

# Algorithmen und Datenstrukturen 2

Sommersemester 2006  
8. Vorlesung

*Peter Stadler*

Universität Leipzig  
Institut für Informatik  
*studla@bioinf.uni-leipzig.de*

# Suche in Texten

## Einführung

### Suche in dynamischen Texten (ohne Indexierung)

- Naiver Algorithmus (Brute Force)
- Knuth-Morris-Pratt (KMP) - Algorithmus
- Boyer-Moore (BM) - Algorithmus
- Signaturen

### Suche in (weitgehend) statischen Texten -> Indexierung

- Suffix-Bäume
- Invertierte Listen
- Signatur-Dateien

### Approximative Suche

- k-Mismatch-Problem
- Editierdistanz
- Berechnung der Editierdistanz

# Einführung

Problem: Suche eines Teilwortes/Musters/Sequenz in einem Text

- String Matching
- Pattern Matching
- Sequence Matching

Häufig benötigte Funktion

- Suchen und Ersetzen in Textverarbeitung
- Durchsuchen von Web-Seiten
- Durchsuchen von Dateisammlungen etc.
- Suchen von Mustern in DNA-Sequenzen (begrenzttes Alphabet: A, C, G, T)

Dynamische vs. statische Texte

- dynamische Texte (z.B. im Texteditor): aufwendige Vorverarbeitung / Indizierung i.a. nicht sinnvoll
- relativ statische Texte: Erstellung von Indexstrukturen zur Suchbeschleunigung

Suche nach beliebigen Strings/Zeichenketten vs. Wörtern/Begriffen

# Exakte Suche vs. approximative Suche (Ähnlichkeitssuche)

Genauere Aufgabenstellung für die exakte Suche

- Gegeben: Zeichenkette  $\text{text}[1..n]$  aus einem endlichen Alphabet  $\Sigma$ ,  
Muster (Pattern)  $\text{pat}[1..m]$  mit  $\text{pat}[i] \in \Sigma$ ,  $m \leq n$
- *Fenster*  $w_i$  ist eine Teilzeichenkette von  $\text{text}$  der Länge  $m$ , die an Position  $i$  beginnt, also  $\text{text}[i]$  bis  $\text{text}[i+m-1]$
- Ein Fenster  $w_i$ , das mit dem Muster  $p$  übereinstimmt, heißt *Vorkommen* des Musters an Position  $i$ .  $w_i$  ist Vorkommen:  
 $\text{text}[i] = \text{pat}[1], \text{text}[i+1] = \text{pat}[2], \dots, \text{text}[i+m-1] = \text{pat}[m]$
- Ein *Mismatch* in einem Fenster  $w_i$  ist eine Position  $j$ , an der das Muster mit dem Fenster nicht übereinstimmt
- Gesucht: ein oder alle Positionen von Vorkommen des Pattern  $\text{pat}$  im Text

Beispiele

Position:	12345678...	12345678...
Text:	dieser testtext ist ...	aaabaabacabca
Muster:	test	aaba

Maß der Effizienz: Anzahl der (Zeichen-) Vergleiche zwischen Muster und Text

# Naiver Algorithmus (Brute Force)

## Brute Force-Lösung 1

- Rückgabe der ersten Position  $i$  an der Muster vorkommt bzw.  $-1$  falls kein Vorkommen

```
FOR i=1 to n -m+1 DO BEGIN
  found := true;
  FOR j=1 to m DO IF text[i] ≠ pat [j] THEN found := false; { Mismatch }
  IF found THEN RETURN i;
END;
RETURN -1;
```

- Komplexität  $O(n \cdot m)$

## Brute Force-Lösung 2

- Abbrechen der Prüfung einer Textposition  $i$  bei erstem Mismatch mit dem Muster

```
FOR i=1 to n -m+1 DO BEGIN
  j := 1;
  WHILE j <= m AND pat[j] = text[i+j-1] DO j := j+1 END;
  IF j = m+1 THEN RETURN i;
END
RETURN -1;
```

- Aufwand oft  $n$
- Worst Case-Aufwand weiterhin  $O(n \cdot m)$

# Naiver Algorithmus: Beispiel

```
Text:  der erste testtext ist kurz
Muster: test
        test
         test
          test
           test
            test
             test
              test
               test
                test
                 test
```

## Verschiedene bessere Algorithmen

- Nutzung der Musterstruktur, Kenntnis der im Muster vorkommenden Zeichen
- Knuth-Morris-Pratt (1974): nutze bereits geprüfter Musteranfang um ggf. Muster um mehr als eine Stelle nach rechts zu verschieben
- Boyer-Moore (1976): Teste Muster von hinten nach vorne

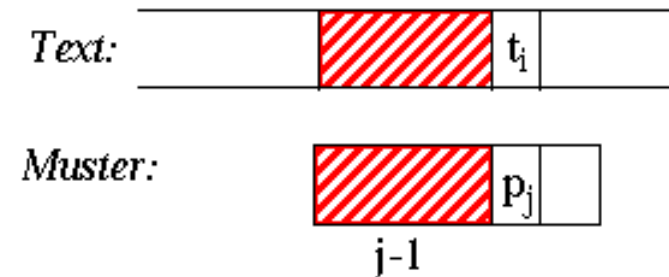
# Knuth-Morris-Pratt (KMP) I

**Idee:** nutze bereits gelesene Information bei einem Mismatch

- verschiebe ggf. Muster um mehr als 1 Position nach rechts
- gehe im Text nie zurück!

## Allgemeiner Zusammenhang

- Mismatch an Textposition  $i$  mit  $j$ -tem Zeichen im Muster
- $j-1$  vorhergehende Zeichen stimmen überein
- mit welchem Zeichen im Muster kann nun das  $i$ -te Textzeichen verglichen werden, so dass kein Vorkommen des Musters übersehen wird?



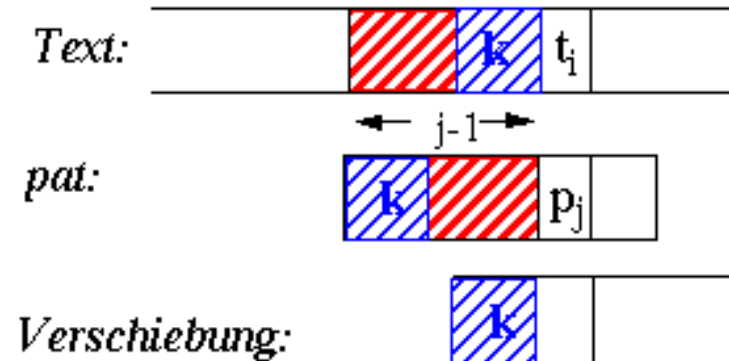
## Beispiele

*Text:* DATENSTRUKTUREN GEgebenenfalls  
*Muster:* DATUM GEGEN

# Knuth-Morris-Pratt (KMP) II

## Beobachtungen

- wesentlich ist das längste Präfix des Musters (Länge  $k < j-1$ ), das Suffix des übereinstimmenden Bereiches ist, d.h. gleich  $pat [ j-k-1.. j-1 ]$  ist
- dann ist Position  $k+1 = next (j)$  im Muster, die nächste Stelle, die mit Textzeichen  $t_j$  zu vergleichen ist (entspricht Verschiebung des Musters um  $j-k-1$  Positionen)
- für  $k=0$  kann Muster um  $j-1$  Positionen verschoben werden



## Hilfstabelle next spezifiziert die nächste zu prüfende Position des Musters

- $next [j]$  gibt für Mismatch an Position  $j > 1$ , die als nächstes zu prüfende Musterposition an
- $next[j] = 1 + k$  (=Länge des längsten echten Suffixes von  $pat[1..j-1]$ , das Präfix von  $pat$  ist)
- $next [1]=0$
- $next$  kann ausschliesslich auf dem Muster selbst (vorab) bestimmt werden

## Beispiel zur Bestimmung der Verschiebetabelle next

P. Stadler	j	1 2 3 4 5	next[j]:
	Muster:	ABABC	



# KMP: Algorithmus

KMP-Suchalgorithmus (setzt voraus, dass next-Tabelle erstellt wurde)

```
j:=1; i:=1;
WHILE (i <= n) DO BEGIN
    IF pat[j]= text[i] DO BEGIN    IF j=m RETURN i-m+1; // Match
                                  j :=j+1; i :=i+1;
                                END
    ELSE                            IF j>1 THEN j := next [j]
                                  ELSE i := i+1;
    END
RETURN -1; // Mismatch
```

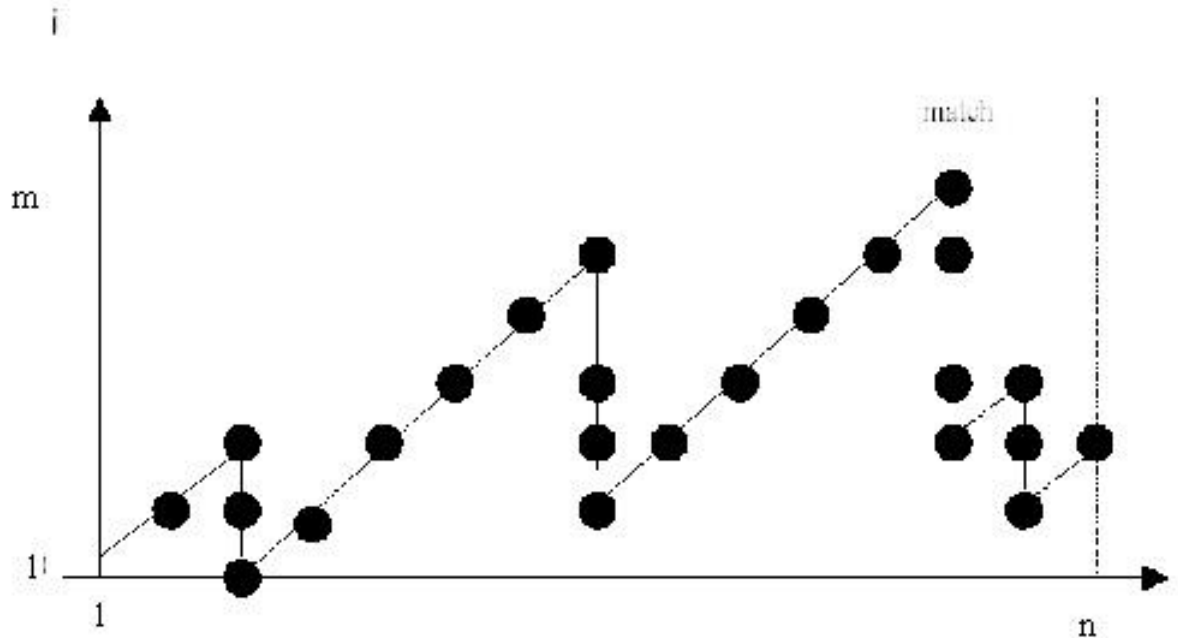
Beispiel

Text: ABCAABABABAABABC  
Muster: ABABC

j 1 2 3 4 5  
Muster: ABABC  
next[j]:

# KMP Verlauf

## Verlauf von $i$ und $j$ bei KMP-Stringsuche



### Lineare Worst-Case-Komplexität $O(n+m)$

- Suchverfahren selbst  $O(n)$
- Vorberechnung der next-Tabelle  $O(m)$

### Vorteilhaft v.a. bei Wiederholung von Teilmustern

# Boyer-Moore

Auswertung des Musters von rechts nach links, um bei Mismatch Muster möglichst weit verschieben zu können

Nutzung von im Suchmuster vorhandenen Informationen, insbesondere vorkommenden Zeichen und Suffixen

Vorkommens-Heuristik ("bad character heuristic")

- Textposition  $i$  wird mit Muster von hinten beginnend verglichen; Mismatch an Muster-Position  $j$  für Textsymbol  $t$
- wenn  $t$  im Muster nicht vorkommt (v.a. bei kurzen Mustern sehr wahrscheinlich), kann Muster hinter  $t$  geschoben, also um  $j$  Positionen
- wenn  $t$  vorkommt, kann Muster um einen Betrag verschoben werden, der der Position des letzten Vorkommens des Symbols im Suchmuster entspricht
- Verschiebeumfang kann für jeden Buchstaben des Alphabets vorab auf Muster bestimmt und in einer Tabelle vermerkt werden

# Boyer-Moore: Algorithmus

**Beispiel:** Text: DATENSTRUKTUREN UND ALGORITHMEN . . .  
Muster: DATUM DATUM

Vorbereitung einer Hilfstabelle last

- für jedes Symbol des Alphabets wird die Position seines letzten Vorkommens im Muster angegeben
- -1, falls das Symbol nicht im Muster vorkommt
- für Mismatch an Musterposition  $j$ , verschiebt sich der Anfang des Musters um  $j - \text{last}[t] + 1$  Positionen

**Algorithmus**

```
i:=1;  
WHILE (i <= n-m) DO BEGIN  
    j := m;  
    WHILE j >= 1 AND pat[j]= text[i+j-1] DO j := j-1;  
    IF j < 1      RETURN i;           // Match  
    ELSE        i := (i+j-1) - last [text[i+j-1]];  
END;  
RETURN -1; // Mismatch
```

**Komplexität:**

- für große Alphabete / kleine Muster wird meist  $O(n/m)$  erreicht, d.h zumeist ist nur jedes  $m$ -te Zeichen zu inspizieren
- Worst-Case jedoch  $O(n*m)$

# Boyer-Moore: Beispiel

Text: PETER PIPER PICKED A PECK  
Muster: PECK

## Last-Tabelle:

A:	N:
B:	O:
C:	P:
D:	...
E:	
...	
J:	Y:
K:	Z:
...	...

# Boyer-Moore: Verbesserungen

## Match-Heuristik ("good suffix heuristic")

- Suffix  $s$  des Musters stimmt mit Text überein
- Fall 1: falls  $s$  nicht noch einmal im Muster vorkommt, kann Muster um  $m$  Positionen weitergeschoben werden
- Fall 2: es gibt ein weiteres Vorkommen von  $s$  im Muster: Muster kann verschoben werden, bis dieses Vorkommen auf den entsprechenden Textteil zu  $s$  ausgerichtet ist
- Fall 3: Präfix des Musters stimmt mit Endteil von  $s$  überein: Verschiebung des Musters bis übereinstimmende Teile übereinander liegen

Text:	CBABBCBBCABA . . .	CBABBCBBCABA . . .
Muster:	ABBABC	ABCCBC

Text:	BAABBCABCABA . . .
Muster:	CBAABC

Lineare Worst-Case-Komplexität  $O(n+m)$

# Signaturen I

## Indirekte Suche über Hash-Funktion

- Berechnung einer Signatur  $s$  für das Muster, z.B. über Hash-Funktion
- für jedes Textfenster an Position  $i$  (Länge  $m$ ) wird ebenfalls eine Signatur  $s_i$  berechnet
- Falls  $s_i = s$  liegt ein potentieller Match vor, der näher zu prüfen ist
- zeichenweiser Vergleich zwischen Muster und Text wird weitgehend vermieden

## Pessimistische Philosophie

- "Suchen" bedeutet "Versuchen, etwas zu finden". Optimistische Ansätze erwarten Vorkommen und führen daher viele Vergleiche durch, um Muster zu finden
- Pessimistische Ansätze nehmen an, dass Muster meist nicht vorkommt. Es wird versucht, viele Stellen im Text schnell auszuschließen und nur an wenigen Stellen genauer zu prüfen
- Neben Signatur-Ansätzen fallen u.a. auch Verfahren, die zunächst Vorhandensein seltener Zeichen prüfen, in diese Kategorie

# Signaturen II

Kosten  $O(n)$  falls Signaturen effizient bestimmt werden können

- inkrementelle Berechnung von  $s_i$  aus  $s_{i-1}$
- unterschiedliche Vorschläge mit konstantem Berechnungsaufwand pro Fenster

Beispiel: Ziffernalphabet; Quersumme als Signaturfunktion

Text:	7 6 2 1 3 0 8 7 2 5 0 8 . . .	Muster: 1 3 0 8
	- 1 6 -	Signatur: $1+3+0+8=12$

- inkrementelle Berechenbarkeit der Quersumme eines neuen Fensters (Subtraktion der herausfallenden Ziffer, Addition der neuen Ziffer)
- jedoch hohe Wahrscheinlichkeit von Kollisionen (false matches)



# Karp-Rabin

## Alternative Signaturfunktion

- Abbildung des Musters / Fensters in Dezimalzahl von max. 9 Stellen (mit 32 Bits repräsentierbar)

- Signatur des Musters:  $s(p_1, \dots, p_m) = \sum_{j=1..m} (10^{j-1} \cdot p_{m+1-j}) \bmod 10^9$

- Signatur  $s_{i+1}$  des neuen Fensters  $(t_{i+1} \dots t_{i+m})$  abgeleitet aus Signatur  $s_i$  des vorherigen Fensters  $(t_i \dots t_{i+m-1})$ :

$$s_{i+1} = ((s_i - t_i \cdot 10^{m-1}) \cdot 10 + t_{i+m}) \bmod 10^9$$

- Signaturfunktion ist auch für größere Alphabete anwendbar

# Statische Suchverfahren

Annahme: weitgehend statische Texte / Dokumente

- derselbe Text wird häufig für unterschiedliche Muster durchsucht

Beschleunigung der Suche durch Indexierung (Suchindex)

Vorgehensweise bei

- Information Retrieval-Systemen zur Verwaltung von Dokumentkollektionen
- Volltext-Datenbanksystemen
- Web-Suchmaschinen etc.

Indexvarianten

- (Präfix-) B\*-Bäume
- Tries, z.B. Radix oder PATRICIA Tries
- Suffix-Bäume
- Invertierte Listen
- Signatur-Dateien

# Suffix-Bäume I

**Suffix-Bäume:** Digitalbäume, die alle Suffixe einer Zeichenkette bzw. eines Textes repräsentieren

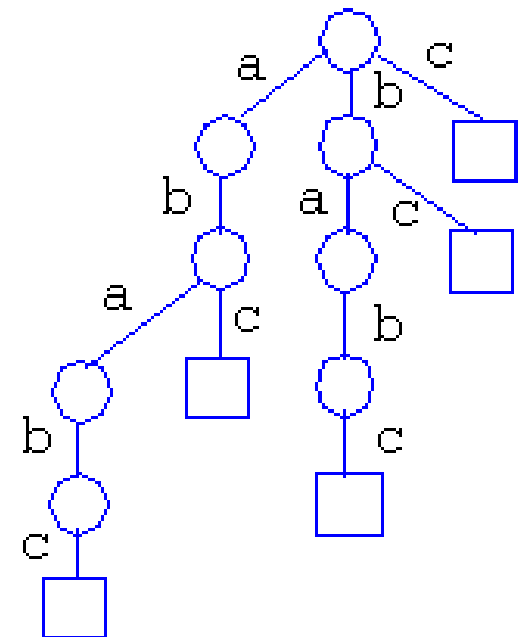
## Unterstützte Operationen:

- Teilwortsuche: in  $O(m)$
- Präfix-Suche: Bestimmung aller Positionen, an denen Worte mit einem Präfix  $p$  auftreten
- Bereichssuche: Bestimmung aller Positionen von Worten, die in der lexikographischen Ordnung zwischen zwei Grenzen  $p1$  und  $p2$  liegen

**Suffix-Tries** basierend auf Tries

- hoher Platzbedarf für Suffix-Tries  
 $O(n^2)$  -> Kompaktierung durch  
Suffix-Bäume

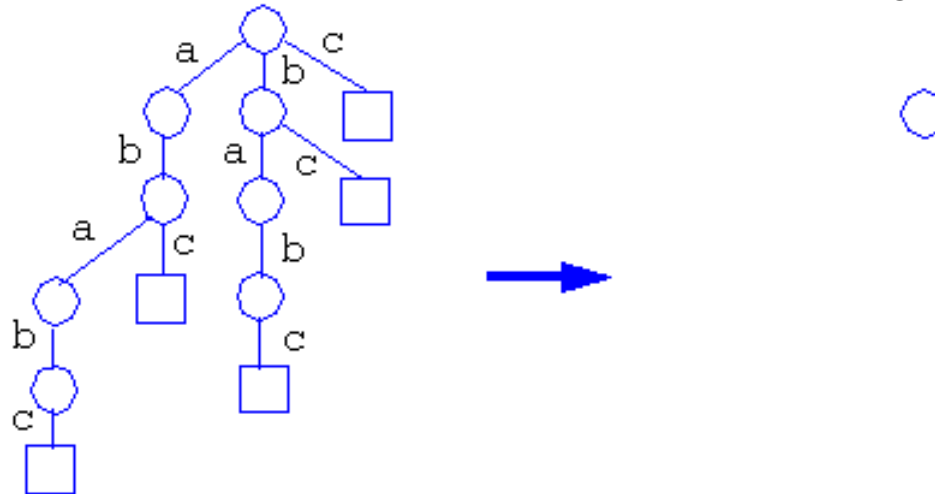
Text ababc  
5 Suffixe: ababc  
babac  
abc  
bc  
c



# Kontraktion von Suffix-Bäumen

Alle Wege im Trie, die nur aus unären Knoten bestehen, werden zusammengezogen

Text: ababc  
Suffixe: ababc  
          babc  
          abc  
          bc  
          c



## Eigenschaften für Suffix-Baum S

- jede Kante in S repräsentiert nicht-leeres Teilwort des Eingabetextes T
- die Teilworte von T, die benachbarten Kanten in S zugeordnet sind, beginnen mit verschiedenen Buchstaben
- jeder innerer Knoten von S (außer der Wurzel) hat wenigstens zwei Söhne
- jedes Blatt repräsentiert ein nicht-leeres Suffix von T


**Linearer Platzbedarf**  $O(n)$ : n Blätter und höchstens n-1 innere Knoten


# Konstruktion von Suffix-Bäumen

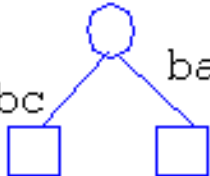
## Vorgehensweise bei Konstruktion

- beginnend mit leerem Baum  $T_0$  wird pro Schritt Suffix  $suff_i$  beginnend an Textposition  $i$  eingefügt und Suffix-Baum  $T_{i-1}$  nach  $T_i$  erweitert
- zur Einfügung ist  $head_i$  zu bestimmen, d.h. längstes Präfix von  $suff_i$ , das bereits im Baum präsent ist, d.h. das bereits Präfix von  $suff_j$  ist ( $j < i$ )

Text: ababc

$T_0 =$  

$T_1 =$   ababc

$T_2 =$  

$T_3 =$  

$suff_3 =$  abc  
 $head_3 =$   
 $tail_3 =$

**naiver Algorithmus:**  $O(n^2)$

**linearer Aufwand  $O(n)$**  gemäß Konstruktionsalgorithmus von McCreight

- Einführung von Suffix-Zeigern
- Einzelheiten siehe Ottmann/Widmayer (2001)