15 - Suche in Texten(1)

Prof. Dr. S. Albers

Suche in Texten

Verschiedene Szenarios:

Dynamische Texte

- Texteditoren
- Symbolmanipulatoren

Statische Texte

- Literaturdatenbanken
- Bibliothekssysteme
- Gen-Datenbanken
- WWW-Verzeichnisse

Suche in Texten

Datentyp **string**:

- array of character
- file of character
- list of character

Operationen: (seien T, P von Typ **string**)

Länge:	length ()
***i*-tes Zeichen** :	T [i]
Verkettung:	cat (T, P) T.P

Problemdefinition

Gegeben:

Text $t_1\ t_2\ ....\ t_n \quad \in \Sigma^n$

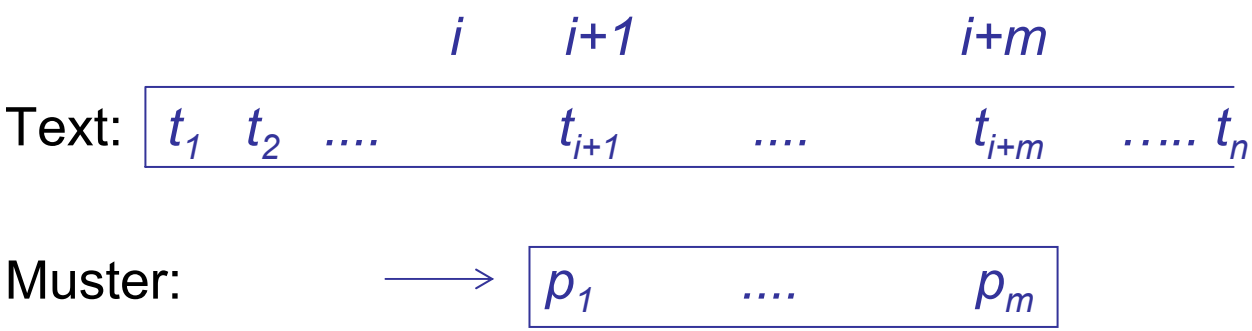
Muster $p_1 p_2\ ...\ p_m \quad \in \Sigma^m$

Gesucht:

Ein oder alle Vorkommen des Musters im Text, d.h. Verschiebungen i mit $0 \leq i \leq n - m$ und

$$p_1 = t_{i+1}$$
$$p_2 = t_{i+2}$$
$$\vdots$$
$$p_m = t_{i+m}$$

Problemdefinition



Aufwandsabschätzung (Zeit) :

1. # mögl. Verschiebungen: $n - m + 1$ # Musterstellen: m
   → $O(n\ m)$

2. mind. 1 Vergleich pro m aufeinander folgende Textstellen:
   → $\Omega\ (\ m + n/m\ )$

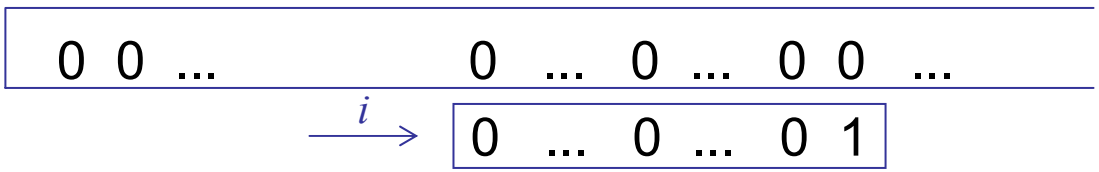
Naives Verfahren

Für jede mögliche Verschiebung $0 \leq i \leq n - m$ prüfe maximal m Zeichenpaare. Bei Mismatch beginne mit neuer Verschiebung.

```
textsearchbf  := proc (T : : string, P : : string)
#  Input:   Text T und Muster P
# Output:  Liste L mit Verschiebungen i, an denen P in T vorkommt
     n := length (T); m := length (P);
     L := [];
     for i from 0 to n-m do
            j := 1;
            while j ≤ m and T[i+j] = P[j]
                        do j := j+1 od;
            if j = m+1 then L := [L [] , i] fi;
     od;
     RETURN (L)
end;
```

Naives Verfahren

Aufwandsabschätzung (Zeit):



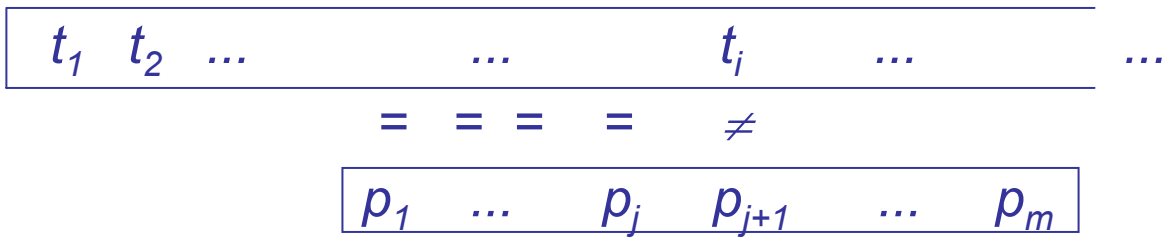
Worst Case: $\Omega(m\ n)$

In der Praxis oft: Mismatch tritt sehr früh auf

→ Laufzeit ~ $c\ n$

Verfahren nach Knuth-Morris-Pratt (KMP)

Seien t_i und p_{j+1} die zu vergleichenden Zeichen:



Tritt bei einer Verschiebung erstmals ein Mismatch auf bei

t_i und p_{j+1} dann gilt:

- Die zuletzt verglichenen *j* Zeichen in *T* stimmen mit den ersten *j* Zeichen in *P* überein.
- $t_i \neq p_{j+1}$

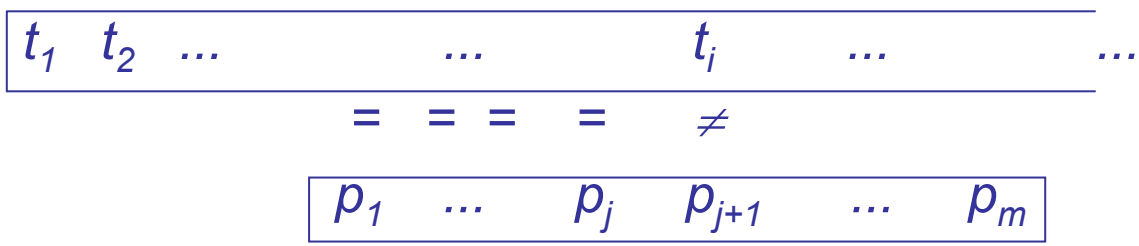
Verfahren nach Knuth-Morris-Pratt (KMP)

Idee:

Bestimme $j' = \text{next}[j] < j$, so dass t_i anschliessend mit $p_{j'+1}$ verglichen werden kann.

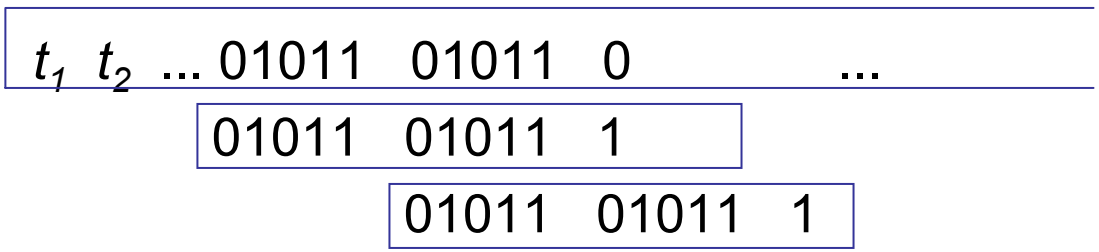
Bestimme $j' < j$, so dass $P_{1\ldots j'} = P_{j-j'+1\ldots j}$.

Bestimme das längste Präfix von P, das echtes Suffix von $P_{1\ldots j}$ ist.



Verfahren nach Knuth-Morris-Pratt (KMP)

Beispiel für die Bestimmung von next[j]:



next[j] = Länge des längsten Präfix von *P*, das echtes Suffix von $P_{1 \ldots j}$ ist.

Verfahren nach Knuth-Morris-Pratt (KMP)

$\Rightarrow$ für P = 0101101011 ist next = [0,0,1,2,0,1,2,3,4,5] :

1	2	3	4	5	6	7	8	9	10
0	1	0	1	1	0	1	0	1	1
		0							
		0	1						
					0				
					0	1			
					0	1	0		
					0	1	0	1	
					0	1	0	1	1

Verfahren nach Knuth-Morris-Pratt (KMP)

```
KMP := proc (T : : string, P : : string)
# Input: Text T und Muster P
# Output: Liste L mit Verschiebungen i, an denen P in T vorkommt
    n := length (T); m := length(P);
    L :=  []; next := KMPnext(P);
    j  :=  0;
    for i from 1 to n do
          while j>0 and T[i] <> P[j+1] do j := next [j] od;
          if T[i]  = P[j+1]  then j := j+1 fi;
          if j = m then L := [L[] , i-m] ;
                        j  := next [j]
          fi;
     od;
     RETURN (L);
end;
```

Verfahren nach Knuth-Morris-Pratt (KMP)

Muster: abrakadabra, next = [0,0,0,1,0,1,0,1,2,3,4]

```
a b r a k a d a b r a b r a b a b r a k ...
| | | | | | | | | | |
a b r a k a d a b r a
```

next[11] = 4

```
a b r a k a d a b r a b r a b a b r a k ...
            - - - - /
            a b r a k
```

next[4] = 1

Verfahren nach Knuth-Morris-Pratt (KMP)

a b r a k a d a b r a b r a b a b r a k ...

\- | | | /|

a b r a k

next [4] = 1

a b r a k a d a b r a b r a b a b r a k ...

\- | /|

a b r a k

next [2] = 0

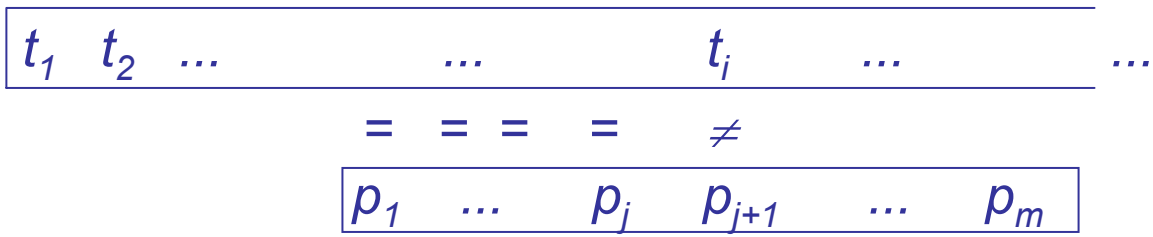
a b r a k a d a b r a b r a b a b r a k ...

| | | | |

a b r a k

Verfahren nach Knuth-Morris-Pratt (KMP)

Korrektheit:



Situation am Beginn der for-Schleife:

$P_{1...j} = T_{i-j...i-1}$ und $j \neq m$

falls $j = 0$: man steht auf erstem Zeichen vom P

falls $j \neq 0$: P kann verschoben werden, solange $j>0$ und $t_i \neq p_{j+1}$

Verfahren nach Knuth-Morris-Pratt (KMP)

Ist dann $T[i] = P[j+1]$, können j und i (am Schleifenende) erhöht werden.

Wurde ganz P verglichen ($j = m$), ist eine Stelle gefunden, und es kann verschoben werden.

Verfahren nach Knuth-Morris-Pratt (KMP)

Laufzeit:

- Textzeiger *i* wird nie zurückgesetzt
- Textzeiger *i* und Musterzeiger *j* werden stets gemeinsam inkrementiert
- Es ist next[j] < *j*; *j* kann nur so oft herabgesetzt werden, wie es heraufgesetzt wurde.

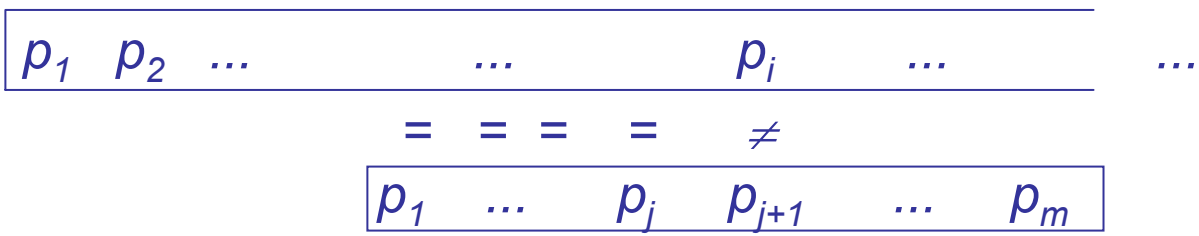
Der KMP-Algorithmus kann in Zeit O(*n*) ausgeführt werden, wenn das next-Array bekannt ist.

Berechnung des next-Arrays

next[i] = Länge des längsten Präfix von *P*, das echtes Suffix von $P_1 \ldots_i$ ist.

next[1] = 0

Sei next[*i*-1] = *j*:



Berechnung des next-Arrays

Betrachte zwei Fälle:

1) $p_i = p_{j+1}$ → next[i] = j + 1

2) $p_i \neq p_{j+1}$ → ersetze j durch next[j] , bis $p_i = p_{j+1}$ oder
j = 0.
Falls $p_i = p_{j+1}$ ist, kann next[i] = j + 1 gesetzt werden, sonst ist next[i] = 0.

Berechnung des next-Arrays

```
KMPnext := proc (P : : string)
#Input    :  Muster P
#Output :   next-Array für P
     m := length (P);
     next := array (1..m);
     next [1] := 0;
     j := 0;
     for i from 2 to m do
        while j > 0 and P[i] <> P[j+1]
             do j := next [j] od;
        if P[i] = P[j+1] then j := j+1 fi;
        next [i] := j
     od;
     RETURN (next);
end;
```

Laufzeit von KMP

Der KMP-Algorithmus kann in Zeit O(n + m) ausgeführt werden.

Kann die Textsuche noch schneller sein?

Verfahren nach Boyer-Moore (BM)

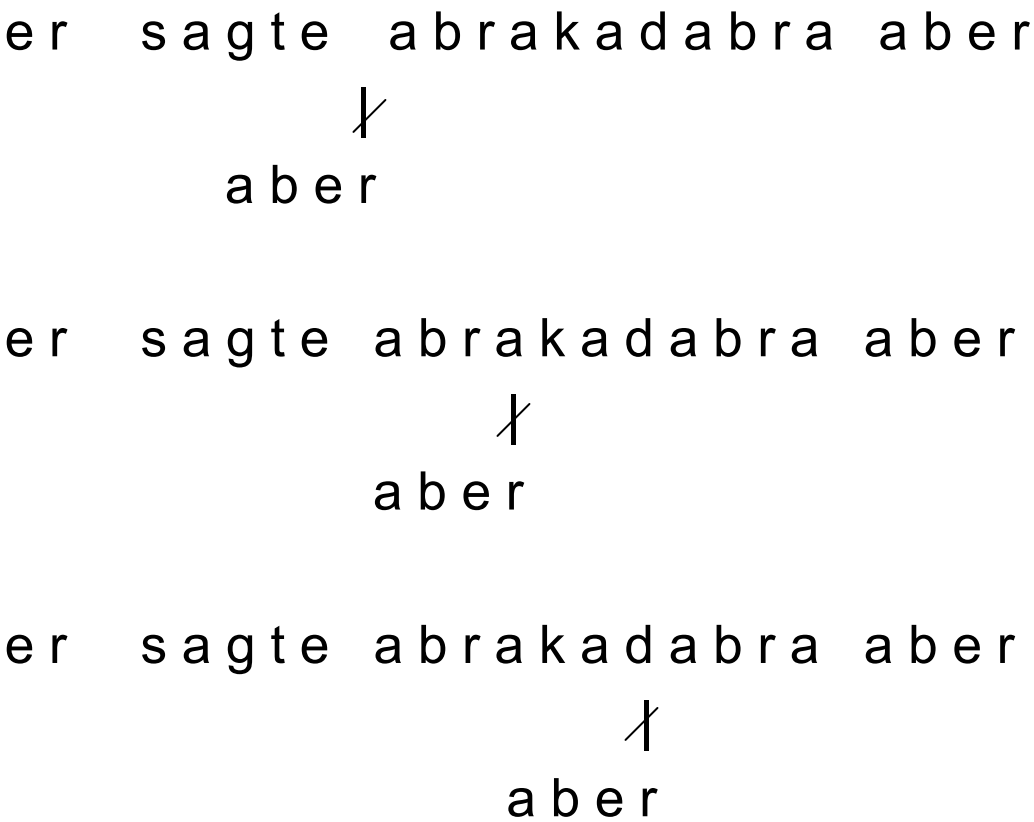
Idee: Das Muster von links nach rechts anlegen, aber zeichenweise von rechts nach links vergleichen

Beispiel:

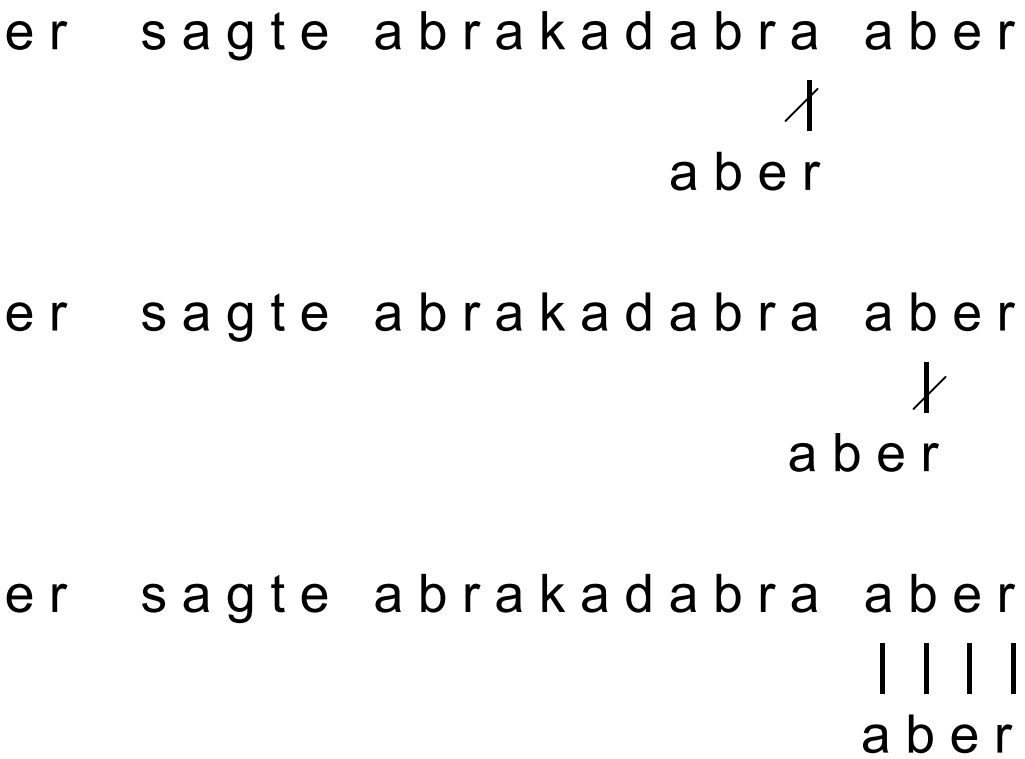
```
e r   s a g t e   a b r a k a d a b r a   a b e r
      ↙
a b e r
```

```
e r   s a g t e   a b r a k a d a b r a   a b e r
            ↙
      a b e r
```

Verfahren nach Boyer-Moore (BM)



Verfahren nach Boyer-Moore (BM)



Große Sprünge: wenig Vergleiche

Erhoffte Laufzeit: O($m + n/m$)

BM – Die Vorkommensheuristik

Für $c \in \Sigma$ und das Muster P sei

$\delta(c)$:= Index des von rechts her ersten Vorkommens von c in P

= max $\{j \mid p_j = c\}$

$$= \begin{cases} 0 & \text{falls } c \notin P \\ j & \text{falls } c = p_j \text{ und } c \neq p_k \text{ für } j < k \leq m \end{cases}$$

Wie teuer ist die Berechnung aller δ-Werte?

Sei $|\Sigma| = l$:

BM – Die Vorkommensheuristik

Seien

c = das den Mismatch verursachende Zeichen

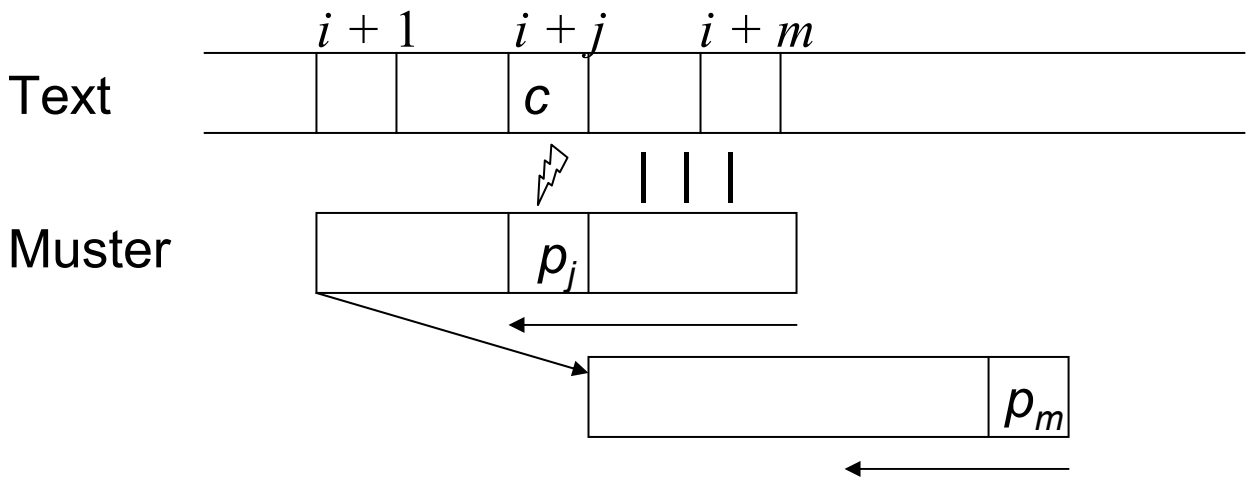
j = Index des aktuellen Zeichens im Muster ($c \neq p_j$)

BM – Die Vorkommensheuristik

Berechnung der Musterverschiebung

Fall 1 *c* kommt nicht im Muster *P* vor. ($\delta(c) = 0$)

Verschiebe das Muster um *j* Positionen nach rechts

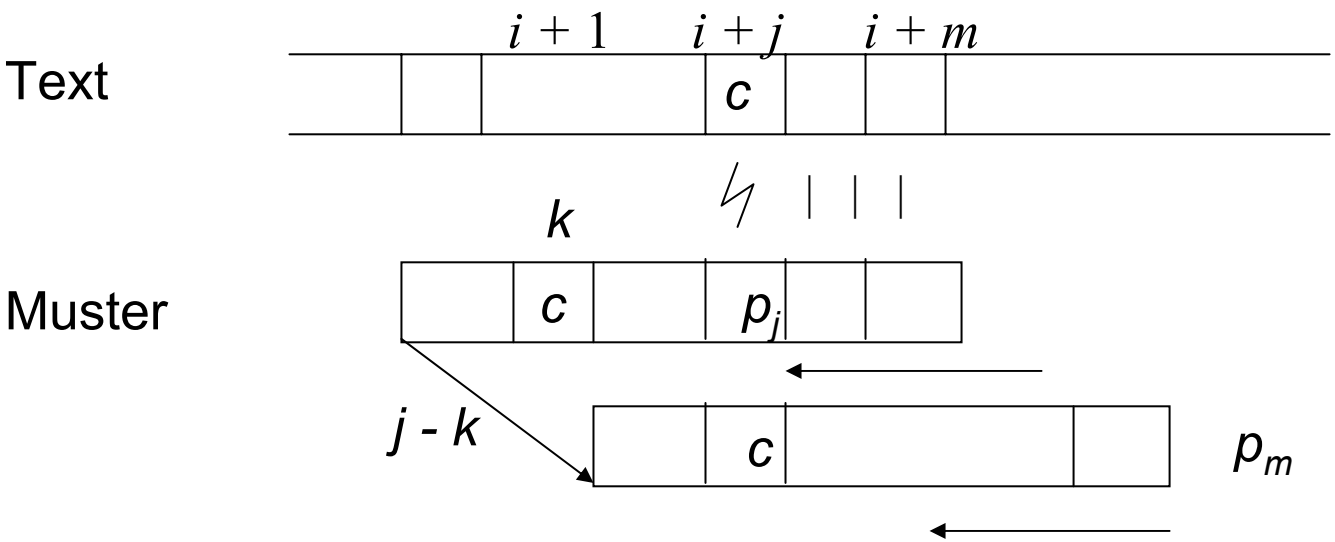


$$\Delta(i) = j$$

BM – Die Vorkommensheuristik

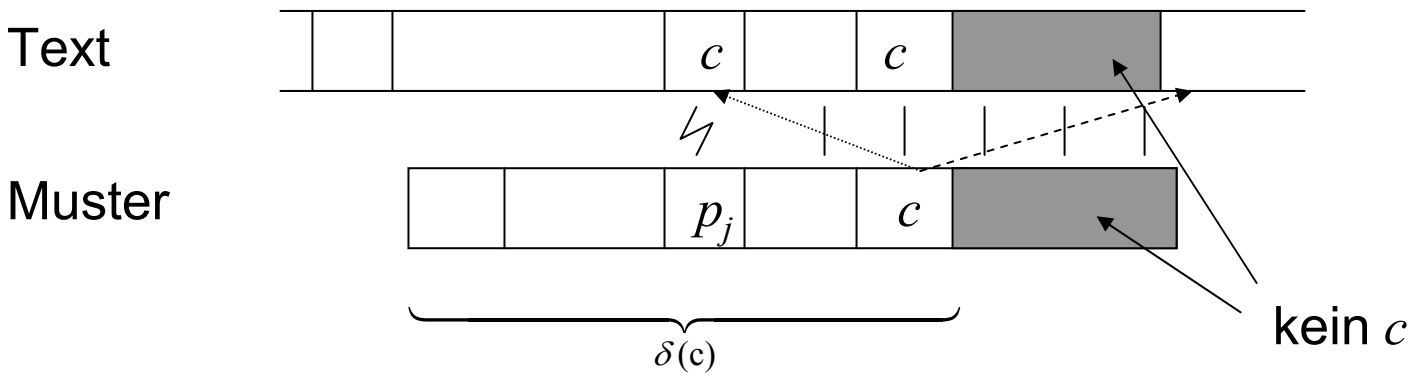
Fall 2 c kommt im Muster vor. ($\delta(c) \neq 0$)

Verschiebe das Muster soweit nach rechts, dass das rechteste c im Muster über einem potentiellen c im Text liegt.



BM – Die Vorkommensheuristik

Fall 2 a: $\delta(c) > j$

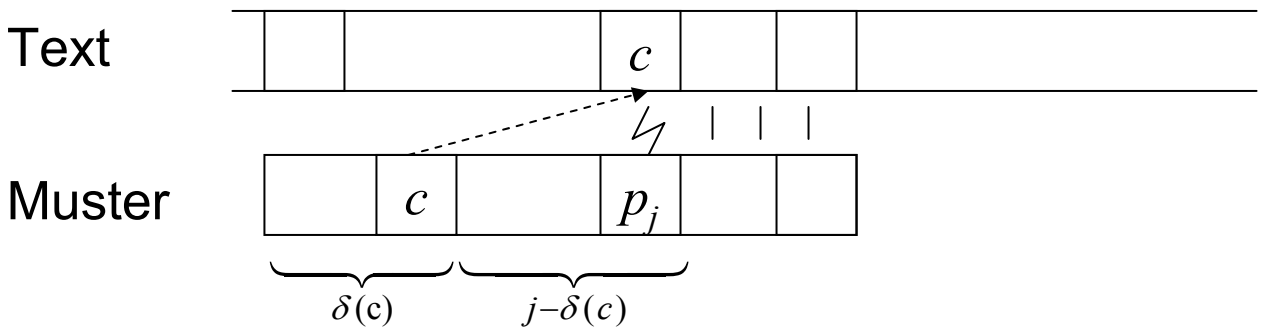


Verschiebung des rechtesten c im Muster auf ein potentielles c im Text.

$$\Rightarrow \text{Verschiebung um } \Delta(i) = m - \delta(c) + 1$$

BM – Die Vorkommensheuristik

Fall 2 b: $\delta(c) < j$



Verschiebung des rechtesten *c* im Muster auf *c* im Text:

$\Rightarrow$ Verschiebung um $\Delta(i) = j - \delta(c)$

BM-Algorithmus (1.Version)

Algorithmus BM-search1

Input: Text T und Pattern P

Output: Verschiebungen für alle Vorkommen von P in T

```
1  n := length(T); m := length(P)
2  berechne δ
3  i := 0
4  while i ≤ n – m do
5       j := m
6       while j > 0 and P[j] = T[i + j] do
7            j := j – 1
        end while;
```

BM-Algorithmus (1.Version)

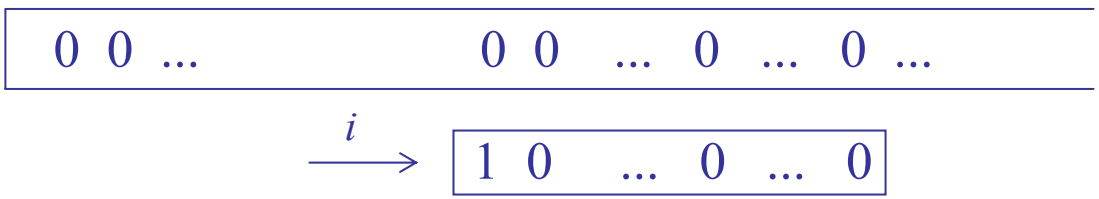
```
8      if j = 0
9       then gebe Verschiebung i aus
10             i := i + 1
11     else if δ(T[i + j]) > j
12          then i := i + m + 1 - δ[T[i + j]]
13          else i := i + j - δ[T[i + j]]
14  end while;
```

BM-Algorithmus (1.Version)

Laufzeitanalyse:

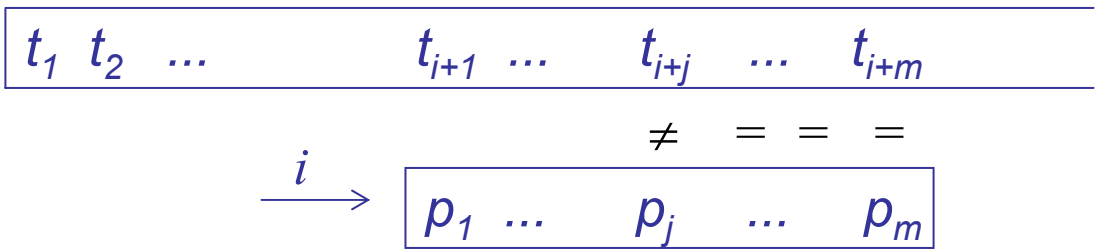
gewünschte Laufzeit : $O(m + n/m)$

worst-case Laufzeit: $\Omega(n\,m)$



Match-Heuristik

Nutze die bis zum Auftreten eines Mismatches $p_j \neq t_{i+j}$ gesammelte Information



$wrw[j]$ = Position, an der das von rechts her nächste Vorkommen des Suffixes $P_{j+1 \ldots m}$ endet, dem nicht das Zeichen P_j vorangeht.

Mögliche Verschiebung: $\gamma[j] = m - wrw[j]$

Beispiel für die *wrw*-Berechnung

wrw[*j*] = Position, an der das von rechts her nächste Vorkommen des Suffix $P_{j+1 \ldots m}$ endet, dem nicht das Zeichen P_j vorangeht.

Muster: banana

wrw[j]	betracht. Suffix	verb. Zeichen	weit. Auft.	Pos.
wrw[5]	a	n	banana	2
wrw[4]	na	a	*** bana na	0
wrw[3]	ana	n	banana	4
wrw[2]	nana	a	banana	0
wrw[1]	anana	b	banana	0
wrw[0]	banana	ε	banana	0

Beispiel für die *wrw*-Berechnung

$\Rightarrow$ *wrw* (banana) = [0,0,0,4,0,2]

a b a a b a b a n a n a n a n a

$\neq$ = = =

b a n a n a

b a n a n a

BM-Algorithmus (2.Version)

Algorithmus BM-search2

Input: Text T und Pattern P

Output: Verschiebung für alle Vorkommen von P in T

```
1  n := length(T); m := length(P)
2  berechne δ und γ
3  i := 0
4  while i ≤ n – m do
5       j := m
6       while j > 0 and P[j] = T[i + j] do
7            j := j – 1
       end while;
```

BM-Algorithmus (2.Version)

```
8       if j = 0
9         then gebe Verschiebung i aus
10              i := i + γ[0]
11        else i := i + max(γ[j], j - δ[T[i + j]])
12  end while;
```

15 - Suche in Texten (2): Suffix-Bäume

Prof. Dr. S. Albers

Suche in Texten

Verschiedene Szenarios:

Dynamische Texte

- Texteditoren
- Symbolmanipulatoren

Statische Texte

- Literaturdatenbanken
- Bibliothekssysteme
- Gen-Datenbanken
- WWW-Verzeichnisse

Eigenschaft von Suffix-Bäumen

Suchindex

zu einem Text σ für Suche nach verschiedenen Mustern α

Eigenschaften:

1. **Teilwortsuche** in Zeit O(|α|).

2. **Anfragen an σ selbst**, z.B.:
   Längstes Teilwort von σ, das an mind. 2 Stellen auftritt.

3. **Präfix-Suche:** Alle Stellen in σ mit Präfix α.

Eigenschaft von Suffix-Bäume

4. **Bereichs-Suche:** Alle Stellen in σ im Intervall $[\alpha, \beta]$ mit $\alpha \leq_{lex} \beta$, z.B.

   abrakadabra, acacia $\in$ [abc, acc],
   abacus $\notin$ [abc, acc] .

5. **Lineare Komplexität:**
   Speicherplatzbedarf und Konstruktionszeit $\in O(|\sigma|)$

Tries

Trie: Baum zur Repräsentation von Schlüsseln.

Alphabet Σ, Menge S von Schlüsseln, $S \subset \Sigma^*$

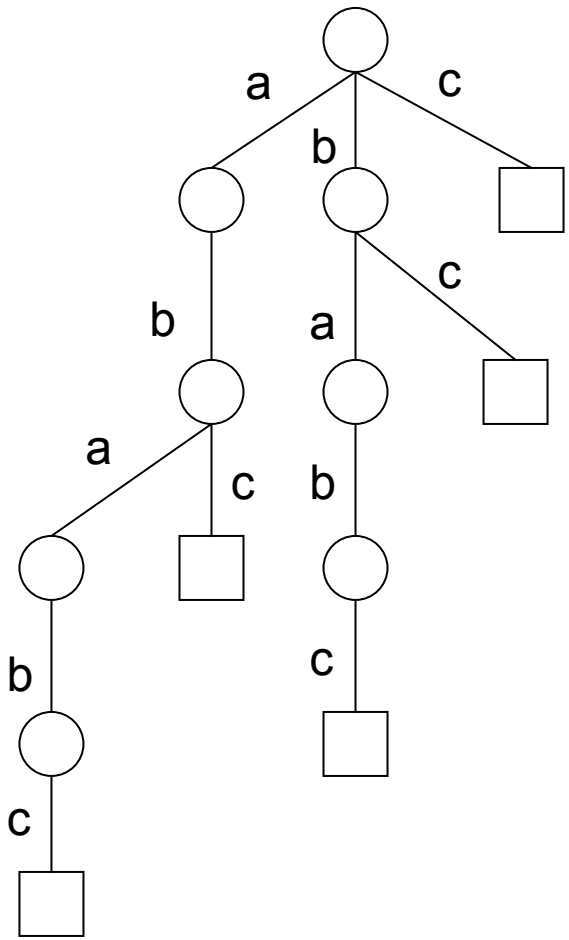
Schlüssel $\hat{=}$ Zeichenkette aus Σ^*

Kante eines Tries T: Beschriftung mit einzelnen Zeichen aus Σ

benachbarte Kanten: verschiedene Zeichen

Tries

Beispiel:



Tries

Blatt repräsentiert Schlüssel:

Entspricht Beschriftung der Kanten des Weges von der Wurzel zu Blatt

! Schlüssel werden nicht in Knoten gespeichert !

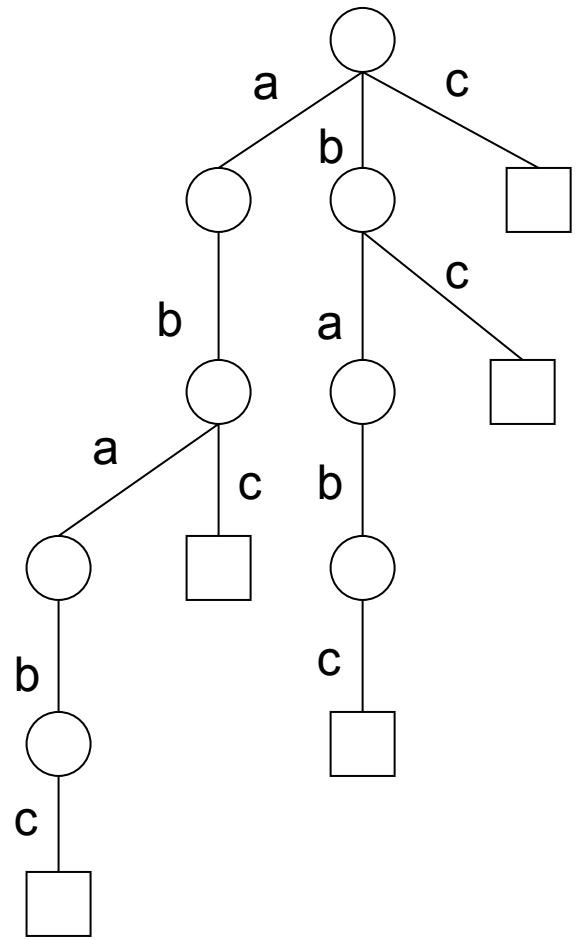
Suffix-Tries

Trie für alle Suffixe eines Wortes

Beispiel: σ = ababc

Suffixe:

ababc = suf_1
babc = suf_2
abc = suf_3
bc = suf_4
c = suf_5



Suffix-Tries

Innere Knoten eines Suffix-Tries = Teilwort von σ.

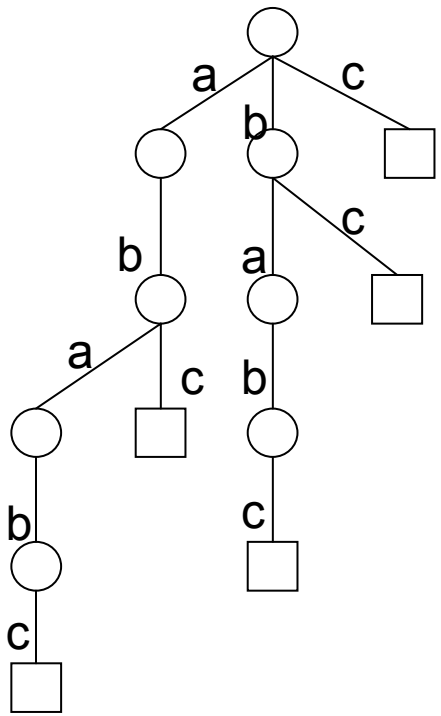
Jedes echte Teilwort von σ ist als innerer Knoten repräsentiert.

Sei $\sigma = a^n b^n$: $\exists$ $n^2 + 2n + 1$ verschied. Teilwörter = innere Knoten

$\Rightarrow$ Speicherplatzbedarf $\in O(n^2)$.

Suffix-Tries

Ein Suffix-Trie *T* erfüllt bereits einige der geforderten Eigenschaften:



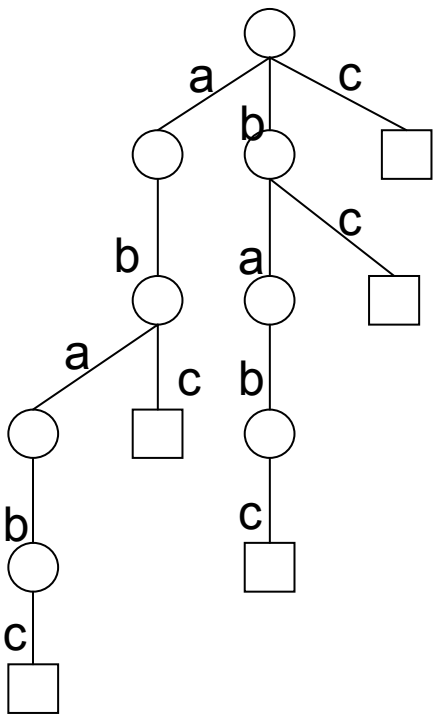
1. Zeichenkettensuche nach α : Folge dem Weg mit Kantenbeschriftung α in *T* in Zeit $O(|\alpha|)$. Blätter des Teilbaumes $\hat{=}$ Vorkommen von α

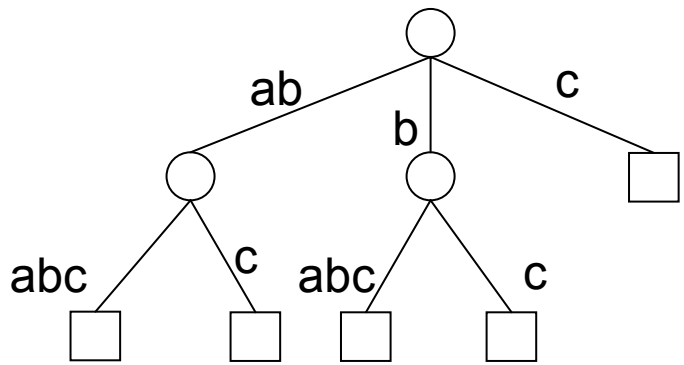
2. Längstes, doppelt auftretendes Wort: Innerer Knoten mit größter Tiefe, der mind. zwei Söhne hat.

3. Präfix-Suche: alle Vorkommen von Zeichenketten mit Präfix α finden sich in dem Teilbaum unterhalb des inneren Knotens von α in T.

Suffix-Bäume

Suffix-Baum entsteht durch Kontraktion von unären Knoten aus Suffix-Trie:





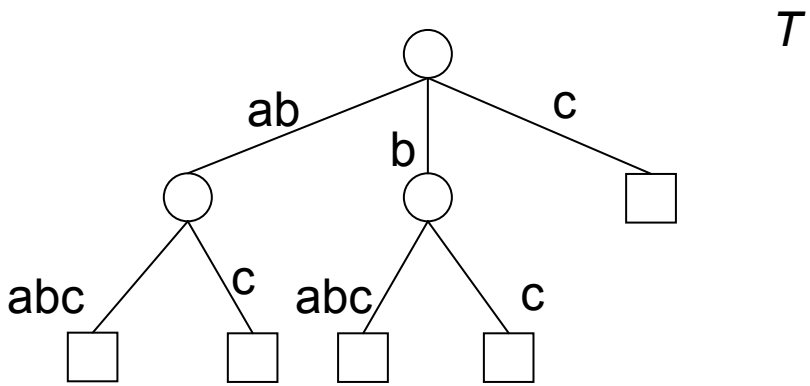
Suffix-Baum = kontraktierter Suffix-Trie

Interne Repräsentation von Suffix-Bäumen

Sohn/Bruder-Repräsentation

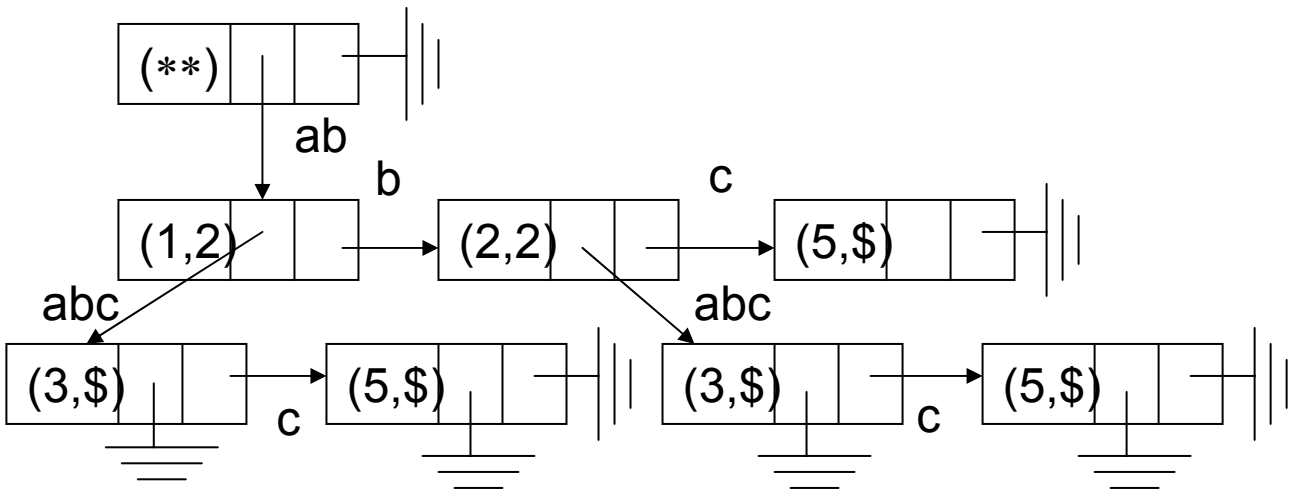
Teilwort: Zahlenpaar (*i,j*)

Beispiel: σ = ababc



Interne Repräsentation von Suffix-Bäumen

Beispiel σ = ababc



Knoten v = ($v.u$, $v.o$, $v.sn$, $v.br$)

Weitere Zeiger (Suffix-Zeiger) kommen später hinzu

Eigenschaften von Suffix-Bäumen

(S1) Kein Suffix ist Präfix eines anderen Suffixes;
gilt, falls (letztes Zeichen von σ) = \$ $\notin \Sigma$

Suche:

(T1) Kante $\hat{=}$ nichtleeres Teilwort von σ.

(T2) Benachbarte Kanten: zugeordnete Teilworte beginnen mit verschiedenen Zeichen.

Eigenschaften von Suffix-Bäumen

Größe

(T3) Innerer Knoten ($\neq$ Wurzel): mind. zwei Söhne.

(T4) Blatt $\triangleq$ (nicht-leeres) Suffix von σ.

Sei $n = |\sigma| \neq 1$

$\overset{(T4)}{\Longrightarrow}$ Anzahl der Blätter : n

$\overset{(T3)}{\Longrightarrow}$ Anzahl der inneren Knoten $\leq n-1$

$\Rightarrow$ Speicherplatz $\in O(n)$

Konstruktion von Suffix-Bäumen

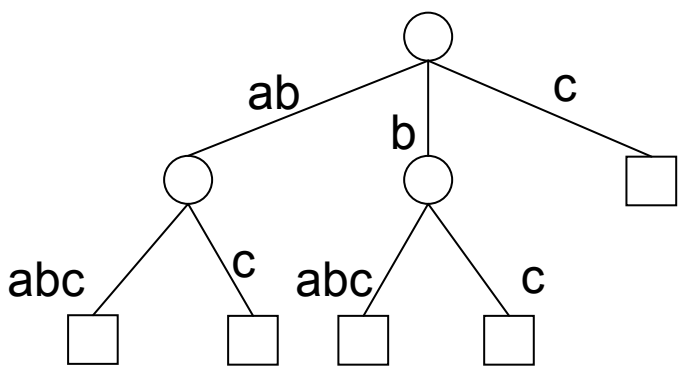
Definition:

partieller Weg: Weg von der Wurzel zu einem Knoten von *T*

Weg: Ein partieller Weg, der bei einem Blatt endet.

Ort einer Zeichenkette α: Knoten am Ende des mit α beschrifteten partiellen Weges (falls er existiert).

T

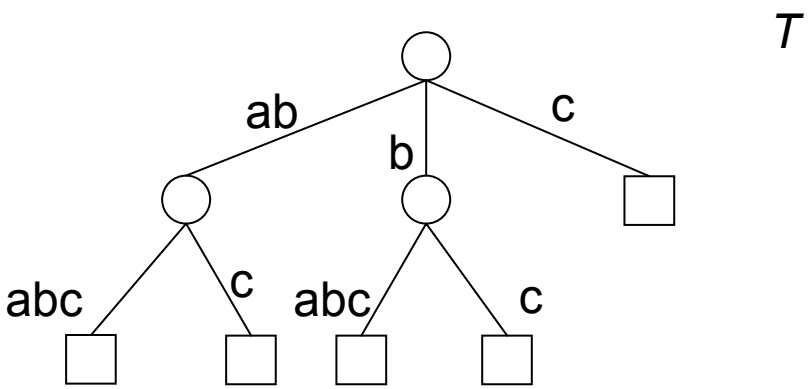


Konstruktion von Suffix-Bäumen

Erweiterung einer Zeichenkette α: Zeichenkette mit Präfix α

erweiterter Ort einer Zeichenkette α: Ort der kürzesten Erweiterung von α, deren Ort definiert ist.

kontraktierter Ort einer Zeichenkette α: Ort des längsten Präfixes von α, dessen Ort definiert ist.



Konstruktion von Suffix-Bäumen

Definitionen:

suf_i: an Position i beginnendes Suffix von σ, also z.B.
$suf_1 = \sigma$, suf_n = \$.

$head_i$: längstes Präfix von suf_i , das auch Präfix von suf_j für ein $j < i$ ist.

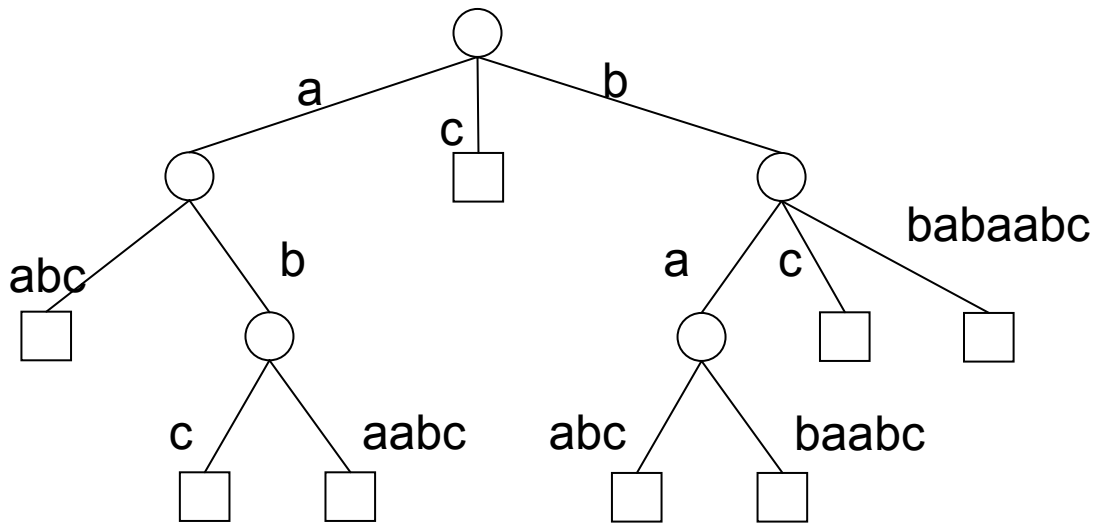
Beispiel: σ = bbabaabc α = baa (hat keinen Ort)

suf_4 = baabc

$head_4$ = ba

Konstruktion von Suffix-Bäumen

σ = bbabaabc



Naive Suffix-Baum-Konstruktion

Beginne mit dem leeren Baum T_0

Der Baum T_{i+1} entsteht aus T_i durch Einfügen des Suffixes suf_{i+1}.

Algorithmus Suffix-Baum

Input: Eine Zeichenkette σ

Output: Der Suffix-Baum T von σ

1 n := | σ |; T_0 := ∅;

2 **for** i := 0 **to** n – 1**do**

3 füge suf_{i+1} in T_i ein, dies sei T_{i+1} ;

4 **end for**

Naive Suffix-Baum-Konstruktion

In T_i haben alle Suffixe suf_j , $j < i$ bereits einen Ort.

→ $head_i$ = längstes Präfix von suf_i, dessen erweiterter Ort in T_{i-1} existiert.

Definition:

$tail_i := suf_i - head_i$, d.h. also $suf_i = head_i tail_i$.

$$\overset{(S1)}{\Longrightarrow} \quad tail_i \neq \varepsilon.$$

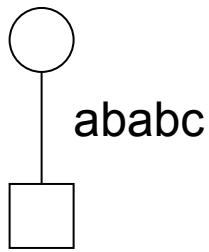
Naive Suffix-Baum-Konstruktion

Beispiel: σ = ababc

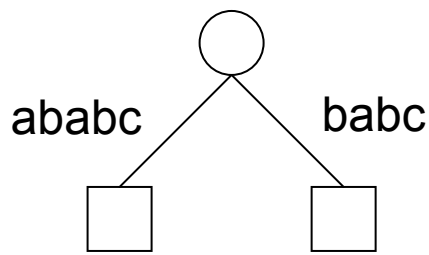
suf_3 = abc
$head_3$ = ab
$tail_3$ = c

T_0 =

T_1 =



T_2 =

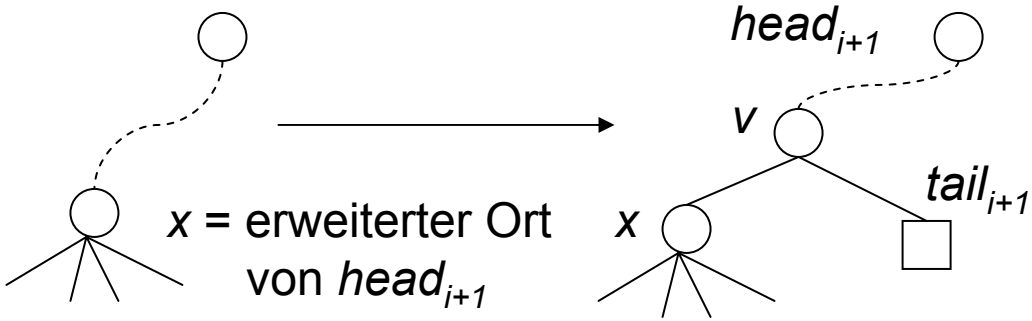


Naive Suffix-Baum-Konstruktion

T_{i+1} kann aus T_i wie folgt konstruiert werden:

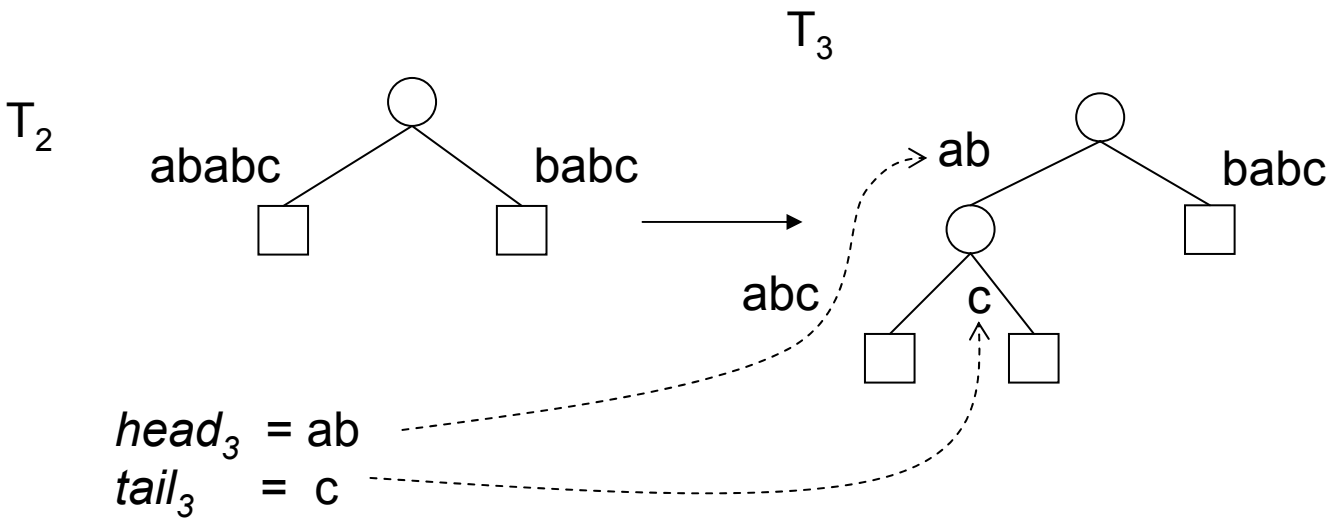
1. Man bestimmt den erweiterten Ort von $head_{i+1}$ in T_i und teilt die letzte zu diesem Ort führende Kante in zwei neue Kanten auf durch Einfügen eines neuen Knotens.

2. Man schaffe ein neues Blatt als Ort für suf_{i+1}



Naive Suffix-Baum-Konstruktion

Beispiel: σ = ababc



Naive Suffix-Baum-Konstruktion

Algorithmus Suffix-Einfügen

Input: Der Baum T_i und der Suffix suf_{i+1}

Output: Der Baum T_{i+1}

1 v := Wurzel von T_i

2 $j := i$

3 **repeat**

4 finde Sohn w von v mit $\sigma_{w.u} = \sigma_{j+1}$

5 $k := w.u - 1$;

6 **while** $k < w.o$ **and** $\sigma_{k+1} = \sigma_{j+1}$ **do**

7 $k := k + 1$; j := $j + 1$

end while

Naive Suffix-Baum-Konstruktion

8 **if** $k = w.o$ **then** $v := w$

9 **until** k $< w.o$ or w = nil

10 /* v ist konktraktierter Ort von $head_{i+1}$ */

11 füge den Ort von $head_{i+1}$ und $tail_{i+1}$ in T_i unter v ein

Laufzeit für Suffix-Einfügen: O( )

Gesamtlaufzeit für naive Suffix-Baum-Konstruktion: O( )

Der Algorithmus M

(Mc Creight, 1976)

Falls erweiterter Ort von $head_{i+1}$ in T_i gefunden: Erzeugen eines neuen Knotens und Aufspalten einer Kante $\in$ O(1) Zeit.

Idee: Erweiterter Ort von $head_{i+1}$ wird in konstanter amortisierter Zeit in T_i bestimmt. (Zusatzinformation erforderlich!)

Analyse des Algorithmus M

Theorem 1

Algorithmus *M* liefert in Zeit $O(|\sigma|)$ einen Suffix-Baum für σ mit $|\sigma|$ Blättern und höchstens $|\sigma| - 1$ inneren Knoten.

Suffix-Baum Anwendung

Verwendung von Suffix-Baum *T*:

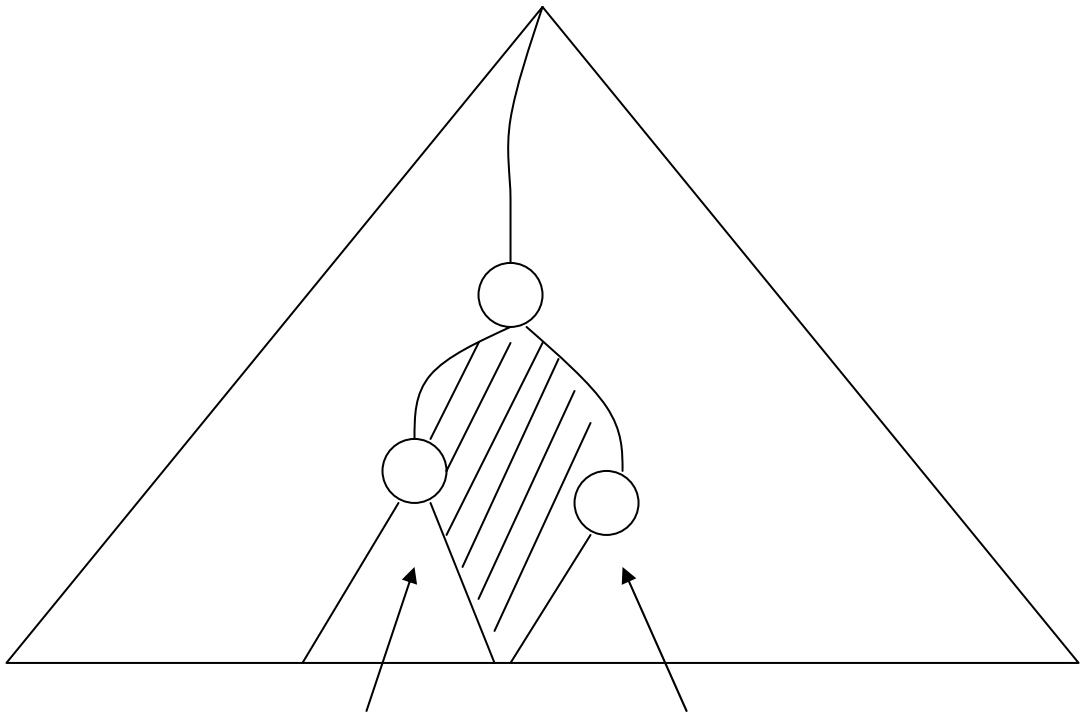
1 Suche nach Zeichenkette α: Folge dem Weg mit Kantenbeschriftung α in *T* in Zeit O(| α |). Blätter des Teilbaumes $\hat{=}$ Vorkommen von α

2 Suche längstes, doppelt auftretendes Wort: Finde Ort eines Wortes mit größter gewichteter Tiefe, der innerer Knoten ist.

3 Suche nach Präfix: Alle Vorkommen von Zeichenketten mit Präfix α finden sich in dem Teilbaum unterhalb des „Ortes“ von α in *T*.

Suffix-Baum Anwendung

4 Bereichssuche nach $[\alpha, \beta]$:



Bereichsgrenzen

Suffix-Baum Beispiel

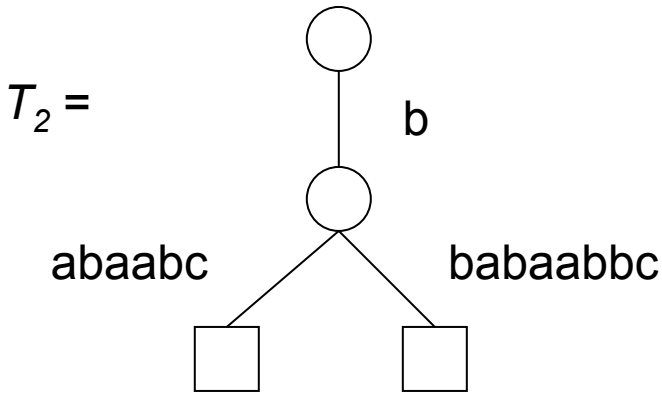
T_0 = 

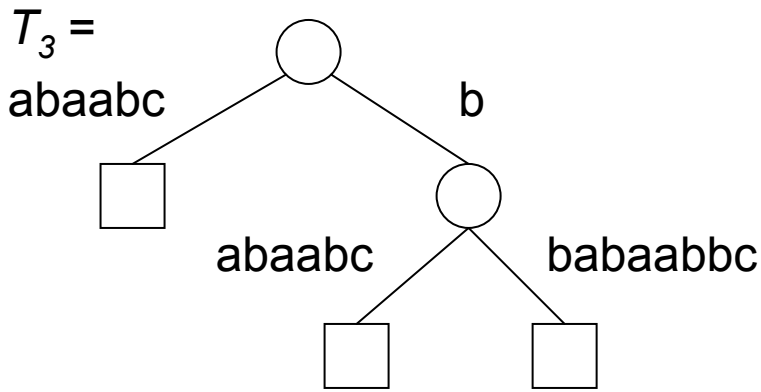
T_1 =  bbabaabc

suf_1 = bbabaabc

suf_2 = babaabc
$head_2$ = b

Suffix-Baum Beispiel

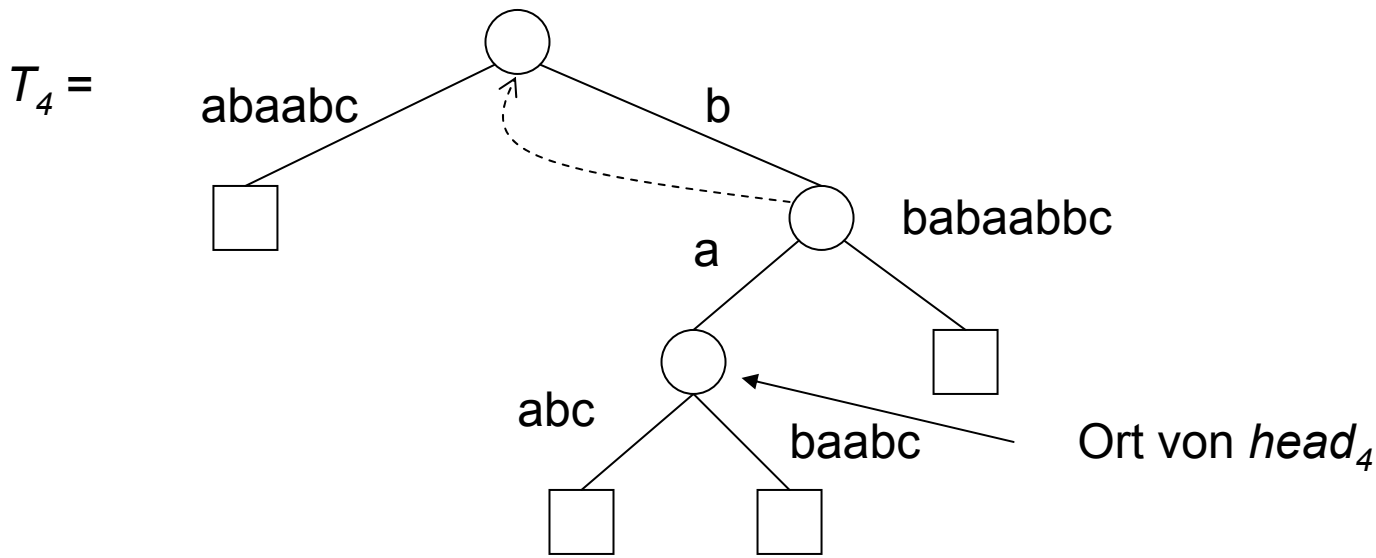




suf_3 = abaabc
$head_3$ = ε

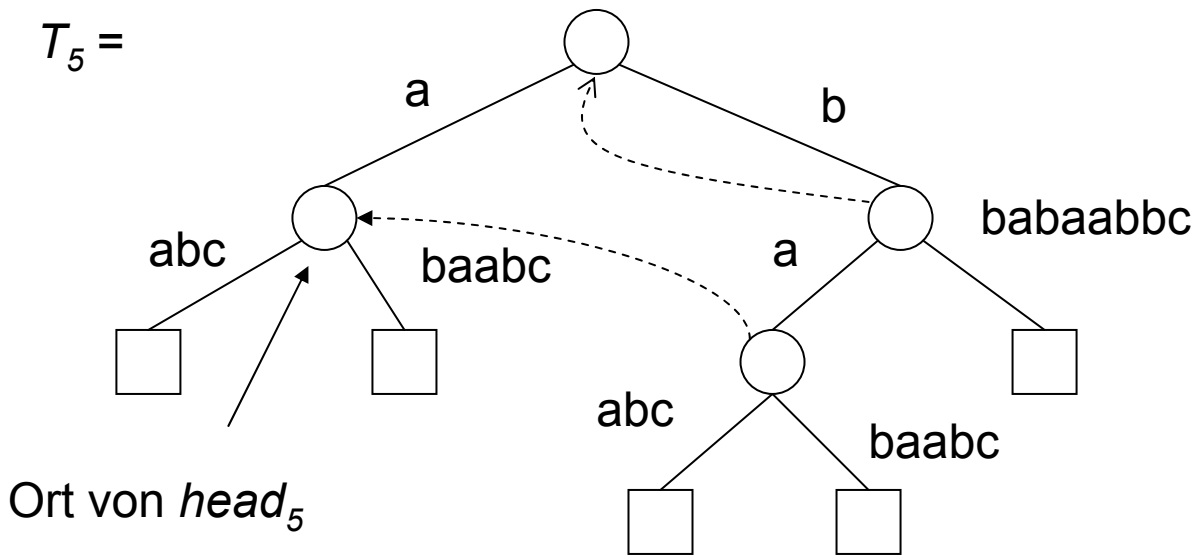
suf_4 = baabc
$head_4$ = ba

Suffix-Baum Beispiel



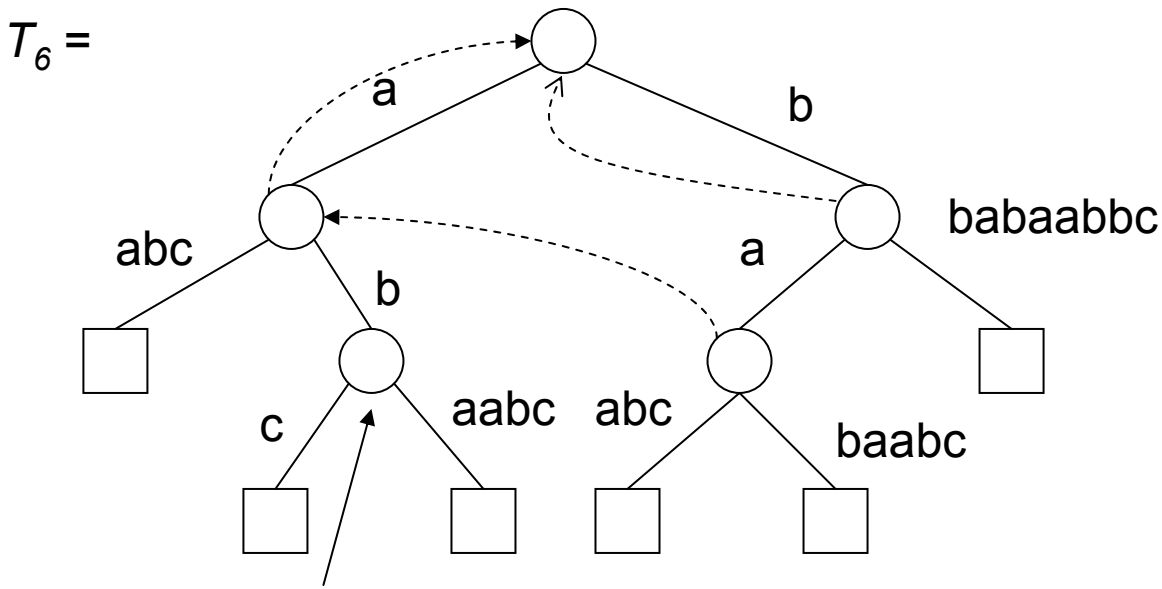
suf_5 = aabc
$head_5$ = a

Suffix-Baum Beispiel



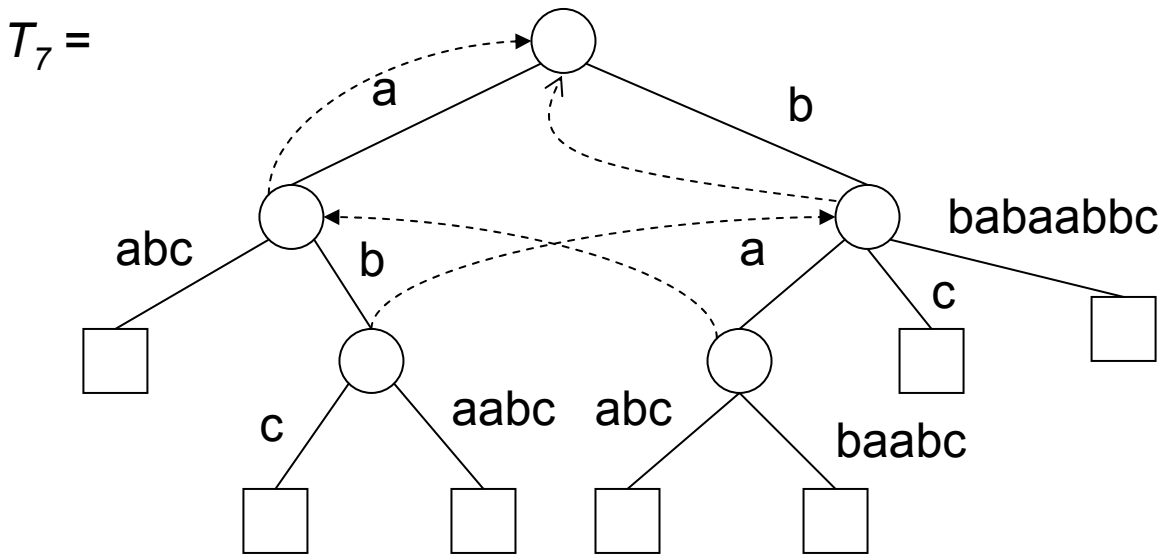
suf_6 = abc
$head_6$ = ab

Suffix-Baum Beispiel



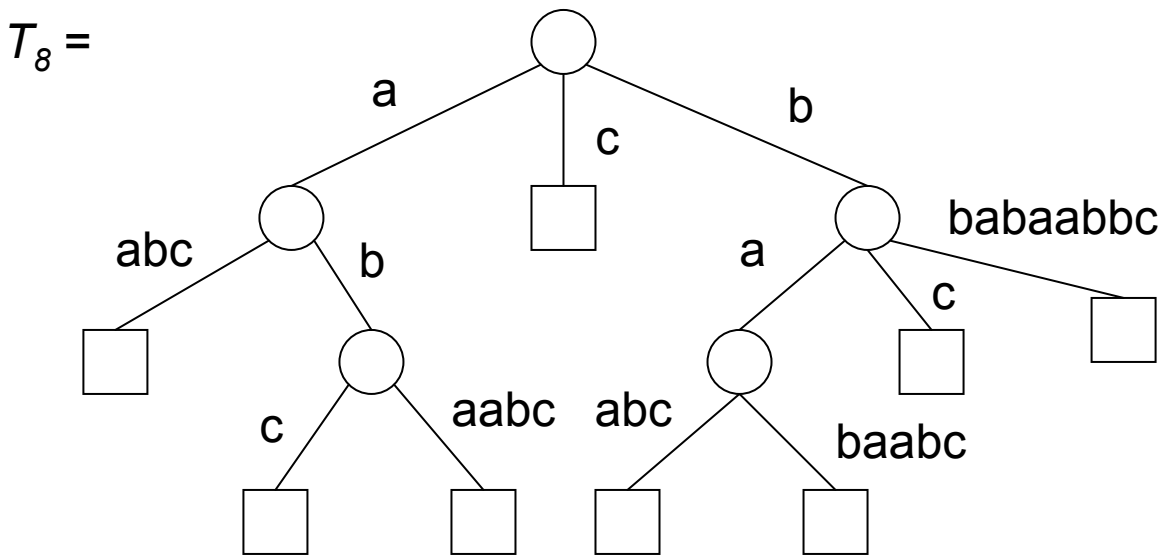
suf_7 = bc
$head_7$ = b

Suffix-Baum Beispiel



suf_8 = c

Suffix-Baum Beispiel

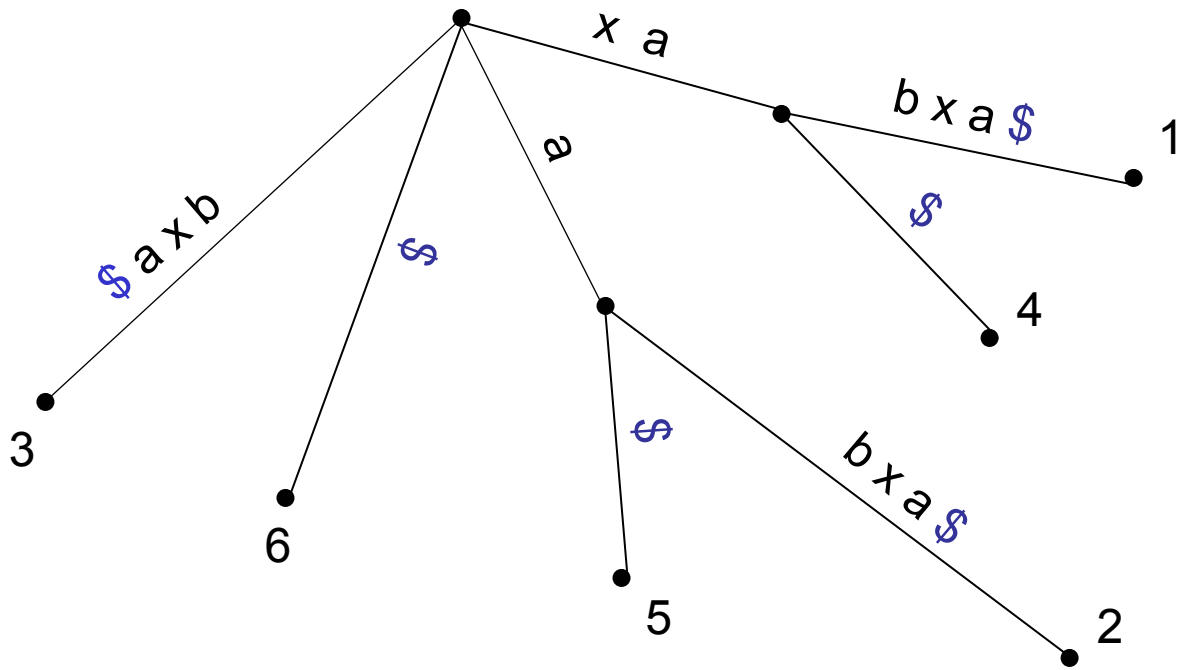


15 - Konstruktion von Suffix-Bäumen (3)

Prof. Dr. S. Albers

Suffix-Baum

t = x a b x a $
1 2 3 4 5 6



Ukkonens Algorithmus: Impliziter Suffix-Baum

Definition: Ein *impliziter Suffix-Baum* ist der Baum, den man aus dem Suffix-Baum für *t*\$ enthält, indem man

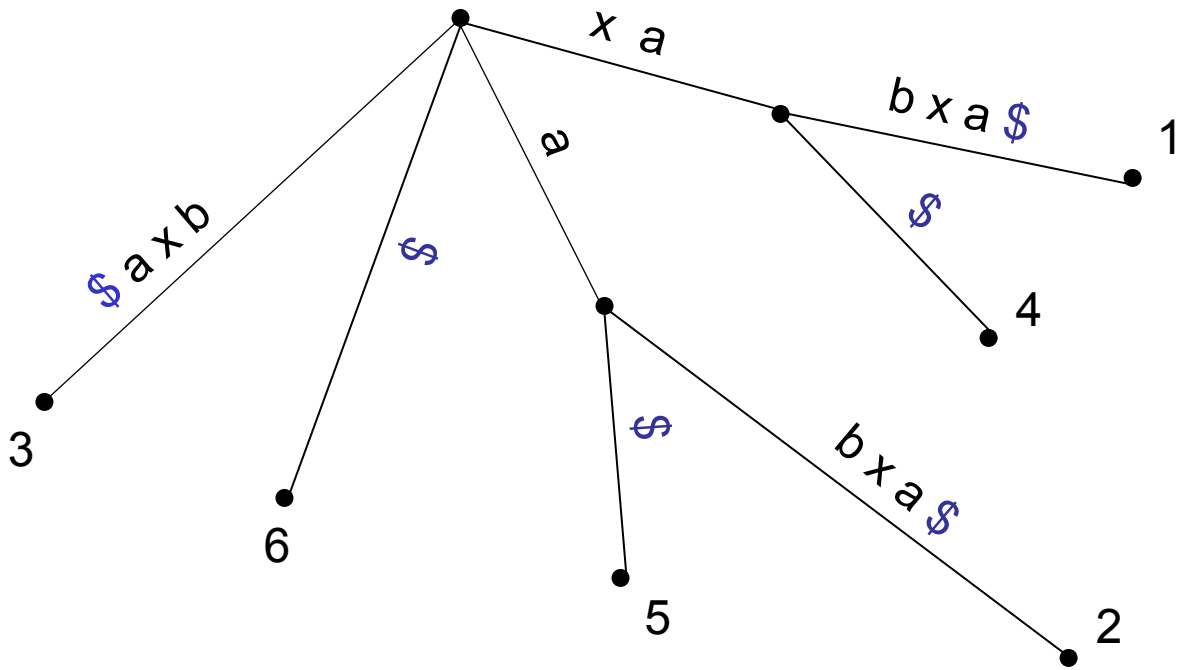
(1) \$ von den Markierungen der Kanten entfernt,

(2) Kanten ohne Markierung entfernt,

(3) Knoten mit nur einem Kind entfernt.

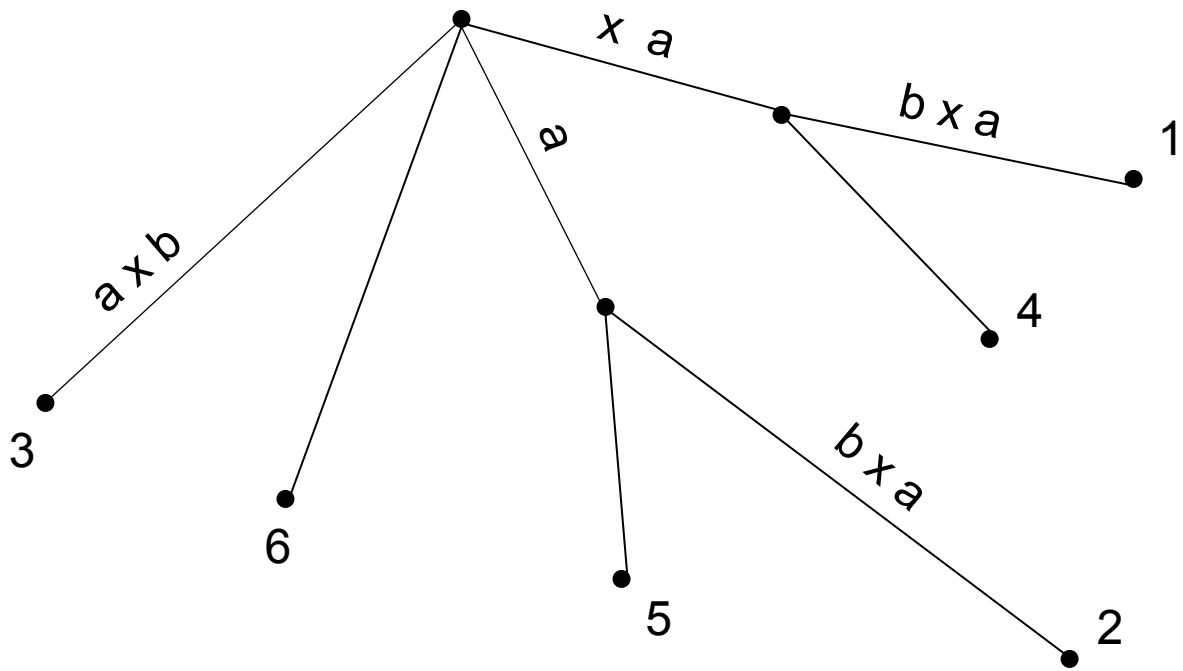
Ukkonens Algorithmus: Impliziter Suffix-Baum

t = x a b x a $
1 2 3 4 5 6



Ukkonens Algorithmus: Impliziter Suffix-Baum

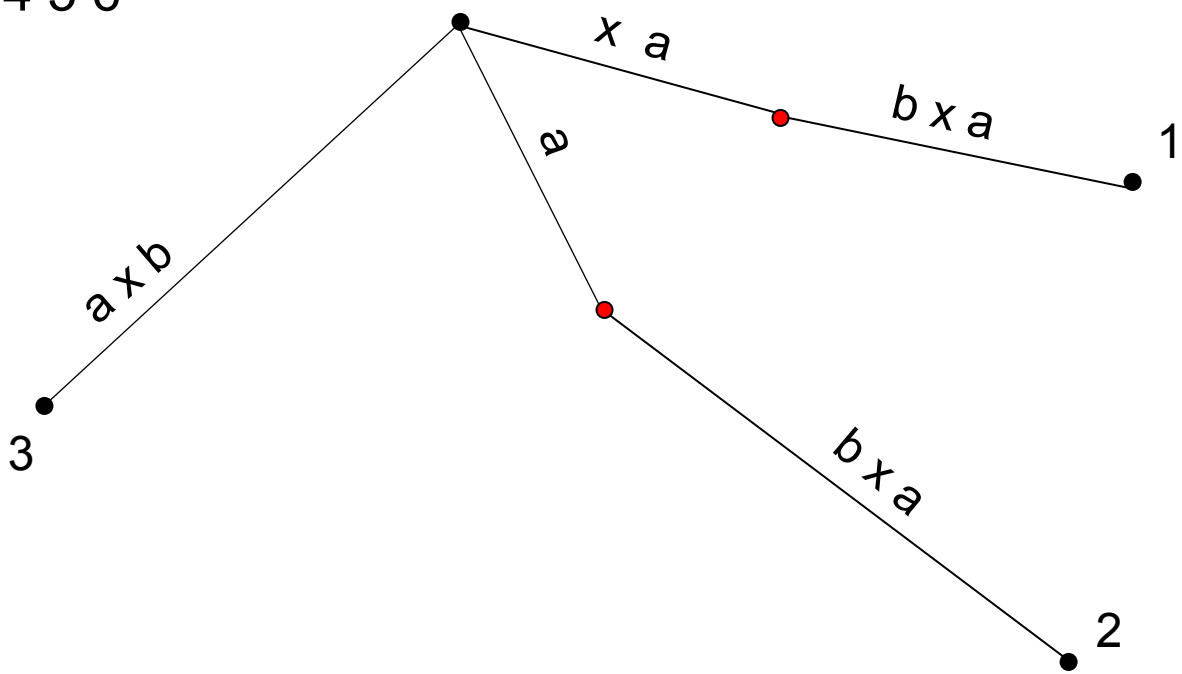
(1) Entfernen der Kantenmarkierung $ für t = x a b x a $



Ukkonens Algorithmus: Impliziter Suffix-Baum

(2) nicht markierten Kanten entfernen

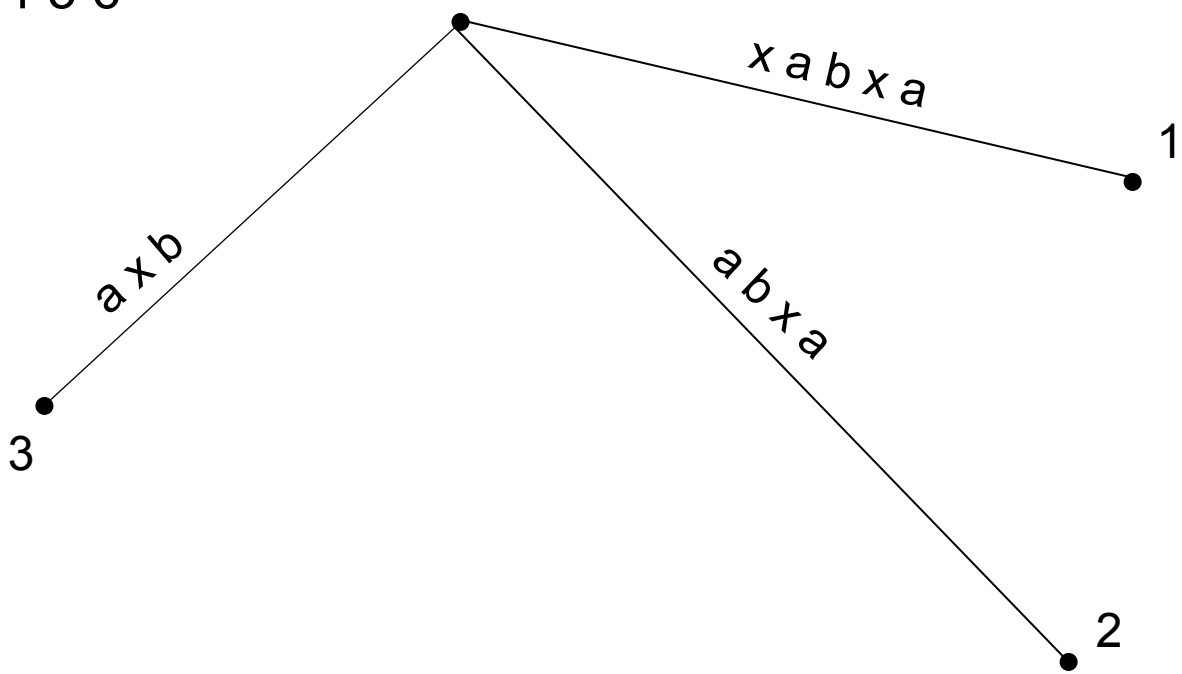
t = x a b x a $
1 2 3 4 5 6



Ukkonens Algorithmus: Impliziter Suffix-Baum

(3) Knoten mit nur einem Kind entfernen

t = x a b x a $
1 2 3 4 5 6



Ukkonens Algorithmus

Sei $t = t_1 t_2 t_3 \ldots t_m$

Ukk arbeitet online: Der Suffix-Baum $ST(t)$ wird schrittweise durch Konstruktion einer Reihe von impliziten Suffix-Bäumen für alle Präfixe von t konstruiert:

$$ST(\varepsilon),\ ST(t_1),\ ST(t_1 t_2),\ \ldots,\ ST(t_1 t_2 \ldots t_m)$$

$ST(\varepsilon)$ ist der leere implizite Suffix-Baum.

Er besteht nur aus der Wurzel.

Ukkonens Algorithmus

Die Methode wird *online* genannt, weil in jedem Schritt der implizite Suffix-Baum für ein Anfangsstück von *t* konstruiert wird, ohne den Rest des Inputstrings zu kennen.

Der Algorithmus arbeitet also inkrementell, da er den Inputstring zeichenweise von links nach rechts liest.

Ukkonens Algorithmus

Inkrementelle Konstruktion des impliziten Suffixbaumes:

Induktionsanfang: $ST(\varepsilon)$ besteht nur aus der Wurzel.

Induktionsschritt: Aus $ST(t_1 \ldots t_i)$ wird $ST(t_1 \ldots t_i t_{i+1})$, für alle $i < m$

- Sei T_i der implizite Suffix-Baum für $t[1...i]$
- Zuerst konstruiert man T_1 : Der Baum hat nur eine Kante, die mit dem Zeichen t_1 markiert ist.
- Die in Phase $i+1$ zu lösende Aufgabe ist, T_{i+1} aus T_i für $i=1$ bis $m-1$ zu konstruieren.

Ukkonens Algorithmus

Pseudo-Code Formulierung von Ukk:

Konstruiere Baum T_1

for *i* = 1 **to** *m*–1 **do**

begin {Phase *i*+1}

  for *j* = 1 **to** *i* +1 **do**

    begin {Erweiterung *j*}

      Finde im aktuellen Baum das Ende des Pfades von der Wurzel, der mit *t*[*j* ... *i*] markiert ist.

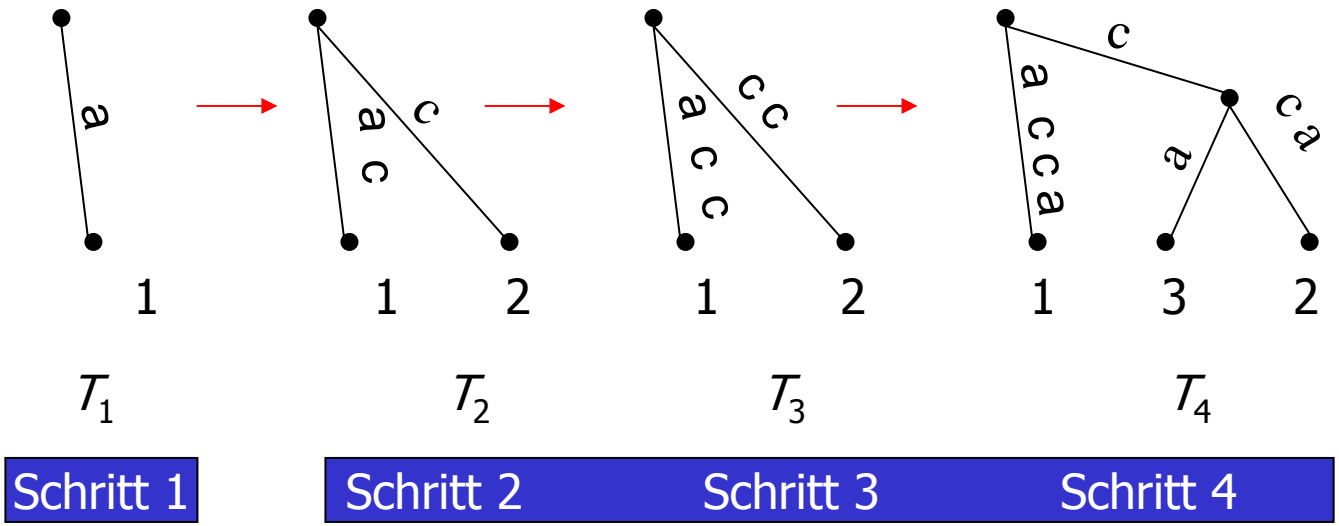
      Falls erforderlich, erweitere ihn durch Hinzufügen des Zeichens *t*[*i*+1], damit gewährleistet ist, dass der String *t*[*j*...*i*+1] im Baum ist.

      end;

end;

Ukkonens Algorithmus

t = a c c a $



Ukkonens Algorithmus

- Jede Erweiterung *j* wird so durchgeführt, dass man das Ende des mit *t*[*j*...*i*] gekennzeichneten Pfades von der Wurzel findet und den Pfad durch das Zeichen *t*[*i*+1] erweitert.

- In Phase *i*+1 wird zuerst der String *t*[1...*i*+1] in den Baum eingefügt, gefolgt von den Strings *t*[2...*i*+1] , *t*[3...*i*+1] ,.... (entsprechend den Erweiterungen 1,2,3,.... in Phase *i*+1).

- Erweiterung *i*+1 in Phase *i*+1 fügt das einzelne Zeichen *t*[*i*+1] in den Baum ein (es sei denn es ist schon vorhanden).

Ukk: Suffix-Erweiterungsregeln

Der Erweiterungsschritt *j* (in Phase *i*+1) wird nach einer der folgenden Regeln durchgeführt:

Regel 1: Wenn *t* [*j*...*i*] in einem Blatt endet, wird *t* [*i*+1] an die Markierung der zum Blatt führenden Kante angehängt.

Regel 2: Falls kein Pfad vom Ende von *t* [*j*...*i*] mit dem Zeichen *t* [*i*+1] anfängt, wird eine neue Kante zu einem neuen Blatt erzeugt, die mit dem Zeichen *t* [*i*+1] markiert wird.

(Das ist die einzige Erweiterung, die die Anzahl der Blätter im Baum erhöht! Das Blatt repräsentiert bei Position *j* beginnende Suffixe.)

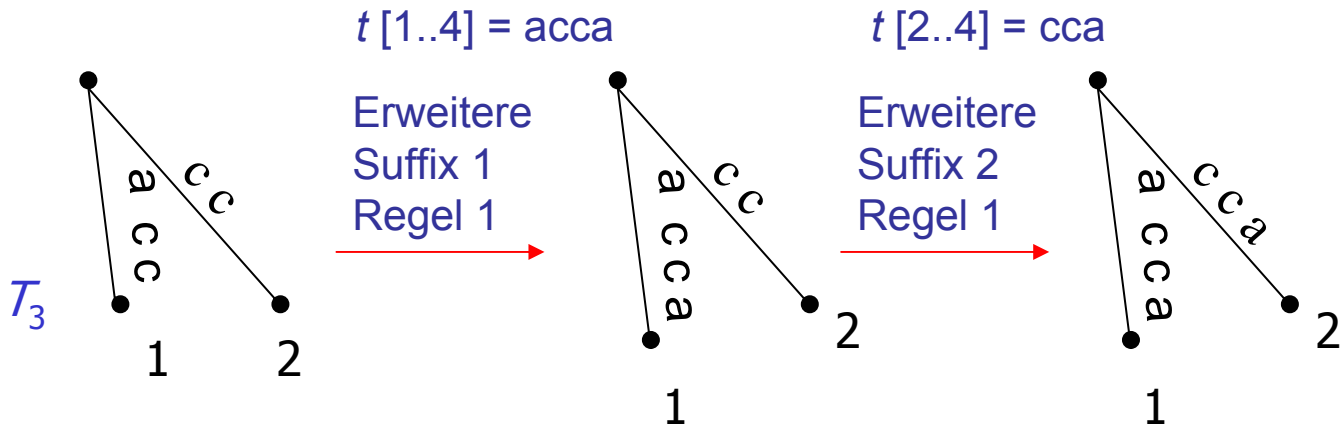
Regel 3: Falls ein mit dem Zeichen *t* [*i*+1] markierter Pfad am Ende von *t* [*j* ...*i*] beginnt, macht man nichts (denn dann tritt *t* [*j*…*i* +1] bereits im Baum auf).

Ukkonens Algorithmus

t = a c c a $

t [1...3] = acc

t [1...4] = acca



Ukkonens Algorithmus

Bei Durchführung von Phase i+1 (der Vorgang, durch den T_{i+1} aus T_i konstruiert wird) kann man beobachten:

(1) Sobald in Erweiterung j erstmals Regel 3 angewandt wird, muss der Pfad, der mit t [j...i] in T_i markiert ist, eine Fortsetzung mit Zeichen t [i+1] haben. Dann muss auch für jedes $j' \geq j$ der Pfad, der mit t [j'... i] markiert ist, ebenfalls eine Fortsetzung mit Zeichen t [i+1] haben.

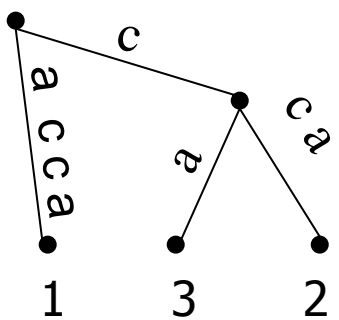
Daher wird Regel 3 auch für Erweiterungen $j' = j$+1,..., i+1 angewandt.

Wenn man also einmal Regel 3 für eine Erweiterung j in Phase i+1 angewandt hat, kann man diese Phase beenden.

Ukkonens Algorithmus

(2) Sobald man in T_i ein Blatt erzeugt, bleibt dies ein Blatt in allen Bäumen $T_{i'}$ für $i' > i$ (einmal ein Blatt, immer ein Blatt!), denn es gibt keine Regel für das Entfernen von Blättern.

t = a c c a b a a c b a ....



T_4

Ukkonens Algorithmus

Folgerung:

- Blatt 1 wird in Phase 1 erstellt. In jeder Phase *i*+1 gibt es eine Anfangssequenz von aufeinanderfolgenden Erweiterungen (angefangen bei Erweiterung 1), bei denen Regel 1 oder Regel 2 angewandt werden.

- Sei j_i die letzte Erweiterung in Phase *i*, die nach Regel 1 oder 2 erfolgt, d.h. jede Erweiterung *j´*, mit $j' > j_i$, erfolgt nach Regel 3.

  Dann gilt: $j_i \leq j_{i+1}$

Ukkonens Algorithmus

Folgerung:

- Wird in Phase *i*+1 bei Erweiterung *k* Regel 2 angewandt, so wurde in Phase *i* bei Erweiterung *k* Regel 3 angewandt.

Erweiterungen der Regel 1 können implizit durchgeführt werden!

Ukkonens Algorithmus

Beispiel:

Phase 1:	berechne Erweiterungen	1 ... j_1
Phase 2:	berechne Erweiterungen	j_1 +1 ... j_2
Phase 3:	berechne Erweiterungen	j_2 +1 ... j_3
....		
Phase *i*-1:	berechne Erweiterungen	j_{i-2} +1 ... j_{i-1}
Phase *i*:	berechne Erweiterungen	j_{i-1} +1 ... j_i

In jeden zwei aufeinanderfolgenden Phasen gibt es höchstens einen Index, der in beiden Phasen betrachtet werden muss, um die erforderlichen Erweiterungen durchzuführen.

Ukkonens Algorithmus

Der Algorithmus kann nun verbessert werden:

Die in Phase i+1 auszuführenden Erweiterungen j für j $\in$ [1, j_i] basieren alle auf Regel 1 und es wird nur konstante Zeit benötigt, um all diese Erweiterungen implizit durchzuführen.

Falls $j \in [j_i + 1, i+1]$ ist, sucht man das Ende des mit $t[j...i]$ markierten Pfades und erweitert ihn mit dem Zeichen t[i+1] indem man Regel 2 oder 3 anwendet.

Falls dabei Regel 3 angewandt wird, setzt man nur $j_{i+1} = j - 1$ und beendet Phase i+1.

Ukkonens Algorithmus

t = pucupcupu

i:	0	1	2	3	4	5	6	7	8	9
	ε	***p**	pu	puc	pucu	pucup	pucupc	pucupcu	pucupcup	pucupcupu
			***u**	uc	ucu	ucup	ucupc	ucupcu	ucupcup	ucupcupu
				***c**	cu	cup	cupc	cupcu	cupcup	cupcupu
					u	***up**	upc	upcu	upcup	upcupu
						p	***pc**	pcu	pcup	pcupu
							c	**cu**	**cup**	***cupu**
								u	up	***upu**
									p	**pu**
										u

- Suffixe, die zu einer Erweiterung nach Regel 2 führen, sind mit * markiert.
- Unterstrichene Suffixe markieren die letzte Erweiterung nach Regel 2.
- Suffixe, die eine Phase beenden (erstmalige Anwendung von Regel 3) sind blau markiert.

Ukkonens Algorithmus

- Solange der Algorithmus explizite Erweiterungen ausführt, merkt man sich im Index j^* den Index der gerade aktuell ausgeführten expliziten Erweiterung.

- Im Verlaufe der Ausführung des Algorithmus nimmt j^* niemals ab.

- Da es nur m Phasen gibt ($m = |t|$), und j^* durch m begrenzt ist, führt der Algorithmus daher nur m explizite Erweiterungen aus.

Ukkonens Algorithmus

Revidierte Pseudo-Code Formulierung von Uuk:

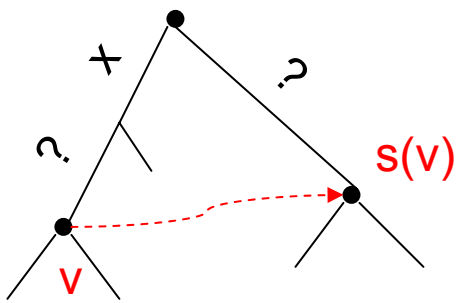
Konstruiere Baum T_1; $j_1 = 1$;
for $i = 1$ **to** $m - 1$ **do**
begin {Phase i+1}
    Führe implizite Erweiterungen durch.
    for $j = j_i + 1$ **to** $i + 1$ **do**
        begin {Erweiterung j}
        Finde das Ende des Pfades von der Wurzel mit Markierung $t[j \ldots i]$ im aktuellen Baum. Falls erforderlich, erweitere ihn durch Hinzufügung des Zeichens $t[i+1]$, damit gewährleistet wird, dass der String $t[j \ldots i+1]$ im Baum ist.
        $j_{i+1} := j$;
        if Regel 3 angewandt **then** $j_{i+1} := j - 1$ und beende Phase i+1;
        end;
end;

Ukkonens Algorithmus

Die Laufzeit kann noch weiter verbessert werden, wenn man Hilfszeiger (sog. Suffix-Links) ausnutzt.

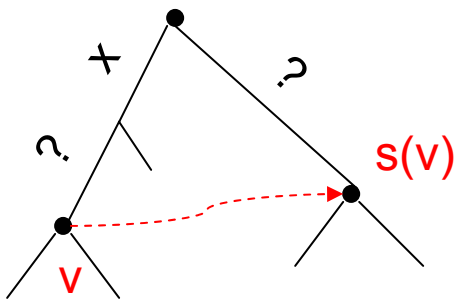
Definition: Sei *x*? ein beliebiger String, wobei *x* ein einzelnes Zeichen darstellt und ? einen (möglicherweise leeren) Teilstring.

Für jeden inneren Knoten *v* mit Kennzeichnung *x*? gilt: Falls es einen weiteren Knoten *s*(*v*) mit Pfad-Markierung ? gibt, so gibt es einen Zeiger von *v* auf *s*(*v*), der als Suffix-Link bezeichnet wird.



Ukkonens Algorithmus

Die Idee ist, Nutzen aus den Suffix-Links zu ziehen, um die Erweiterungspunkte effizienter, d.h. in amortisiert konstanter Zeit, zu finden, ohne bei jeder expliziten Erweiterung an der Wurzel beginnen zu müssen.



Ukkonens Algorithmus

Beispiel: Angenommen, man führt Erweiterung 3 in Phase 6 für

t = „acacag“ aus

Man muss also *t*[3...5] = „aca“ finden, indem man von der Wurzel herunterläuft und überprüft, ob „ acag“ eingefügt werden soll oder ob es bereits im Baum vorhanden ist. Dann muss man wieder von der Wurzel herunterlaufen für die Suffixe „cag“, „ag“ und „g“ und Regel 2 auf sie anwenden.

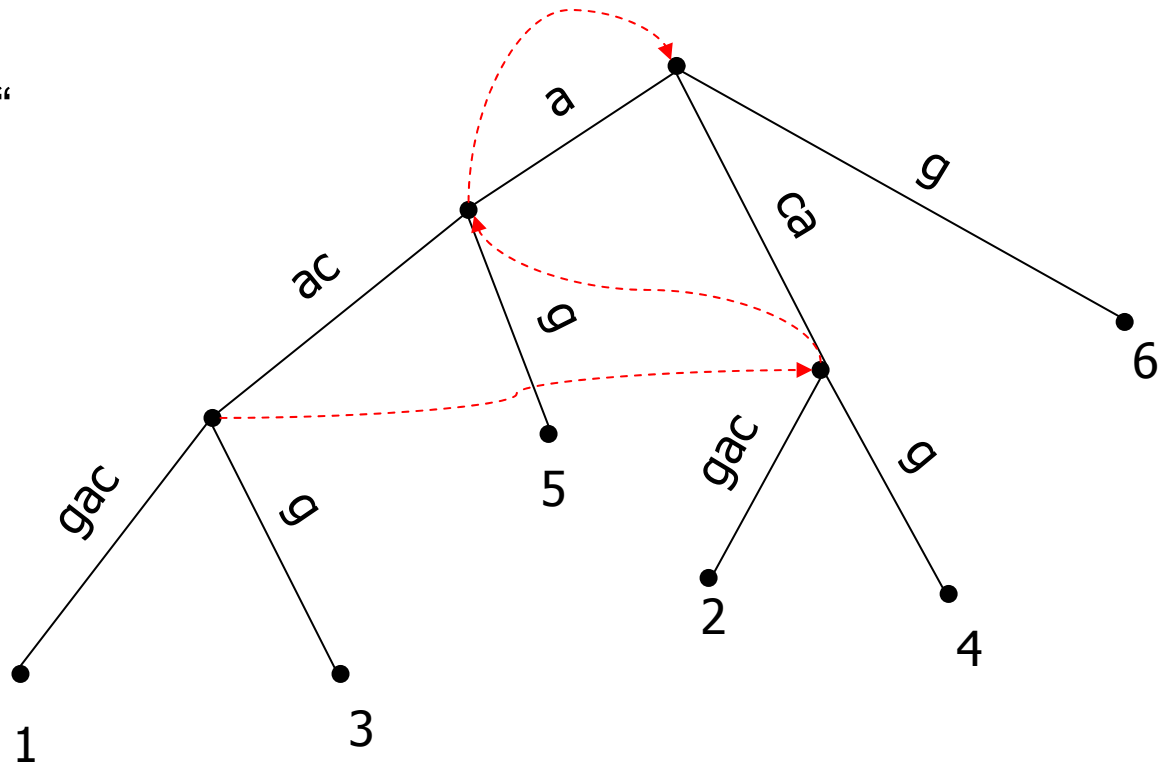
Ukkonens Algorithmus

Beispiel: Angenommen, man führt Erweiterung 3 in Phase 6 für

t = „acacag“ aus

Finde *t*[3 ...5] = „aca“
und erweitere:

„aca g“
„ca g“
„a g“
„g“



Ukkonens Algorithmus

- Mit der Hilfe von Suffix-Links werden die Erweiterungen basierend auf Regel 2 oder Regel 3 vereinfacht.
- Jede explizite Erweiterung kann in amortisierter Zeit O(1) ausgeführt werden (hier nicht gezeigt).
- Weil nur m explizite Erweiterungen durchgeführt werden, beträgt die Gesamtlaufzeit von Ukkonens Algorithmus $O(m)$ ($m = |t|$).

Ukkonens Algorithmus

Der echte Suffix-Baum:

Der endgültige implizite Suffix-Baum T_m kann in einen wirklichen Suffix-Baum in Zeit O(m) umgewandelt werden.

(1) Füge ein Terminalsymbol $ ans Ende vom String t ein.

(2) Lasse Ukkonens Algorithmus mit diesem Zeichen weiterlaufen.

Das Resultat ist der echte Suffix-Baum in dem kein Suffix ein Präfix eines anderen Suffix ist und jedes Suffix in einem Blatt endet.