogar überholt. Dieses Muster beschreibt (verallgemeinert für längere Sequenzen) ebenfalls lineares Verhalten mit dem konstanten Faktor 1.33. Ab der Textlänge $n = 26$ findet sich wiederum ein Muster mit schlechterem Verhalten ($t = \mathtt{aab}^{4i}\mathtt{ab}^{2i}\mathtt{ab}^i\mathtt{a}$, konstanter Faktor 1.43) usw.

Es stellt sich nun die Frage, ob gegen eine, und wenn ja, gegen welche, Sequenz diese Folge schlechtester Muster konvergiert. Wird die Konvergenzfolge ebenfalls noch lineares Verhalten beschreiben, oder steigt der Faktor beliebig weit an, so daß sich ein schlechteres als lineares Verhalten ergibt?

Eine mögliche Verallgemeinerung obiger *worst cases* wäre die Sequenz

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

[4] Tatsächlich wurde auf diese Weise der Text $ta = \mathtt{aa}(\mathtt{ba})^i\mathtt{a}$ als Beispiel für die quadratische Komplexität von *find_aLink* „oben herum“ gefunden.

Textlänge n	# Schritte im *worst case*	(#)	Texte
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)	$\mathtt{a(ab)^{16}b}$

Tabelle 5.1: *Worst cases* von *find_aLink* für Texte bis zur Länge 34 über dem Alphabet $\{\mathtt{a}, \mathtt{b}\}$.

Textlänge n	# Schritte im *worst case*	(#)	Texte
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)	$\mathtt{a(ab)^{16}b}$

Tabelle 5.2: *Worst cases* von *canonize_up* für Texte bis zur Länge 34 über dem Alphabet $\{\mathtt{a}, \mathtt{b}\}$.

Textlänge n	# Schritte im *worst case*		(#)	Texte
0	0			—
1	0			—
2	0			—
3	0			—
4	1	(+1)	(1)	$\mathtt{ab}^3$
5	2	(+1)	(1)	$\mathtt{ab}^4$
6	3	(+1)	(1)	$\mathtt{ab}^5$
7	4	(+1)	(1)	$\mathtt{ab}^6$
8	5	(+1)	(1)	$\mathtt{ab}^7$
9	6	(+1)	(1)	$\mathtt{ab}^8$
10	7	(+1)	(1)	$\mathtt{ab}^9$
11	8	(+1)	(1)	$\mathtt{ab}^{10}$
12	9	(+1)	(1)	$\mathtt{ab}^{11}$
13	10	(+1)	(2)	$\mathtt{ab}^{12}, \mathtt{aab}^6\mathtt{ab}^3\mathtt{a}$
14	11	(+1)	(3)	$\mathtt{ab}^{13}, \mathtt{aab}^7\mathtt{ab}^3\mathtt{a}, \mathtt{aab}^6\mathtt{ab}^3\mathtt{ab}$
15	12	(+1)	(6)	$\mathtt{ab}^{14}, \mathtt{aab}^8\mathtt{ab}^3\mathtt{a}, \mathtt{aab}^7\mathtt{ab}^4\mathtt{a}, \ldots$
16	14	(+2)	(1)	$\mathtt{aab}^8\mathtt{ab}^4\mathtt{a}$
17	15	(+1)	(2)	$\mathtt{aab}^9\mathtt{ab}^4\mathtt{a}, \mathtt{aab}^8\mathtt{ab}^4\mathtt{ab}$
18	16	(+1)	(6)	...
19	18	(+2)	(2)	$\mathtt{aab}^{10}\mathtt{ab}^5\mathtt{a}, \mathtt{aab}^8\mathtt{ab}^4\mathtt{ab}^2\mathtt{a}$
20	19	(+1)	(4)	...
21	20	(+1)	(10)	...
22	22	(+2)	(3)	$\mathtt{ab}^{12}\mathtt{ab}^6\mathtt{a}, \mathtt{aab}^{10}\mathtt{ab}^5\mathtt{ab}^2\mathtt{a}, \mathtt{aab}^8\mathtt{ab}^4\mathtt{ab}^2\mathtt{ab}^2\mathtt{a}$
23	23	(+1)	(7)	...
24	24	(+1)	(18)	...
25	26	(+2)	(5)	$\mathtt{aab}^{14}\mathtt{ab}^7\mathtt{a}, \mathtt{aab}^{11}\mathtt{ab}^6\mathtt{ab}^3\mathtt{a}, \ldots$
26	28	(+2)	(1)	$\mathtt{aab}^{12}\mathtt{ab}^6\mathtt{ab}^3\mathtt{a}$
27	29	(+1)	(2)	$\mathtt{aab}^{13}\mathtt{ab}^6\mathtt{ab}^3\mathtt{a}, \mathtt{aab}^{12}\mathtt{ab}^6\mathtt{ab}^3\mathtt{ab}$
28	30	(+1)	(10)	...
29	32	(+2)	(2)	$\mathtt{aab}^{14}\mathtt{ab}^7\mathtt{ab}^3\mathtt{a}, \mathtt{aab}^{11}\mathtt{ab}^6\mathtt{ab}^3\mathtt{ab}^3\mathtt{a}$
30	34	(+2)	(1)	$\mathtt{aab}^{12}\mathtt{ab}^6\mathtt{ab}^3\mathtt{ab}^3\mathtt{a}$
31	35	(+1)	(2)	$\mathtt{aab}^{13}\mathtt{ab}^6\mathtt{ab}^3\mathtt{ab}^3\mathtt{a}, \mathtt{aab}^{12}\mathtt{ab}^6\mathtt{ab}^3\mathtt{ab}^3\mathtt{ab}$
32	36	(+1)	(8)	...
33	38	(+2)	(3)	$\mathtt{aab}^{16}\mathtt{ab}^8\mathtt{ab}^4, \mathtt{aab}^{14}\mathtt{ab}^7\mathtt{ab}^3\mathtt{ab}^3,$ $\mathtt{aab}^{11}\mathtt{ab}^6\mathtt{ab}^3\mathtt{ab}^3\mathtt{ab}^4$
34	40	(+2)	(1)	$\mathtt{aab}^{12}\mathtt{ab}^6\mathtt{ab}^3\mathtt{ab}^3\mathtt{ab}^4$

Tabelle 5.3: *Worst cases* von *find_pLoc* für Texte bis zur Länge 34 über dem Alphabet $\{\mathtt{a}, \mathtt{b}\}$.

In der folgenden Tabelle ist für einige Werte von k und i die Anzahl Schritte von *find_pLoc* zusammengestellt:

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	?
allgemein	$4i-2$	$10i-2$	$22i-2$	$94i-2$	$3070i-2$	$98302i-2$
n	$3i+4$	$7i+5$	$15i+6$	$63i+8$	$2047i+13$	$65535i+18$
$\frac{\#\text{ Schritte}}{n}$	$\sim \frac{4}{3}$ ≈ 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

Für Sequenzen dieser Art scheint das Verhältnis der Anzahl Schritte von *find_pLoc* zur Textlänge n also stets unterhalb von 1.5 zu bleiben. Auch die erste Ableitung der Anzahl Schritte im *worst case* (Zahlen in Klammern in der mittleren Spalte von Tabelle 5.3) deutet auf diesen konstanten Faktor hin: Für größere n wechseln sich jeweils zwei Schritte (+1) mit zwei Schritten (+2) ab.

Leider ließen sich die empirischen Untersuchungen in dieser Form nicht weiter führen, da die exponentiell ansteigende Zahl zu untersuchender Sequenzen schnell an ihre natürlichen Grenzen stieß. Insbesondere für größere Alphabete waren kaum Messungen möglich. Da die hierzu durchgeführten Messungen ($|\mathcal{A}| = 3$ bis $n = 20$, $|\mathcal{A}| = 4$ bis $n = 15$) im *worst case* aber immer exakt dieselben Ergebnisse wie für $|\mathcal{A}| = 2$ ergaben, kann davon ausgegangen werden, daß mit dem Alphabet $\{\mathtt{a}, \mathtt{b}\}$ bereits alle „schlechten“ Sequenzen erzeugt werden können.

5.4 Laufzeitmessungen

Schließlich sollten die so ermittelten Ergebnisse auch an für praktische Einsätze relevanten, längeren Texten validiert werden. Dazu wurden Laufzeitmessungen für zufällige Texte über verschieden großen Alphabeten (wiederum

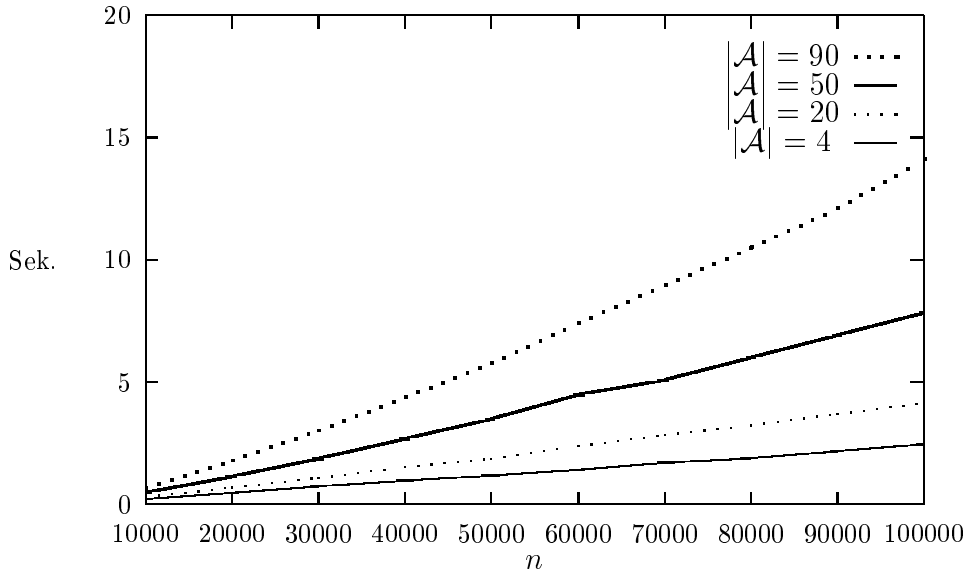


Abbildung 5.4: Laufzeiten von *bigrep* für zufällige Texte, abhängig von der Textlänge n und der Alphabetgröße $|\mathcal{A}|$.

Bernoulli-Verteilung) durchgeführt, die in Abb. 5.4 dargestellt sind[5].

Man erkennt deutlich die auch bei *ukk* vorhandene Alphabetgrößenabhängigkeit aufgrund des höheren Verzweigungsgrades bei Bäumen von Texten über einem großen Alphabet. Ebenfalls geben diese Kurven einen weiteren Hinweis auf die erhoffte Linearität des Verfahrens, wenn auch ein leichtes „Durchhängen“ zu beobachten ist. Da solche Effekte aber von der Speicherverwaltung des Rechners herrühren können, muß dies nicht ein schlechteres als lineares Verhalten des Algorithmus bedeuten. Zumindest im *average case* scheint sich also die Vermutung der Zeitkomplexität $\mathcal{O}(|\mathcal{A}|n)$ zu bestätigen.

Bei einem quantitativen Vergleich der Laufzeiten mit denen von *ukk*[6], die in der folgenden Tabelle zusammengestellt sind, ist zu beobachten, daß zur Konstruktion von *cat* generell etwa die dreifache Zeit benötigt wird[7]:

[5]Die Messungen wurden auf einer SPARCstation 20 durchgeführt, die über einen Hauptspeicher von 32 MB verfügt, der aber an keiner Stelle voll ausgeschöpft wurde (max. Prozeßgröße: 13 MB). Das Betriebssystem war Solaris v2.4, gemessen wurde mit dem Unix-Tool *rusage*. Es wurde jeweils über zehn Messungen gemittelt.

[6]Verwendet wurde die in [GK95] beschriebene C-Implementierung.

[7]Die Zeitangaben sind in Sekunden, jeweils über zehn Messungen gemittelt.

Art des Textes ($n = 100000$)	*ukk*	*cat*
Random-Sequenz, $\|\mathcal{A}\| = 4$	0.7	2.2
Random-Sequenz, $\|\mathcal{A}\| = 20$	1.6	4.3
Random-Sequenz, $\|\mathcal{A}\| = 50$	2.8	7.9
Random-Sequenz, $\|\mathcal{A}\| = 90$	4.8	14.6
Genetische Daten[8]	0.8	2.3
Manual-Page	1.2	2.9
Fibonacci-String	0.2	1.3

Außerdem wird die vierfache Menge an Speicher benötigt, was auf die wesentlich größere zu erzeugende Datenstruktur zurückgeführt werden kann, da mit dem Affixbaum nicht nur zwei Suffixbäume erzeugt werden müssen; es ist auch notwendig, alle Kanten bidirektional einzutragen und Blätter ebenfalls als vollständige Knoten zu repräsentieren, da sie gleichzeitig innere Knoten des jeweils anderen Baumes sind.

Dennoch reichten die Ressourcen des oben beschriebenen Rechners aus, die Affixbäume von Sequenzen bis zu einer Länge von 250000 Zeichen problemlos innerhalb des zur Verfügung stehenden Hauptspeichers zu konstruieren.

[8] Ausschnitt aus der Nucleotidsequenz des Chromosoms XI von Saccharomyces cerevisiae über dem Alphabet $\mathcal{A} = \{C, G, T, A\}$

Kapitel 6

Das Programm *bigrep*

In diesem Kapitel wird kurz eine einfache Anwendung für Affixbäume beschrieben, die im Rahmen dieser Arbeit erstellt wurde. Es handelt sich hierbei um das Programm *bigrep*, das ähnlich wie das bekannte Unix-Tool *grep* zur Suche von (kurzen) Mustern in (langen) Texten verwendet werden kann.

Das Spezielle an *bigrep* ist, daß im Gegensatz zu *grep* und allen bekannten Varianten davon Text und Muster bidirektional verwaltet werden. Außerdem können in ein- und demselben Text nacheinander mehrere Muster gesucht und der Text interaktiv erweitert werden.

In *bigrep* stehen die folgenden Kommandos zur Verfügung:

`r` ⟨*text*⟩	fügt ⟨*text*⟩ rechts hinter den bisher eingegebenen Text,
`l` ⟨*text*⟩	fügt $\langle text \rangle^{-1}$ links vor den bisher eingegebenen Text,
`f` ⟨*pattern*⟩	sucht nach ⟨*pattern*⟩ im bisher eingegebenen Text,
`b` ⟨*pattern*⟩	sucht nach $\langle pattern \rangle^{-1}$ im bisher eingegebenen Text,
`t`	zeigt den bisher eingegebenen Text an,
`a`	zeigt den kompakten Affixbaum des eingegebenen Textes an,
`c`	löscht den bisherigen Text,
`q`	beendet *bigrep*,
`?`	gibt eine kurze Hilfemeldung aus.

Intern wird *online* bei Eingabe von `r` und `l` der kompakte Affixbaum des eingegebenen Textes erzeugt. Das Kommando `f` bzw. `b` bewirkt, daß der so erzeugte Baum gemäß ⟨*pattern*⟩ entlang den Suffix- bzw. Präfixkanten „abgewandert“ wird. Falls auf diese Weise ein Auftreten von ⟨*pattern*⟩ bzw. $\langle pattern \rangle^{-1}$ im Text ermittelt wird, wird `SUCCESS` ausgegeben, falls an irgendeiner Stelle keine Fortsetzung im Baum existiert, `FAIL`.

Da es sich bei der Implementierung von *bigrep* um eine direkte Übertragung des in Kapitel 4 dieser Arbeit beschriebenen Verfahrens zur *online*-

Konstruktion von kompakten Affixbäumen handelt, sind hier weitere Erläuterungen nicht notwendig.

Kapitel 7

Zusammenfassung

Die vorliegende Arbeit beschäftigt sich mit Affixbäumen. Hierbei handelt es sich um eine Weiterentwicklung von Suffixbäumen für bidirektionale Anwendungen, mit Hilfe derer exakte Suche auf flexible Art und Weise möglich ist.

Zunächst wurden grundlegende Datenstrukturen eingeführt. Es wurde ein allgemeiner Dualitätsbegriff für $\mathcal{A}^+$-Bäume definiert und die Datenstruktur des Bi-Baumes entwickelt, auf Basis dessen dann Affixbäume eingeführt wurden. Deren Dualität und der lineare Platzbedarf der kompakten Variante wurden nachgewiesen, ihre maximale Größe bestimmt, verschiedene Repräsentationsformen diskutiert und die für Affixbäume günstigste ausgewählt. Da Affixbäume die Struktur der Suffixbäume vollständig enthalten, konnte festgestellt werden, daß sich sämtliche Anwendungen von Suffixbäumen auf Affixbäume übertragen lassen. Es wurden drei Konstruktionsmöglichkeiten für Affixbäume vorgeschlagen, von denen die bidirektionale *online*-Konstruktion im vierten Kapitel ausführlich besprochen wurde. Dabei stellte sich heraus, daß es sich nicht um eine einfache Verallgemeinerung des entsprechenden Verfahrens von Ukkonen für kompakte Suffixbäume handelt, sondern umfangreiche Erweiterungen notwendig wurden. Bei der anschließenden Komplexitätsanalyse konnte nicht, wie erhofft, die Linearität des Verfahrens nachgewiesen werden, obwohl dies für die Mehrzahl der Schritte möglich war. An drei Stellen traten aber nicht zu lösende Probleme auf. Aus diesem Grunde wurden empirische Untersuchungen angefertigt, die zumindest bei zweien dieser kritischen Punkte ebenfalls auf lineares Verhalten hinwiesen. Der dritte Problemfall konnte auch empirisch nicht eindeutig eingeordnet werden. Eine Verallgemeinerung der ermittelten *worst cases* sowie Laufzeitmessungen an Random-Sequenzen gaben aber Hinweise auf ebenfalls lineares Verhalten.

Es wurden zwei Verfahren zur Konstruktion von kompakten Affixbäumen implementiert, zur frühzeitigen Kontrolle ein ineffizientes Programm in der *lazy*-funktionalen Programmiersprache Haskell und unter Effizienzgesichtspunkten das ausführlich beschriebene *online*-Verfahren in C. Letzteres war, verglichen mit einer entsprechenden Implementierung des Verfahrens für Suffixbäume, in seinem Laufzeitverhalten etwa um den Faktor drei langsamer.

Demgegenüber hat die aufwendigere Datenstruktur aber auch Vorteile: Text und Muster werden bidirektional verwaltet, was neue Verfahren zum flexiblen *pattern matching* ermöglicht, wie sie in der Bioinformatik oder in Bereichen mit ähnlichen Suchproblemen Verwendung finden können. Darüber hinaus könnte man untersuchen, ob es sinnvoll ist, für genetische Anwendungen statt der reversen Vereinigung eine „revers-komplementäre Vereinigung“ zu definieren, so daß sich die beiden komplementären Stränge einer DNA-Doppelhelix in einem Baum repräsentieren und untersuchen lassen. Auch eine Erweiterung des Suchverfahrens für approximatives Matching oder ein Einsatz zur Datenkompression ist denkbar.

Es hat sich gezeigt: Affixbäume leisten alles, was Suffixbäume leisten, und noch viel mehr; oder mit Apostolicos Worten:

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

Literaturverzeichnis

[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.

[GK94b] R. Giegerich and S. Kurtz. Suffix trees in the functional programming paradigm. In *Proceedings of the European Symposium on Programming (ESOP'94)*, number 788 in Lecture Notes in Computer Science, 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 suffixbaumkonstruktionen. Diplomarbeit, Technische Fakultät der Universität Bielefeld, 1995.

[Kur95] S. Kurtz. *Fundamental Algorithms for a Declarative Pattern Matching System*. Dissertation thesis, Technische Fakultät der Universität Bielefeld, 1995.

[McC76] E. M. McCreight. A space-economical suffix tree construction algorithm. *J. 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.

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