Algorithmentheorie
- Suche in Texten(1) -

Prof. Dr. S. Albers
Prof. Dr. Th. Ottmann

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 $t_1\, t_2 \ldots\ldots t_n \quad \in \Sigma^n$

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

Gesucht:

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

$$
\begin{aligned}
p_1 &= t_{i+1} \\
p_2 &= t_{i+2} \\
&\vdots \\
p_m &= t_{i+m}
\end{aligned}
$$

Problemdefinition

i $\quad$ $i+1$ $\quad$ $i+m$

Text: $t_1 \; t_2 \; .... \; t_{i+1} \; .... \; t_{i+m} \; ..... \; t_n$

Muster: $\longrightarrow$ $p_1 \; .... \; p_m$

Aufwandsabschätzung (Zeit) :

1. # mögl. Verschiebungen: n – m + 1 # Musterstellen: m
   → O(n m)

2. mind. 1 Vergleich pro m aufeinander folgende Textstellen:
   → Ω (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):

0 0 ... 0 ... 0 ... 0 0 ...

$\xrightarrow{i}$ | 0 ... 0 ... 0 1 |

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:

$$t_1 \quad t_2 \quad \ldots \qquad \ldots \qquad t_i \qquad \ldots \qquad \ldots$$

$$= \quad = \quad = \quad = \quad \neq$$

$$p_1 \quad \ldots \quad p_j \quad p_{j+1} \quad \ldots \quad p_m$$

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

$t_1 \quad t_2 \quad \ldots \quad \ldots \quad t_i \quad \ldots \quad \ldots$

$= \; = \; = \; = \; \neq$

$p_1 \quad \ldots \quad p_j \quad p_{j+1} \quad \ldots \quad p_m$

Verfahren nach Knuth-Morris-Pratt (KMP)

Beispiel für die Bestimmung von next[j]:

t_1 t_2 ... 01011 01011 0 ...

01011 01011 1

01011 01011 1

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

Verfahren nach Knuth-Morris-Pratt (KMP)

⇒ 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:

$$
\begin{array}{ccccccccccc}
\boxed{t_1 \quad t_2 \quad \ldots \quad \ldots \quad t_i \quad \ldots} & \ldots \\
= \;\; = \;\; = \;\; = \;\; \neq & \\
\boxed{p_1 \quad \ldots \quad p_j \quad p_{j+1} \quad \ldots \quad p_m} &
\end{array}
$$

Situation am Beginn der for-Schleife:

$P_{1...j} = T_{i\text{-}j...i\text{-}1}$ 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 Prefixes von P,
das echtes Suffix von $P_{1 \ldots i}$ ist.

next[1] = 0
Sei next[i-1] = j:

$p_1 \quad p_2 \quad \ldots \qquad \ldots \qquad p_i \qquad \ldots \qquad \ldots$

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

$p_1 \quad \ldots \quad p_j \quad p_{j+1} \quad \ldots \quad p_m$

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)

```
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

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)

```
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

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
```

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

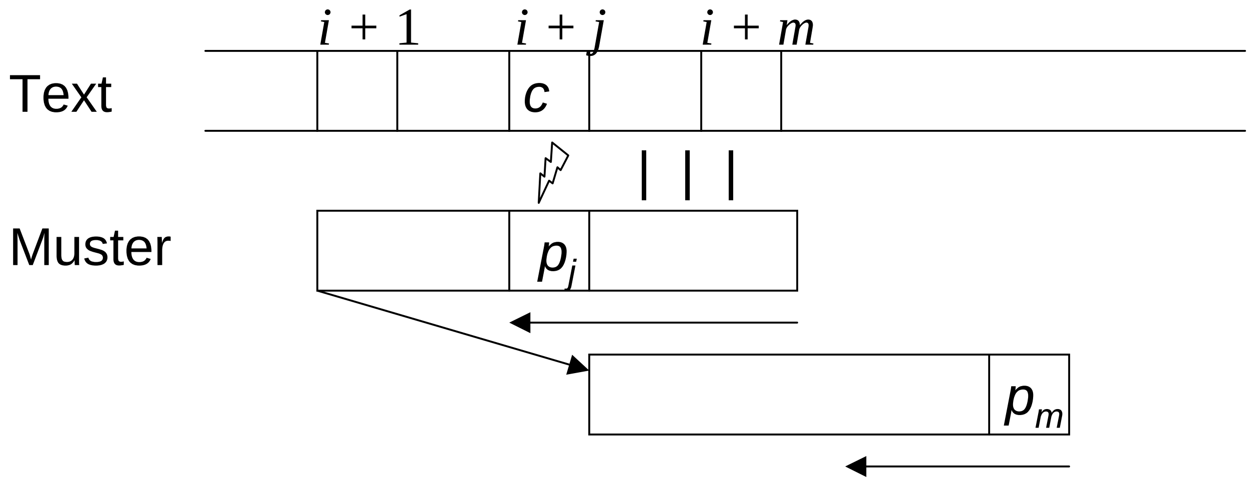
j = Index des aktuellen Zeichens im Muster ($c \neq p_j$)

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

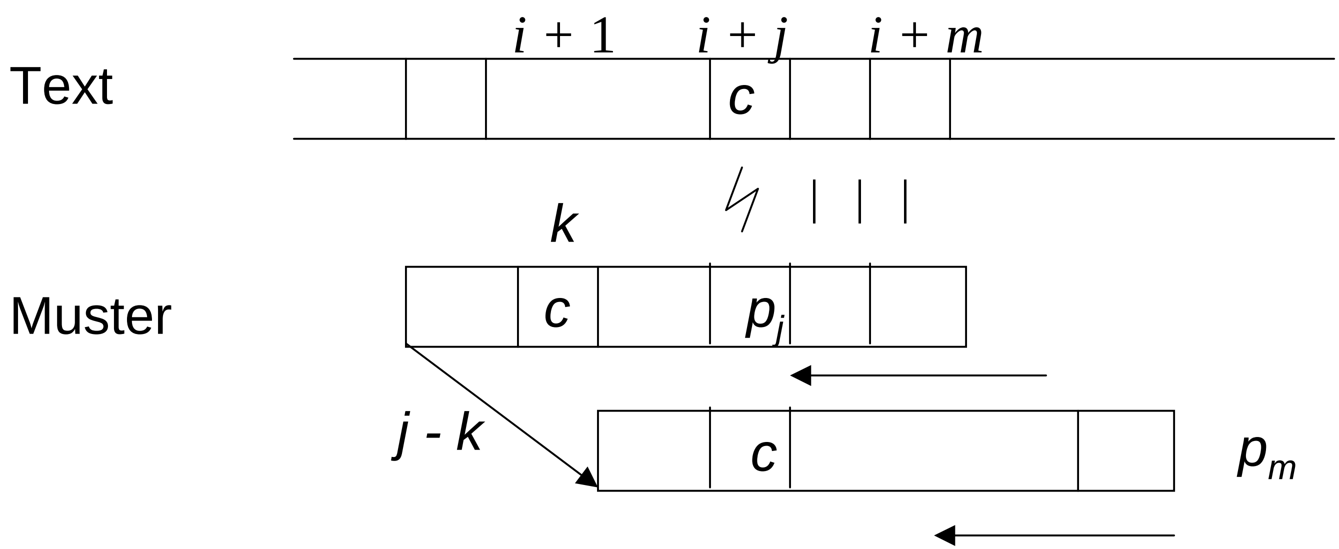


$$\Delta(i) = j$$

BM – Die Vorkommensheuristik

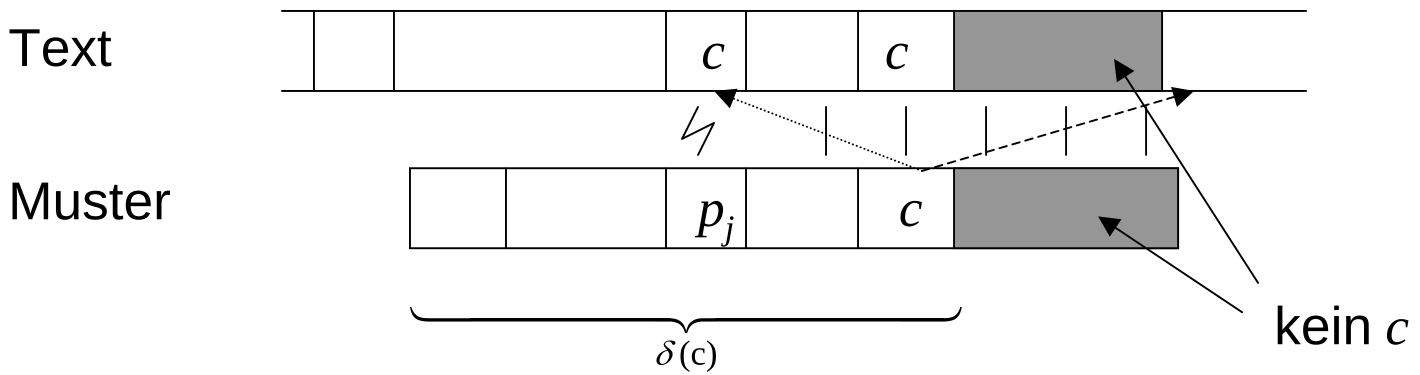
Fall 2 *c* kommt im Muster vor. ($\delta(c) \neq 0$)

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: $\delta(c) > j$

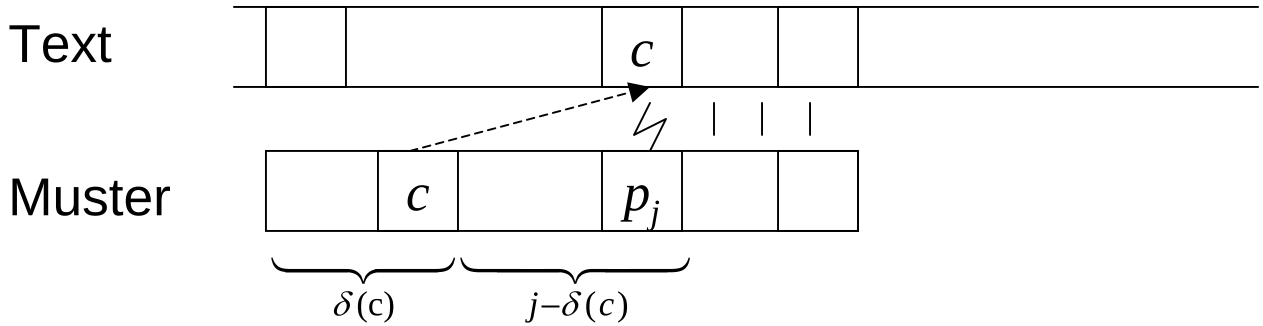


Verschiebung des rechtesten *c* im Muster auf ein potentielles *c* im Text.

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

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

n := length(T); m := length(P)

berechne δ

$i := 0$

while $i \leq n - m$ **do**

    $j := m$

    while $j > 0$ **and** $P[j] = T[i + j]$ **do**

        $j := j - 1$

  end while;

BM-Algorithmus (1.Version)

```
if j = 0
    then gebe Verschiebung i aus
         i := i + 1
else if δ(T[i + j]) > j
    then i := i + m + 1 - δ[T[i + j]]
    else i := i + j - δ[T[i + j]]
end while;
```

BM-Algorithmus (1.Version)

Laufzeitanalyse:

gewünschte Laufzeit : $c(m + n/m)$

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

$$\boxed{0\ 0\ \ldots\qquad 0\ 0\ \ldots\ 0\ \ldots\ 0\ \ldots}$$

$$\xrightarrow{i}\ \boxed{1\ 0\ \ldots\ 0\ \ldots\ 0}$$

Match-Heuristik

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

$t_1 \; t_2 \; \ldots \quad t_{i+1} \; \ldots \quad t_{i+j} \; \ldots \quad t_{i+m}$

$\neq \quad = \quad = \quad =$

$\xrightarrow{i}$ $p_1 \; \ldots \quad p_j \; \ldots \quad p_m$

$wrw[j]$ = 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
            ≠ = = =
        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

n := length(T); m := length(P)
berechne δ und γ
$i := 0$
while $i \leq n - m$ **do**
    $j := m$
    while $j > 0$ **and** $P[j] = T[i + j]$ **do**
        $j := j - 1$
    end while;

BM-Algorithmus (2.Version)

if $j = 0$
  then gebe Verschiebung i aus
    $i := i + \gamma[0]$
  else $i := i + \max(\gamma[j], j - \delta[T[i + j]])$
end while;