Algorithmentheorie

15 - Suche in Texten

Robert Elsässer

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

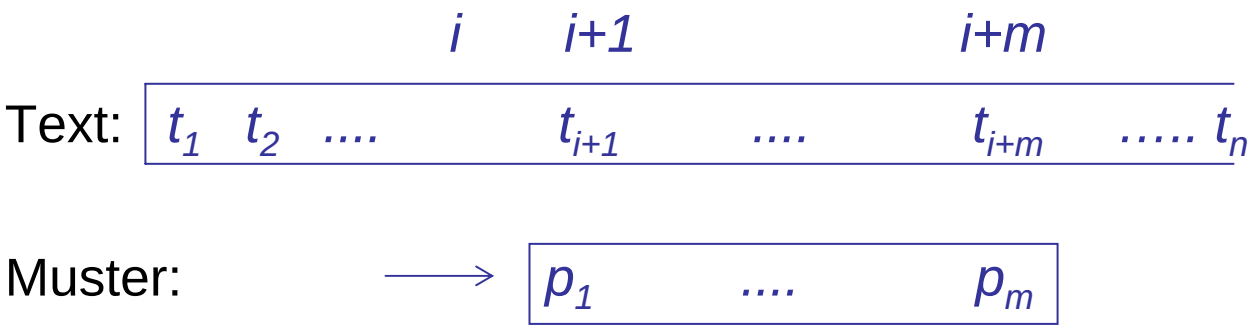
Gesucht:

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:
   → Ω ($m + n/m$)

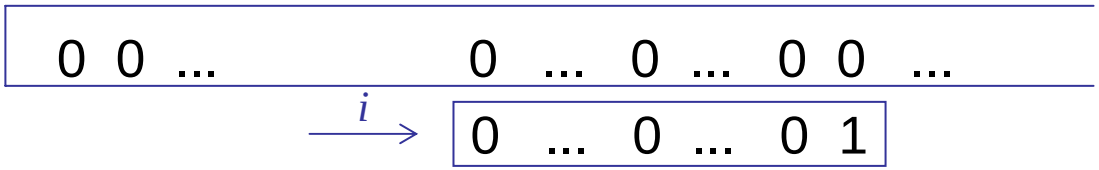
Naives Verfahren

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):



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

In der Praxis tritt Mismatch oft 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	t_2	...		...		t_i	...		...
		=	=	=	=	≠			
p_1	...		p_j	p_{j+1}	...		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.
- $t_i \neq p_{j+1}$

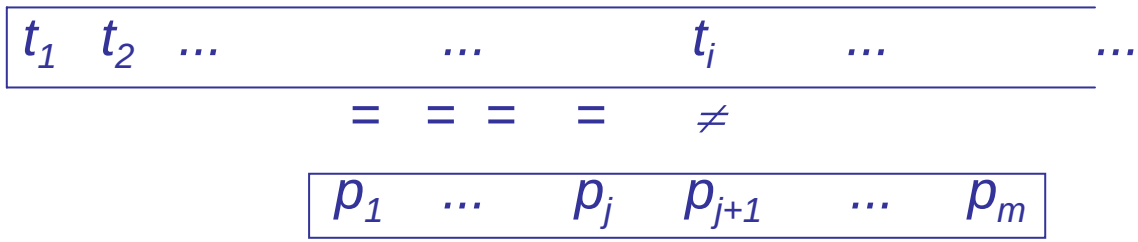
Verfahren nach Knuth-Morris-Pratt (KMP)

Idee:

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

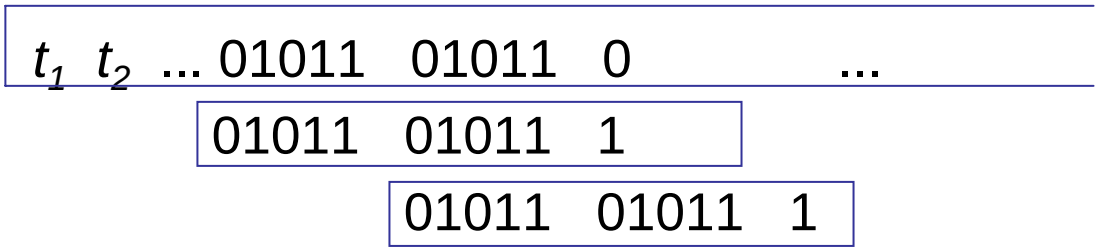
Bestimme größtes $j' < j$, so dass $P_{1\ldots j'} = P_{j-j'+1\ldots j}$.

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)

⇒ 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

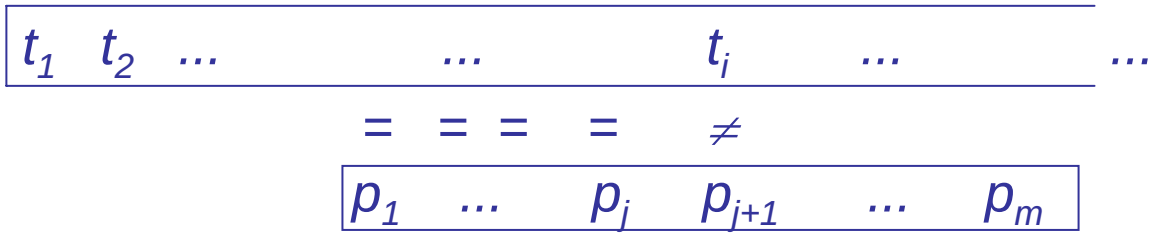
a b r a k a d a b r a b r a b a b r a k ...

| | | | |

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.

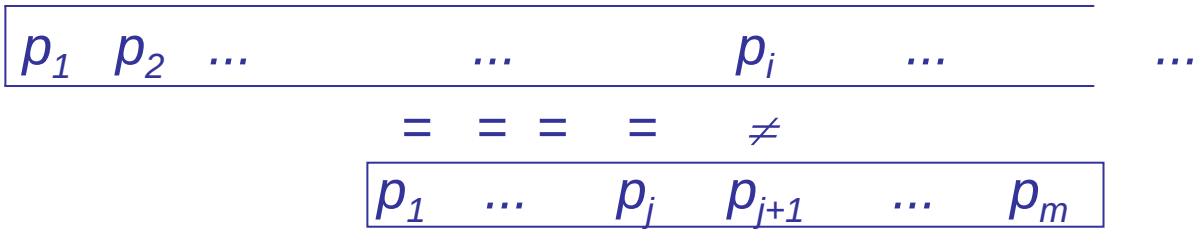
Der KMP-Algorithmus kann in Zeit O(*n*) ausgeführt werden, wenn das next-Array bekannt ist.

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, dann next[i] = j + 1 sonst 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.