

# Proseminar

## Effiziente Algorithmen

### Kapitel 3: Strings

Prof. Dr. Christian Scheideler

WS 2018

# Übersicht

- Grundlegende Notation
- Ein naiver Algorithmus
- Knuth-Morris-Pratt Algorithmus
- Aho-Corasick Algorithmus

# Grundlegende Notation

- **Alphabet**  $\Sigma$ : endliche Menge an **Symbolen**  
 $|\Sigma|$ : Kardinalität von  $\Sigma$
- **Wort (String, Zeichenkette)**  $s$ : endliche Folge von Symbolen über  $\Sigma$
- $|s|$ : Länge von  $s$
- $\varepsilon$ : leeres Wort, d.h.,  $|\varepsilon|=0$
- $\Sigma^n$ : Menge aller Worte über  $\Sigma$  der Länge  $n$   
 $\Sigma^0 = \{\varepsilon\}$
- $\Sigma^* = \bigcup_{i \geq 0} \Sigma^i$ : Menge aller Worte über  $\Sigma$
- $\Sigma^+ = \bigcup_{i \geq 1} \Sigma^i$ : Menge aller Worte über  $\Sigma$  außer  $\varepsilon$

# Grundlegende Notation

**Definition 1:** Seien  $s = s_1 \dots s_n$  und  $s' = s'_1 \dots s'_m$  Worte über  $\Sigma$ .

- $s'$  heißt **Teilwort** von  $s$  falls es ein  $i \geq 1$  gibt mit  $s' = s_i s_{i+1} \dots s_{i+m-1}$
- $s'$  heißt **Präfix** von  $s$  falls  $s' = s_1 s_2 \dots s_m$
- $s'$  heißt **Suffix** von  $s$  falls  $s' = s_{n-m+1} s_{n-m+2} \dots s_n$

Es gibt zwei Varianten des String Matching Problems.  
Gegeben zwei Worte  $s$  (das **Suchwort**) und  $t$  (der **Text**),

1. Bestimme, ob  $s$  ein Teilwort von  $t$  ist oder
2. Bestimme alle Positionen, ab denen  $s$  ein Teilwort von  $t$  ist.

# Grundlegende Notation

Beispiel: finde `avoctdfytv` in

`kvjlixapejrbxveenpphkhthbkwyrwamnugzhppfxiyjyanhapfwbghx  
mshrlyujfjhrsovkvveylnbxnawavgizyvmfohigeabgksfnbkmffxjdf  
ffqbualeytrphyrbjqjqavctgxjifqgfydhoiwhrvwqbxgrixyszdfss  
bpajnhopvlamhhfavoctdfytvvggikngkwzixgjtlxkozjlefilbrboiegwf  
gnbzsudssvqymnapbpqvlubdoyxkkwhcoudvtkmikansgsutdjyth  
apawlvliygjkmxorzeoafeoffbfxuhkzukeftnrfmocylculksedgrdsfe  
lvayjpgkrtedehwhrvvbbldkctq`

Im Allgemeinen ist `|t|>>|s|` (Google web search)

# Naiver Algorithmus

Input: Text  $t$ , Suchwort  $s$  ( $|t|=n$ ,  $|s|=m$ )

Algorithm SimpleSearch:

for  $i:=1$  to  $n-m+1$  do

$j:=1$

  while  $j \leq m$  and  $s[j]=t[i+j-1]$  do

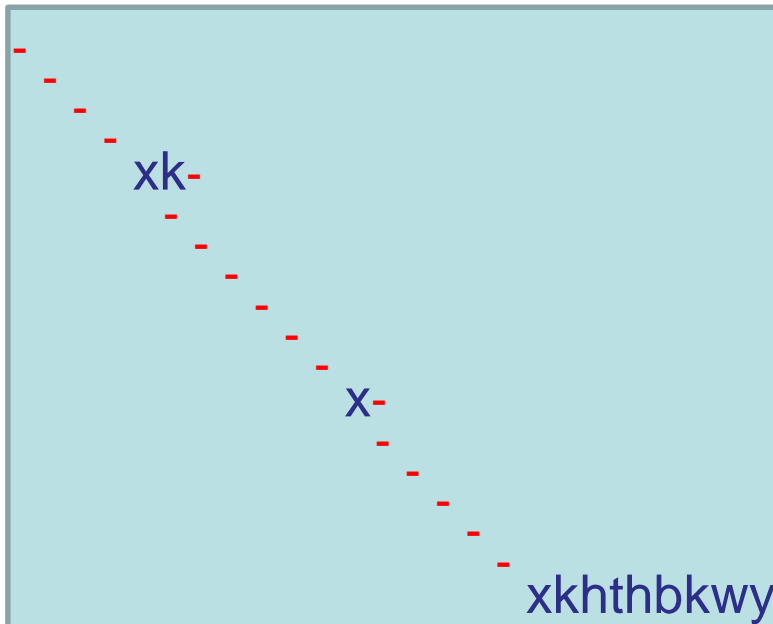
$j:=j+1$

  if  $j > m$  then output  $i$

# Naiver Algorithmus

Suchstring **s**: xkhthbkwy

Text **t**: kvavixkpejrbxeenppxkhthbkwy



Ist SimpleSearch immer gut?

Anzahl verglichener Symbole:  $n+3$





# Knuth-Morris-Pratt Algorithmus

**Beobachtung:** bei einem Mismatch beim  $i$ -th Symbol des Suchworts kennen wir die vorigen  $i-1$  Symbole im Text.

**Idee:** berechne im vornherein, wie wir dieses Wissen bestmöglich bei einem Mismatch ausnutzen können

**Beispiel:**

- Suchwort  $s$ : ababcab

- Text: ababa....

ababcab

ababcab (schiebe  $s$  um zwei für nächstmögl. Übereinstimmung, führe Abgleich bei aktueller Pos.  $a$  im Text fort)

# Knuth-Morris-Pratt Algorithm

## Im Allgemeinen:

- Angenommen,  $(s_1 \dots s_i) = (t_1 \dots t_i)$  aber  $s_{i+1} \neq t_{i+1}$ .
- Dann gehe zur ersten Position  $d$  in  $t$ , so dass  $(s_1 \dots s_{i-d+1}) = (t_d \dots t_i)$  und fahre mit dem Abgleich bei Position  $t_{i+1}$  fort.
- In diesem Fall gilt  $(s_1 \dots s_{i-d+1}) = (s_d \dots s_i)$ .
- Wir möchten alle möglichen Sprünge in einem Preprocessing berechnen.

## Ziel des Preprocessings:

- Finde für jede Position  $i$  in  $s$  das minimale  $d > 1$ , so dass  $(s_1 \dots s_{i-d+1}) = (s_d \dots s_i)$ . Gibt es kein  $d$ , setzen wir  $d = i + 1$ .
- Wir benennen das entsprechende  $d$  für  $i$  mit  $d_i$ .
- Die  $d_i$ 's werden in einem Feld gespeichert, so dass sie schnell zugreifbar sind.

# Knuth-Morris-Pratt Algorithm

Die  $d_i$ -Werte können effizient berechnet wie folgt werden:

Algorithm KMP-Preprocessing:

```
d0 := 2; d1 := 2 // Verschiebung von s um 1
j := d1 // aktuelle Verschiebung von s
for i := 2 to m do
  while j ≤ i and si ≠ si-j+1 do
    // (s1...si-j) = (sj...si-1) aber si-j+1 ≠ si
    j := j + di-j - 1
  di := j
```

# Knuth-Morris-Pratt Algorithm

Beispiel:  $s=ababaca$

i	0	1	2	3	4	5	6	7
$d_i$	2	2	3	3	3	3	7	7

**Satz:** Die Laufzeit des KMP-Preprocessing ist  $O(m)$ .

# Knuth-Morris-Pratt Algorithm

Algorithm KMP:

KMP-Preprocessing

$i:=1$  // aktuelle Position in  $t$

$j:=1$  // aktuelle Startposition von  $s$  in  $t$

while  $i \leq n$  do

    if  $j \leq i$  and  $t_i \neq s_{i-j+1}$  then

$j:=j+d_{i-j}-1$

    else

        if  $i-j+1=m$  then //  $s$  ab Pos.  $j$  gefunden

            output  $j$

$j:=j+d_m-1$

$i:=i+1$

# Boyer-Moore Algorithmus

- Im Gegensatz zum KMP Algo vergleicht der Boyer-Moore Algorithmus die Zeichen des Suchworts von rechts nach links.
- Dadurch hat der Boyer-Moore Algorithmus in der Praxis eine Laufzeit von  $O(n/m)$  statt  $O(n+m)$ . Er ist allerdings etwas aufwändiger zu implementieren (siehe den TUM-Spickzettel).


# Aho-Corasick Algorithmus

**Problem:** suche in Text  $t$  nach allen Positionen, in denen ein Suchwort in  $S = \{s_1, \dots, s_k\}$  startet.

Im folgenden sei  $m_i = |s_i|$  und  $m = \sum_{i=1}^k m_i$ .

**Erste Idee:** lass KMP Algorithmus parallel für alle Suchworte laufen.

**Laufzeit:**  $O(m+k \cdot n)$

  
Preprocessing      Hauptalgorithmus

# Aho-Corasick Algorithmus

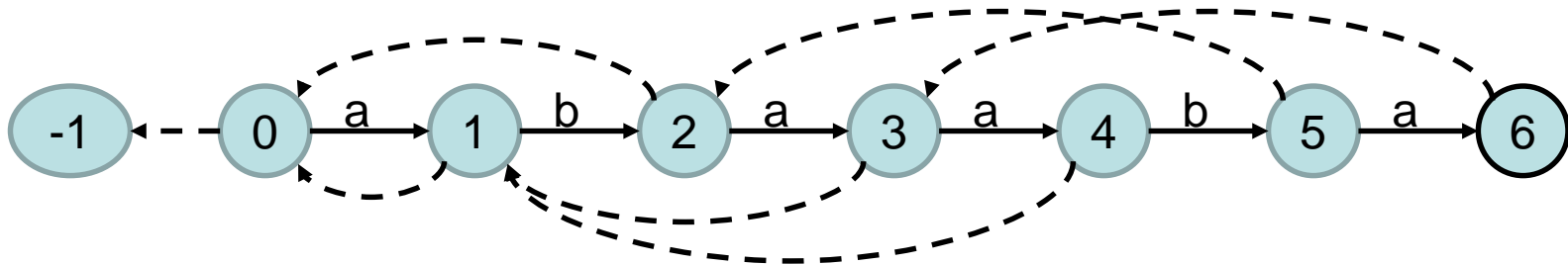
**Bessere Idee:** verwende anstelle von Tabellen mit  $d_i$ -Werten einen endlichen Automaten.

**Beispiel:** sei  $s=abaaba$

- Tabelle der  $d_i$ -Werte:

<b>i</b>	0	1	2	3	4	5	6
<b><math>d_i</math></b>	2	2	3	3	4	4	4

- Endlicher Automat:





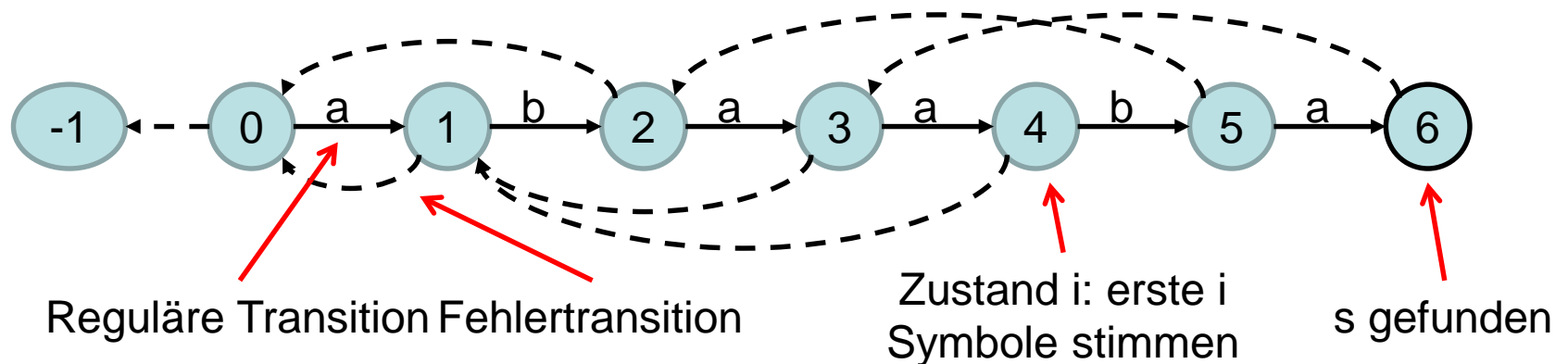
# Aho-Corasick Algorithmus

Beispiel: sei  $s=abaaba$

- Tabelle der  $d_i$ -Werte:

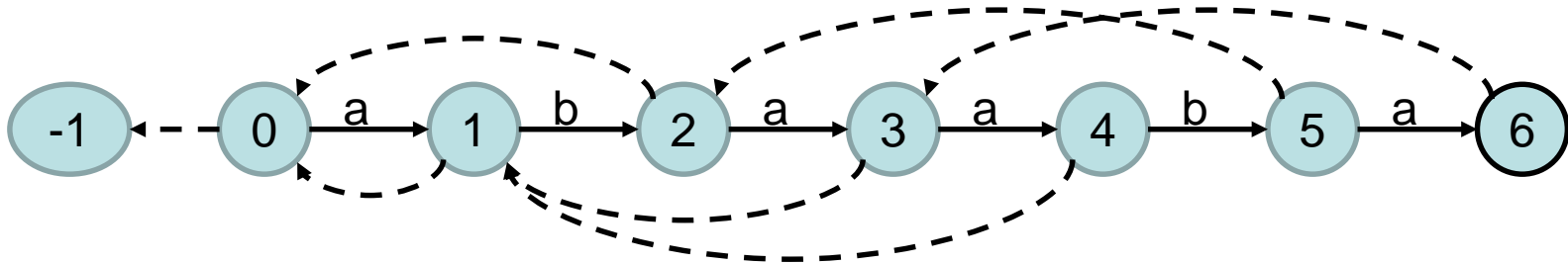
i	0	1	2	3	4	5	6
$d_i$	2	2	3	3	4	4	4

- Endlicher Automat:



# Aho-Corasick Algorithmus

Beispiel: sei  $s=abaaba$



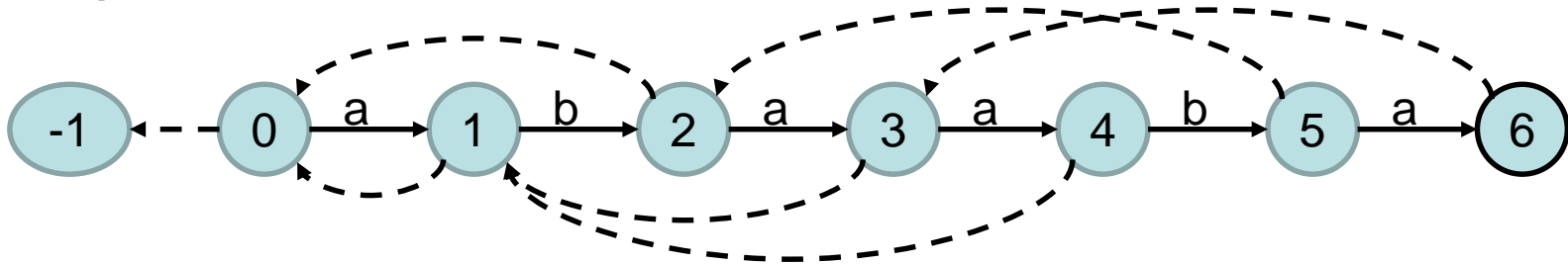
Dieser wird AC-Automat genannt.

**Definition:** Ein **AC-Automat** besteht aus:

- $Q$ : endliche Zustandsmenge
- $\Gamma = \Sigma \cup \{\text{fail}\}$ : endliches Alphabet (mit Eingabealphabet  $\Sigma$ )
- $\delta: Q \times \Gamma \rightarrow Q$ : Übergangsfunktion
- $q_0$ : Anfangszustand
- $F \subseteq Q$ : Menge akzeptierender Zustände

# Aho-Corasick Algorithmus

Beispiel: sei  $s=abaaba$



AC-Automat für  $s \in \Sigma^*$  mit  $|s|=m$ :

- $Q = \{-1, 0, 1, \dots, m\}$ ,  $q_0 = 0$  und  $F = \{m\}$
- $\Gamma = \Sigma \cup \{\text{fail}\}$
- Für alle  $i \in \{0, \dots, m-1\}$ ,  $\delta(i, s_{i+1}) = i+1$
- Für alle  $i \in \{0, \dots, m\}$ ,  $\delta(i, \text{fail}) = i - d_i + 1$

Die **Fehlertransition** wird benutzt, falls für gelesenes Symbol keine reguläre Transition existiert.

# Aho-Corasick Algorithmus

AC Preprocessing für ein einzelnes Suchwort  $s$ :

Algorithm AC-Preprocessing:

```
d0:=2; d1:=2 // Verschiebung von s um 1
j:=d1 // aktuelle Verschiebung von s
for i:=2 to m do
  while j ≤ i and si ≠ si-j+1 do
    // (s1...si-j)=(sj...si-1) aber si-j+1 ≠ si
    j:=j+di-j - 1
  di:=j
// berechne Fehlertransitionen f0,...,fm
for i:=0 to m do fi:=i-di+1
```

**Satz:** Das AC Preprocessing hat Laufzeit  $O(m)$ .

# Aho-Corasick Algorithmus

Aho-Corasick Algorithmus für ein Suchwort:

AC-Preprocessing

$j := 0$  // Startposition im Automaten

for  $i := 1$  to  $n$  do

    while ( $j \neq -1$  and  $t_i \neq s_{j+1}$ ) do

$j := f_j$

$j := j + 1$

    if  $j = m$  then output  $i - m + 1$

**Satz:** Der AC algorithm hat eine Laufzeit von  $O(n)$ .

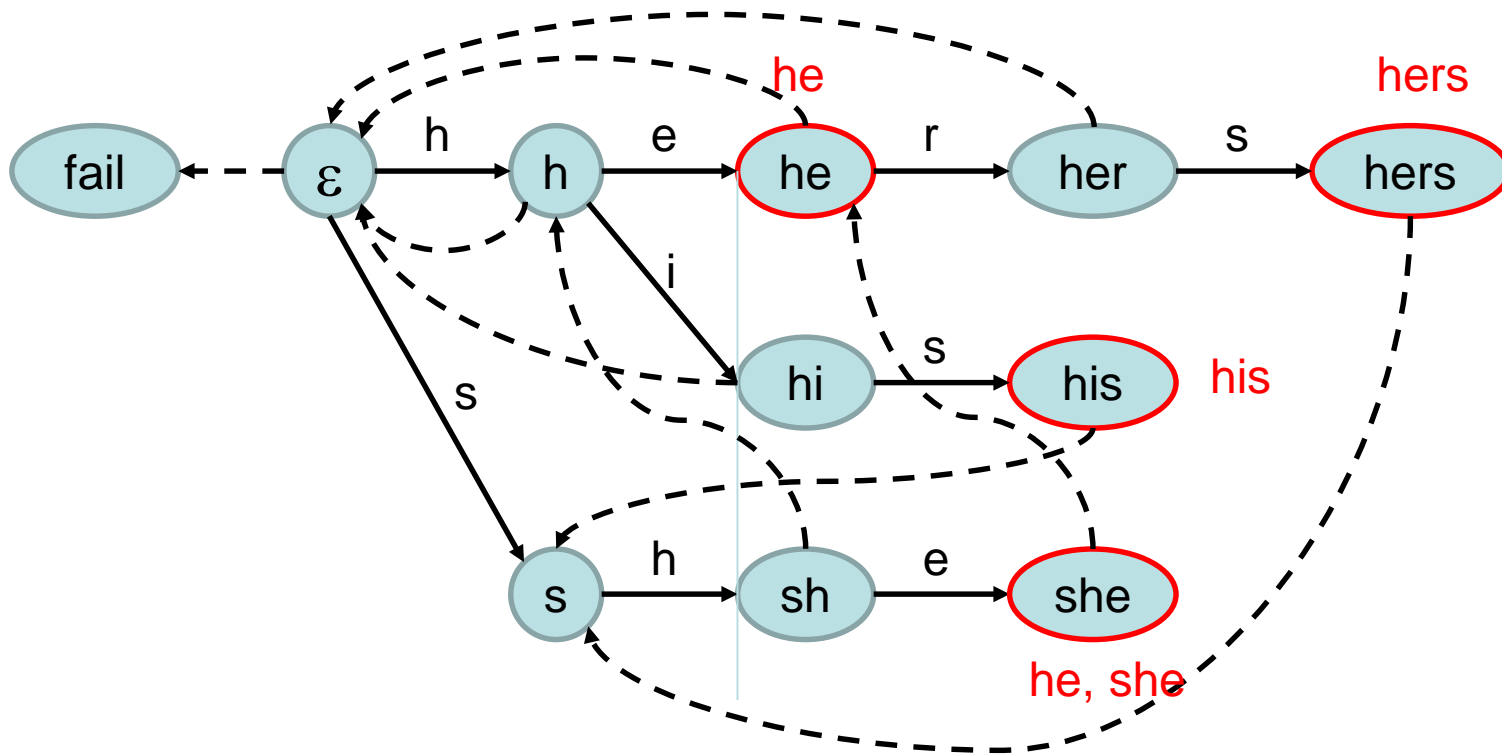
# Aho-Corasick Algorithmus

AC Automat für Menge  $S$  von Suchworten:

- $Q = \{ w \in \Sigma^* \mid w \text{ ist Präfix eines } s \in S \} \cup \{\text{fail}\}$  und  $q_0 = \varepsilon$
- $F = F_1 \cup F_2$  wobei
  - $F_1 = S$  und
  - $F_2 = \{ w \in \Sigma^* \mid \exists s \in S: s \text{ ist Suffix von } w \}$
- Für alle  $w \in Q$  und  $a \in \Sigma$  gilt:
  - $\delta(w, a) = w \circ a$  falls  $w \circ a \in Q$ , sonst
  - $\delta(w, \text{fail}) = w'$  für das  $w' \in Q$ , das das größte Suffix von  $w$  ist. Für  $w = \varepsilon$  ist  $\delta(w, \text{fail}) = \text{fail}$  (wobei „fail“ den Zustand repräsentiert, der vorher „-1“ war).

# Aho-Corasick Algorithmus

Beispiel:  $S = \{he, she, his, hers\}$



# Aho-Corasick Algorithmus

Aho-Corasick Algorithmus für eine Menge  $S$  an Suchworten:

- $m$ : Summe der Längen aller  $s \in S$
- $f_w$ : Zustand, der für  $\delta(w, \text{fail})$  erreicht wird
- $S_w$ : Menge aller  $s \in S$ , die ein Suffix von  $w$  sind

AC-Preprocessing2

$w := \varepsilon$  // Startposition im AC Automat

for  $i := 1$  to  $n$  do

    while ( $w \neq \text{fail}$  and  $\delta(w, t_i)$  is not defined) do

$w := f_w$

    if  $w = \text{fail}$  then  $w := \varepsilon$  else  $w := w \circ t_i$

    if  $w \in F$  then output  $(i, S_w)$

**Satz:** Der AC Algorithmus hat Laufzeit  $O(n+m)$ .



# Aho-Corasick Algorithmus

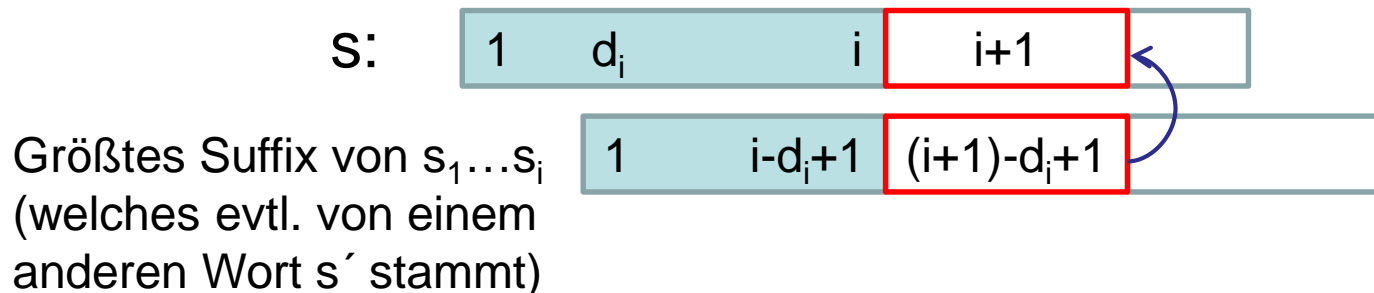
Der AC Automat für  $S$  kann in zwei Phasen konstruiert werden:

**Phase I:** konstruiere den Präfixbaum von  $S$  mit den regulären Transitionen (Zeit  $O(m)$ )

**Phase II:** berechne die Fehlertransitionen ab Zustand  $\varepsilon$  in der Reihenfolge wie bei der Breitensuche

Berechnung für Phase II ist ähnlich zum KMP Preprocessing:

- Betrachte den Zustand  $s_1 \dots s_{i+1}$  des AC Automaten. Starte mit der Fehlertransition für  $s_1 \dots s_i$  als das größtmögliche Suffix für die Fehlertransition von  $s_1 \dots s_{i+1}$ .



# Aho-Corasick Algorithmus

## Phase II:

- Initialization:
  - $f_\varepsilon := \text{fail}$
  - $f_a := \varepsilon$  for all  $a \in \Sigma$
- For all prefixes  $w \in \Sigma^*$  in BFS order:
  - $f_w := f_{\text{pred}(w)}$  //  $\text{pred}(w)$ :  $w$  ohne letztes Symbol
  - while ( $f_w \neq \text{fail}$  and  $\delta(f_w, \text{last}(w))$  undefined) do
    - //  $\text{last}(w)$ : letztes Symbol von  $w$
    - $f_w := f_{f_w}$
  - if  $f_w = \text{fail}$  then  $f_a := \varepsilon$  else  $f_w := \delta(f_w, \text{last}(w))$

Satz: AC Preprocessing2 hat Laufzeit  $O(m)$ .

# Aho-Corasick Algorithmus

Aho-Corasick Algorithmus für reguläre Ausdrücke (grundlegende Idee):

- Bilde endlichen Automat für regulären Ausdruck (siehe EBKFS)
- Füge Fehlertransitionen hinzu, so dass der Automat zurück zum Zustand gelangt, der dem längsten Suffix des Wortes repräsentiert durch den vorherigen Zustand entspricht.



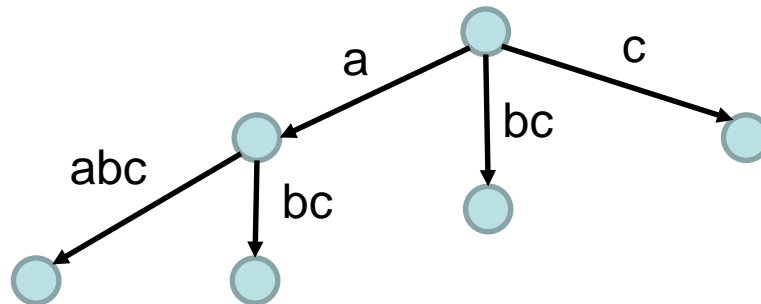
# Suffix Bäume

**Ziel:** bereite Text  $t$  so auf, dass für jedes Suchwort  $s$  schnell herausgefunden werden kann, ob  $s$  in  $t$  ist.

**Suffix Baum:** Baum, der alle Suffixe eines Textes  $t$  enthält

**Beispiel:** Sei  $t=aabc$

komprimiert:  
(Patricia trie)



Damit ist die Suche nach einem Wort  $s$  in Zeit  $O(m \log |\Sigma|)$  Zeit möglich, wobei  $|s|=m$  und  $\Sigma$  das Eingabealphabet ist.

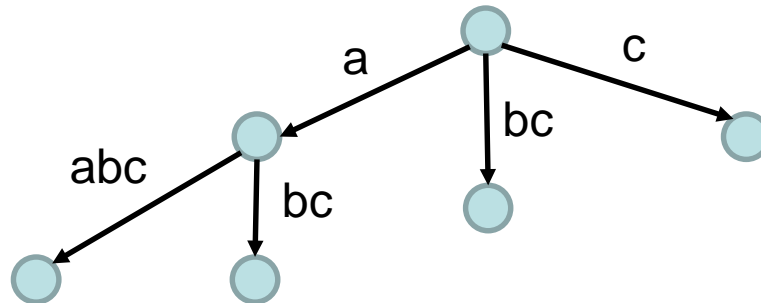
# Suffix Bäume

**Ziel:** bereite Text  $t$  so auf, dass für jedes Suchwort  $s$  schnell herausgefunden werden kann, ob  $s$  in  $t$  ist.

**Suffix Baum:** Baum, der alle Suffixe eines Textes  $t$  enthält

**Beispiel:** Sei  $t=aabc$

komprimiert:  
(Patricia trie)



**Problem:** evtl. hoher Speicherplatz (bis zu  $\Theta(n^2)$  )

Besser: **Suffix Arrays** (nur linearer Speicherplatz)

# Probleme

- 10252: Common Permutation
- 454: Anagrams
- 10340: All in All
- 760: DNA Sequencing
- 850: Crypt Kicker II
- 10010: Where's Waldorf
- 129: Krypton Factor

Hausaufgabe:

- 10132: File Fragmentation