Kap. 3: String Matching

Professor Dr. Petra Mutzel

Lehrstuhl für Algorithm Engineering, LS11

Fakultät für Informatik, TU Dortmund

7./10./12./13. VO A&D WS 08/09 4./18./25./27. November 2008

Literatur für diese VO

- T.H. Cormen, C.E. Leiserson, R. Rivest, C. Stein: Algorithmen – Eine Einführung, Oldenbourg, 2004 (oder das englische Original, MIT Press)

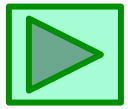
- P. Kilpelainen, University of Kuopio: Biosequence Algorithms, Lecture 4: Set Matching and Aho-Corasick Algorithm

A.V. Aho and M.J. Corasick: Efficient String Matching: An Aid to Bibliographic Search, Communications of the ACM, 1975

Eigener Skript-Teil (teilweise) (s. Web)

Literatur für diese VO

- P. Kilpelainen, University of Kuopio: Molecular Sequence Algorithms, Lecture 5: The Shift-And Method



Überblick

3.1 Einführung
- Motivation und Definitionen
- Naives Verfahren

3.2 Knuth-Morris-Pratt

3.3 Boyer-Moore

3.4 Aho-Corasick Algorithmus

3.5 Bit-Parallelismus

3.1 Einführung

String Matching (bzw. Pattern Matching):

- Suche einer gegebenen Zeichenfolge P (**Muster** bzw. ***pattern***) in einem Text T.
- Meist gilt: |P|<<|T|

Anwendungen:

- Suchfunktionen in Texteditoren, *grep* bei Unix
- Google in Web,
- Suche nach Mustern in DNS-Sequenzen
- Text Mining, ...

Definitionen

Gegeben:

- **Text** (Zeichenkette) $T_1...T_N$ von N Zeichen aus einem endlichen Alphabet Σ
- Pattern (kurze Zeichenkette) $P_1...P_M$ der Länge M, wobei $M<<N$

Problem: Finde alle Vorkommen von P innerhalb von T, d.h. jeweils die Anfangs-Indizes i ($1 \leq i \leq N-M+1$), so dass für $j=1,...,M$ gilt: $T_{i+j-1}=P_j$

Wir nehmen dabei an, dass der Text und das Muster in jeweils einem Feld von $T[1..N]$ bzw. $P[1..M]$ gespeichert ist.

Zwei verschiedene Szenarien

- Das Pattern ist vorgegeben, und dann sollen beliebige Texte durchsucht werden

  ➔ String Matching Algorithmen: 3.2-3.5

- Der Text ist vorgegeben, dann sollen beliebige Suchmasken im Text gefunden werden

  ➔ Aufbereitung des Texts: Indizierung, z.B. Suffix Arrays: Kap. 3.6

neben dem exakten gibt es auch approximatives Matching

Naives Verfahren

- **Idee:** Anlegen des Musters von links nach rechts an jeden Teilstring des Textes mit Länge M, und auf Übereinstimmung prüfen.

```
NAIVE_SEARCH(T,P)
1.  i = 0  // Position vor der Anlegestelle von P
2.  while (i ≤ N-M) {
3.     j = 1
4.    while ((j≤M) && (P[j]==T[i+j])) {  j=j+1 }
5.    if { j==M+1} then Ausgabe: an Stelle i+1 in T gefunden
6.    i=i+1
7.  }
```

Problem: Information, die durch Vergleiche erhalten wird, wird nicht benutzt, sondern vergessen.

- **Analyse:** Muster wird genau $N-M+1$ Mal an den Text angelegt. Im schlimmsten Fall pro Anlegen $\Theta(M)$ Vergleiche. Gesamt: Worst Case: $O(NM)$.

```
NAIVE_SEARCH(T,P)
1.  i = 0  // Position vor der Anlegestelle von P
2.  while (i ≤ N-M) {
3.     j = 1
4.    while ((j≤M) && (P[j]==T[i+j])) {  j=j+1 }
5.    if { j==M+1} then Ausgabe: an Stelle i+1 in T gefunden
6.    i=i+1
7.  }
```

3.2 Verfahren von Knuth-Morris-Pratt

- **Idee:** Nutze die Informationen, die wir jeweils bis zu einem Mismatch über den Text an den jeweiligen Stellen erhalten haben.
- Dabei wird versucht das Muster nach jedem Mismatch um mehr als nur eine Position nach rechts zu rücken.
- **Ziel:** Verhindere, dass Vergleiche noch einmal über bereits bekannte Zeichen geführt werden müssen.

Beispiel:

Position	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
Text 1	N	A	N	A	N	A		D	A	S
Muster	A	N	A	N	A	S				
Muster		A	N	A	N	A	S			

Verfahren von Knuth-Morris-Pratt

Wie weit können wir das Muster nach rechts schieben?

- Nach der Verschiebung muss garantiert sein, dass links von der aktuellen Position im Muster nur Zeichen stehen, die alle mit dem jeweiligen Zeichen im Text übereinstimmen
- Achtung: Muster nicht zu weit nach rechts schieben!

Berechnung allein aus P!

zu weit rechts

Beispiel:

Position	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
Text 1	N	A	N	A	N	A		D	A	S
Muster	A	N	A	N	A	S				
Muster		A	N	A	N	A	S			

Knuth-Morris-Pratt (KMP)

- **Annahmen:** Die letzten q gelesenen Zeichen im Text stimmen mit den ersten q Zeichen des Musters überein. Es sei P_q das Teilmuster von $P[1]$ bis $P[q]$
- Das gerade gelesene i-te Zeichen im Text ist verschieden vom $q+1$.-ten Zeichen im Muster

- Wir bestimmen vom Anfangsstück des Musters mit Länge q (dem gematchten Teil des Musters) ein Endstück (den **suffix(P_q)**) maximaler Länge $l<q$, das ebenfalls Anfangsstück (**prefix(P)**) des Musters ist.
- Dann kann das Muster um $q-l$ Stellen nach rechts geschoben werden.
- Position $l+1$ ist dann in M die erste Stelle die man mit dem i-ten Zeichen in T als nächstes vergleichen muss.

Beispiel

q	abababababca	l=next[q]	Shift nach rechts (q-l)
1	a #	0	1
2	ab #	0	2
3	aba #	1	2
4	abab #	2	2
5	ababa #	3	2
6	ababab #	4	2
7	abababa #	5	2
8	abababab #	6	2
9	ababababc #	0	9
10	abababababca #	1	9

Algorithmus Knuth-Morris-Pratt

```
Knuth-Morris-Pratt(T,P)
1. InitNext(P)   // Initialisiere das Feld next[]
2.  j = 0
3.  for (i = 1 to N) {
4.     while ((j>0) && (P[j+1]≠T[i])) {  j=next[j] }
5.     if ( P[j+1]==T[i] ) then  j=j+1
6.     if ( j==M) then { P gefunden: Ausgabe; j=next[j] }
7.  }
```

next[j] enthält die Position aus P, die unter die i-te Position in T kommt

Prozedur InitNext(P)

```
InitNext(P)        // Initialisiert das Feld next[]
1.  next[1]=0; l=0
2.  for (q = 2,..., M) {
3.     while ((l>0) && (P[l+1]≠P[q])) {  l=next[l] }
4.     if ( P[l+1]==P[q] ) then  l=l+1
5.     next[q]=l
6.  }
```

Korrektheit

Behauptung: **`InitNext()`** ist korrekt

- **Fall 1:** $P[l+1]==P[q]$ in Zeile (3). In diesem Fall ist das Endstück des Musters P_q das Anfangsstück von P_q ist, das bisherige (von $=q-1$) vereinigt mit $P[q]$.
- **Fall 2:** $P[l+1]\neq P[q]$ in Zeile (3). Dann geht man am besten genauso vor wie im KMP: man vergleicht $P[q]$ mit $next[l]$, $next[next[l]]$,..., bis man bei 0 angekommen ist.
- Dann ist KMP auch korrekt.

Analyse der Laufzeit

- Index i wird genau N Mal und der Index j maximal N Mal erhöht.
- Index j kann allerdings höchstens so oft zurückgesetzt werden, wie er erhöht wurde, also insgesamt maximal N Mal.
- Gleiches Argument für **`InitNext(P)`**
- Dort kann l höchstens so oft zurückgesetzt werden, wie es insgesamt erhöht wurde, also maximal M.
- **Gesamtlaufzeit von KMP:** $\Theta(N+M)$, dabei Vorverarbeitung $\Theta(M)$ und Suche $\Theta(N)$

Bemerkungen

- KMP durchläuft den Text ausschließlich sequentiell: der Index i wird niemals zurückgesetzt.
- Manchmal ist dies von Vorteil, wenn z.B. der Text auf externen Medien gespeichert ist (z.B. Bänder).

ENDE

3.3 Algorithmus von Boyer-Moore

Idee:

- **Last-Verschiebung:** hängt vom Zeichen im Text ab, das für den Mismatch verantwortlich ist. Wenn es z.B. im Muster nicht auftaucht (wahrscheinlich, da $M<<N$), dann kann P um die ganze Länge verschoben werden.
- Dazu: P wird von links nach rechts an T angelegt, aber von **rechts nach links** gelesen.
- Dies allein würde jedoch immer noch Laufzeit $O(NM)$ ergeben. Deswegen außerdem noch:
- **Suffix-Verschiebung:** Verschiebe so weit, bis die letzten gematchten Zeichen im Text mit den ersten Zeichen in P übereinstimmen.

Die beiden Verschiebungen sind voneinander unabhängig.

Algorithmus von Boyer-Moore

Bestimme Maximum der beiden Verschiebungen:

- **Last-Verschiebung:** Sei zum ersten Mal $P[j] \neq T[i+j]$ für ein j, $1 \leq j \leq M$. Das Zeichen an dieser Stelle im Text sei c. Verschiebe P so weit nach rechts, bis c im Text über dem rechtesten Zeichen gleich c in P ist. Falls c nicht in P vorkommt, dann kann P bis hinter die Position $i+j$ verschoben werden.
- **Suffix-Verschiebung:** Verschiebe so weit, bis die letzten gematchten Zeichen im Text mit den ersten Zeichen in P übereinstimmen.

Die beiden Verschiebungen sind voneinander unabhängig.

Algorithmus von Boyer-Moore

```
Boyer-Moore(T,P)
1.InitLast(P)        // Initialisiere das Feld last[]
2.InitSuffix(P)      // Initialisiere das Feld suffix[]
3.  i = 0            // Position vor der Anlegestelle von P
4. while (i ≤N-M) {
5.      j=M
6.      while ((j>0) && (P[j]==T[i+j])) {  j=j-1 }
7.      if { j==0} then  {
8.          P gefunden: Ausgabe
9.          i = i+suffix[0]
10.     }
11.     else { i = i+max( suffix[j] , j-last[T[i+j]] ) }
12.}
```

Berechnung von last[]

- **Lemma:** Sei k der größte Index aus $1 \le k \le M$, für den gilt $T[i+j]=P[k]$, falls vorhanden, sonst $k=0$. Dann ist es korrekt, i um $j-k$ zu erhöhen.

- **Beweis: Fall 1:** $k=0$. Dann sind alle anderen Zeichen in P links von j ungleich $T[i+j]$ ➔$i=i+j$ ✔
- **Fall 2:** $k<j$. Das rechteste Erscheinen des Mismatch-Zeichens c in T liegt links von j in P. In diesem Fall ist eine Verschiebung um $j-k>0$ nach rechts korrekt.✔
- **Fall 3:** $k>j$. In diesem Fall ist $j-k<0$ und es wird keine Verschiebung (wegen $last$) durchgeführt. ✔

Prozedur InitLast(P)

`InitLast(P)`

1. **for all** c aus dem Alphabet: last[c] = 0
2. **for all** j=1,...,M: **last[P[j]]** = j

Berechnung von suffix[]

- **Idee:** Verschiebe P soweit nach rechts, bis die bisher untersuchten gematchten Zeichen im Text mit den erst-nächsten passenden Zeichen im Muster übereinstimmen.
- Falls keine Übereinstimmung herrscht, dann schiebe das Muster bis ganz hinter Position $i+j$ im Text.

- Wir definieren **suffix[j]** als die kleinste Verschiebung s, so dass $P[j+1-s]=P[j+1], ..., P[M-s] = P[M]$.

- Alternative Sichtweise: **suffix[j]** = $\min(M-k, 0 \leq k < M)$, so dass $P[j+1],...,P[M]$ gleich dem Suffix von P_k ist, wobei $P_k=P[1]...P[k]$.

Berechnung von suffix[]

- **Fall 1:** **suffix[j]** ist am Anfang von P. Dann gilt $suffix[j]=M-\pi[M]$ für alle j, wobei $\pi[M]$ die Länge des längsten Präfixes von P ist, das echter Suffix von $P_M=P$ ist.

- **Fall 2:** **suffix[j]** ist nicht am Anfang von P. Dann kann der Suffix im invertierten Muster P^{-1} mit Hilfe von $next[]$ ausgerechnet werden. Es gilt $suffix[j] \leq q-next[q]$ für alle $1 \leq q \leq M$ und $j=M-next[q]$.

- **Intuitiv:** wir suchen alle diejenigen Längen q, deren Endungen mit dem gesuchten Präfix übereinstimmen. Das sind die mit $next[q]=M-j$.

Berechnung von suffix[]

- **Fall 1:** **suffix[j]** ist am Anfang von P. Dann gilt $suffix[j]=M-\pi[M]$ für alle j, wobei $\pi[M]$ die Länge des längsten Präfixes von P ist, das echter Suffix von $P_M=P$ ist.

- **Fall 2:** **suffix[j]** ist nicht am Anfang von P. Dann kann der Suffix im invertierten Muster P^{-1} mit Hilfe von $next[]$ ausgerechnet werden. Es gilt $suffix[j] \leq q-next[q]$ für alle $1 \leq q \leq M$ und $j=M-next[q]$.

$$suffix[j] = \min(\{M-\pi[M]\} \cup \{q-next[q]: 1 \leq q \leq M \text{ und } j=M-next[q]\})$$

Prozedur InitSuffix(P)

```
InitSuffix(P)
1.Berechne InitNext(P)➔next[]//Fall 1
2.Berechne InitNext(P^-1)➔next[]//Fall 2
3.  for all j=0,...,M  suffix[j] = M - next[M]
4.  for all q=1,...,M {
5.      j = M - next[q]
6.      if suffix[j] > q - next[q] then {
7.          suffix[j] = q - next[q]
8.      }
9.  }
```

Analyse von Boyer-Moore

Theorem: Die Laufzeit von Boyer-Moore für die einmalige Suche nach P ist $O(N+M)$ plus einer Vorverarbeitungszeit von $O(M+|\Sigma|)$.

Beweis:

- **last[]**-Berechnung: $O(M+|\Sigma|)$
- **suffix[]**-Berechnung: $O(M)$
- BM: ohne Beweis

Analyse von Boyer-Moore

- Die Laufzeitschranke gilt nur, wenn **beide** Verschiebeverfahren verwendet werden.
- Verschiebung mittels **last[]** ist sehr einfach und effektiv, die **suffix[]**-Verschiebung ist etwas aufwändiger. Deswegen in der Praxis oft ohne die **suffix[]**-Verschiebung.
- In beiden Fällen ist der Algorithmus in der Praxis meist sehr schnell.

3.4 Algorithmus von Aho-Corasick

Exaktes Matching mit mehreren Mustern

- Seien T ein Text der Länge N und P eine Menge von Pattern $P=\{P_1,P_2,...,P_z\}$ der Gesamtlänge $\sum|P_i|=M$.
- **Gesucht:** Alle Vorkommen der P_i in T.

- Naive Anwendung von z Mal KMP ergibt Laufzeit $O(zN+M)$
- **Ziel:** Laufzeit $O(N+M+k)$, wobei k die Anzahl der Vorkommen der Pattern im Text entspricht.
- Beachte: k kann echt größer sein als $N+M$.

Beispiel: Sei $T=aa...a$ und $P=\{\underbrace{aa...a}_{N/2},\underbrace{aa...a}_{N/4},...\}$

Dann gilt: $|P|=O(N)$ und

Anzahl der Vorkommen: $N/2+3N/4+... = O(N \log N)$

$\log_2 N$ Summanden

3.4 Algorithmus von Aho-Corasick

Exaktes Matching mit mehreren Mustern

- Seien T ein Text der Länge N und P eine Menge von Pattern $P=\{P_1,P_2,...,P_z\}$ der Gesamtlänge $\sum|P_i|=M$.
- **Gesucht:** Alle Vorkommen der P_i in T.

- Naive Anwendung von z Mal KMP ergibt Laufzeit $O(zN+M)$
- **Ziel:** Laufzeit $O(N+M+k)$, wobei k die Anzahl der Vorkommen der Pattern im Text entspricht.
- Beachte: k kann echt größer sein als $N+M$.

Idee: Mischung aus Knuth-Morris-Pratt mit endlichen Automaten

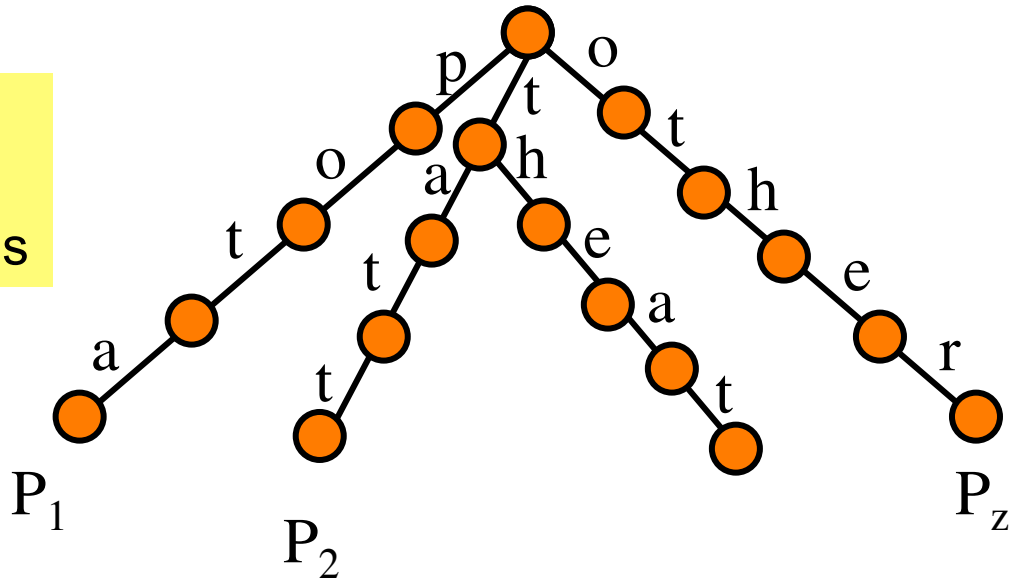
Dazu nötig: Keyword-Baum (engl. *Trie*) für Patternmenge P

Keyword-Baum (Trie)

Def.: Die Datenstruktur **Keyword-Baum** (**engl.** ***Trie***) **K** ist ein gerichteter Baum mit Wurzel r und Eigenschaften 1-4:

1. Die Kanten des Baums tragen Zeichen aus dem Alphabet
2. Je 2 Kanten, die von einem Knoten ausgehen, tragen verschiedene Zeichen
3. Jeder Weg von r zu einem Blatt entspricht einem Muster P_i.
4. Für jedes Muster gibt es einen Knoten v mit $L(v)=P$.

$L(v)$ heißt „Label von v“ und ist die Konkatenation aller Zeichen entlang des (r,v) Weges



Trie für $P=\{pota,tatt,theat,other\}$

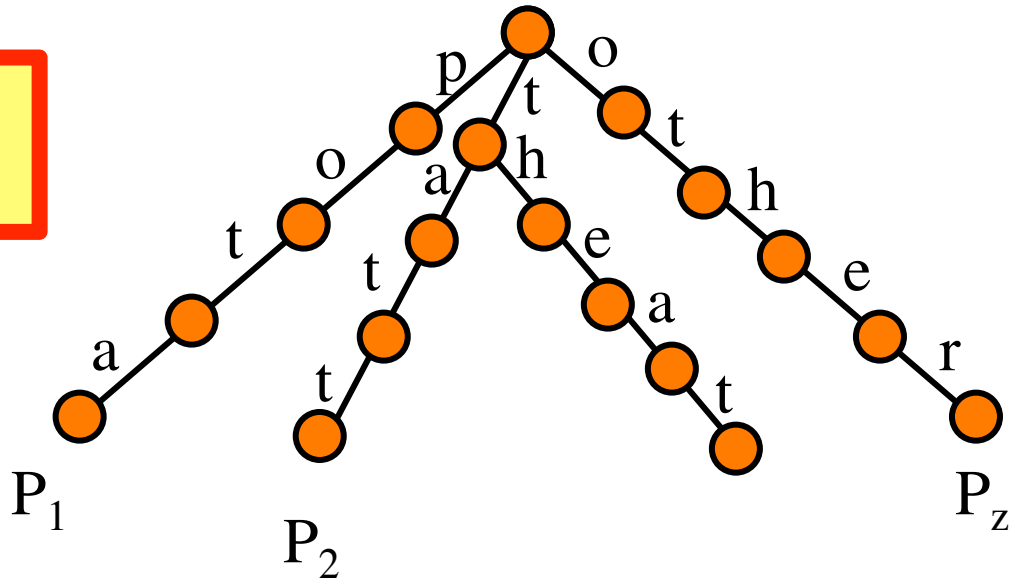
Konstruktion des Tries

Starte mit Wurzel r und füge die Pattern nacheinander hinzu:

Beginne bei Wurzel und folge dem Weg entlang der Zeichen von P_i:

- Falls der Weg vor P_i zu Ende ist, vervollständige den Weg durch Hinzufügen der fehlenden Zeichen von P_i
- Speichere einen Identifier i von P_i am Endknoten des Weges.

Die **Konstruktion** geht in Zeit $O(|P_1|+...+|P_z|)=O(M)$.



Trie für P={pota,tatt,theat,other}

Suchen im Trie

Suchen eines P_i im Trie K:

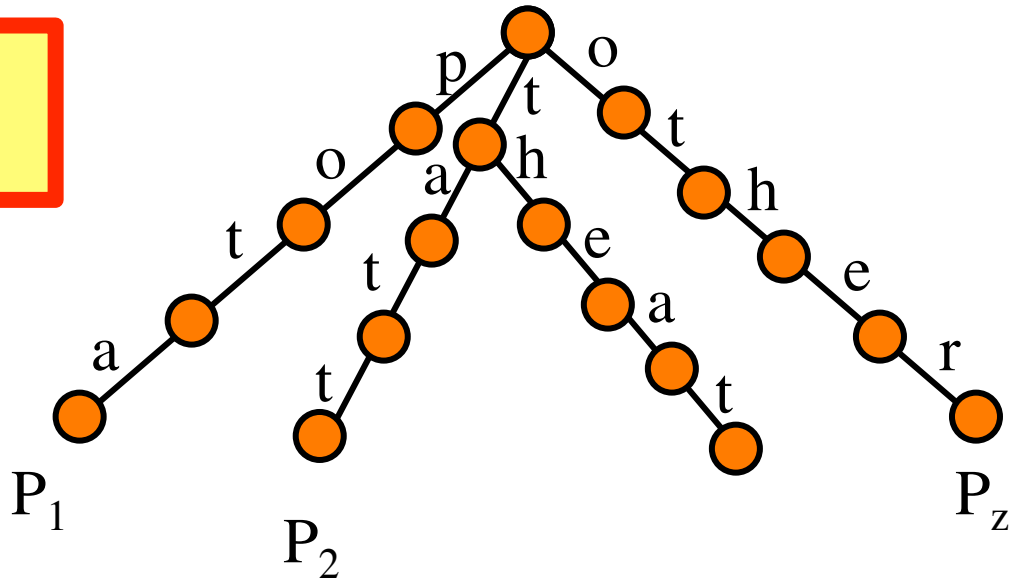
Starte mit Wurzel r und folge dem Weg der Zeichen von P_i so lange wie möglich:

- Falls der Weg an einem Knoten mit Identifier endet, ist P_i in K enthalten
- Falls der Weg endet bevor P_i zu Ende ist, dann ist der String nicht in K enthalten.

Die **Suche** eines Musters geht in Zeit $O(|P_i|)$.

JETZT: Erweiterung zu endlichem Automat.

Trie für P={pota,tatt,theat,other}



Algorithmus von Aho-Corasick

Idee:

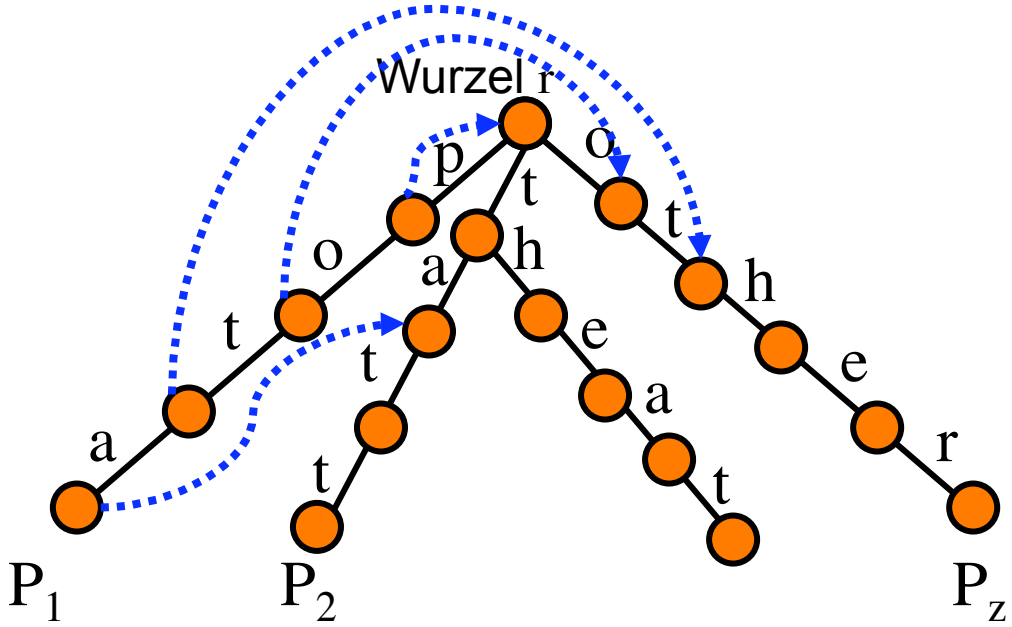
- Teste den Trie gegen den Text (analog zu KMP)
- Wie weit kann nach einem Mismatch verschoben werden?
- Dazu: Vorverarbeitung des Trie: setze **Fehler-Links**

Definitionen: Sei $v \in K$ ein Knoten und $L(v)$ das Label von v.

- Dann sei α der längste echte Suffix von $L(v)$, der Präfix eines Musters aus P ist.
- Wir setzen Fehler-Links n_v für jedes $v \in K$, wobei n_v=Knoten in K mit $L(n_v)=\alpha$.

Beispiel: Trie

mit Fehler-Links am linken Ast



Definitionen: Sei $v \in K$ ein Knoten und $L(v)$ das Label von v.

- Dann sei α der längste echte Suffix von $L(v)$, der Präfix eines Musters aus P ist.
- Wir setzen Fehler-Links n_v für jedes $v \in K$, wobei n_v=Knoten in K mit $L(n_v)=\alpha$.

Aho-Corasick Automat

- **Zustände:** Knoten des Trie K
- **Anfangszustand:** 0 = Wurzel
- **Aktionen** durch die Funktionen: Übergangsfunktion ***goto***, Fehlerfunktion ***failure***, Ausgabefunktion ***output*** (wie folgt).

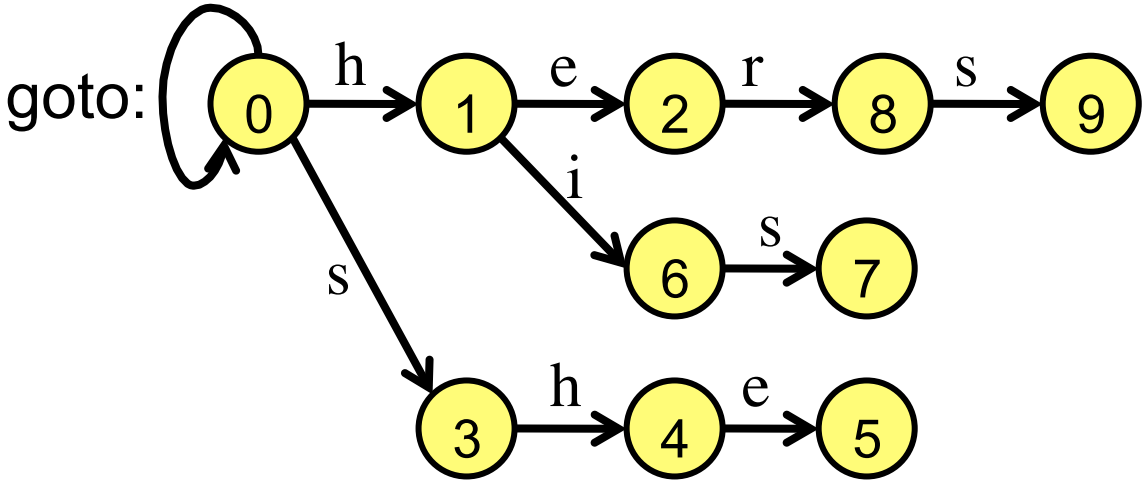
- **goto Funktion g(q,a):** gibt den Zustand nach aktuellem Zustand q und *matching* Zeichen a wie folgt:
- Falls die Kante (q,v) mit a markiert ist, dann $g(q,a):=v$
- Sonst:
  - Falls $q=0$: $g(0,a)=0$ für jedes Zeichen a, das keine ausgehende Kante von 0 markiert.
  - Sonst ($q\neq 0$): $g(q,a)=\emptyset$

Aho-Corasick Automat

- ***failure* Funktion $f(q)$ für $q \neq 0$:** gibt den Zustand nach einem Mismatch
- $f(q)$ ist der Knoten mit dem längsten echten Suffix α von $L(q)$, so dass α Präfix eines Patterns P_i ist
- das ist immer definiert, weil $L(0)=\varepsilon$ ist Präfix jedes Patterns
- kein Vorkommen wird verpaßt

- ***output* Funktion $out(q)$:** gibt die Menge der Patterns, die beim Eingang in den Zustand q akzeptiert werden.

Beispiel: P={he,she,his,hers}



failure:

i	1	2	3	4	5	6	7	8	9
f(i)	0	0	0	1	2	0	3	0	3

output:

i	2	5	7	9
output(i)	{he}	{she,he}	{his}	{hers}

AC Suche im Text T[1..N]

```
1.  q=0                                          // Anfangszustand
2.  for i:=1 to N {
3.      while g(q,T[i])==∅ do q=f(q)             // fail
4.      q=g(q,T[i])                              // goto
5.      if out(q)≠∅ then { Ausgabe von i und out(q) }
6.  }
```

Beispiel: Suche nach „ushers“

Analyse der AC Suche

Theorem: Die Suche in einem Text $T[1...N]$ mit einem AC Automaten geht in Zeit $O(N+z)$, wobei z die Anzahl der Vorkommen der Patterns ist.

Beweis: Für jeden Index über T gibt es 0 oder mehrere *fails* und jeweils ein *goto*.

- Jedes *goto* bleibt entweder bei r oder erhöht die Tiefe von q durch 1 ➔ die Tiefe von q wird $\leq N$ Mal erhöht
- Jedes *fail* bringt q näher an r ➔ Anzahl der *fails* ist begrenzt durch $\leq N$
- Die z Vorkommen können in Zeit $O(z)$ ausgegeben werden (*pattern identifiers* und Startpositionen).

Konstruktion eines AC Automaten: Phase 1

- Konstruiere den Trie für P
- Für jedes $P_i \in P$, das zum Trie hinzugenommen wird, setze $out(v) := \{P_i\}$ für alle Knoten v, die mit P_i markiert sind
- Vervollständige die *goto* Funktion für r durch $g(0,a) := 0$ für alle restlichen Zeichen $a \in \Sigma$

Phase 1 läuft in Zeit $\Theta(M+|\Sigma|)$.

Konstruktion eines AC Automaten: Phase 2

```
(1) Q = emptyQueue()
(2) for a∈Σ do
(3)    if g(0,a)≠0 then { q=g(0,a); f(q)=0; enqueue(q,Q) }
(4) while not isEmpty(Q) do {
(5)    r = dequeue(Q)
(6)    for a∈Σ do {
(7)       if g(r,a) ≠ ∅ then {
(8)          u=g(r,a); enqueue(u,Q); v=f(r) //r ist Elter von u
(9)          while g(v,a)==∅ do v=f(v)
(10)         f(u)=g(v,a)
(11)         out(u) = out(u) ∪ out(f(u))
(12)} } }
```

Erläuterung zu Phase 2

- Die Fehlerfunktion $f()$ sowie die Ausgabefunktion $out()$ werden mittels BFS ermittelt ➔die Knoten näher an Wurzel sind bereits berechnet
- Betrachte Knoten r und $u=g(r,a)$ (r ist Elter von u und $L(u)==L(r)a$) sowie $v=f(r)$
- Falls $g(v,a)\neq\emptyset$, dann hat auch v als Nachfolger ein „a“, d.h. die Fehlerfunktion wird auf $g(v,a)$ gesetzt.
- Sonst: Springe auf $v=f(v)$ (der nächstgrößte Präfix α eines Musters, der Suffix von $L(v)$ ist)
- v und $g(v,a)$ können beide $r=0$ sein
- ➔$f(u)$ wird korrekt berechnet

Komplexität der Phase 2

- Phase 2 kann mit Laufzeit $\Theta(M\ |\Sigma|)$ implementiert werden.

Beweis: BFS Traversierung des Baumes➔$O(M)$

- Die **while**-Schleife (Zeile 4) wird maximal M Mal durchlaufen
- Die **for**-Schleife wird genau $|\Sigma|$ Mal durchlaufen
- Die **while**-Schleife (Zeile 9) wird insgesamt maximal M Mal durchlaufen, denn der Abstand zur Wurzel verringert sich immer und man kann nur soviel zurücklaufen, wie man jeweils vorgelaufen ist.)

Laufzeit des AC Algorithmus

- **Theorem:** Der AC Algorithmus läuft bei konstanter Alphabetgröße Σ in Zeit $O(N+M)$.

Beweis: folgt aus bisherigen Betrachtungen.

Anwendung: Matching mit Wildcards

Definition: Eine Wildcard ist ein Zeichen $\Phi \notin \Sigma$, das mit jedem Zeichen $a \in \Sigma$ übereinstimmt (*matcht)*.

Aufgabe: Exaktes Matching eines Musters P_Φ mit *Wildcards* gegen einen Text T ohne *Wildcards*.

Idee: Das Muster hat die Form $P=\Phi\Phi P_1 \Phi\Phi ... \Phi P_2 \Phi ... \Phi P_k \Phi$, wobei die P_i Teilmuster ohne *Wildcards* sind und seien l_i deren Endpositionen in P.

- Anwendung des Aho-Corasick Algorithmus wie folgt:

Anwendung: Matching mit Wildcards

- Starte mittels AC-Automaten die Suche nach der Patternmenge $P=\{P_1,...,P_k\}$
- Sei C Array der Länge $|T|$: wir addieren bei einem Matching eines P_j an der Position $i \geq l_j$ im Text in C an der Stelle $C(i-l_j+1)$ (Anfangsposition) eine +1.
- D.h. jedes Auftreten eines P_j an Stelle i dient als Zeuge einer evtl. Anfangsposition von ganz P bei $i-l_j+1$.
- Liegen am Ende k Zeugen an einer Stelle i in C vor, so ist dies Anfangsposition von P. Oder kurz:
- $C(i)=k \Leftrightarrow P$ startet in $T(i)$

Beispiel: P_Φ=ΦATCΦΦTCΦATC, P={ATC,TC,ATC} mit $l_1=4, l_2=8, l_3=12$ und Text T= ACGATCTCTCGATC...
➔C[1]=C[7]=C[11]=1, C[3]=3

Analyse

Annahme: Die Alphabetgröße ist konstant.

- **Theorem:** Der Aho-Corasick Algorithmus in Verbindung mit *Wildcards* läuft in Zeit $O(N+|P_\Phi|+z)$, wobei z die Anzahl der Vorkommen von Mustern P_i aus P bezeichnet.
- Falls die Anzahl der *Wildcards* konstant ist, dann ist die Laufzeit $O(N+M)$.

- AC läuft in Zeit $O(M+z)$, wenn $M=\Sigma|P_i|$.
- Die Länge von C ist $|T|$, und es wird höchstens $z \leq kN$ Mal hochgezählt, k ist konstant.

3.5 Bit-Parallelismus

- **Idee:** Aktualisiere an jeder Position j im Text T während des Durchlaufs den **Zustandsvektor** (**Bitvektor**) $S_j[1..m]$ der Länge m.
- $S_j[1..m]$ kodiert die Existenz aller Präfixe von P, der bei der aktuellen Textposition j endet:
- $S_j[i]=1 \quad \Leftrightarrow \quad P[1..i] = T[j-i+1..j]$ für alle $i=1,...,m$

- **Beispiel:** P=„ENNEN“, T=„HENNENFUTTER“, $j=6$:
- Präfixe von P, die bei j enden: „en“, „ennen“
- Zustandsvektor: $S_6[1..5] = [0,1,0,0,1]$ für alle $i=1,...,N$

- P in T an Pos. j gefunden $\Leftrightarrow$ Zustandsvektor $S_j[m] = 1$

Berechnung des Zustandsvektors

$S_j[1..m]$: Spalten der Matrix

	T	H	E	N	N	E	N	F	U	T	T	E	R
P		1	2	3	4	5	6	7	8	9	10	11	12
E	1	0	1	0	0	1	0	0	0	0	0	1	0
N	2	0	0	1	0	0	1	0	0	0	0	0	0
N	3	0	0	0	1	0	0	0	0	0	0	0	0
E	4	0	0	0	0	1	0	0	0	0	0	0	0
N	5	0	0	0	0	0	1	0	0	0	0	0	0

- $M[i,j]=1 \quad \Leftrightarrow \quad P[1..i]$ ist gleich Suffix von $T[1..j]$

- $M[i,j]=1 \quad \Leftrightarrow \quad P[1..i] = T[j-i+1..j]$
- $\Leftrightarrow \quad P[1..i-1]=T[j-i+1..j-1]$ und $P[i]=T[j]$
- $\Leftrightarrow \quad M[i-1,j-1]=1$ und $P[i]=T[j]$

- Berechne $M[]$ spaltenweise: $j=1,...,N$

Schnelle Berechnung von S[1..m]

- **Annahme:** $|P| \leq w$, wobei w der Prozessorwortlänge entspricht (üblicherweise: 32 oder 64 Bit)
- **Dann** entspricht der Zustandsvektor einer einzigen Zahl ➔Aktualisierung ist schnell!
- ➔Berechnung von $M[,]$ geht dann in Zeit $\Theta(N)$

- Aktualisierung des Zustandsvektors in einem Schritt:
- Berechne für jedes Zeichen $a \in \Sigma$ eine Erscheinens-Maske (Vektor) $U[a]$ für Muster P:
- $U[a,i] = 1$, falls $P[i]=a$ und 0 sonst (für $i=1,...,m$)
- $U[\ ]$ kann in Zeit $\Theta(|\Sigma|\ m)$ berechnet werden

Schnelle Berechnung von S[1..m]

Aktualisierung des Zustandsvektors in einem Schritt:

- Berechne für jedes Zeichen $a \in \Sigma$ eine Erscheinens-Maske **U[a]** für Muster P:
- U[a,i]= 1, falls P[i]=a und 0 sonst (für i=1,...,m)
- U[ ] kann in Zeit $\Theta(|\Sigma| m)$ berechnet werden

- Sei `BitShift`(S) eine Operation auf S, die Elemente von S um eine Position nach hinten schiebt, und eine 1 an Position eins einträgt:
- `BitShift`(S)[i]=1 für i=1 und =S[i-1] für i=2,...,m
- Implementierung von `BitShift`, z.B. S= (S << 1) | 1

- **Berechnung:** `BitShift`(S) AND U[T[j]]

Aktualisierung für unser Beispiel

Wir berechnen S_4 aus S_3:

S_3	BitShift(S3)	AND	U[`N´]	=	S_4
0	1		0		0
1	0		1		0
0	1		1		1
0	0		0		0
0	0		1		0

Shift-And Implementierung Bit-Parallelismus

```
(1)  Berechne U(P)
(2)  S=0
(3)  one_at_row_m = 1 << (m-1) //ergibt Vektor: (0,0,...0,1)
                               //aus (1,0,0,...0) und m-1 shifts
(4)  for j=1,...,N do {
(5)     S = BitShift(S) & U[T[j]]
(6)     if (S & one_at_row_m) then
(7)        Ausgabe des Musters an Position j-m+1
(8)  }
```

Bemerkungen

- Offensichtlich ist der Algorithmus korrekt und läuft in Zeit $\Theta(N)$, wenn die Musterlänge konstant ist.
- Der Bit-Parallelismus Algorithmus (bzw. *Shift-And* Implementierung) wurde ursprünglich von Baeza-Yates & Gonnet 1992 vorgestellt (Variante: *Shift-Or*)
- Praktische Experimente zeigen, dass diese Methode sehr schnell ist, wenn die Muster kurz sind.
- Kann leicht auf *Wildcards* und auf inexaktes Matching verallgemeinert werden.

ENDE STRING MATCHING