Algorithmen auf Strings

0. Motivation

Algorithmen auf Strings spielen u.a. eine große Rolle bei

- der Textverarbeitung,
- der Untersuchung von biologischen Sequenzen,
- der Internetsuche.

Typische Fragestellungen sind u.a.

- exaktes Finden eines Musterstrings in einem Textstring (Stringmatchingproblem)
- Finden von zu einem gegebenen Musterstring "ähnliche" Teilstrings in einem Textstring (approximatives Stringmatchingproblem).
- Komprimierung von Textstrings.
- Finden von übergeordneten Strukturen in Strings

Wir werden uns mit derartigen Fragestellungen beschäftigen.

Literatur:

Maxime Crochemore, Wojciech Rytter: Text Algorithms, Oxford University Press, 1994.

Maxime Crochemore, Wojciech Rytter: Jewels of Stringology, World Scientific, 2003.

Maxime Crochemore, Christophe Hancart, Thierry Lecroq: Algorithms on Strings, Cambridge University Press, 2007.

Bill Smyth: Computing Patterns in Strings, Pearson - Addison Wesley, 2003.

Dan Gusfield: Algorithms on Strings, Trees, and Sequences, Cambridge University Press, 1997.

Michael S. Waterman: Introduction to Computational Biology: Maps, Sequences and Genomes, Chapman & Hall, 1995.

Pavel A. Pevzner: Computational Molecular Biology: An Algorithmic Approach, MIT Press, 2000.

James A. Storer: Data Compression: methods and theory, Computer Science Press, 1988.

Timothy C. Bell, John G. Cleary, Ian H. Witten: Text Compression, Prentice Hall 1990.

David Sankoff, Joseph B. Kruskal (eds.): Time Warps, String Edits, and Macromolecules: The Theory and Practice of Sequence Comparison, Addison-Wesley, 1983.

M. Lothaire: Applied Combinatorics on Words
Cambridge University Press, 2005.

1. Das Stringmatchingproblem

gegeben: Textstring x über einem endlichen Alphabet Σ, $|x| = n$.
Musterstring y über Σ, $|y| = m$.

gesucht: das erste (alle) Vorkommen von y in x.

1.1 Der Algorithmus von Knuth, Morris und Pratt

Das Stringmatchingproblem kann leicht in $O(n \cdot m)$ Zeit gelöst werden. Hierzu testet man alle möglichen $n-m+1$ Positionen in x, ob dort ein Teilstring z mit $z = y$ beginnt.

Aufwand:
pro Test $O(m)$ $\Rightarrow$ insgesamt $O(n \cdot m)$

Frage:
Kann der obige naive Ansatz verbessert werden?

Zur Beantwortung dieser Frage analysieren wir den naiven Ansatz genauer. Wir schieben synchron zwei Zeiger über den Text- und über den Musterstring.

$$
\begin{array}{lllllllllll}
x = a_1 a_2 a_3 \dots & a_z & a_{z+1} & a_{z+2} & \dots & \overset{\downarrow}{a_j} & \dots & a_{z+m} & \dots & a_n \\
 & & b_1 & b_2 & \dots & \underset{\uparrow}{b_i} & \dots & b_m & &
\end{array}
$$

Wir unterscheiden zwei Fälle:

1. Fall: $a_j = b_i$

$\Rightarrow \quad a_{j-i+1} a_{j-i+2} \dots a_j = b_1 b_2 \dots b_i$

Beide Zeiger werden um eine Position weitergeschoben.

2. Fall: $a_j \neq b_i$

$\Rightarrow \quad a_{j-i+1} a_{j-i+2} \dots a_{j-1} = b_1 b_2 \dots b_{i-1}$

und

$a_j \neq b_i$.

Der naive Ansatz schiebt den Musterstring um eine Position nach rechts (d.h., erhöht k um 1) und setzt beide Zeiger um $i-1$ Positionen zurück.

$\Rightarrow$

Das Wissen $a_{j-i+1} \dots a_{j-1} = b_1 \dots b_{i-1}$ wird nicht berücksichtigt und auch vergessen.

Frage:
Mit welchem Gewinn können wir dieses Wissen verwerten?

Idee:
Setze den Textstringzeiger überhaupt nicht zurück. D.h., jedes Textstringsymbol wird nur einmal gelesen

$\Rightarrow$

Der Musterstring muss derart nach rechts verschoben werden, dass

1) links vom Textstringzeiger Muster und Text übereinstimmen und

2) jede geringere Verschiebung des Musterstrings 1) verletzen würde.

Bemerkung:
Den Musterstring nach rechts verschieben bedeutet den Musterstringzeiger nach links zurücksetzen. Demzufolge müsste der Musterstringzeiger so wenig wie möglich zurückgesetzt werden, so dass links vom Muster- und vom Textstringzeiger beide Strings übereinstimmen.

Ziel:
Ermittlung der gemäß obigen Überlegungen neue Position des Musterstringzeigers.

Hierzu definieren wir für $1 \leq r \leq m$

$$H(r) := \max_{e<r} \{ e \mid b_1 b_2 \ldots b_{e-1} \text{ ist Suffix von } b_1 b_2 \cdots b_{r-1} \}.$$

Dies bedeutet, dass $H(r) - 1$ die Länge des maximalen Präfixes von $b_1 b_2 \ldots b_{r-2}$, der auch Suffix von $b_1 b_2 \ldots b_{r-1}$ ist, bezeichnet

Nehmen wir an, dass $H(r)$, $1 \leq r \leq m$, bereits berechnet ist. Dann können wir die naive Methode wie folgt verfeinern:

- Verschiebe den Musterstring derart, dass $b_{H(i)}$ unter a_j steht und verändere den Textstringzeiger nicht. Der Musterstringzeiger zeigt danach auf $b_{H(i)}$.

Wir erhalten dann folgenden Algorithmus:

<u>Algorithmus KMP</u>

<u>Eingabe</u>: Textstring $x = a_1 a_2 \ldots a_n$,
Musterstring $y = b_1 b_2 \ldots b_m$,
Tabelle H.

<u>Ausgabe</u>: k minimal, so dass

$$a_{k+1} a_{k+2} \ldots a_{k+m} = b_1 b_2 \ldots b_m,$$

falls solches k existiert und "Fehlanzeige" sonst.

<u>Methode</u>:

```
i := 1 ; j := 1 ;
while i ≤ m and j ≤ n
   do
      while i > 0 cand b_i ≠ a_j
         do
            i := H(i)   (* Beachte H(1) = 0 *)
         od ;
      i := i+1 ; j := j+1
   od ;
```

```
if i > m
  then
    k := j - (m+1)
  else
    "Fehlanzeige"
fi.
```

Das <u>cand</u> in der inneren <u>while</u> - Schleife ist ein "bedingtes <u>and</u>" und vergleicht b_i mit a_j nur, falls $i > 0$. Jede Iteration der inneren <u>while</u> - Schleife schiebt den Musterstring so lange um $(i - H(i))$ Zeilen nach rechts, bis $i = 0$ oder $b_i = a_j$. Im ersten Fall ist kein nichtleerer Präfix von y Suffix von $a_1 a_2 \dots a_j$. Im zweiten Fall gilt $b_1 b_2 \dots b_i = a_{j-i+1} a_{j-i+2} \dots a_j$. Die äußere <u>while</u> - Schleife erhöht synchron den Textstring- und den Musterstringzeiger.

<u>Lemma 1.1</u>

Der Algorithmus KMP ist korrekt.

<u>Beweis:</u>

Zu zeigen ist, dass der Algorithmus KMP das kleinste k mit

$$a_{k+1} a_{k+2} \dots a_{k+m} = b_1 b_2 \dots b_m$$

berechnet, falls solches existiert und ansonsten "Fehlanzeige" ausgibt.

Falls KMP die äußere *while*-Schleife wegen $i = m+1$ verlässt, dann gilt gemäß Konstruktion

a) $j \leq n+1$ und
b) $a_{k+1} a_{k+2} \ldots a_{k+m} = b_1 b_2 \ldots b_m$.

Dies kann leicht mittels Induktion bewiesen werden.

Wir müssen uns noch davon überzeugen, dass kein $k' \leq n-m$ mit

i) $a_{k'+1} a_{k'+2} \ldots a_{k'+m} = b_1 b_2 \ldots b_m$ und

ii) $k' < k$, falls KMP k ausgibt,

existiert.

Annahme: Es existiert solches k'.

Zur Herbeiführung eines Widerspruches betrachten wir den Ablauf des Algorithmus KMP.

Da die Ausgabe von KMP $k > k'$ oder "Fehlanzeige" ist, gibt es folgende Situation:

$x = a_1 a_2 \ldots a_e a_{e+1} \ldots a_{k'} \ldots a_q \ldots a_j \ldots a_{e+m} \ldots a_n$

$\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \neq$

vor Shift $\quad b_1 b_2 \ldots \qquad b_i \ldots b_m$

nach Shift $\quad b_1 b_2 \ldots \qquad b_m$

- Unmittelbar vor der Abarbeitung des Blockes der inneren *while*-Schleife steht der Anfang des Musterstrings echt links von $a_{k'+2}$ und

danach echt rechts von $a_{k'+1}$.

$\Rightarrow \quad l < k' < j-1$

Wegen

$$a_{k'+1} a_{k'+2} \dots a_j \dots a_{k'+m} = b_1 b_2 \dots b_m$$

und

$$a_{l+1} a_{l+2} \dots a_{l+(i-1)} = b_1 b_2 \dots b_{i-1}$$

gilt:

$b_1 b_2 \dots b_{j-k'-1}$ ist Suffix von $b_1 b_2 \dots b_{i-1}$

Gemäß obiger Situation gilt aber

$$j - k' > H(i),$$

was der Definition von $H(i)$ widerspricht.
Also ist unsere Annahme falsch. ∎

Aufwandsanalyse:

benötigte Zeit:

- Der Textstringzeiger j wird niemals vermindert.
- Eine Erhöhung des Musterstringzeigers um eins impliziert auch die Erhöhung des Textstringzeigers um eins.

$\Rightarrow$

Der Musterstringzeiger i wird maximal n-mal um eins erhöht.

- Eine Verminderung von i in der inneren <u>while</u>-Schleife setzt eine entsprechende Erhöhung von i voraus.

$\Longrightarrow$

Die benötigte Zeit ist $O(n)$.

<u>benötigter Platz</u>:

Der zusätzliche Platzbedarf setzt sich aus dem Platzbedarf für die beiden Zeiger und für die Tabelle H zusammen, ist also $O(m)$.

<u>Berechnung der Tabelle H</u>:

<u>Idee</u>: Induktive Berechnung von H.

1) Gemäß Definition gilt: $H(1) = 0$ und $H(2) = 1$.

2) Seien $H(1), H(2), \ldots, H(i)$, $i \geq 2$ berechnet. Wir werden nun $H(i+1)$ berechnen. Hierzu unterscheiden wir zwei Fälle:

   a) $b_i = b_{H(i)}$

      Dann folgt direkt $H(i+1) = H(i) + 1$

   b) $b_i \neq b_{H(i)}$

      Dann liegt folgende Situation vor:

$b_1 b_2 \dots b_{H(i)+1}$ ist Suffix von $b_1 b_2 \dots b_{i-1}$,
aber $b_i \neq b_{H(i)}$.

D.h., $b_1 b_2 \dots b_{H(i)}$ ist kein Suffix von $b_1 b_2 \dots b_i$.

Wir iterieren nun obige Betrachtungsweise und überprüfen $H(H(i))$, welches gemäß Voraussetzung bereits berechnet ist.

- Falls nun $b_i = b_{H(H(i))}$, dann gilt $H(i+1) = H(H(i)) + 1$. Andernfalls wiederholen wir obige Vorgehensweise.

Dies tun wir so lange, bis wir entweder $H(i+1)$ berechnet haben oder zum Anfang des Musterstrings y gelangt sind.

Somit ergibt sich folgender Algorithmus:

<u>Algorithmus</u> BERECHNUNG VON H

<u>Eingabe</u>: $y = b_1 b_2 \dots b_m$
<u>Ausgabe</u>: Tabelle H
<u>Methode</u>:

```
H(1) := 0 ; H(2) := 1 ;
for i from 2 to m-1
   do
      j := H(i) ;
      while j > 0 cand b_i ≠ b_j
         do
            j := H(j)
         od ;
```

$H(i+1) := j+1$

<u>od.</u>

Analog zur Analyse des Algorithmus KMP kann man zeigen, dass der Algorithmus BERECHNUNG von H $O(m)$ Zeit benötigt.

<u>Übung:</u>

a) Beweisen Sie die Korrektheit des Algorithmus BERECHNUNG von H. Führen Sie für diesen Algorithmus formal eine Aufwandsanalyse durch.

b) Analysieren Sie exakt die Anzahl der von dem Algorithmus KMP durchgeführten Vergleiche.

c) Übungsaufgabe 5.4.b) aus "Algorithmen und Datenstrukturen".

Der Algorithmus KMP berechnet das <u>erste</u> Vorkommen des Musterstrings y im Textstring x. Es stellt sich nun die Frage, wie wir <u>alle</u> Vorkommen des Musterstrings y im Textstring x in $O(n+m)$ Zeit berechnen können. Es verbietet sich, nach jedem Finden von y in x den Musterstring einfach um eine Position nach rechts zu verschieben und KMP erneut zu starten. Wie folgendes Beispiel zeigt, könnte dies zu einer Laufzeit von $\Omega(m \cdot n)$ führen.

Beispiel:

$$x = \underbrace{aaa\ldots a}_{n\text{-mal}} \quad , \quad y = \underbrace{aa\ldots a}_{m\text{-mal}}$$

Für jedes der $n-m+1$ Vorkommen von y in x würden bei obiger Vorgehensweise m Vergleiche durchgeführt werden. Die Gesamtanzahl der durchgeführten Vergleiche wäre somit

$$m(n-m+1) = mn - m^2 + m$$

◆

Ziel:

Erweiterung des Algorithmus KMP, so dass alle Vorkommen von y in x in Zeit $O(n+m)$ berechnet werden.

Idee:

Erweitere die Tabelle H um den Wert

$$H(m+1) := \max_{\ell \leq m} \{ \ell \mid b_1 b_2 \ldots b_{\ell-1} \text{ ist Suffix von } b_1 b_2 \ldots b_m \}.$$

Wir können dann mit Hilfe von $H(m+1)$ den minimalen Rechtsshift berechnen und den Algorithmus KMP im Textstring dort fortsetzen, wo er mit der letzten Ausgabe terminiert ist.

Übung:

a) Modifizieren Sie den Algorithmus KMP, so dass alle Vorkommen von y in x in Zeit $O(n+m)$ berechnet werden.

b) Erweitern Sie den Algorithmus BERECHNUNG VON H, so dass auch $H(m+1)$ berechnet wird.

Literatur:

D.E. Knuth, J.H. Morris, V.R. Pratt, Fast Pattern Matching in Strings, SIAM J. Comput. 6 (1977), 323 - 350.

Insgesamt erhalten wir folgenden Satz:

Satz 1.1

Unter Verwendung des Algorithmus KMP kann das Stringmatchingproblem in $O(n+m)$ Zeit gelöst werden. Der benötigte zusätzliche Speicherplatz ist $O(m)$.

Bemerkung:

Anstatt der Tabelle H, welche die Größe m hat, kann eine Tabelle F der Größe $|\Sigma| \cdot m$ berechnet werden, mittels der erreicht werden kann, dass in der inneren <u>while</u>-Schleife stets der Musterstringzeiger auf die richtige Position gesetzt wird.

(siehe Vorlesung Alg. auf Strings, WS 06/07 S. 14 - 18.)

1.2. Der Algorithmus von Boyer und Moore

Der Algorithmus KMP betrachtet stets den gesamten Textstring.

Frage:
Gibt es einen Algorithmus, der häufig nicht den gesamten Textstring betrachtet?

Idee:

```
      ↓         ↓         ↓
Gibt es für das Stringmatchingproblem
problem
      ↑problem
                ↑problem
                          ↑
```

allgemein:

$$x = a_1 a_2 a_3 \ldots \overset{\downarrow}{a_m}\, a_{m+1} a_{m+2} \ldots a_n$$
$$y = b_1 b_2 b_3 \ldots \underset{\uparrow}{b_m}$$

Plaziere zunächst wie beim Algorithmus KMP den Musterstring $y = b_1 b_2 \ldots b_m$ linksbündig zum Text=string $x = a_1 a_2 \ldots a_n$.

Aber:
Vergleiche y mit dem korrespondierenden Teilstring $a_1 a_2 \ldots a_m$ des Textstrings von rechts nach links.

Beobachtung:
Im Mittel wird häufig $a_m \neq b_m$ erfüllt sein. Dann kann wie folgt verfahren werden:

1. Fall: $a_m \notin y$

Platziere y unter $a_{m+1} a_{m+2} \cdots a_{2m}$.

2. Fall: $a_m \in y$

Platziere y derart, dass a_m von rechts gesehen über dem ersten Symbol b_r mit $b_r = a_m$ steht. D.h.,

i) $a_m = b_r$, wobei b_r unter a_m steht und
ii) $a_m \neq b_j \quad \forall j > r$.

Danach liegt folgende Situation vor:

$$\begin{array}{lllllllll} x = & a_1 a_2 \ldots & a_{m-r+1} & a_{m-r+2} & \cdots & a_m\, a_{m+1} & \cdots & \overset{\downarrow}{a_{2m-r}} & \cdots\, a_n \\ & & b_1 & b_2 & \cdots & b_r\, b_{r+1} & \cdots & \underset{\uparrow}{b_m} & \end{array}$$

Wir vergleichen y mit dem korrespondierenden Teilstring $a_{m-r+1} a_{m-r+2} \cdots a_{2m-r}$ des Text Strings von rechts nach links.

y und der korrespondierende Teilstring von x können in einem Suffix gleich sein und sich erst dann unterscheiden. D.h., allgemeiner liegt folgende Situation vor:

Annahme:
Wir vergleichen y mit dem Teilstring $a_{k+1} a_{k+2} \cdots a_m$ von x und folgendes ist erfüllt:

i) $a_{k+j} \neq b_j$ und
ii) $a_{k+j+1} a_{k+j+2} \cdots a_{k+m} = b_{j+1} b_{j+2} \cdots b_m$.

Idee:

- Shifte $y = b_1 b_2 \ldots b_m$ unter Anwendung von "sinnvollen" Regeln nach rechts.
- Vergleiche den neuen korrespondierenden Teilstring des Textstrings x von rechts nach links mit y.

Frage:
Was sind sinnvolle Regeln bzw. Heuristiken für den Rechtsshift?

Derartige Regeln müssen die Eigenschaft besitzen, dass kein Shift über ein Vorkommen von y in x hinaus erfolgt. Boyer und Moore geben in ihrer Arbeit

R. S. Boyer, J. S. Moore, A fast string searching algorithm, CACM 20 (1977), 762 - 772.

zwei Heuristiken an, die jeweils einen Wert δ_1 bzw. δ_2 für einen sinnvollen Rechtsshift berechnen.

δ_1 basiert auf folgender Idee:

Der Textstringbuchstabe a, der einen Mismatch verursacht hat, muss einem Musterstringbuchstaben, der identisch zu ihm ist, zugewiesen werden. Dies bedeutet, dass wir den Musterstring y so weit nach rechts schieben, bis das in y am

weitesten rechts stehende c unter dem Textbuchstaben c, der den Mismatch verursacht hat, steht. Für $c \in \Sigma$ ist somit $\delta_1(c)$ definiert durch:

$$\delta_1(c) := \begin{cases} m & \text{falls } c \notin y \\ \min\limits_{0 \le i \le m} \{ i \mid b_{m-i} = c \} & \text{sonst.} \end{cases}$$

Bemerkung:

- Im obigen Szenario, d.h., $a_{z+j} \neq b_j$ und $a_{z+j+1} \dots a_{z+m} = b_{j+1} \dots b_m$ definiert $\delta_1(a_{z+j})$ genau dann einen Rechtsshift von y, wenn $m - \delta_1(a_{z+j}) < j$. Ist dies der Fall, dann kann y um $\delta_1(a_{z+j}) - (m-j)$ Positionen nach rechts geshiftet werden.

- δ_1 kann durch eine Tabelle der Größe $|\Sigma|$ beschrieben werden.

- $\delta_1(c)$ gibt das in y links vom Stringende am weitesten rechts stehende c an. Natürlich kann man für jede Position j, $1 \le j \le m$ einen entsprechenden Wert im voraus berechnen. Dies würde allerdings die Vorbereitungszeit erheblich erhöhen und es wäre ein zusätzlicher Speicher der Größe $m \cdot |\Sigma|$ notwendig, wenn man dies für alle j tun würde.

δ_2 ist eine Funktion derjenigen Position in y, in der sich y vom gerade geprüften Teilstring von x

unterscheidet.

Annahme:

$a_{z+j} \neq b_j$ und $a_{z+j+1} \dots a_{z+m} = b_{j+1} \dots b_m$.

Idee:

Berechne eine Tabelle $\delta_{2,1}$ die die notwendige Information über den kleinsten Rechtsshift > 0 enthält, so dass zwischen $a_{z+j+1} \dots a_{z+m}$ und dem korrespondierenden Teilstring y' im Musterstring y kein Mismatch vorkommt und unmittelbar links von y' in y ein Symbol $\neq b_j$ steht.

Bemerkung:

Falls links von y' in y das Symbol b_j stehen würde, dann hätten wir nach dem Rechtsshift unter a_{z+j} wiederum das Symbol b_j stehen und somit wiederum ein Mismatch.

Interpretation:

Äquivalent zur obigen Idee könnten wir y unter y plazieren und für das untere y den geringstmöglichen Rechtsshift > 0 durchführen, so dass zwischen $b_{j+1} \dots b_m$ und dem darunterstehenden Teilstring y' von y kein Mismatch vorkommt und unmittelbar links von y' ein Symbol $\neq b_j$ steht. D.h., wir benötigen das maximale $g < j$, so dass

i) $b_{g+1} b_{g+2} \dots b_{g+(m-j)} = b_{j+1} b_{j+2} \dots b_m$ und
ii) $b_g \neq b_j$.

Falls g nicht existiert, dann benötigen wir den maximalen Präfix von $b_1 b_2 \dots b_{m-j}$, der auch Suffix von $b_{j+1} b_{j+2} \dots b_m$ ist.

Wir berechnen zunächst eine Tabelle δ_2' und dann mit Hilfe dieser die Tabelle δ_2.

$\delta_2'(j)$ enthält den Index desjenigen Symbols, das unter b_m geschoben wird. Falls das untere y ganz über das obere y hinaus geschoben wird, dann erhält $\delta_2'(j)$ den Wert 0. Somit ergibt sich für $\delta_2'(j)$ folgender Wert:

$$\delta_2'(j) := \begin{cases} \bullet \max\{ g + (m-j) \mid 1 \leq g < j,\ b_g \neq b_j \text{ und } b_{g+1} \dots b_{g+(m-j)} = b_{j+1} \dots b_m \} & \text{falls solches } g \text{ existiert.} \\ \bullet \max\{ \ell \mid b_1 \dots b_\ell \text{ ist Suffix von } b_{j+1} \dots b_m \} & \text{falls nicht } g \text{ aber solches } \ell \text{ existiert} \\ \bullet\ 0 & \text{sonst} \end{cases}$$

Falls der erste Fall eintritt, dann wird der Muster=string y derart nach rechts geshiftet, dass b_g unter a_{z+j} und somit auch $b_{g+(m-j)}$ unter a_{z+m} stehen.

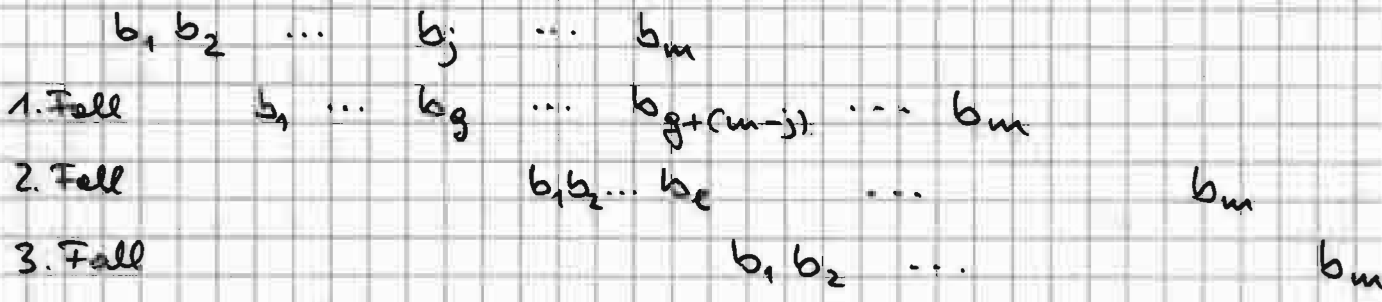
Falls der zweite Fall eintritt, dann wird der Muster=string y derart nach rechts geshiftet, dass b_ℓ unter a_{z+m} steht.

Im dritten Fall wird y ganz über a_{z+m} hinaus geschoben. D.h., b_1 steht dann unter a_{z+m+1}.

Bemerkung:
Im Gegensatz zur Tabelle δ_2' geben die Komponenten der Tabelle δ_1 die Länge eines möglichen Rechtsshiftes an. Besser wäre es, wenn $\delta_2'(j)$ dies auch tun würde. Dann könnte der Algorithmus den maximalen der beiden Rechtsshifte einfach bestimmen und diesen durchführen.

Ziel:
Äquivalente Umformulierung von $\delta_2'(j)$ zu $\delta_2(j)$, so dass $\delta_2(j)$ die Länge des durch $\delta_2'(j)$ spezifizierten Rechtsshiftes angibt.

Situation:



Somit ergibt sich als Länge des korrespondierenden Rechtsshiftes:

1. Fall: $m - (g + (m-j)) = j - g$
2. Fall: $m + \ell$
3. Fall: m

Also erhalten wir

$$\delta_2(j) = m - \delta_2'(j).$$

δ_2 kann durch eine Tabelle der Größe m spezifiziert werden. Da δ_2 nur von y und nicht von x abhängt, kann diese Tabelle im voraus berechnet werden. Gegeben die Tabellen δ_1 und δ_2 sieht der Algorithmus von Boyer und Moore folgendermaßen aus:

Algorithmus BM

Eingabe: Textstring $x = a_1 a_2 \ldots a_n$,
Musterstring $y = b_1 b_2 \ldots b_m$ und
Tabellen δ_1 und δ_2.

Ausgabe: k minimal, so dass

$$a_{k+1} a_{k+2} \ldots a_{k+m} = b_1 b_2 \ldots b_m,$$

falls solches k existiert und "Fehlanzeige" sonst.

Methode:

```
Ausgabe := n;
msende := m; (* markiert Musterstringende in x *)
while msende ≤ n
   do
      k := msende; (* Zeiger in x *)
      j := m;      (* Zeiger in y *)
      while j > 0 and a_k = b_j
         do
            j := j-1; k := k-1
         od;
```

```
if j = 0
  then
    msende := n+1 ;
    Ausgabe := k
  else
    msende := max { k + δ1(a_k) , msende + δ2(j) }
fi
od;
if Ausgabe < n
  then
    return (Ausgabe)
  else
    return ("Fehlanzeige")
fi.
```

Bemerkung:

Aus der Definition der Tabellen δ_1 und δ_2 folgt direkt, dass keiner der durch diese Tabellen definierten Shifts den Musterstring y über ein Vorkommen von y im Textstring x hinausschiebt. Somit müssen wir uns für den Beweis der Korrektheit des Algorithmus BM davon überzeugen, dass BM diese Shifts korrekt durchführt.

Nach dem durch $\delta_1(a_k)$ definierten Shift muss die Position des Musterstrings y die folgende sein:

$$\begin{cases} b_1 \text{ steht unter } a_{k+1} & \text{falls } a_k \notin y \\ b_{m-q} \text{ steht unter } a_k & \text{sonst,} \end{cases}$$

wobei $q = \min_{0 \leq i < m} \{ i \mid b_{m-i} = a_k \}$.

Falls der Algorithmus BM den zu $\delta_1(a_k)$ korrespondierenden Shift durchführt, dann gilt danach

$$msende = k + \delta_1(a_k).$$

D.h., b_m steht unter

$$\begin{cases} a_{k+m} & \text{falls } \delta_1(a_k) = m \\ a_{k+q} & \text{falls } \delta_1(a_k) = \min_{0 \leq i < m} \{ i \mid b_{m-i} = a_k \} =: q \end{cases}$$

Also steht

$$\begin{cases} b_1 \text{ unter } a_{k+1} & \text{falls } a_k \notin y \\ b_{m-q} \text{ unter } a_k & \text{sonst} \end{cases}$$

Demzufolge führt der Algorithmus BM den Shift korrekt durch.

$\delta_2(j)$ gibt die Länge des Rechtsshifts, um die der Musterstring nach rechts geschoben wird, an. Falls BM den zu $\delta_2(j)$ korrespondierenden Shift durchführt, dann gilt danach

$$msende = msende + \delta_2(j).$$

Dies bedeutet, dass der Musterstring exakt um $\delta_2(j)$ nach rechts geschoben wird. Also führt der

Algorithmus BM den Shift korrekt durch.

Bemerkung:
Falls für $q := \delta_1(a_k)$ der Wert $m-q$ größer als j ist, dann wäre der durch $\delta_1(a_k)$ definierte Shift ein Linksshift. Wegen $\delta_2(j) > 0$ gilt dann

$$\text{msende} + \delta_2(j) > k + \delta_1(a_k),$$

so dass der durch $\delta_2(j)$ definierte Rechtsshift vom Algorithmus BM durchgeführt wird. Also führt der Algorithmus BM stets einen Rechtsshift durch.

Bevor wir uns mit der Laufzeitanalyse des Algorithmus BM beschäftigen, überlegen wir uns, wie wir die Tabellen δ_1 und δ_2 effizient berechnen.

Die Berechnung der Tabelle δ_1 ist einfach und wird von folgenden Algorithmus durchgeführt:

Algorithmus δ_1
Eingabe: $y = b_1 b_2 \ldots b_m$
Ausgabe: Tabelle δ_1
Methode:

```
for alle c ∈ Σ
    do
        δ1(c) := m
    od;
```

for j **from** m **to** 1
  do
    if $\delta_1(b_j) = m$
      then
        $\delta_1(b_j) := m-j$
    fi

Übung:
Beweisen Sie die Korrektheit des Algorithmus δ_1 und führen Sie eine Laufzeitanalyse durch.

Die Berechnung der Tabelle δ_2 gestaltet sich wesentlich schwieriger. Für jedes $1 \le j \le m$ ist genau ein Wert $\delta_2(j)$ zu berechnen. Unser Ziel ist es, die Gesamttabelle in $O(m)$ Zeit zu berechnen. Für $1 \le j \le m$ betrachten wir noch einmal $\delta_2(j)$. Drei Fälle sind zu unterscheiden:

1. Fall:

$\exists\, 1 \le g < j$ mit $b_{g+1} \dots b_{g+(m-j)} = b_{j+1} \dots b_m$ und $b_g \neq b_j$.

Dann ist g_{max}, das maximale solche g zu berechnen. Es gilt dann

$$\delta_2(j) = j - g_{max}.$$

2. Fall:

$\neg$ Fall 1 und $\exists\, 1 \le \ell \le m-j$ mit $b_1 \dots b_\ell$ ist Suffix von $b_{j+1} b_{j+2} \dots b_m$.

Dann ist l_{max}, das maximale solche l zu berechnen. Es gilt dann

$$\delta_2(j) = m - l_{max}.$$

<u>3. Fall:</u>

$\neg$ Fall 1 und $\neg$ Fall 2.

Dann gilt

$$\delta_2(j) = m.$$

Demzufolge ist folgende Grobstruktur für den Algorithmus zur Berechnung der Tabelle δ_2 sinnvoll:

(1) Für alle $1 \leq j \leq m$ initialisiere

$$\delta_2(j) := m.$$

(2) Für alle $1 \leq j \leq m$ berechne g_{max}, d.h., das maximale $1 \leq g < j$ mit

$$b_{g+1} \cdots b_{g+(m-j)} = b_{j+1} \cdots b_m \quad \text{und} \quad b_g \neq b_j$$

und setze

$$\delta_2(j) := j - g_{max},$$

falls solches g existiert.

(3) Für alle $1 \leq j \leq m$, für die in (2) kein g_{max} berechnet wurde, für die also

$$\delta_2(j) = m$$

gilt, berechne l_{max}, d.h., das maximale $1 \leq l \leq m-j$ mit

$b_1 \ldots b_l$ ist Suffix von $b_{j+1} \ldots b_m$

und setze

$$\delta_2(j) := m - l_{max},$$

falls solches l existiert.

Bemerkung:
Für die Durchführung von (3) benötigen wir lediglich $O(m)$ Zeit. Wir können hierzu den Algorithmus BERECHNUNG VON H verwenden.

Ziel:
Durchführung von (2) in $O(m)$ Zeit.

Idee:
Reduktion von (2) auf ein Problem, dessen Lösung in $O(m)$ Zeit die Durchführung von (2) in $O(m)$ Zeit impliziert

Hierzu ist es zunächst sinnvoll, die Aufgabenstellung (2) äquivalent umzuformen. Dafür betrachten wir nochmals die Situation, in der ein Wert $\delta_2(j)$ zu berechnen ist. Sei hierzu $g_0 := g_{max}$.

$$x = a_1 a_2 \ldots a_k a_{k+1} \ldots a_{k+j} a_{k+j+1} \ldots a_{k+m} \ldots a_n$$

$$\neq \text{ (unter } a_{k+j}\text{)}$$

$$b_1 \ldots b_{g_0} \underbrace{b_{g_0+1} \ldots b_j}_{j-g_0} \; b_{j+1} \ldots b_s \underbrace{b_{s+1} \ldots b_m}_{j-g_0}$$

wobei $s := g_0 + m - j \iff g_0 = s + j - m$

Es gilt:

$$m - s = m - (g_0 + m - j) = j - g_0$$

Somit erhalten wir aus (2) folgende äquivalente Aufgabenstellung:

(2') Für alle $1 \leq j \leq m$ berechne maximales $s < m$ mit

$$b_{g+1} b_{g+2} \ldots b_s = b_{j+1} b_{j+2} \ldots b_m \text{ und } b_g \neq b_j,$$

wobei $g := s + j - m$ und setze

$$\delta_2(j) := m - s,$$

falls solches s existiert.

Zur Lösung dieser Aufgabenstellung möchten wir einmal den Musterstring y lesen und dabei die benötigten Werte $\delta_2(j)$ berechnen. Da die zu bestimmenden Eigenschaften vom Suffix von y abhängen, macht lesen von links nach rechts keinen Sinn. Beim Lesen von rechts nach links haben wir die Schwierigkeit, dass zum Zeitpunkt, wenn

Bereits zum Zeitpunkt, wenn b_s betrachtet wird, muss die jetzt benötigte Information zumindest vorbereitet werden und spätestens dann, wenn das zu b_j korrespondierende b_g betrachtet wird, berechnet sein. Hierzu ist es nützlich, die Aufgabenstellung (2') nochmals äquivalent umzuformen. Folgende Beobachtung hilft uns hierbei:

- Für jedes $1 \leq s < m$ gibt es höchstens ein g, so dass

$$b_{g+1} b_{g+2} \dots b_s = b_{j+1} b_{j+2} \dots b_m \quad \text{und } b_g \neq b_j ,$$

wobei $j := m + g - s$

Bew. d. Beob.:

Betrachten wir $s < m$ fest. Falls kein $g < s$, das obige Eigenschaften erfüllt, existiert, dann ist nichts mehr zu beweisen.

Betrachten wir $g < s$ minimal, so dass

$b_{g+1} b_{g+2} \dots b_s = b_{j+1} b_{j+2} \dots b_m$ und $b_g \neq b_j$, wobei $j = m + g - s$

$\Longrightarrow$

Für alle $g < h =: g + \ell \leq s$ gilt

$$b_h = b_{g+\ell} = b_{j+\ell} = b_{m+g-s+\ell} = b_{m+h-s}$$

$\Longrightarrow$

Es existiert kein $h > g$, so dass

$b_{h+1} b_{h+2} \dots b_s = b_{j+1} b_{j+2} \dots b_m$ und $b_h \neq b_j$,
wobei $j = m + h - s$.

□

Aufgrund obiger Beobachtung erhalten wir aus (2') folgende äquivalente Aufgabenstellung:

(2") Für alle $1 \leq s < m$ berechne, falls es existiert, das eindeutige $g \in \{1, 2, \dots, s\}$ mit

$b_{g+1} b_{g+2} \dots b_s = b_{j+1} b_{j+2} \dots b_m$ und $b_g \neq b_j$,
wobei $j := m + g - s$.

Falls g existiert und $\delta_2(j) > m - s$, dann führe

$$\delta_2(j) := m - s$$

aus.

Für $1 \leq s < m$ bezeichne N_s die Länge des längsten Suffixes des Teilstrings $b_1 b_2 \dots b_s$, der auch Suffix von $b_1 b_2 \dots b_m$ ist. Dann gilt für das zu s gemäß der Aufgabenstellung (2") korrespondierende g:

$$g = s - N_s$$

und daher

$$\begin{aligned} j &= m + g - s \\ &= m + s - N_s - s \\ &= m - N_s. \end{aligned}$$

Also können wir anstatt g auch N_s berechnen. Nach der Berechnung von N_s können wir, falls $\delta_2(m - N_s) > m - s$

$$\delta_2(m - N_s) := m - s$$

ausführen.

Wir arbeiten lieber mit aufsteigenden als mit absteigenden Indizes. Daher definieren wir den String y^R durch

$$y^R := b_m b_{m-1} \dots b_1$$

Nach umindizieren erhalten wir den String

$$z := c_1 c_2 \dots c_m.$$

D.h.,

$$c_q = b_{m-q+1} \quad \text{bzw.} \quad b_q = c_{m-q+1}$$

für $1 \leq q \leq m$.

Für $1 < i \leq m$ bezeichne Z_i die Länge des längsten Präfixes von $c_i c_{i+1} \dots c_m$, der auch Präfix von $c_1 c_2 \dots c_m$ ist.

Konstruktion ⟹

$$N_s = Z_{m-s+1}$$

und

$$Z_r = N_{m-r+1}.$$

Demzufolge genügt zur Lösung der Aufgabenstellung (2") die Entwicklung eines Algorithmus für folgendes maximale Präfixproblem.

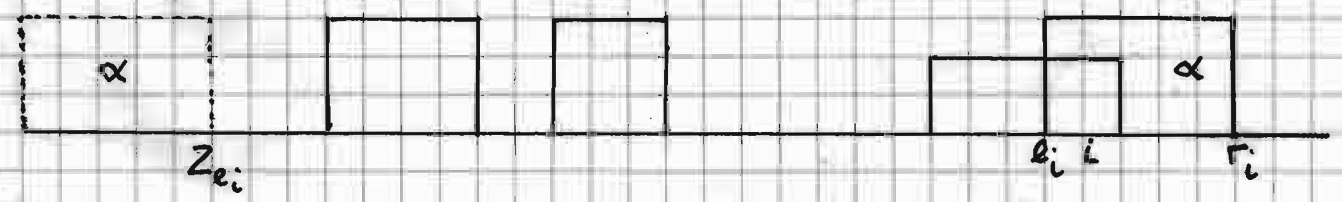
<u>maximale Präfixproblem:</u>

<u>gegeben</u>: String $z = c_1 c_2 \dots c_m$

<u>gesucht</u>: Z_i für $1 < i \leq m$.

Unser Ziel ist nun die Entwicklung eines effizienten Algorithmus für das maximale Präfixproblem. Hierfür ist folgendes Konzept nützlich:

Betrachte die im folgenden Bild gezeichneten Boxen:



Jede Box startet in einer Position j mit $Z_j > 0$. Die Länge der Box, die in j startet, ist Z_j. Dies bedeutet, dass jede Box einen Teilstring maximaler Länge, der gleich einem Präfix von z ist und

nicht in Position 1 beginnt, repräsentiert.

D.h., wir definieren die <u>Z-Box bzgl. j</u> als das Intervall $[j, j+Z_j-1]$.

Für jedes $i > 1$ bezeichnet r_i den am weitesten rechts liegenden Endpunkt aller Z-Boxen, die in i oder links von i beginnen. D.h.,

$$r_i := \max_{1<j\leq i} \{ j+Z_j-1 \mid Z_j > 0 \}.$$

Für jedes $i > 1$ bezeichnet l_i den am weitesten links liegenden Endpunkt aller Z-Boxen, deren rechter Endpunkt r_i ist. D.h.,

$$l_i := \min_{1<j\leq i} \{ j \mid j+Z_j-1 = r_i \}.$$

<u>Idee:</u>

Beginnend in $i=2$ berechnen wir sukzessive die Werte Z_i, r_i und l_i. Dabei werden wir bereits erworbene Kenntnisse derart verwenden, dass insgesamt nur $O(n)$ Vergleiche benötigt werden.

Da wir in jeder i-ten Iteration lediglich die Werte r_{i-1} und l_{i-1} benötigen, verwenden wir zwei Variablen r bzw. l, in denen die zuletzt berechneten r- und l-Werte gespeichert werden.

In der nachfolgenden Beschreibung des Algorithmus enthalten die Klammern (* *) Kommentare.

Algorithmus MAXPREF

Eingabe: String $z = c_1 c_2 \dots c_m$

Ausgabe: Werte Z_i für $2 \le i \le m$.

Methode:

(1) $i := 2$;

(2) Vergleiche $c_2 c_3 \dots c_m$ und $c_1 c_2 \dots c_m$ von links nach rechts bis ein Mismatch gefunden ist.

(3) $Z_2 :=$ Länge des gemeinsamen Präfixes von $c_2 c_3 \dots c_m$ und z.

(4) **if** $Z_2 > 0$
  then
    $r := 2 + Z_2 - 1$;   (* dann gilt $r = r_2$ *)
    $l := 2$ (* dann gilt $l = l_2$ *)
  else
    $r := 0$; $l := 0$
  fi;

(5) **for** $i := 3$ **to** m
  do
    if $i > r$
      then
        • Vergleiche $c_i c_{i+1} \dots c_m$ und z von links nach rechts bis ein Mismatch gefunden ist.
        • $Z_i :=$ Länge des gemeinsamen Präfixes von $c_i c_{i+1} \dots c_m$ und z.
        • **if** $Z_i > 0$
          then

$r := i + Z_i - 1;$

$\ell := i$

fi

else (* $i \leq r$

D.h., die Position i ist in der Z-Box $[\ell, r]$ und c_i im Teilstring $\alpha := c_\ell c_{\ell+1} \ldots c_r$. Der Teilstring α ist auch Präfix von $z = c_1 \ldots c_m$.

$\Longrightarrow$

$c_{i-\ell+1} = c_i$

und auch

$$\underbrace{c_{i-\ell+1}}_{i'} c_{i-\ell+2} \ldots c_{Z_\ell} = \underbrace{c_i c_{i+1} \ldots c_r}_{\beta}$$

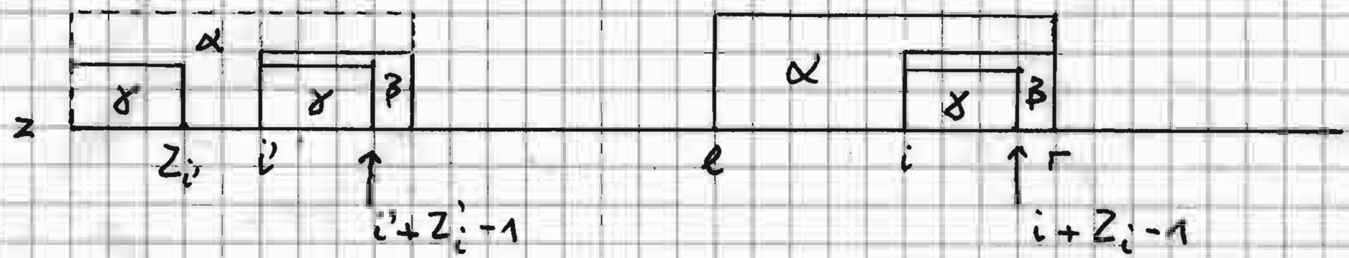
Da der Teilstring beginnend in Position i' ein Präfix der Länge $Z_{i'}$ von z hat, folgt somit:

Der Teilstring beginnend in Position i ist ein Präfix von z der Länge $\geq \min\{Z_{i'}, |\beta|\}$.

*)

if $Z_{i'} < |\beta|$
then
$Z_i := Z_{i'}$

(*



*)

else (* $Z_{i'} \geq |\beta|$

$\Longrightarrow$

$c_i c_{i+1} \dots c_r$ ist Präfix von z und $Z_i \geq |\beta| = r - i + 1$.

*)

- Vergleiche $c_{r+1} c_{r+2} \dots c_m$ mit $c_{|\beta|+1} c_{|\beta|+2} \dots c_m$ von links nach rechts bis ein Mismatch gefunden ist. Sei $q \geq r+1$ diejenige Position, die den Mismatch enthält.
- $Z_i := q - i$;
  if $q - 1 > r$
  then
  $r := q-1$, $l := i$
  fi

od. ~~fi~~ **fi**

Übung:

a) Beweisen Sie die Korrektheit des Algorithmus MAXPREF.

b) Arbeiten Sie den Algorithmus MAXPREF aus.

Lemma 1.2

Der Algorithmus MAXPREF hat die Laufzeit $O(m)$, wobei m die Länge des Eingabestrings ist.

Beweis:

Jede Iteration benötigt konstante Zeit zuzüglich der Anzahl der Vergleiche, die während der Iteration durchgeführt werden.

$\Longrightarrow$

$$\text{Gesamtzeit} = O(m) + O(GAV),$$

wobei GAV für die Gesamtanzahl der durchgeführten Vergleiche steht.

Ziel: Abschätzung von GAV

Jeder Vergleich resultiert in einen Match oder in einen Mismatch. Bezeichne

A_1 die Gesamtanzahl der Matches und
A_2 die Gesamtanzahl der Mismatches.

$\Longrightarrow$

$$GAV = A_1 + A_2.$$

Jede Iteration, die Vergleiche durchführt, endet, sobald ein Mismatch auftritt.

$\Longrightarrow$

$$A_2 \leq m.$$

Für $i \geq 3$ gilt: $r_i \geq r_{i-1}$.

Sei k eine Iteration, in der q Vergleiche mit einem Match enden. Dann gilt:

$$r_k = r_{k-1} + q.$$

Ferner gilt für alle i : $r_i \leq m$.

Insgesamt folgt somit

$$A_1 \leq m.$$

Also gilt

$$GAV \leq 2m.$$

■

Wir haben bewiesen, dass die Vorbereitungszeit für den Algorithmus BM linear in der Länge des Musterstrings ist. Offen ist noch die Laufzeitanalyse für den Algorithmus BM. Wie immer können zwei Kostenmaße zugrunde gelegt werden.

a) erwartete Laufzeit
b) worst case Laufzeit

a) <u>erwartete Laufzeit</u>

Diese kann ermittelt werden durch

- <u>Experimente</u>
  Dabei stellt sich die Frage: Was ist ein geeignetes Experiment? D.h., wie sehen die Problemstellungen in der Praxis aus?

  Literatur hierzu: Siehe Buch von Gusfield, S. 16.

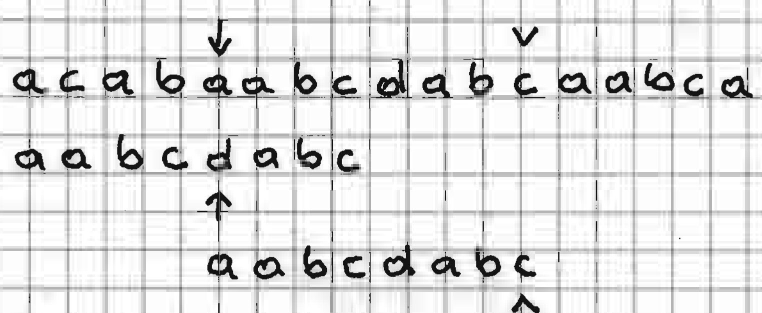
- <u>theoretische Analyse</u>
  Hierbei stellt sich die Frage: Was ist ein realistisches Modell?

b) <u>Laufzeit im worst case</u>

Es ist nahezu trivial zu zeigen, dass der KMP-Algorithmus höchstens $2n$ Vergleiche benötigt. Wesentlich komplizierter ist es, den BM-Algorithmus zu analysieren. Dies beruht auf folgender Eigenschaft des Algorithmus:

- Im Gegensatz zum KMP-Algorithmus vergisst der BM-Algorithmus bereits berechnete Information.

<u>Beispiel</u>:



In ihrer Arbeit beweisen Knuth, Morris und Pratt mit einer komplizierten Analyse eine $7n$ obere Schranke für die Anzahl der Vergleiche, die der BM-Algorithmus durchführt. Guibas und Odlyzko haben diese obere Schranke auf $4n$ verbessert. Ihre Analyse ist auch kompliziert.

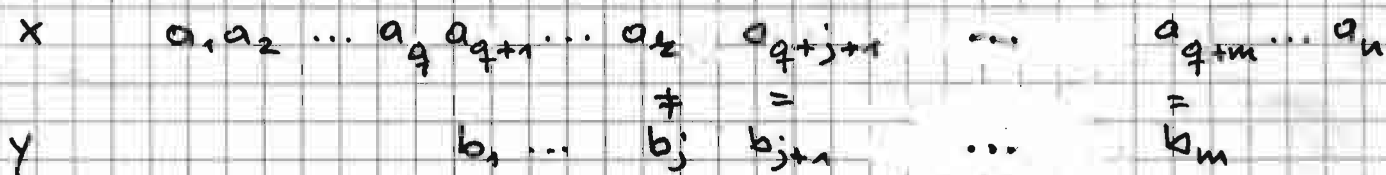
L. J. Guibas, A. M. Odlyzko: A new proof of the linearity of the Boyer-Moore string searching algorithm, SIAM J. Computing 9 (1980), 672-682.

Cole hat 1994 einen wesentlich einfacheren $4n$-Beweis und einen schwierigen $3n$-Beweis publiziert. In derselben Arbeit hat er Eingaben für den BM-Algorithmus konstruiert und gezeigt, dass diese $3(n-m)$ Vergleiche benötigen.

R. Cole: Tight bounds on the complexity of the Boyer-Moore string matching algorithm, SIAM J. Computing 23 (1994), 1075-1091.

Wir werden uns Cole's $4n$-Analyse erarbeiten. Wir betrachten hierzu den BM-Algorithmus und unterteilen diesen in sogenannte Vergleiche/Shift-Phasen, die wir mit 1 beginnend durchnummerieren.

Vergleiche/Shift-Phase:



Das rechte Ende des Musterstrings y steht unter einer Position im Eingabestring x. Der Musterstring y wird von rechts nach links mit dem darüber stehenden Teilstring von x verglichen, bis entweder y in x gefunden ist oder ein Mismatch geschieht. Falls ein Mismatch vorliegt, dann wird y gemäß den Regeln δ_1 und δ_2 nach rechts geshiftet.

<u>Ziel</u>:
Abschätzung der Anzahl der in einer Vergleichs/Shift-Phase vorgenommenen Vergleiche.

Wir ordnen jeden Vergleich demjenigen Textsymbol zu, das an dem Vergleich teilnimmt und unterscheiden zwei Typen von Vergleichen:

$$\begin{cases} \underline{\text{unkritisch}} & \text{falls das Textsymbol zum ersten Mal an einem Vergleich teilnimmt} \\ \underline{\text{kritisch}} & \text{sonst.} \end{cases}$$

Da jedes Textsymbol höchstens einmal an einem unkritischen Vergleich teilnimmt, ist die Anzahl aller unkritischen Vergleiche durch die Anzahl n der Textsymbolen beschränkt.

Falls für die Anzahl aller kritischen Vergleiche der Beweis gelingt, dass diese höchstens das Dreifache der Anzahl aller unkritischen Vergleiche ist, dann haben wir eine $4n$ obere Schranke für die Gesamtanzahl aller Vergleiche bewiesen.

Idee:

Zu jeder Vergleiche/Shift-Phase korrespondieren Länge des durchgeführten Shiftes viele unkritische Vergleiche. Diese möchten wir kritischen Vergleichen derart zuordnen, dass wir heraus eine obere Schranke für die Anzahl dieser kritischen Vergleiche herleiten können.

Frage:

Welche kritische Vergleiche sind sinnvolle Kandidaten für diese Zuordnung?

Zur Beantwortung dieser Frage betrachten wir die Situationen, unmittelbar vor und unmittelbar nach dem in der Vergleiche/Shift-Phase durchgeführten Shift.

Vergleiche/Shift-Phase:

$$x = a_1 a_2 \ldots a_q a_{q+1} \ldots a_k\ a_{q+j+1} \ldots a_{q+m} \ldots a_{q+m+p} \ldots a_n$$

(*)

$$\begin{array}{ccccccc} & \neq & = & & = & & \\ b_1 \ldots & b_j & b_{j+1} & \ldots & b_m & & \\ & ? & = & \ldots & = & & \\ b_1 \ldots & b_{g_0} & b_{g_0+1} & \ldots & b_{g_0+(m-j)} & \ldots & b_m \end{array}$$

naheliegende Antworten:

1) Für jedes der Textsymbole $a_{q+j+1}, a_{q+j+2}, \ldots, a_{q+m}$, die in der aktuellen Vergleiche/Shift-Phase den Match mit $b_{j+1} b_{j+2} \ldots b_m$ bilden, der nächste Vergleich, an dem dieses Textsymbol teilnimmt.

2) Die kritischen Vergleiche, die in der aktuellen Vergleichs/Shift-Phase vor dem Shift stattgefunden haben.

Bemerkung:
Beide Zuordnungsarten sehen plausibel aus und sind es auch wert, weiter verfolgt zu werden. Die zweite Zuordnungsart wird zum Ziel führen. Wir werden in der Vorlesung nur die zweite Zuordnungsart analysieren.

Der Algorithmus BM führt den Shift mittels der Operation

$$msende := \max\{k + \delta_1(a_k),\ msende + \delta_2(j)\}$$

durch. Jenachdem, ob dieser gemäß der δ_1- oder der δ_2-Heuristik durchgeführt wird, unterscheiden wir zwei Fälle.

<u>1. Fall</u>: Durchführung gemäß δ_2-Heuristik.

D.h., $$msende := msende + \delta_2(j)$$

Es gilt:

$$\delta_2(j) = m - \delta_2'(j),$$

wobei

$$\delta_2'(j) = \begin{cases} \max\{ g + (m-j) \mid 1 \leq g < j,\ b_g \neq b_j \\ \quad \text{und } b_{g+1} b_{g+2} \cdots b_{g+(m-j)} = b_{j+1} \cdots b_m \}, & \text{falls solches } g \text{ existiert,} \\ \max\{ \ell \mid b_1 b_2 \dots b_\ell \text{ ist Suffix von } b_{j+1} \cdots b_m \} & \text{falls nicht } g \text{ aber solches } \ell \text{ existiert} \\ 0 & \text{sonst} \end{cases}$$

Somit können drei Unterfälle eintreten:

1.1

$$\delta_2'(j) = \max\{ g + (m-j) \mid 1 \leq g < j,\ b_g \neq b_j \text{ und } b_{g+1} \cdots b_{g+(m-j)} = b_{j+1} \cdots b_m \}.$$

$\Longrightarrow$

$$\delta_2(j) = j - g_0 ,$$

wobei $g_0 + (m-j) = \delta_2'(j)$.

Somit liegt in der Tat die oben beschriebene Situation (*) vor, wobei $p := j - g_0$ die Länge des durchgeführten Shiftes ist.

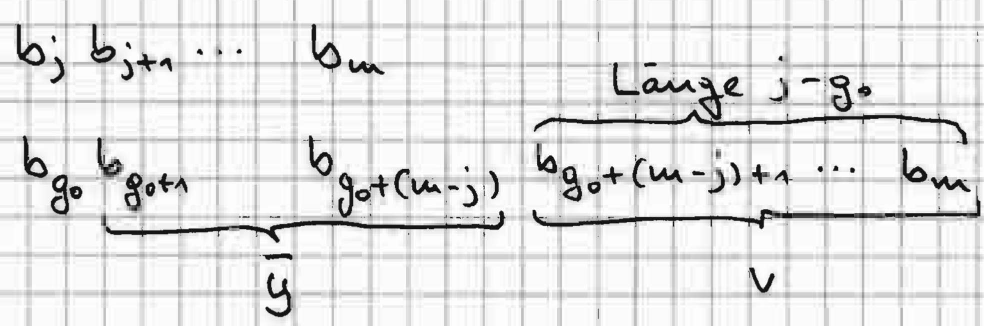
Sei

$$\bar{y} := b_{j+1} b_{j+2} \cdots b_m = b_{g_0+1} b_{g_0+2} \cdots b_{g_0+(m-j)}.$$

Falls $j - g_0 \geq m - j$, dann können alle in der aktuellen Vergleichs/Shift-Phase durchgeführten kritische Vergleiche paarweise verschiedene durch den Shift der Länge $j - g_0$ bedingte unkritische Vergleiche zugeordnet werden. Somit ist der vorliegende Fall unproblematisch.

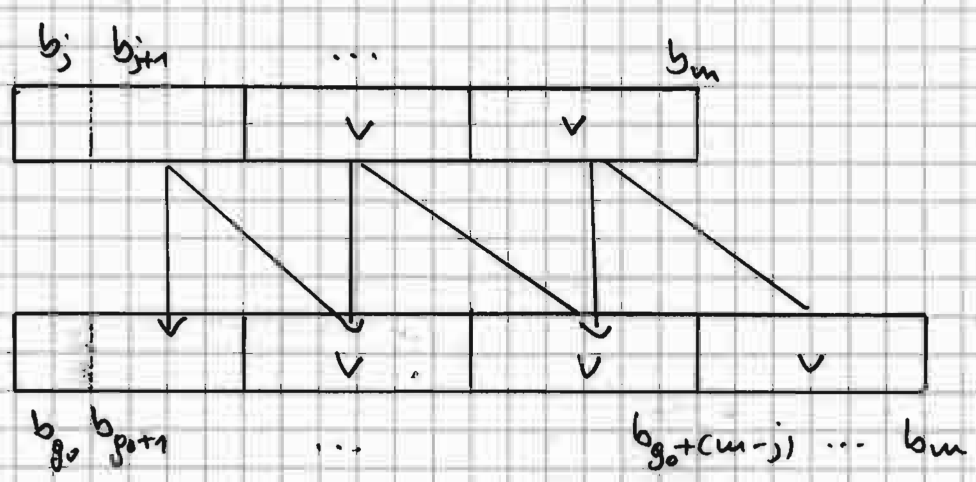
<u>Annahme</u>: $j - g_0 < m - j$.

Dann liegt folgende Situation vor:



Wegen $j - g_0 < m - j$ ist $v := b_{g_0+(m-j)+1} \cdots b_m$ ein echter Suffix von $\bar{y} = b_{j+1} b_{j+2} \cdots b_m$.

Gemäß obiger Situation ist v auch Suffix von $b_{g_0+1} b_{g_0+2} \cdots b_{g_0+(m-j)}$. Diese Betrachtungsweise setzt sich fort, so dass sich folgende Situation ergibt:



Also können wir $\bar{y}$ wie folgt schreiben:

$$\bar{y} = u v^k ,$$

wobei $k \geq 1$ und $u \in \Sigma^*$ ein Suffix von v ist. Ferner ist $b_j u$ auch ein Suffix von v. Falls $u = \varepsilon$, dann gilt $k > 1$.

Wegen $|uv^k| > |v|$ können wir nicht auf triviale Art und Weise die zu uv^k korrespondierenden kritische Vergleiche paarweise unterschiedlichen zu v korrespondierenden unkritische Vergleiche zuordnen.

$\Longrightarrow$

Eine genauere Analyse der obigen Situation ist notwendig.

<u>Bezeichnungen:</u>

Eine <u>Periode</u> eines Strings s ist die Länge $|v|$ eines Suffixes v von s, so dass $s = uv^k$ für ein $k \geq 1$ und einen möglicherweise leeren Suffix u von v. <u>Periode(s)</u> bezeichnet stets die kleinste Periode von s.

<u>Lemma 1.3</u>

Seien v und w zwei nichtleere Strings, so dass $vw = wv$. Dann existieren ein String u und natürliche Zahlen i und j, für die gilt:

$$v = u^i \quad \text{und} \quad w = u^j .$$

Beweis: (mittels Induktion über $|v| + |w|$)

$|v| + |w| = 2$:

Dann gilt: $vw = wv \Rightarrow v = w$.

Setze $u := v$, $i := 1$ und $j := 1$

$\Rightarrow$

$v = u^i$ und $w = u^j$.

Annahme:

Die Behauptung des Lemmas ist erfüllt für alle $v, w \in \Sigma^+$ mit $|v| + |w| \leq \ell$, wobei $\ell \geq 2$.

$\ell \leadsto \ell+1$:

Betrachte $v, w \in \Sigma^+$ mit $|v| + |w| = \ell + 1$ beliebig, aber fest.

1. Fall: $|v| = |w|$

Dann gilt wiederum $vw = wv \Rightarrow v = w$

Setze $u := v$, $i := 1$ und $j := 1$

$\Rightarrow$

$v = u^i$ und $w = u^j$.

2. Fall: $|v| \neq |w|$

O.B.d.A. sei $|v| < |w|$. Dann gilt

$(vw = wv \wedge |v| < |w|) \Rightarrow$

v ist ein echter Präfix von w. D.h.,

$\exists z \in \Sigma^+$ mit $w = vz$.

Substitution von vz für w in der Gleichung $vw = wv$ ergibt dann

$$vvz = vzv.$$

Streichen der linken Kopie von v auf beiden Seiten der Gleichung ergibt

$$vz = zv.$$

Wegen

$$|v| + |z| = |w| < |v| + |w| = \ell + 1$$

gilt

$$|v| + |z| \leq \ell.$$

Induktionsannahme $\Rightarrow$

$$\exists\, u \in \Sigma^+,\ i, j \in \mathbb{N} \text{ mit } v = u^i \text{ und } z = u^j$$

Also gilt

$$w = vz = u^i u^j = u^{(i+j)},$$

womit das Lemma bewiesen ist. ■

Übung:
Seien $s = vw = wv$ für $v, w \in \Sigma^+$. Zeigen Sie, dass s Periodizität t mit $t \leq \min\{|v|, |w|\}$ besitzt.

Nun können wir mit der Analyse des Algorithmus BM fortfahren. Folgende Situation liegt vor:

$$\begin{array}{l} a_1 a_2 \ldots a_q a_{q+1} \ldots a_{q+p+1} \ldots a_k\, a_{q+j+1} \ldots a_{q+m} \ldots a_{q+m+p} \ldots a_n \\ \qquad\qquad\qquad\qquad\qquad\qquad \neq \quad = \quad \ldots \quad = \\ \qquad\qquad b_1 \quad \ldots \qquad\qquad b_j\, b_{j+1} \ldots b_m \\ \qquad\qquad\qquad\qquad\qquad\qquad \overset{?}{=} \quad = \quad \ldots \quad = \\ \qquad\qquad\qquad\qquad\qquad\qquad b_{g_0} b_{g_0+1} \ldots \underbrace{b_{g_0+(m-j)} \ldots b_m}_{v} \end{array}$$

wobei $p := j - g_0$ und $j - g_0 < m - j$.

<u>Idee</u>:
Zeige, dass in der aktuellen Vergleichs/Shift-Phase viele der durchgeführten Vergleiche unkritisch sind, so dass die Anzahl der kritischen Vergleiche die Länge $j - g_0$ des durchgeführten Shiftes nicht allzu sehr überschreiten kann.

<u>Durchführung</u>:

Sei z der kürzeste Teilstring von v, so dass $v = z^\ell$ für ein $\ell \in \mathbb{N}$. Wegen $v = v^1$ existiert der Teilstring z. z heißt <u>Generator</u> von v.

Oben haben wir uns überlegt, dass

$$\bar{y} = u v^k ,$$

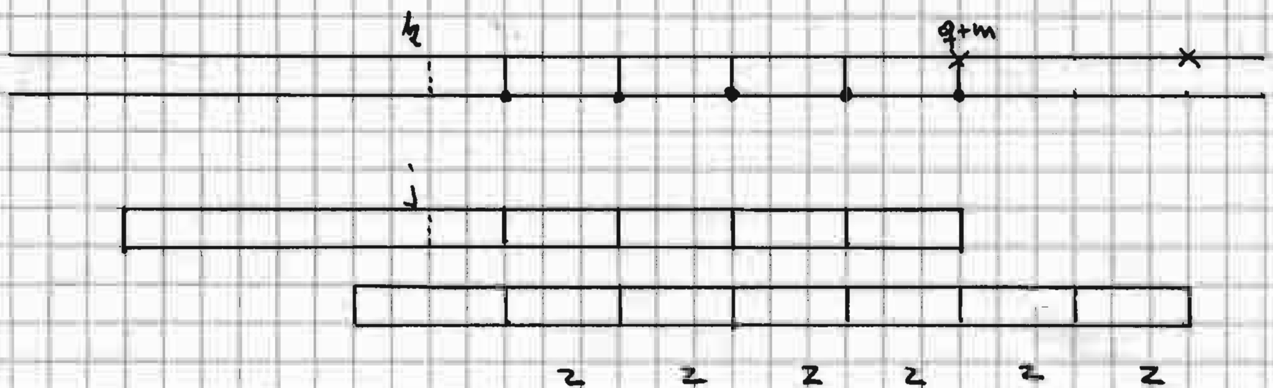
wobei $k \geq 1$ und $u \in \Sigma^*$ ein Suffix von v ist. Ferner ist auch $b_j u$ ein Suffix von v. Also gilt folgendes

Lemma:

Lemma 1.4

$|z|$ ist Periode von $\bar{y}$ und auch von $b_j\bar{y}$.

Lemma 1.4 in oben skizzierte Situation mit einbezogen ergibt folgendes Bild:



• Generatorendposition in x

× Mstringendposition in x.

Zur Charakterisierung derjenigen Vergleiche, die kritisch sein können, müssen wir obige Situation sorgfältig analysieren und einige Eigenschaften herausarbeiten. Zunächst werden wir zeigen, dass in vorangegangenen Phasen die Mstringendposition niemals gleich einer aktuellen Generatorendposition war.

Lemma 1.5

In keiner vorangegangenen Vergleiche/Shift-Phase war die Endposition des Musterstrings gleich einer aktuellen Generatorendposition.

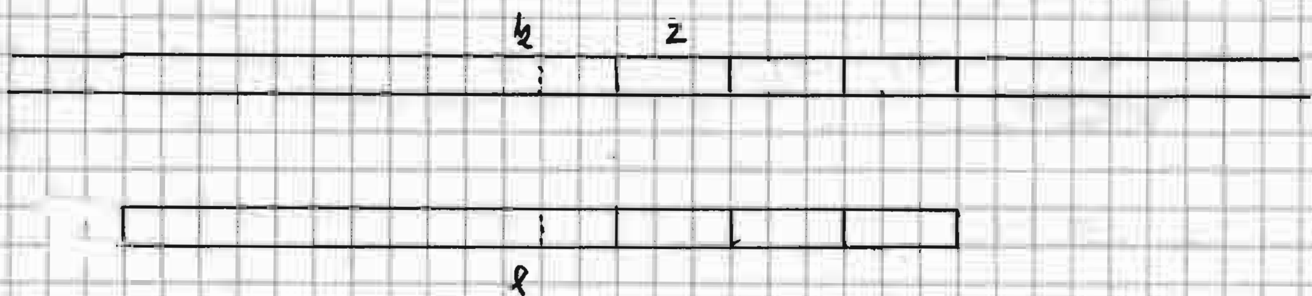
Beweis:

Solange der Musterstring nicht gefunden ist, schließt jede Vergleichs/Shift-Phase mit einem Rechtsshift ab. Da die Generatorendposition q+m die aktuelle Endposition des Musterstrings ist, muss in jeder Vorgängerphase das Musterstringende echt links von q+m liegen.

Annahme:

Die Endposition des Musterstrings liegt in einer Vorgängerphase in einer aktuellen Generatorendposition echt links von q+m.

Dann liegt folgende Situation vor:



Lemma 1.4 $\Rightarrow$ $b_l = b_j$

Also muss die Vergleichsphase der betrachteten Vorgängerphase mit einem Mismatch zwischen b_l und a_k enden.

Wir betrachten nun den Shift, der gemäß der δ_2-Heuristik durchgeführt würde.

Da nach dem Shift gemäß der δ_2-Heuristik unter a_2 ein $b_1 \neq b_2$ steht, muss das Ende des Shifts echt innerhalb eines der Generatoren liegen.

Nach dem Shift liegt somit folgende Situation vor:

Gemäß unserer δ_2-Heuristik stimmen der unter $\bar{x}$ stehende Teilstring des Musterstrings und $\bar{x}$ überein.

$\Rightarrow$

$$z = z_1 z_2 = z_2 z_1 \quad \text{für zwei nichtleere Strings } z_1 \text{ und } z_2$$

Lemma 1.3 $\Rightarrow$

z ist nicht der kürzeste Teilstring von v, so dass $v = z^l$ für ein $l \in \mathbb{N}$.

Dies ist ein Widerspruch zur Wahl von z.

■

Bemerkung:
Für den Beweis des obigen Lemmas ist es unerheblich, ob der Algorithmus BM in der Tat den Shift gemäß der δ_2-Heuristik durchführt oder nicht.

Mit Hilfe von Lemma 1.5 lässt sich leicht das folgende Lemma beweisen.

Lemma 1.6

In jeder vorangegangenen Vergleichs/Shift-Phase matcht der Musterstring y in $a_{q+j+1} a_{q+j+2} \cdots a_{q+m}$ höchstens $|z|-1$ Symbole.

Beweis:

Annahme:

y matcht $a_{q+j+1} \cdots a_{q+m}$ in einem Teilstring der Länge $\geq |z|$.

Lemma 1.5 $\Rightarrow$

Die Endposition von y ist nicht gleich einer Generatorendposition

Also liegt folgende Situation vor:

Demzufolge wäre $z = z_1 z_2 = z_2 z_1$ für zwei nicht=leere Strings z_1 und z_2, was wegen der Wahl von z nicht sein kann.

■

Folgendes Lemma grenzt die möglichen Endpositionen des Musterstrings y während vorangegangenen Vergleichs/Shift-Phasen innerhalb von $a_{q+j+1} \cdots a_{q+m}$ ein.

Lemma 1.7

Falls in einer vorangegangenen Vergleichs/Shift-Phase die Endposition des Musterstrings y innerhalb von $a_{q+j+1} a_{q+j+2} \dots a_{q+m}$ liegt, dann befindet sich diese innerhalb der ersten $|z|-1$ Positionen oder innerhalb der letzten $|z|$ Positionen in $a_{q+j+1} \dots a_{q+m}$.

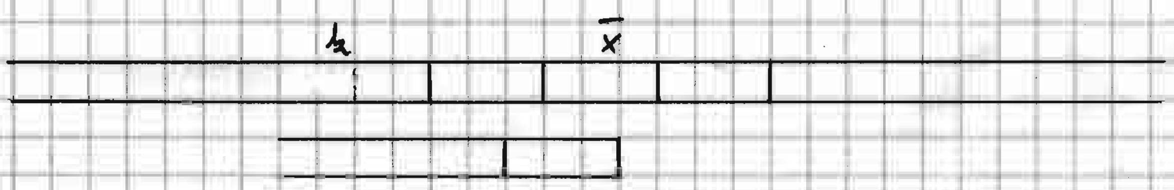
Beweis:

Annahme:

Die Endposition des Musterstrings befindet sich in einer vorangegangenen Phase rechts von den ersten $|z|-1$ und links von den letzten $|z|$ Positionen innerhalb von $a_{q+j+1} a_{q+j+2} \dots a_{q+m}$.

Lemma 1.5 $\Rightarrow$

Die Endposition des Musterstrings y befindet sich echt innerhalb eines Generators $\bar{x}$. D.h., es liegt folgende Situation vor:



Lemma 1.6 $\Rightarrow$

Die Länge des Matches, mit dem die Vergleichsphase endet ist $\leq |z|-1$.

$\Rightarrow$

Der Mismatch erfolgt innerhalb von $a_{q+j+1} \dots a_{q+m}$.

Gemäß der δ_2-Heuristik wird der Musterstring nach rechts verschoben und zwar so wenig wie möglich, so dass

- der aktuelle Match nach dem Shift wieder gematcht wird und
- unter dem Symbol im Textstring, das den Mismatch bedingt hat, nach dem Shift ein anderes Musterstringsymbol steht.

Betrachten wir den Shift, der die Endposition des Musterstrings unter die Endposition von $\bar{x}$ plazieren würde.

Da $\bar{x} = z$ Generator von $\bar{y}$ ist, würde innerhalb von $a_{q+j+1} a_{q+j+2} \ldots a_{q+m}$ kein Mismatch erfolgen.

$\Rightarrow$

- der aktuelle Match wird nach dem Shift wieder gematcht und
- unter dem Symbol in x, das den Mismatch bedingt hat, steht nun ein anderes Musterstringsymbol.

Somit erfüllt der betrachtete Shift bis eventuell auf die Minimalität die δ_2-Heuristik.

Lemma 1.5 $\Rightarrow$

Dieser Shift ist nicht möglich.

⇒

Gemäß der δ_2-Heuristik muss ein Shift erfolgen, der das Musterstringende echt innerhalb $\bar{x}$ belässt.

Übung:
Zeigen Sie, dass auch gemäß der δ_1-Heuristik nur ein Shift erfolgen kann, der das Musterstringende echt innerhalb $\bar{x}$ belässt.

Da dies für jede Phase, die mit Musterstringende echt innerhalb $\bar{x}$ startet, gilt, kann das Musterstringende niemals über die Endposition von $\bar{x}$ hinausgeschoben werden.

Dies ist unmöglich, da in der aktuellen Vergleiche/Shift-Phase das Musterstringende echt rechts von $\bar{x}$ steht.

Also ist unsere Annahme falsch und das Lemma somit bewiesen.

■

Nun können wir die Zuordnung der während der aktuellen Vergleiche/Shift-Phase erfolgten kritischen Vergleiche an die zum Shift korrespondierenden unkritischen Vergleiche durchführen.

Gemäß unserer Konstruktion ist die Länge $|v|$ des erfolgten Shiftes mindestens $|z|$, wobei z der Generator von v ist.

Lemma 1.7 $\Rightarrow$

Falls in einer vorangegangenen Vergleichs/Shift-Phase die Endposition des Musterstrings y innerhalb des aktuellen Matches $a_{q+j+1} a_{q+j+2} \dots a_{q+m}$ liegt, dann befindet sich diese innerhalb der ersten $|z|-1$ oder innerhalb der letzten $|z|$ Positionen des Matches.

Lemma 1.6 $\Rightarrow$

In jeder vorangegangenen Vergleichs/Shift-Phase matcht der Musterstring y in $a_{q+j+1} \dots a_{q+m}$ höchstens $|z|-1$ Symbole.

$\Rightarrow$

Ein Musterstring, dessen Endposition innerhalb der letzten $|z|$ Positionen sich befindet, kann nur Symbole innerhalb der letzten $2|z|-1$ Positionen matchen.

$\Rightarrow$

Höchstens $3|z|-2$ der $m-j$ Vergleiche können kritisch sein.

$\Rightarrow$

Wir können die kritischen Vergleiche der aktuellen Vergleichs/Shift-Phase derart den $\geq |z|$ zum Shift korrespondierenden unkritischen Vergleiche zuordnen, dass

1) jeder unkritische Vergleich höchsten drei kritische Vergleiche zugeordnet bekommt und
2) mindestens ein unkritischer Vergleich weniger als

drei kritische Vergleiche zugeordnet bekommt.

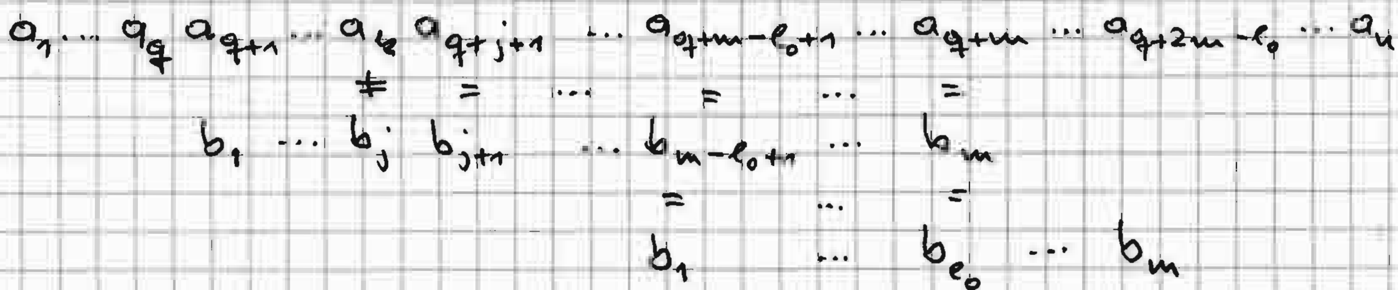
<u>1.2</u>

$\delta_2'(j) = \max\{ l \mid b_1 b_2 \dots b_l$ ist Suffix von $b_{j+1} \dots b_m \}$.

$\Rightarrow$

$\delta_2(j) = m - l_0$, wobei $l_0 := \delta_2'(j)$.

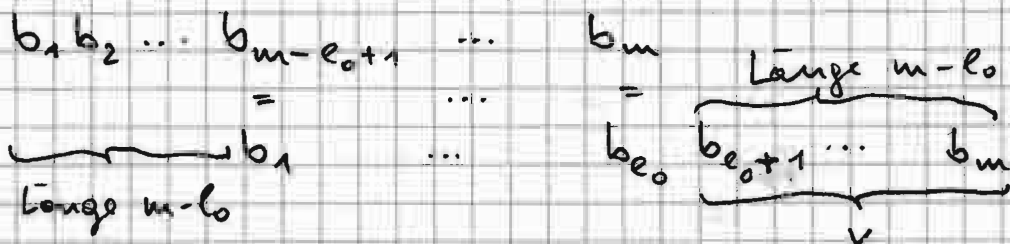
Somit liegt folgende Situation vor:



Falls $m - l_0 \geq m - j$, dann können alle in der aktuellen Vergleiche/Shift-Phase durchgeführten kritische Vergleiche paarweise verschiedene durch den Shift der Länge $m - l_0$ bedingte unkritische Vergleiche zugeordnet werden. Somit ist der vorliegende Fall unproblematisch.

<u>Annahme:</u> $m - l_0 < m - j$

Dann liegt folgende Situation vor:



Analog zum Fall 1.1 überlegen wir uns, dass $m - l_0$ eine Periode des Musterstrings y ist. Wir betrachten nun den kürzesten Teilstring z von v mit $v = z^l$ für ein $l \in \mathbb{N}$.

Wir können nun zu den Lemmata 1.5 - 1.7 analoge Lemmata formulieren, wobei y die Rolle von $\bar{y}$ spielt. Diese können dann genauso wie die Lemmata 1.5 - 1.7 bewiesen werden.

Übung:
Führen Sie die Analyse des Falles 1.2 durch.

1.3

$$\delta_2'(j) = 0$$
$$\Rightarrow$$
$$\delta_2(j) = m.$$

D.h., der Musterstring wird um m Positionen nach rechts geschoben. Wegen $m > m-j$ ist der vorliegende Fall unproblematisch.

2. Fall Durchführung gemäß δ_1-Heuristik.

Die Analyse des 1. Falles hängt nicht davon ab, der Shift in der Tat gemäß der δ_2-Heuristik erfolgt. Falls der Shift nicht gemäß der δ_2-Heuristik durchgeführt wird, dann stehen mehr unkritische Vergleiche für die Zuordnung zur Verfügung, als dies nach Durchführung des Shiftes

gemäß der δ_2-Heuristik der Fall wäre. Somit kann auch bei der Durchführung des Shiftes gemäß der δ_1-Heuristik die Zuordnung durchgeführt werden.

In allen diskutierten Fällen können wir den zum Mismatch korrespondierenden Vergleich einem zum Shift korrespondierenden unkritischen Vergleich, dem maximal zwei kritische Vergleiche zugeordnet wurden, zuordnen. Insgesamt haben wir folgenden Satz bewiesen:

Satz 1.2

Der Algorithmus BM führt maximal $4n$ Vergleiche durch, wobei $n := |x|$ die Länge des Textstrings ist.

Ziel:

Berechnung aller Vorkommen des Musterstrings y im Textstring x in $O(n+m)$ Zeit.

Idee:

Nach jedem Vorkommen von y in x führe den minimalen Rechtsshift, so dass übereinander stehender Suffix gemäß der alten Position und Präfix gemäß der neuen Position von y übereinstimmen, durch. Setze dann den Algorithmus BM fort.

Wie folgendes Beispiel zeigt, könnte dies zu einer Laufzeit von $O(m \cdot n)$ führen:

Beispiel:

$$x = \underbrace{aa \ldots a}_{n\text{-mal}} \quad , \quad y = \underbrace{aa \ldots a}_{m\text{-mal}}$$

Der Algorithmus BM würde bei obiger Vorgehensweise für jedes der $n-m+1$ Vorkommen von y in x m Vergleiche durchführen. Die Gesamtanzahl der durchgeführten Vergleiche wäre dann $m \cdot n - m^2 + m$.

Bemerkung:
Im obigen Beispiel hat der Musterstring y maximal mögliche Periodizität m. Eine kleine Periode des Musterstrings haben alle Beispiele, bei denen obige Vorgehensweise eine hohe Laufzeit nach sich zieht.

In

Zvi Galil, On improving the Worst Case Running Time of the Boyer-Moore String Matching Algorithm, CACM 22 (1979), 505-508.

hatte Galil folgende Idee:

- Immer wenn der Musterstring y im Textstring x gefunden ist, vermeide mit Hilfe der Kenntnis von Periode(y) redundanten Testen.

Für die Durchführung obiger Idee, insbesondere für den Beweis der Korrektheit der Methode,

benötigen wir eine weitere Eigenschaft der Perioden eines Strings.

Erinnerung:
Für die Analyse des Algorithmus BM war folgende Definition der Periode eines Strings s sinnvoll:

Eine Periode eines Strings s ist die Länge $|v|$ eines Suffixes v von s, so dass $s = uv^k$ für ein $k \geq 1$ und einen möglicherweise leeren Suffix u von v.

Folgende äquivalente Definition ist für die Durchführung der obigen Idee geeigneter:

Eine Periode eines Strings s ist die Länge $|v|$ eines Präfixes v von s, so dass $s = v^k u$ für ein $k \geq 1$ und einen möglicherweise leeren Präfix u von v.

Übung:
Beweisen Sie, dass beide Definitionen einer Periode eines Strings s äquivalent sind.

Folgendes Lemma ist das in der Literatur wohlbekannte so genannte Periodizitätslemma.

Lemma 1.8
Seien p und q zwei Perioden eines Strings s. Falls $p + q \leq |s|$, dann ist $ggT(p, q)$ auch eine Periode des Strings s.

Beweis:

Falls $p = q$, dann ist wegen $ggT(p,q) = p$ die Behauptung trivialerweise erfüllt. Sei $p \neq q$. O.B.d.A. können wir $p > q$ annehmen.

Beh. (*):

Sei s ein String. Für alle Perioden a und b von s gilt: $(a+b \leq |s| \wedge a > b) \Rightarrow a-b$ ist Periode von s.

Zunächst werden wir uns davon überzeugen, dass die Beh. (*) das Lemma impliziert und danach die Beh. (*) beweisen.

Für die Reduktion des Lemmas auf die Beh. (*) betrachten wir die Berechnung des $ggT(p,q)$ durch den Euklid'schen Algorithmus. Diese sieht wie folgt aus:

$$
\begin{aligned}
p &= c_1 q + r_1 & 0 < r_1 < q \\
q &= c_2 r_1 + r_2 & 0 < r_2 < r_1 \\
r_1 &= c_3 r_2 + r_3 & 0 < r_3 < r_2 \\
&\vdots \\
r_{j-2} &= c_j r_{j-1} + r_j & 0 < r_j < r_{j-1} \\
r_{j-1} &= c_{j+1} r_j
\end{aligned}
$$

Es gilt dann: $ggT(p,q) = r_j$.

c_1-maliges Anwenden von (*), wobei a zu Beginn den Wert p und danach den Wert der aktuellen Differenz und b stets den Wert q erhalten,

ergibt:

r_1 ist Periode von s.

c_2-maliges Anwenden von (*), wobei q die Rolle von p und r_1 die Rolle von q spielt, ergibt:

r_2 ist Periode von s

⋮

r_{j-1} ist Periode von s

c_j-maliges Anwenden von (*) ergibt

r_j ist Periode von s

Wegen $ggT(p,q) = r_j$ folgt somit das Lemma.

Also fehlt nur noch der Beweis der Beh. (*).

<u>Bew. d. Beh. (*)</u>:

Sei $s := a_1 a_2 \dots a_n$. Zu zeigen ist, dass $a-b$ eine Periode von s ist. Hierzu genügt es zu zeigen, dass für $1 \le i \le n$ folgendes erfüllt ist:

$$a_i = \begin{cases} a_{i+(a-b)} & \text{falls } i+(a-b) \le n \\ a_{i-(a-b)} & \text{falls } i-(a-b) \geq 1 \end{cases}$$

Betrachte $i \in \{1, 2, \dots, n\}$ beliebig, aber fest.

$a + b \le n \implies$

i) $i + a \leq n \lor i - b \geq 1$ und

ii) $i - a \geq 1 \lor i + b \leq n$.

Wir können nun $i + (a-b)$ bzw. $i - (a-b)$ auf zwei Arten und Weisen umschreiben:

$$i + (a-b) = (i+a) - b \quad \text{oder}$$
$$i + (a-b) = (i-b) + a$$

bzw.

$$i - (a-b) = i - a + b \quad \text{oder}$$
$$i - (a-b) = i + b - a.$$

Wir diskutieren zunächst den Fall $i + (a-b) \leq n$. Falls $i+a \leq n$, dann folgt wegen a und b sind Perioden von s:

$$a_i = a_{i+a} = a_{(i+a)-b} = a_{i+(a-b)}.$$

Falls $i - b \geq 1$, dann folgt wegen a und b sind Perioden von s

$$a_i = a_{i-b} = a_{(i-b)+a} = a_{i+(a-b)}.$$

Den Fall $i - (a-b) \geq 1$ beweist man analog. ∎

Übung:
Führen Sie im Beweis der Beh. a) den Fall $i - (a-b) \geq 1$ durch.

Folgendes Lemma besagt, dass eine große Anzahl von Vorkommen des Musterstrings y im Textstring x nur möglich ist, wenn Periode(y) klein ist.

Lemma 1.9

Seien $n := |x|$, $m := |y|$, $m < n$ und r die Anzahl der Vorkommen des Musterstrings y im Textstring x.

Dann gilt: $r \geq \frac{2n}{m} \implies$ Periode$(y) \leq \frac{m}{2}$.

Beweis:

Zunächst überlegen wir uns, dass es zwei Vorkommen von y in x, die sich mindestens in $\frac{m}{2}$ Positionen überlappen, gibt.

Annahme:

Keine Vorkommen von y in x überlappen sich in $\geq \frac{m}{2}$ Positionen.

$\implies$

Nach einem Vorkommen von y in x kann das nächste Vorkommen von y frühestens nach $\frac{m}{2}+1$ Positionen beginnen.

$\implies$

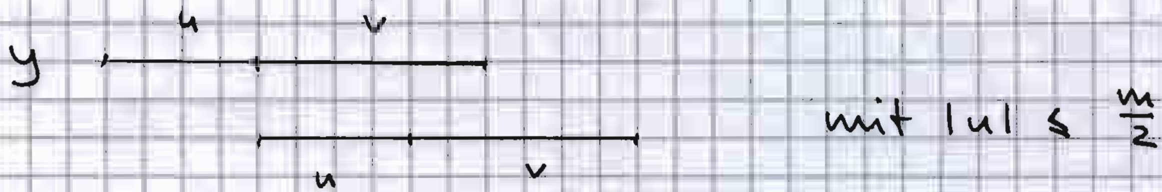
$$|x| \geq \left(\frac{2n}{m} - 1\right)\left(\frac{m}{2}+1\right) + m$$

$$= n + \frac{2n}{m} - \frac{m}{2} - 1 + m$$

$$> n$$

Widerspruch!

Somit liegt folgende Situation vor:



mit $|u| \leq \frac{m}{2}$

Also ist $|u|$ eine Periode von y.

$\Rightarrow$

$\text{Periode}(y) \leq \frac{m}{2}$

■

Übung

Beweisen Sie: $\text{Periode}(y) > \frac{m}{2} \Rightarrow$ Die Laufzeit des Algorithmus BM ist $O(n+m)$.

Für die folgenden Überlegungen nehmen wir stets $\text{Periode}(y) \leq \frac{m}{2}$ an. u bezeichnet stets denjenigen Präfix von y mit $|u| = \text{Periode}(y)$.

Übung:

Entwickeln Sie einen effizienten Algorithmus, der für einen gegebenen String s $\text{Periode}(s)$ berechnet.

Ziel:

Mittels einer einfachen Modifikation des Algorithmus BM dafür zu sorgen, dass auch bei Musterstrings y mit $\text{Periode}(y) \leq \frac{m}{2}$ stets der modifizierte Algorithmus lineare Laufzeit hat.

Betrachten wir hierzu die Vorkommen des Muster-strings y im Textstring x. Seien p und q zwei Positionen in x, in denen jeweils ein Vorkommen von y beginnt. Wir sagen, die Vorkommen p und q sind <u>nahe beieinander</u>, falls $|p-q| \leq \frac{m}{2}$.

Wir definieren die <u>Nachbarschaftsrelation</u> als reflexive, transitive Hülle von "nahe beieinander".

Ein <u>Block</u> ist eine maximale Klasse von Nachbarn. Folgendes Lemma ist leicht zu beweisen:

<u>Lemma 1.10</u>

Sei B ein Block und sei p die maximale Position in B. Dann gilt:

i) Ein neuer Block B' kann nicht vor Position $p + \frac{m}{2}$ beginnen.

ii) Die Anzahl der Blöcke ist $\leq \frac{2n}{m}$.

<u>Beweis:</u>

Übung

■

Folgendes Lemma besagt, dass aufeinanderfolgende Positionen innerhalb eines Blockes stets den Abstand Periode (y) haben.

<u>Lemma 1.11</u>

Sei B ein Block und seien $p_1, p_2, \dots, p_r$, $r > 1$

die Anfangspositionen von y in B in streng monoton wachsender Ordnung. Dann gilt für $1 < i \leq r$ $\quad p_i - p_{i-1} = \text{Periode}(y)$.

<u>Beweis:</u>

Sei
$$k := \text{Periode}(y).$$

Betrachten wir $1 < i \leq r$ beliebig, aber fest.
Sei
$$k' := p_i - p_{i-1}.$$

Da p_i und p_{i-1} nahe beieinander liegen, gilt:
$$k' \leq \frac{m}{2}$$

Genauso, wie im Beweis von Lemma 1.9 folgt nun
$$k' \text{ ist eine Periode von } y.$$

Wegen $k' + k \leq m$ folgt aus Lemma 1.8
$$ggT(k', k) \text{ ist eine Periode von } y.$$

Wegen $\text{Periode}(y) = k$ gilt: $ggT(k', k) = k$

$\Rightarrow$
$$k' = l \cdot k \quad \text{für ein } l \in \mathbb{N}.$$

Falls $l = 1$, dann ist nichts mehr zu beweisen.

<u>Annahme:</u> $l > 1$

Aus

- Periode(y) $= k$ und
- in p_{i-1} und $p_{i-1} + \ell \cdot k$, $\ell > 1$ starten Vorkommen von y in x

folgt direkt:

In Position $p_{i-1} + k < p_i$ startet ein Vorkommen von y in x.

Dies ist ein Widerspruch zur Konstruktion des Blockes B. Somit war unsere Annahme falsch und das Lemma ist bewiesen.

■

Idee:
Der Algorithmus BM findet jedes Vorkommen des Musterstrings y im Textstring x zu einem Zeitpunkt. Geschickter wäre es, stattdessen stets einen gesamten Block zu einem Zeitpunkt zu finden.

Den resultierenden modifizierten Algorithmus nennen wir BM'. Der Algorithmus BM' arbeitet wie folgt:

- Solange kein Musterstring y in x gefunden ist, arbeitet BM' genauso, wie der Algorithmus BM.

- Sobald ein Vorkommen von y in einer Position q gefunden wird, startet BM' genauso wie BM die Suche nach dem nächsten Vorkommen in Position $q+k$, wobei k = Periode(y). Im Gegensatz zu BM untersucht BM' jedoch nur die letzten k Symbole des Musterstrings y, ob diese mit den korrespondierenden Symbole im Textstring übereinstimmen.

- Ist dies der Fall, dann ist das nächste Vorkommen von y in x gefunden und BM' sucht nun genauso, wie oben beschrieben, in Position $q+2k$ weiter.

- Ist dies nicht der Fall, dann arbeitet BM' genauso wie BM weiter, bis das nächste Vorkommen von y in x gefunden ist.

Übung:

a) Arbeiten Sie den Algorithmus BM' aus.
b) Beweisen Sie die Korrektheit des Algorithmus BM'.
c) Zeigen Sie, dass die Laufzeit des Algorithmus BM' linear ist.

1.3 Suffixbäume

Der KMP-Algorithmus und auch der BM-Algorithmus haben den Musterstring vorbereitet um dann in Linearzeit alle Vorkommen des Musterstrings im Textstring zu finden. Falls viele verschiedene Musterstrings im Textstring gesucht werden, ist es mitunter sinnvoller, den Textstring vorzubereiten, so dass die einzelnen Musterstrings effizient im Textstring gefunden werden.

Ziel:

Entwicklung einer Datenstruktur für den Textstring, die die effiziente Lösung von Stringmatchingproblemen ermöglicht. Insbesondere sollen alle Vorkommen eines Musterstrings y mit $|y| = m$ in Zeit $O(m + A(y))$ bestimmt werden können, wobei $A(y)$ die Anzahl der Vorkommen von y im Textstring bezeichnet.

Literatur:

Dan Gusfield, Algorithms on Strings, Trees, and Sequences, Cambridge University Press, 1997, 89 - 107.

Esko Ukkonen, Online-Construction of Suffix Trees, Algorithmica 14 (1995), 249 - 260

Zunächst benötigen wir einige Definitionen:

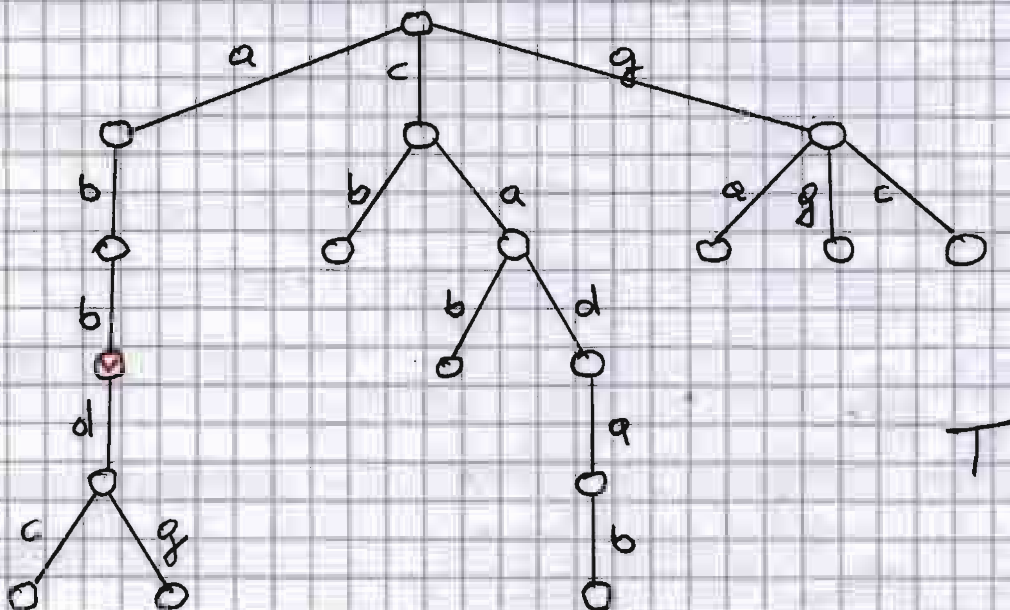
Ein <u>Trie</u> bezüglich eines Alphabets Σ, $|\Sigma| = k$ ist ein Baum $T = (V, E)$, für den gilt:

i) Jeder innere Knoten hat Ausgangsgrad $\leq k$.
ii) Die ausgehenden Kanten eines inneren Knotens sind mit paarweise verschiedenen Elementen aus Σ beschriftet

Ein Pfad P von der Wurzel des Tries T zu einem Blatt v korrespondiert zu demjenigen String, den wir durch Konkatenation der Kantenbeschriftungen auf P in der Reihenfolge, in der diese auf P vorkommen, erhalten.

Somit repräsentiert ein Trie T eine Menge $S(T)$ von Strings. Diese korrespondiert zur Gesamtheit aller Pfade von der Wurzel von T zu einem Blatt.

<u>Beispiel</u>:



$S(T) = \{ abbdc, abbdg, cb, cab, cadab, ga, gg, gc \}$

Obige Menge $S(T)$ ist <u>präfixfrei</u>, d.h., kein Element von $S(T)$ ist Präfix eines anderen Elementes von $S(T)$. Falls wir zu $S(T)$ noch das Element abb hinzunehmen, dann ist die resultierende Menge nicht mehr präfixfrei, da der String abb auch Präfix des Strings abbdc ist.

Folgende Erweiterungen der Definition eines Tries erlauben auch die Repräsentation von Stringmengen S, die nicht präfixfrei sind:

1) Innere Knoten, in denen solch ein Präfix endet, werden markiert, so dass die Pfade von der Wurzel des Tries zu einem markierten Knoten oder zu einem Blatt eineindeutig zu den Elementen von S korrespondieren.

2) Die eingehenden Kanten von Blättern erhalten keine Beschriftung mit einem Symbol aus dem Alphabet Σ und der Trie ist so organisiert, dass auch bei nicht präfixfreien Mengen S die Pfade von der Wurzel zu einem Blatt eineindeutig zu den Elementen von S korrespondieren.

Wir werden nachfolgend stets die erste Erweiterung der Definition eines Tries verwenden.

Beispiel (Fortführung)

Wenn wir im obigen Trie T den Knoten v markieren, dann erhalten wir einen Trie T', der die Menge

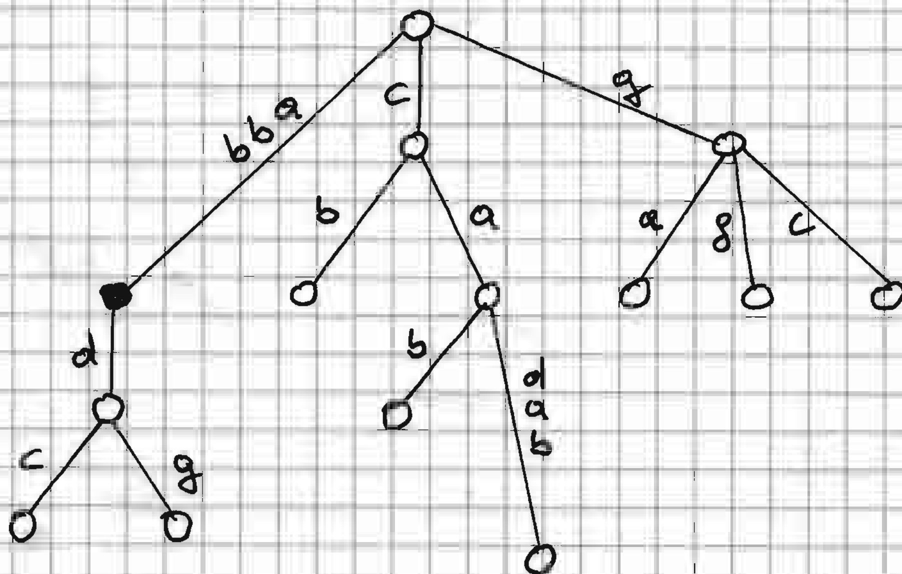
$$S(T') = \{abb, abbcd, abbcg, cb, cab, cadab, ga, gg, gc\}$$

repräsentiert

◊

Aus einem Trie T erhalten wir den korrespondierenden kompakten Trie T_c, indem wir alle Pfade P, auf denen nur nichtmarkierte Knoten mit Ausgangsgrad eins liegen, zu einer Kante zusammenfassen und diese mit dem zu P korrespondierenden String beschriften.

Beispiel (Fortführung):



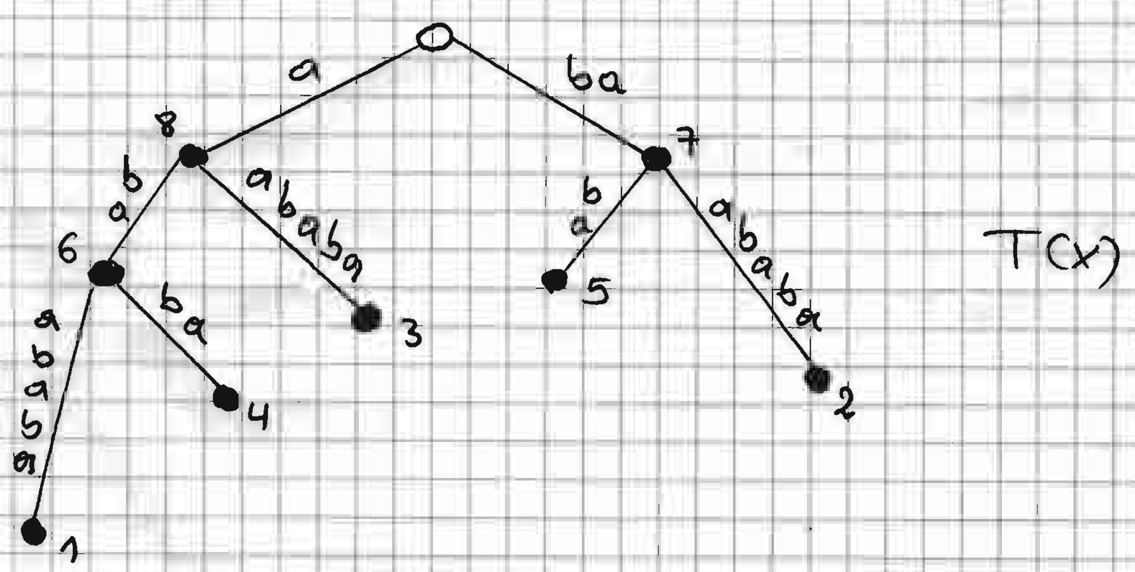
Der zu T' korrespondierende kompakte Trie T_c'.

◆

Ein <u>Suffixbaum</u> $T(x)$ für einen String $x = a_1 a_2 \ldots a_n \in \Sigma^n$ ist ein kompakter Trie bezüglich des Alphabets Σ, der n markierte Knoten, die mit den Zahlen $1, 2, \ldots, n$ numeriert sind, enthält. Der Pfad von der Wurzel zum Knoten mit Nummer i korrespondiert zu demjenigen Suffix von x, der in der i-ten Position von x startet. D.h., zu $a_i a_{i+1} \ldots a_n$.

<u>Beispiel</u>:

Sei $x = abaababa$



Bevor wir einen effizienten Algorithmus zur effizienten Berechnung eines Suffixbaumes $T(x)$ für einen gegebenen Textstring x entwickeln, zeigen wir, wie mit Hilfe des Suffixbaumes das Stringmatchingproblem effizient gelöst werden kann.

gegeben: Textstring $x := a_1 a_2 \dots a_n$, ein Suffix-baum $T(x)$ und ein Musterstring $y := b_1 b_2 \dots b_m$.

gesucht: Alle Positionen in x, in denen der Musterstring y beginnt.

Lösungsmethode:

- Starte in der Wurzel von $T(x)$ und konstruiere den längstmöglichen Pfad P in $T(x)$ mit Kantenmarkierungen $b_1 b_2 \dots b_j$, $0 \leq j \leq m$.
- Zwei Fälle können eintreten:

1. Fall: $j < m$

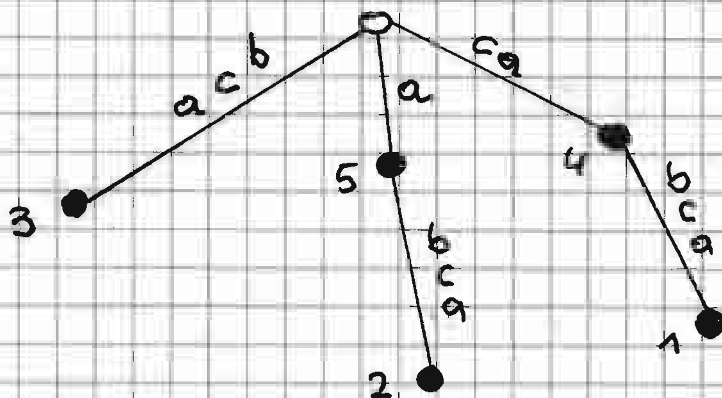
Dann ist der Musterstring y kein Teilstring von x.

2. Fall: $j = m$

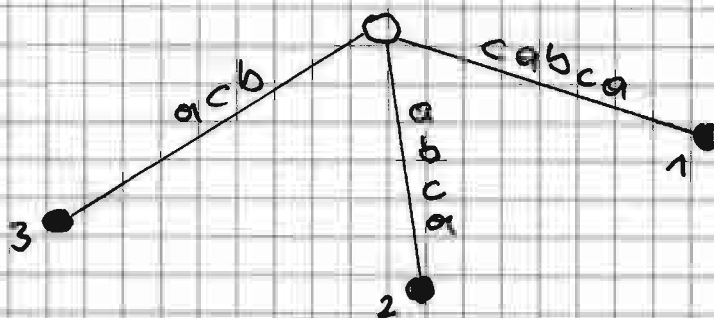
Sei v derjenige Knoten in $T(x)$, in dem der konstruierte Pfad P endet. Dann bezeichnen die Nummern der markierten Knoten im Teilbaum mit Wurzel v exakt diejenigen Positionen in x, in denen der Musterstring y beginnt.

Übung:
Entwickeln Sie einen formalen Beweis für die Korrektheit des obigen Verfahrens.

Betrachten wir für den String $x = cabca$ folgenden Suffixbaum $T(x)$



$T(x)$ enthält Knoten, die den Ausgangsgrad 1 haben. Da $T(x)$ kompakt ist, sind alle Knoten mit Ausgangsgrad 1 markiert. Falls wir in $T(x)$ alle Pfade, auf denen nur Knoten mit Ausgangsgrad 1 liegen, zu einer Kante zusammenfassen und diese mit denjenigen String, der zu dem Pfad korrespondiert, beschriften, dann erhalten wir folgenden Baum $T'(x)$:



Allgemein definieren wir: Sei $T(z)$ ein Suffixbaum für einen String z. Aus $T(z)$ erhalten wir einen <u>impliziten Suffixbaum</u> $T'(z)$ für z, indem wir Pfade, auf denen nur Knoten mit Ausgangs=

grad eins liegen, zu einer Kante zusammenfassen und diese mit denjenigen String, der zu dem Pfad korrespondiert, beschriften.

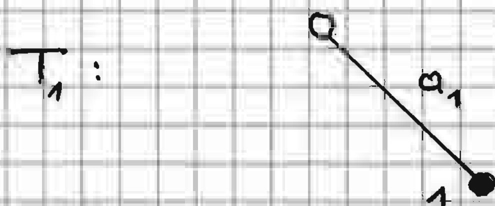
Sei $x := a_1 a_2 \dots a_n$ derjenige String, für den wir einen Suffixbaum konstruieren möchten. Die Idee besteht darin, zunächst einen impliziten Suffixbaum für x zu konstruieren und dann diesen zu einem Suffixbaum zu erweitern.

Hierzu konstruieren wir mit dem Präfix a_1 beginnend sukzessive für jeden Präfix $a_1 a_2 \dots a_i$ von x einen impliziten Suffixbaum T_i. Gegeben einen impliziten Suffixbaum T_n für x konstruieren wir dann einen Suffixbaum für x.

Demzufolge ist der Algorithmus in n Phasen unterteilt. In Phase i, $1 \le i \le n$ wird ein impliziter Suffixbaum T_i für den Präfix $a_1 a_2 \dots a_i$ von x konstruiert. Die Phasen sehen wie folgt aus:

<u>Phase 1:</u>

T_1 besitzt nur eine Kante. Diese ist mit a_1 markiert. D.h.,



Annahme:

Für $a_1 a_2 \ldots a_i$, $i \geq 1$ ist ein impliziter Suffixbaum T_i konstruiert.

Phase i+1:

Die (i+1)-te Phase ist in i+1 Erweiterungs-schritte unterteilt. Dabei korrespondiert der j-te Erweiterungsschritt, $1 \leq j \leq i+1$ zum Suffix $a_j a_{j+1} \ldots a_{i+1}$ von $a_1 a_2 \ldots a_{i+1}$.

Nehmen wir an, dass die Erweiterungsschritte $1, 2, \ldots, j-1$ bereits erfolgt sind. Wir werden nun den j-ten Erweiterungsschritt beschreiben.

Erweiterung j:

Ziel:

Dafür zu sorgen, dass, in der Wurzel des impliziten Suffixbaumes T_{i+1} startend, ein Pfad mit Markierung $a_j a_{j+1} \ldots a_{i+1}$ existiert.

Da T_{i+1} ein impliziter Suffixbaum ist, kann solch ein Pfad auf einer Kante, in einem inneren Knoten oder in einem Blatt enden.

Durchführung:

Finde, in der Wurzel des aktuellen Baumes startend, das Ende des Pfades mit Markierung $a_j a_{j+1} \ldots a_i$.

Da T_i solch einen Pfad enthielt, befindet sich auch solcher im aktuellen Baum. Jenachdem, wo dieser Pfad endet, unterscheiden wir drei Fälle:

<u>1. Fall:</u> Pfad mit Markierung $a_j a_{j+1} \dots a_i$ endet in einem Blatt.

- Konkateniere a_{i+1} an das Ende der Markierung derjenigen Kante, die in dem Blatt des Pfades endet.

<u>2. Fall:</u> Pfad mit Markierung $a_j a_{j+1} \dots a_i$ endet in einem inneren Knoten v.

Zwei Unterfälle können eintreten.

<u>2.1</u> Die Markierung einer der ausgehenden Kanten von v hat Präfix a_{i+1}.

Dann ist der gewünschte Pfad mit Markierung $a_j a_{j+1} \dots a_{i+1}$ bereits im aktuellen Suffixbaum.

$\leadsto$

Tue nichts.

<u>2.2</u> Alle Markierungen der ausgehenden Kanten von v beginnen mit einem Symbol $\neq a_{i+1}$.

$\leadsto$

v erhält einen neuen Nachfolgerknoten w. Die

Kante (v,w) erhält die Markierung a_{i+1} und w die Nummer j. Der Knoten w ist dann ein Blatt im aktuellen Suffixbaum.

<u>3. Fall</u> Pfad mit Markierung $a_j a_{j+1} \ldots a_i$ endet auf einer Kante e.

Sei $\beta c \gamma$ die Markierung der Kante e, wobei β der Suffix von $a_j a_{j+1} \ldots a_i$ ist. D.h., c ist das erste Symbol der Kantenmarkierung, das nicht zu $a_j a_{j+1} \ldots a_i$ korrespondiert. Da der Pfad auf der Kante e und nicht in einem Knoten endet, existiert c.

Zwei Unterfälle können eintreten:

<u>3.1</u> $c = a_{i+1}$

Dann ist der gewünschte Pfad mit Markierung $a_j a_{j+1} \ldots a_{i+1}$ bereits im aktuellen Suchbaum

$\leadsto$

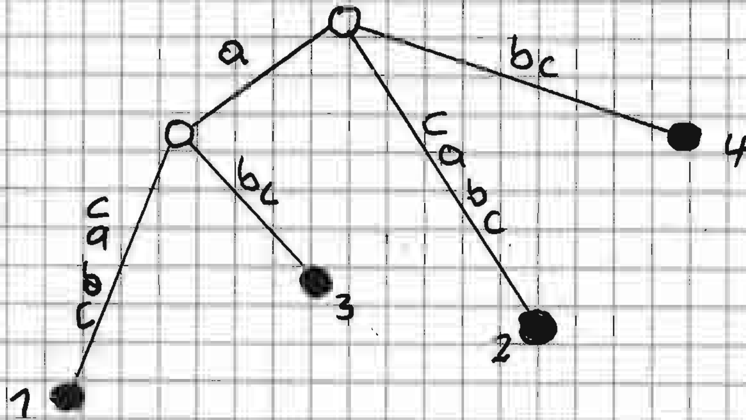
Tue nichts.

<u>3.2</u> $c \neq a_{i+1}$

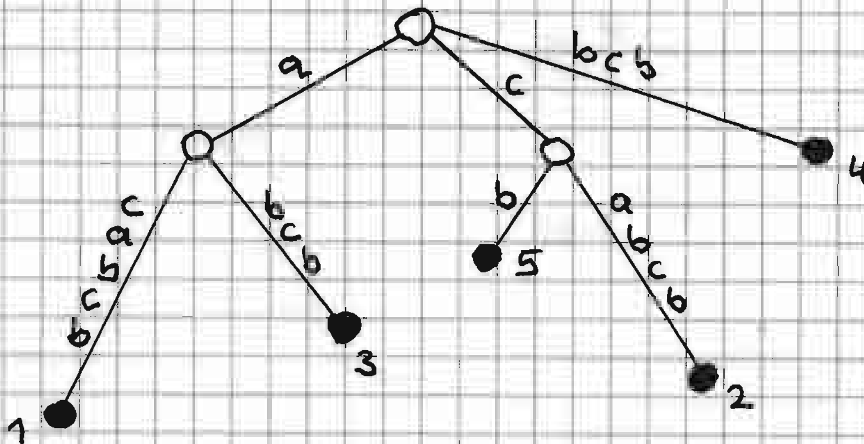
Führe folgende Transformation durch:

Beispiel:

Betrachten wir folgenden impliziten Suffixbaum für acabc:



Nach Hinzufügen des Symbols b erweitert sich obiger implizite Suffixbaum zu folgenden impliziten Suffixbaum für acabcb:



Ziel:

Abschätzung des Zeit- und Platzbedarfes des obigen Algorithmus.

Platzbedarf:

In jedem Blatt von T_n endet eines der möglichen n Suffixe. Also enthält T_n maximal n Blätter und somit auch maximal $n-1$ innere Knoten und höchstens $2n-2$ Kanten. Eine Kante ist mit einem Teilstring von x der Länge $\leq n$ beschriftet.

$\Longrightarrow$

Der Platzbedarf von T_n ist $O(n^2)$.

Zeitanalyse:

- Kosten $K(T_{i+1})$ für die Konstruktion von T_{i+1}:

Für die Erweiterung j ist zunächst das Ende des Pfades mit Markierung $a_j a_{j+1} \ldots a_i$ zu finden. Falls man hierzu in der Wurzel des aktuellen Baumes startet und den Pfad abläuft, dann benötigt man Länge des Pfades, also $O(i-(j-1)) = O(i+1-j)$ Zeit. Ansonsten benötigt man für die Erweiterung konstante Zeit.

$\Longrightarrow$

$$K(T_{i+1}) = O\left(\sum_{j=1}^{i} i+1-j\right)$$

$$= O\left(\sum_{j=0}^{i-1} j+1\right)$$

$$= O(i^2).$$

- Gesamtkosten:

$$O\left(\sum_{i=1}^{n} i^2\right) = O(n^3).$$

Unser Ziel ist nun die Reduktion der benötigten Zeit und des benötigten Platzes.

Ziel 1:

Reduktion auf $O(n^2)$ Zeit und $O(n^2)$ Platz.

Falls wir für jedes j das Ende des Pfades mit Markierung $a_j a_{j+1} \dots a_i$ in konstanter Zeit finden, dann reduzieren sich die Kosten für die Kosten von T_i auf $O(i)$.

$\Rightarrow$

Die Gesamtkosten betragen dann

$$O\left(\sum_{i=1}^{n} i\right) = O(n^2).$$

Idee:

Nach der Konstruktion von T_i, $1 \le i < n$ kennen wir für $1 \le j \le i$ das Ende des Pfades von der Wurzel aus beginnend mit Markierung $a_j a_{j+1} \dots a_i$. Wenn wir uns dieses merken, dann erhalten wir bei der Konstruktion von T_{i+1} dieses in konstanter Zeit.

Lediglich für $j = i+1$ muss das Ende des Pfades mit Markierung q_{i+1} explizit durch Ablaufen des Pfades ermittelt werden, falls dieser überhaupt existiert. Hierfür reicht konstante Zeit.

$\Rightarrow$

Wir haben somit die Zeit auf $O(n^2)$ reduziert. Da wir an der Struktur von T_n nichts geändert haben, hat sich der benötigte Platzbedarf nicht geändert und ist nach wie vor $O(n^2)$.

Ziel 2:

Reduktion auf $O(n)$ Zeit und $O(n)$ Platz.

Folgendes Beispiel zeigt, dass der bisherige Suffixbaum in der Tat $\Omega(n^2)$ Platz benötigen kann.

Beispiel:

Sei

$x = abcdefghijklmnopqrstuvwxyz$

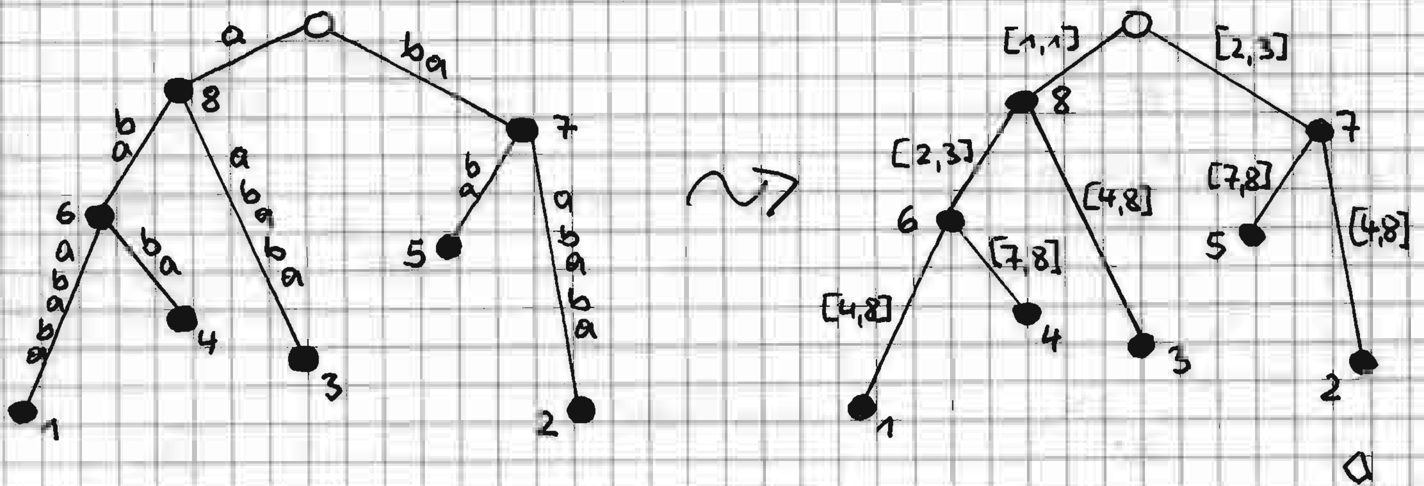
Jedes Suffix beginnt mit einem unterschiedlichen Symbol. Also besitzt die Wurzel des impliziten Suffixbaumes 26 Söhne. Jede korrespondierende Kante ist mit einem kompletten Suffix markiert. Somit benötigen wir für die Markierungen $\sum_{j=1}^{26} j = \frac{26 \cdot 27}{2}$ Platz

Idee:

Jede Kantenmarkierung korrespondiert zu einem Teilstring des Textstrings x. Anstatt diesen explizit hinzuschreiben genügt es, den Anfang und das Ende dieses Teilstrings zu spezifizieren.

Beispiel:

Sei x = abaababa. Wir erhalten folgende Suffixbäume:



⇒

Wir haben den Platzbedarf von $O(n^2)$ auf $O(n)$ reduziert.

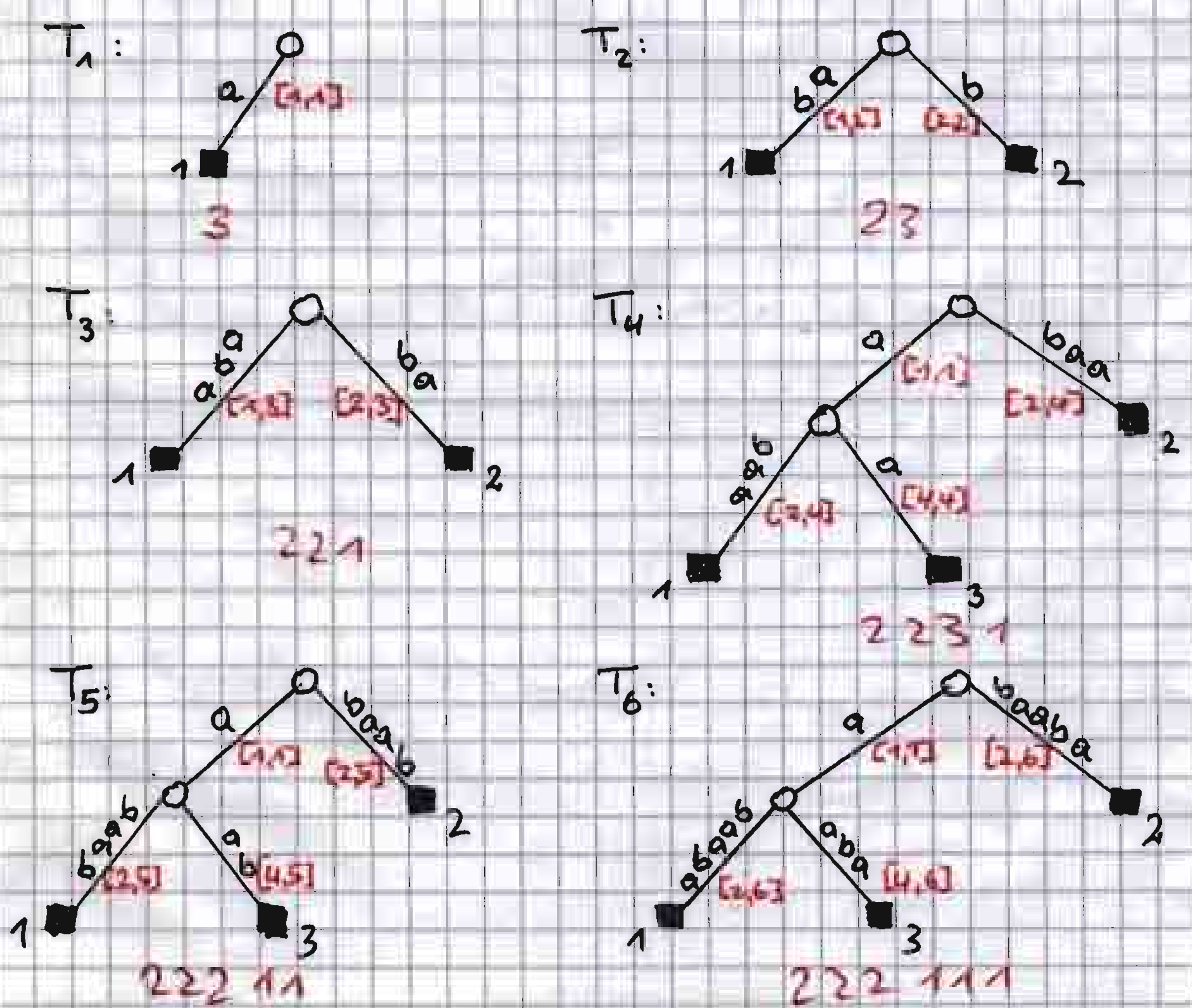
Für die Reduktion der benötigten Zeit betrachten wir nochmals die Durchführung von Erweiterung j. Je nachdem, welche Aktion der Erweiterungsschritt durchführt, fassen wir die Fälle und Unterfälle von Erweiterung j zusammen. Wir erhalten dann:

Aktion	Fall bzw. Unterfall
1. Tue nichts	2.1 und 3.1
2. Erweitere Blattmarkierung	1.
3. Füge eventuell inneren Knoten plus Blatt ein.	2.2 und 3.2

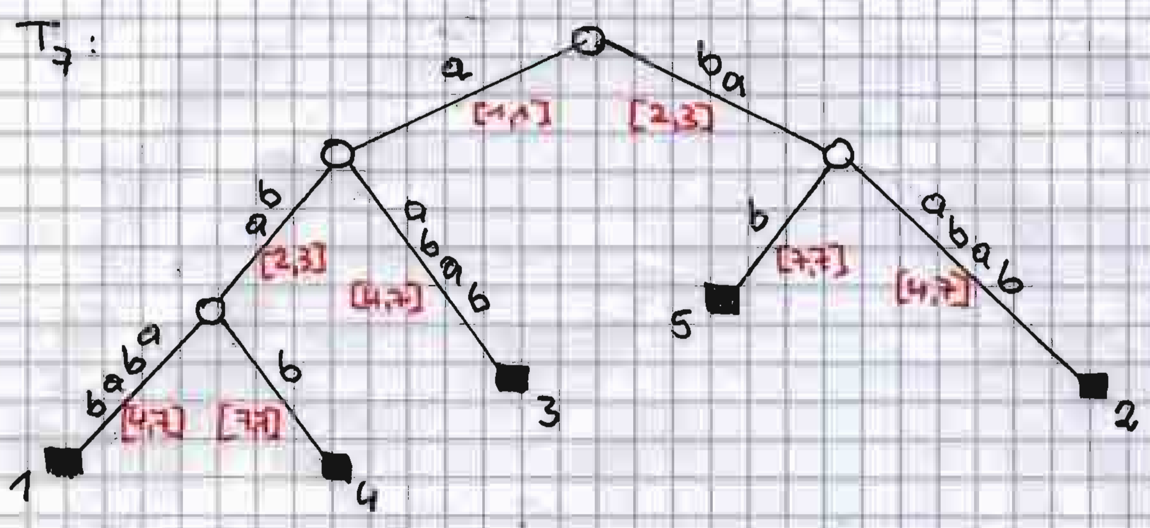
Um zu verdeutlichen, was im einzelnen passiert, führen wir die Konstruktion anhand obigen Beispieles durch.

<u>Beispiel</u> (Fortführung):

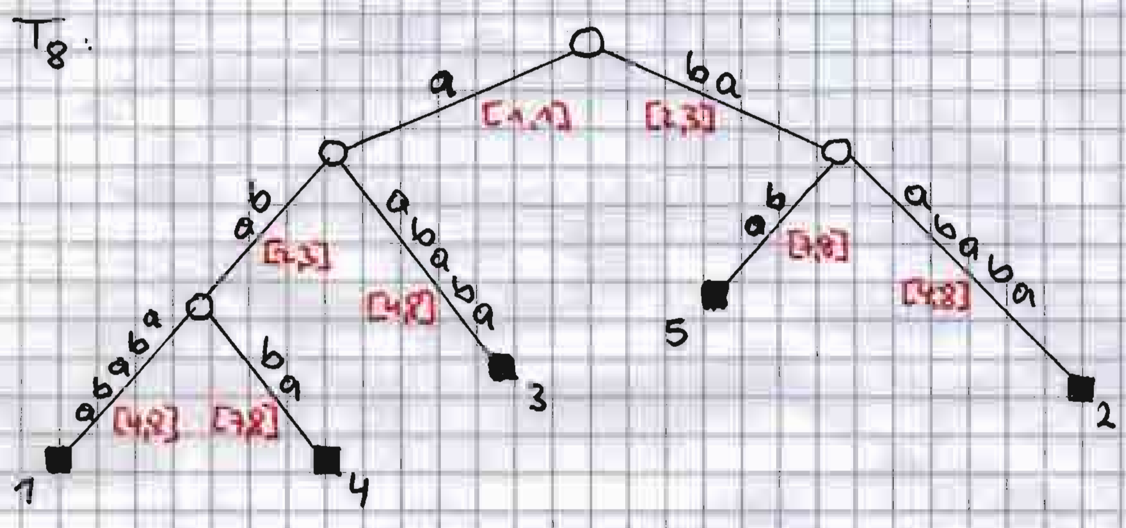
Sei $x = abaababa$. Wir zeichnen Blätter als Quadrate und innere Knoten als Kreise.



T_7:



T_8:



<u>Beobachtung:</u>

1) Falls im aktuellen Suffixbaum T ein Pfad in der Wurzel startend mit Markierung $c_1 c_2 \ldots c_k$ existiert, dann existiert in T für jedes $1 \leq l \leq k$ ein Pfad, der in der Wurzel anfängt, mit Markierung $c_l c_{l+1} \ldots c_k$.

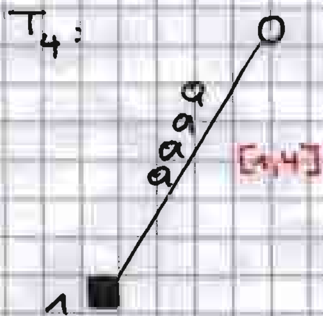
⇒

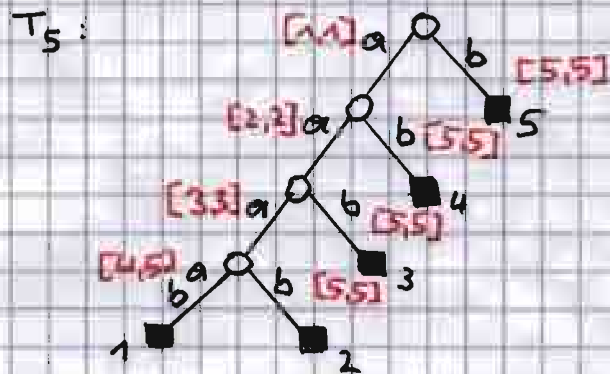
Wenn bei der Konstruktion von T_{i+1} bei der Erweiterung j die 1. Aktion "tue nichts" durchgeführt wird, dann wird auch bei jeder nachfolgenden Er=

weiterung ℓ, $j+1 \leq \ell \leq i+1$ diese Aktion durchgeführt.

2) Wenn ein Blatt mit Nummer k kreiert wird, dann bleibt dieser Knoten auch in den nachfolgenden Suffixbäumen ein Blatt mit Nummer k.

Zur Verdeutlichung der nächsten Beobachtung betrachten wir für $x = aaaab$ die Konstruktion von T_5 aus T_4.





<u>Beobachtung:</u>

3) Bei der Konstruktion von T_{i+1} aus T_i hat die Folge der Erweiterungsschritte folgende Struktur:

- Folge von Aktionen vom Typ 2 "Erweitere Blattmarkierung"

gefolgt von

- Folge von Aktionen vom Typ 3 "Füge eventuell inneren Knoten plus Blatt ein."

gefolgt von

- Folge von Aktionen vom Typ 1 "Tue nichts".

Dabei können eine oder zwei der oben definierten Folgen leer sein.

$\Longrightarrow$

Für $1 \leq k \leq n$ gilt: Wenn ein Blatt mit Nummer k kreiert wird, dann gilt für $1 \leq l < k$, dass das Blatt mit Nummer l bereits existiert.

Für unseren Konstruktionsalgorithmus haben obige Beobachtungen folgende Implikationen:

a) Die Erweiterungen der Blattmarkierungen müssen nicht explizit durchgeführt werden. D.h., für eingehende Kanten der Blätter ist lediglich der Anfang des Teilstrings von x, der zur Kantenmarkierung korrespondiert, zu spezifizieren. Das Ende ist stets das aktuelle Ende in x.

$\Longrightarrow$

Die Folge von Aktionen vom Typ 2 wird nicht explizit durchgeführt.

b)

<u>Annahme:</u>

Aus T_i soll der Suffixbaum T_{i+1} konstruiert werden. In T_i existieren die Blätter mit Nummern $1, 2, \ldots, k$. Blatt mit Nummer $k+1$

existiert nicht.

Um im Fall $k < i$ bei der Konstruktion von T_{i+1} mit der expliziten Durchführung der Konstruktion beginnen zu können, benötigen wir

- in T_i das Ende des Pfades P, beginnend in der Wurzel mit Markierung $a_{k+1} a_{k+2} \dots a_i$.

Gemäß der obigen Überlegungen endet P in einem inneren Knoten oder auf einer Kante.

<u>Beobachtung:</u>
Wegen obiger Beobachtung 1 und der Wahl von k ist P genau derjenige Pfad, mit dem die Konstruktion von T_i beendet wurde.

$\Rightarrow$

Das Ende von P ist bekannt, so dass die Konstruktion von T_{i+1} am Ende von P gestartet werden kann.

Zwei Fälle können nun eintreten:

<u>1. Fall</u>: Aktion vom Typ 1 wird durchgeführt.

Wegen Beobachtung 1 sind wir mit der Konstruktion von T_{i+1} fertig. Des Weiteren kennen wir nun das Ende des Pfades, beginnend in der Wurzel mit der Markierung $a_{k+1} a_{k+2} \dots a_{i+1}$, so dass die explizite Konstruktion von T_{i+2} hier gestartet werden kann.

2. Fall Aktion vom Typ 3 wird durchgeführt.

Je nachdem, wo der Pfad P endet, unterscheiden wir zwei Unterfälle.

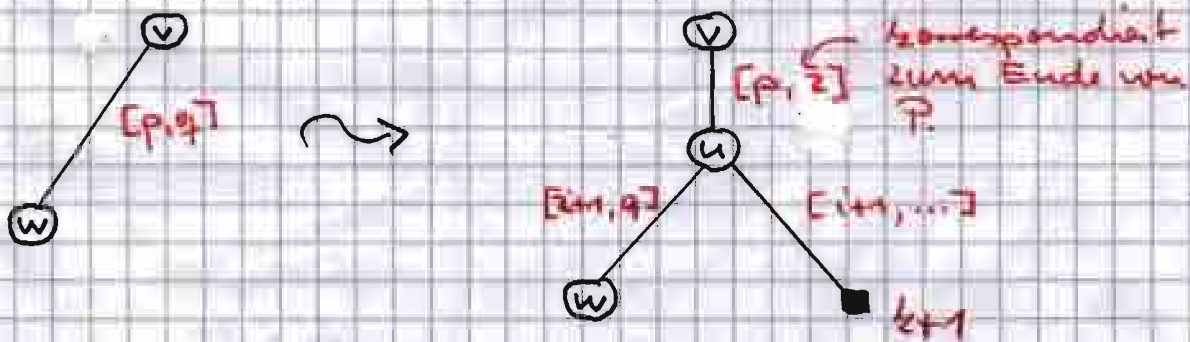
2.1 P endet in einem inneren Knoten v.

Gemäß unserer Konstruktion beginnen alle Markierungen der ausgehenden Kanten von v mit einem Symbol $\neq a_{i+1}$. Der Knoten v erhält ein neues Blatt b mit Nummer k+1 als weiteren Sohn. Die Kante (v, b) wird mit [i+1, ...] markiert.

2.2 P endet auf einer Kante e := (v, w).

Sei [p, q] die Markierung von e. Falls w ein Blatt ist, dann interpretieren wir q als i+1.

Situation nebst Modifikation:



Um die Konstruktion fortsetzen zu können, benötigen wir das Ende desjenigen Pfades P', der in der Wurzel anfängt und Markierung $a_{z+2}\, a_{z+3} \dots a_i$ hat. Gemäß obiger Beobachtung 1 existiert P'.

Ziel:

Entwicklung einer Datenstruktur, die den Zugriff auf das Ende von P' in konstanter Zeit ermöglicht.

Zunächst benötigen wir einige Bezeichnungen. Sei v ein Knoten im aktuellen Suffixbaum T. Dann bezeichnet m(v) die zum Pfad von der Wurzel von T zum Knoten v korrespondierende Kantenmarkierung. Sei $m(v) := a\alpha$ mit $a \in \Sigma$ und $\alpha \in \Sigma^*$. Dann definieren wir

$$s(v) := \begin{cases} \text{Knoten } w \text{ in } T \text{ mit } m(w) = \alpha & \text{falls } w \text{ existiert} \\ \text{undefiniert} & \text{sonst.} \end{cases}$$

Beobachtung:

Sei v ein Blatt mit Nummer p und u ein Blatt mit Nummer p+1 in T. Dann gilt $s(v) = u$.

Annahme:

Jeder Knoten v im aktuellen Suffixbaum T enthält einen Zeiger auf den Knoten s(v), falls dieser existiert.

Dann können wir wie folgt das Ende des Pfades P' finden. Je nachdem, ob oben Fall 2.1 oder Fall 2.2 zutreffend war, unterscheiden wir zwei Fälle:

2.1 P endet in einem inneren Knoten v.

Da jeder innere Knoten mindestens zwei Söhne besitzt, gibt es Pfade mit Markierungen

$a_{z+1} a_{k+2} \ldots a_i c \ldots$ und auch $a_{z+1} a_{k+2} \ldots a_i d \ldots$

mit $c \neq d$ von der Wurzel zu einem Blatt.

Beobachtung 1 $\Rightarrow$

Es gibt Pfade mit Markierungen

$a_{z+2} \ldots a_i c \ldots$ und $a_{z+2} \ldots a_i d \ldots$

von der Wurzel zu einem Blatt.

$\Rightarrow$

Der Pfad P'', der in der Wurzel anfängt und die Markierung $a_{z+2} \ldots a_i$ hat, endet in einem inneren Knoten.

$\Rightarrow$

Der Knoten $s(v)$ existiert im aktuellen Suffixbaum T.

Annahme. $\Rightarrow$

Wir haben in T einen sogenannten Suffixzeiger von v nach $s(v)$.

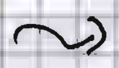
Übung

Geben Sie ein Beispiel an, in dem $s(v)$ eine aus=

gehende Kante, deren Markierung mit einem Symbol $g \in \Sigma$ beginnt, besitzt, obwohl dies für v nicht der Fall ist.

Obige Übungsaufgabe besagt, dass nun zwei Fälle eintreten können:

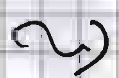
2.1.1 Eine der ausgehende Kanten von $s(v)$ hat Markierung, die mit a_{i+1} beginnt.



Führe Aktion vom Typ 1 "Tue nichts" durch.

Wegen Beobachtung 1 sind wir mit der Konstruktion von T_{i+1} fertig. Des Weiteren kennen wir nun das Ende des Pfades, beginnend in der Wurzel mit der Markierung $a_{k+2} a_{k+3} \dots a_{i+1}$, so dass die Konstruktion von T_{i+2} hier gestartet werden kann.

2.1.2 Die Markierungen aller ausgehenden Kanten von $s(v)$ beginnen mit einem Symbol $\neq a_{i+1}$.



$s(v)$ erhält ein neues Blatt b mit Nummer $k+2$ als weiteren Sohn. Die Kante $(s(v), b)$ erhält die Markierung $[i+1, \dots]$.

Wir folgen dem Zeiger auf $s(s(v))$. Dieser

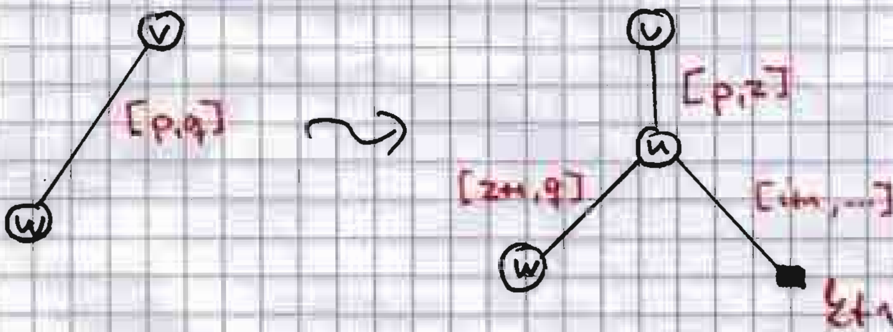
existiert, falls $s(v)$ nicht die Wurzel des aktuellen Suffixbaumes ist, was in der Definition von $s(v)$ $\alpha = \varepsilon$ implizieren würde.

Falls der Zeiger auf $s(s(v))$ nicht existiert, dann ist lediglich noch Erweiterung $i+1$ durchzuführen. Oben haben wir gezeigt, wie diese in konstante Zeit durchführbar ist.

<u>2.2</u> P endet auf einer Kante $e = (v, w)$.

Dann ist v ein innerer Knoten. Oben haben wir uns überlegt, dass $s(v)$ existiert.

<u>Situation nebst Modifikation</u>:



Sei

$$a_{z+1}\alpha\beta = a_{z+1} a_{z+2} \ldots a_i ,$$

wobei $a_{z+1}\alpha$ die Markierung des Pfades $\bar{P}$ von der Wurzel zum Knoten v und β die Markierung der Kante (v,u) sind.

Es gilt: $|\beta| = z - p + 1$

Gemäß unserer Annahme enthält v einen Zeiger auf $s(v)$.

- Wir folgen nun dem Zeiger auf $s(v)$ und dann von $s(v)$ aus dem Pfad mit Markierung β, bis das Ende dieses Pfades gefunden ist.

Ziel: Analyse dieser Pfadverfolgung.

Sei

$$\beta = a_e a_{e+1} \dots a_i .$$

Immer wenn ein innerer Knoten eingefügt wird, erhält dieser ein neues Blatt als Sohn. Dieses Blatt verbleibt während der gesamten Konstruktion als Blatt im Suffixbaum und ändert auch niemals seine Nummer.

Somit folgt aus der Tatsache, dass P auf einer Kante endet und der Konstruktion, dass $s(v)$ genau eine ausgehende Kante

$$e_1 := (s(v), w_1)$$

besitzt, deren Markierung mit a_e beginnt.

Sei $[p_1, q_1]$ die Markierung der Kante e_1.

Im Folgenden bezeichnet L stets die Länge des noch zu folgenden Teilpfades. D.h., zu Be=

gimm gilt: $L = z - p + 1.$

Jenachdem, ob der noch zu betrachtende Pfad über die aktuelle Kante hinausgeht oder nicht unterscheiden wir zwei Fälle.

<u>1. Fall</u>: $L \leq q_1 - p_1 + 1$

Dann endet der Pfad $\overline{\overline{P}}$ mit Markierung β

$$\begin{cases} \text{im Knoten } w_1 & \text{falls } L = q_1 - p_1 + 1 \\ \text{auf der Kante } e_1 & \text{falls } L < q_1 - p_1 + 1 \end{cases}$$

<u>2. Fall</u>: $L > q_1 - p_1 + 1$

Dann entspricht der zur Kante $e_1 = (s(v), w_1)$ korrespondierende Teilpfad dem echten Präfix der Länge $q_1 - p_1 + 1$ von β

Wir modifizieren L durch

$$L := L - (q_1 - p_1 + 1)$$

und folgen von w_1 aus $\overline{\overline{P}}$ weiter.

Wir verfahren nun bezüglich dem Pfad mit Markierung $a_{k+2} a_{k+3} \cdots a_i$ wie oben bezüglich dem Pfad mit Markierung $a_{k+1} a_{k+2} \cdots a_i$ beschrieben.

Zwei Dinge haben wir noch herauszuarbeiten. Zum einen müssen wir uns noch überlegen, wie wir die Suffixzeiger der inneren Knoten setzen.

(Für Blätter benötigen wir keine Suffixzeiger.) Zum anderen ist noch abzuschätzen, welchen Aufwand wir insgesamt für das Zurückverfolgen der Teilpfade haben.

Zunächst überlegen wir uns, wie wir die Suffixzeiger der inneren Knoten effizient setzen können.

Idee:

Falls wir uns davon überzeugen könnten, dass nach Hinzufügen eines neuen inneren Knotens u in einem Erweiterungsschritt einer Phase der Knoten $s(u)$ im nächsten Erweiterungsschritt derselben Phase sowieso besucht wird, dann könnte nach Hinzunahme eines neuen inneren Knotens u der Suffixzeiger auf $s(u)$ im nachfolgenden Erweiterungsschritt mit lediglich konstantem Mehraufwand gesetzt werden.

Dies leistet gerade folgendes Lemma:

Lemma 1.12

Falls bei der Konstruktion von T_{i+1} bei der Erweiterung j ein neuer innerer Knoten u mit $m(u) = a_j a_{j+1} \dots a_i$ dem aktuellen Suffixbaum als Sohn des Knotens v hinzugefügt wird, dann endet bei der Erweiterung $j+1$ die Zurückverfolgung des Pfades von $s(v)$ aus beginnend in einem inneren Knoten w mit $m(w) = a_{j+1} \dots a_i$,

falls solcher Knoten existiert, oder es wird ein Knoten w mit $m(w) = a_{j+1} a_{j+2} \dots a_i$ dem aktuellen Suffixbaum hinzugefügt.

<u>Beweis</u>:

Übung

■

Es verbleibt also noch die Analyse des Gesamtaufwands für das Zurückverfolgen der Teilpfade. Falls hierbei ein Gesamtaufwand von $O(n)$ herauskommt, dann folgt aus obigen Überlegungen, dass der implizite Suffixbaum T_n in $O(n)$ Zeit konstruierbar ist.

<u>Erinnerung</u>: (Definition Tiefe eines Knotens u):

Sei u ein Knoten eines Baumes T. Dann ist die <u>Tiefe von u bzgl. T</u> Tiefe(u,T) definiert durch

$$\text{Tiefe}(u,T) := \begin{cases} 0 & \text{falls } u = \text{Wurzel}(T) \\ 1 + \text{Tiefe}(u,T_i) & \text{sonst, wobei } T_i \text{ derjenige direkte Unterbaum von } T \text{ ist, der } u \text{ enthält.} \end{cases}$$

Tiefe(u,T) ist somit die Anzahl der Knoten auf dem Pfad von Wurzel(T) nach u, wobei der Knoten u selbst nicht mitgezählt wird.

Idee:

Wir interpretieren den Algorithmus als einen Agenten, der in der Wurzel des anfänglichen Suffixbaumes startet und wie der Algorithmus die Pfade zurückverfolgt, indem er sich von Knoten zu Knoten bewegt. Unser Ziel ist es, den Gesamtaufwand für die Wanderung des Agenten durch den Suffixbaum mit Hilfe der Tiefe des Knotens, in dem sich der Agent am Schluss befindet, abzuschätzen.

Beobachtung:

1) Die Tiefe des Startknotens ist null.
2) Das Passieren einer Baumkante erhöht die Tiefe des aktuellen Knotens für den Agenten um eins.
3) Nur das Durchschreiten eines Suffixzeigers kann die Tiefe des aktuellen Knotens für den Agenten verringern.
4) Nach Terminierung des Algorithmus ist die aktuelle Knotentiefe für den Agenten $\leq n$.

Somit gilt für die Gesamtlänge GL durchschrittenen Pfades

$$GL \leq n + 2 \cdot GR,$$

wobei GR die gesamte Verringerung der aktuellen Tiefe bezeichnet.

Somit müssen wir nur noch GR abschätzen. Dies ist mit Hilfe des folgenden Lemmas einfach.

Lemma 1.13

Sei $(v, s(v))$ ein Suffixzeiger, den der Agent durchschreitet. Dann gilt zum Zeitpunkt des Durchschreitens für den aktuellen Suffixbaum T

$$\text{Tiefe}(v, T) \leq \text{Tiefe}(s(v), T) + 1.$$

Beweis:

Seien

P der Pfad von Wurzel(T) nach v in T

und

P' der Pfad von Wurzel(T) nach $s(v)$ in T.

Sei w ein beliebiger Vorgängerknoten von v mit

$$m(w) = a\beta \quad , a \in \Sigma, \beta \in \Sigma^+.$$

Lemma 1.12 $\Rightarrow$

$s(w)$ existiert und w besitzt Suffixzeiger $(w, s(w))$.

Seien w und w' zwei verschiedene Vorgängerknoten von v mit $|m(w)| \geq 2$ und $|m(w')| \geq 2$.

$\Rightarrow$

$s(w) \neq s(w')$ und sowohl $s(w)$ als auch $s(w')$ liegen auf P'.

Lediglich der direkte Nachfolger von Wurzel(T) auf P besitzt eventuell keinen Suffixzeiger.

Insgesamt folgt aus obigen Betrachtungen

$$|P| \leq |P'| + 1$$

$\Rightarrow$

$$\text{Tiefe}(v, T) \leq \text{Tiefe}(s(v), T) + 1.$$

■

Da maximal n-mal ein Suffixzeiger durchlaufen wird, folgt direkt aus Lemma 1.13

$$GR \leq n.$$

Insgesamt haben wir folgenden Satz bewiesen:

<u>Satz 1.3</u>

Die Länge des Pfades, den der Agent während der Konstruktion des impliziten Suffixbaumes durchschreitet, ist $\leq 3 \cdot n$.

Da die sonstige Arbeit pro Phase konstant ist und n Phasen durchlaufen werden, wird der implizite Suffixbaum in $O(n)$ Zeit konstruiert.

Offen ist noch die Konstruktion eines Suffixbaumes aus dem impliziten Suffixbaum. Hierzu konkatenieren wir den Textstring x und den

String, bestehend aus dem Sonderzeichen \$. Der Effekt ist, dass kein Suffix von $x\$$ Präfix eines anderen Suffix ist. Also ist der implizite Suffixbaum für $x\$$ auch der Suffixbaum für $x\$$.

Übung:
Konstruieren Sie in Linearzeit aus einem Suffixbaum für $x\$$ einen Suffixbaum für x.

Insgesamt haben wir folgenden Satz bewiesen:

Satz 1.4
Seien Σ ein endliches Alphabet und $x \in \Sigma^+$ ein String der Länge n über Σ. Ein Suffixbaum für x kann in $O(n)$ Zeit konstruiert werden.

1.4 Anwendungen von Suffixbäumen

Suffixbäume finden unmittelbar ihre Anwendung bei der Implementierung eines Indexes für einen Text. Dies überrascht nicht, da jeder Teilstring eines Textes Präfix eines Suffixes des Textes ist.

Sei $x := a_1 a_2 a_3 \dots a_n$ ein Text. Der kürzeste Präfix von $a_i a_{i+1} \dots a_n$, $1 \leq i \leq n$, so dass dieser nirgendwo sonst im Text x vorkommt, heißt Identifizierer der Position i im Text x. Falls der Text x mit einem Sonderzeichen endet, dann für jede Position i der Identifizierer definiert. Der Positionsbaum eines Textes

$x = a_1 a_2 \dots a_n \in \Sigma^n$ ist ein kompakter Trie bezüglich des Alphabets Σ, der $\leq n$ markierte enthält. Die Blätter sind mit paarweise verschiedenen Zahlen aus $\{1, 2, \dots, n\}$ nummeriert. Der Pfad von der Wurzel zum Blatt mit Nummer i korrespondiert zum Identifizierer der Position i im Text x.

Der Positionsbaum für x kann leicht aus dem Suffixbaum für x konstruiert werden.

Übung:
Entwickeln Sie einen Algorithmus, der in Linearzeit aus dem Suffixbaum für x den Positionsbaum für x konstruiert.

Folgende vier Operationen soll der Index eines Textes unterstützen:

1) Für $y \in \Sigma^*$ finde den längsten Präfix, der im Text x enthalten ist.

2) Finde das erste Vorkommen eines Strings $y \in \Sigma^*$ im Text x.

3) Bestimme die Anzahl der Vorkommen von $y \in \Sigma^*$ im Text x.

4) Berechne eine Liste aller Vorkommen von $y \in \Sigma^*$ im Text x.

Übung:
Entwickeln Sie Algorithmen für die vier Operationen, die der Index eines Textes unterstützen soll. Zeigen Sie insbesondere, dass der Suffixbaum für einen Text x derart in Linearzeit vorbereitet werden kann, dass die Anzahl der Vorkommen eines Strings y im Text x in Zeit $O(|y|)$ ermittelt werden kann.

1970 hat Don Knuth vermutet, dass es für folgendes Problem keinen Linearzeitalgorithmus gibt:

Längste gemeinsame Teilstringproblem

gegeben: Strings x_1 und x_2 über einem endlichen Alphabet Σ.
gesucht: ein längster gemeinsamer Teilstring von x_1 und x_2.

Die Verwendung von Suffixbäumen ermöglicht einen einfachen Linearzeitalgorithmus für dieses Problem.

Beobachtung:
Ein längster gemeinsamer Teilstring von x_1 und x_2 ist Präfix sowohl eines Suffixes von x_1 als auch eines Suffixes von x_2.

$\leadsto$

Idee:
Konstruktion eines gemeinsamen Suffixbaumes für die Strings x_1 und x_2.

<u>Ziel:</u>

Konstruktion eines gemeinsamen Suffixbaumes für eine Menge $\{x_1, x_2, \dots, x_t\}$ von Strings.

Seien $\$_1, \$_2, \dots, \$_t$ paarweise verschiedene Symbole, die nicht in einem der Strings $x_1, x_2, \dots, x_t$ vorkommen.

- Berechne den Suffixbaum für den String

$$x_1 \$_1 x_2 \$_2 \dots \$_{t-1} x_t \$_t .$$

<u>Eigenschaften:</u>

- Wahl der Sonderzeichen $\$_1, \$_2, \dots, \$_t$ und Konstruktionsalgorithmus

Der konstruierte Baum hat die gewünschte Struktur. Da die Blattmarkierungen nicht explizit erweitert werden, interpretieren wir, dass diese mit dem korrespondierenden Sonderzeichen $\$_i$ enden.

Zur Lösung des längsten gemeinsame Teilstring-problem konstruieren wir für die beide Eingabe-strings x_1 und x_2 den gemeinsamen Suffixbaum und merken dabei uns für jeden inneren Knoten v die Anzahl der Symbole auf dem Pfad von der Wurzel zu v.

Danach traversieren wir bottom-up den Suffix=baum und entscheiden dabei für jeden inneren Knoten, ob in seinem Teilbaum bezüglich beiden Strings x_1 und x_2 ein Blatt existiert. Ein solcher mit maximaler Anzahl von Symbolen auf dem Pfad von der Wurzel zu ihm definiert einen längsten gemeinsamen Teilstring von x_1 und x_2.

Übung:

a) Arbeiten Sie den Algorithmus zur Lösung des längsten gemeinsame Teilstringproblems aus.

b) Verallgemeinern Sie den Algorithmus, so das ein längster gemeinsamer Teilstring von k Strings $x_1, x_2, \ldots, x_k$ in Zeit $O(k \cdot n)$ be=stimmt wird, wobei $n := |x_1| + |x_2| + \ldots + |x_k|$.

Bemerkung:

Eine Vielzahl von weiteren Anwendungen von Suffixbäumen findet man in

Dan Gusfield: Algorithms on Strings, Trees, and Sequences, Kapitel 7 und 9.