4. 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
  - Textverarbeitung
  - Durchsuchen von Web-Seiten
  - Durchsuchen von Dateisammlungen etc.
  - Suchen von Mustern in DNA-Sequenzen (begrenztes 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. Wörten/Begriffen
- Exakte Suche vs. approximative Suche (Ähnlichkeitssuche)

Einführung (2)

- Genauere Aufgabenstellung (exakte Suche)
  - *Gegeben:* Zeichenkette *text [1..n]* aus einem endlichen Alphabet Σ,
    Muster (Pattern) *pat [1..m]* mit pat[i] $\in \Sigma$, m <= n
  - *Fenster* w_i ist eine Teilzeichenkette von text der Länge m, die an Position i beginnt, also text [i] bis 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 [i] = pat [1], text[i+1]=pat[2], ..., text[i+m-1]=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 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-m)*m) = O (n*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 nur O(n+m)
  - Worst Case-Aufwand weiterhin O(n*m)

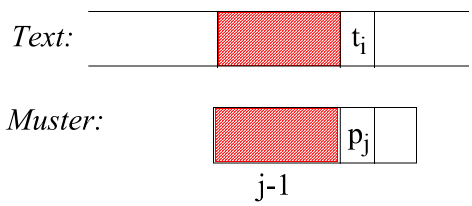
Naiver Algorithmus (2)

```
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)

- 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 Text-zeichen verglichen werden, so daß kein Vorkommen des Musters übersehen wird?

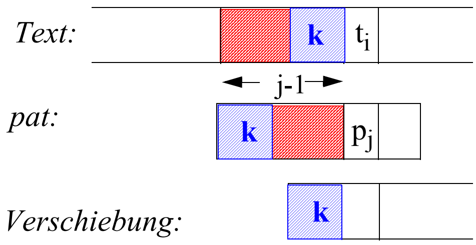


- Beispiele

```
Text:    DATENSTRUKTUREN      GEGEBENENFALLS
Muster:  DATUM                GEGEN
```

KMP (2)

- 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_i 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 Postion 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

j	1 2 3 4 5
Muster:	A B A B C

next[j]:

KMP (3)

- 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
```

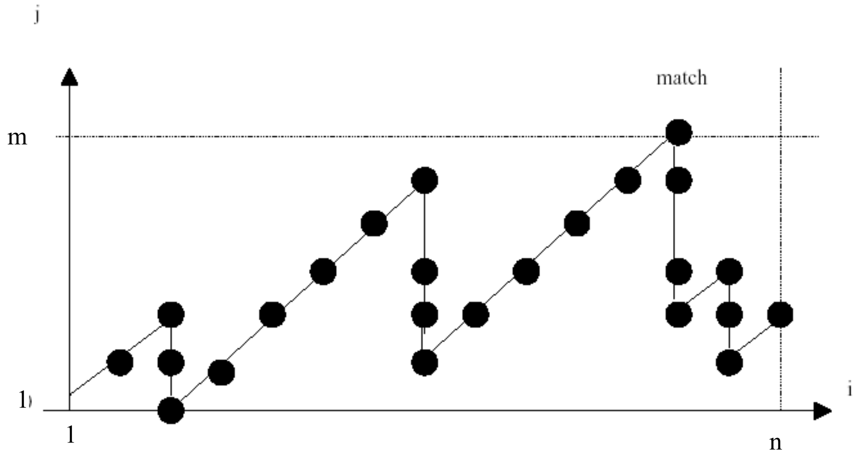
- Beispiel

Text:	A B C A A B A B A A B A B C
Muster:	A B A B C

j	1 2 3 4 5
Muster:	A B A B C
next[j]:	

KMP (4)

- 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
- Beispiel:

```
Text:    DATENSTRUKTUREN UND ALGORITHMEN ...
Muster:  DATUM               DATUM
```

Boyer-Moore (2)

- Vorberechnung 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 - 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: P E T E R   P I P E R   P I C K E D   A   P E C K
Muster: P E C K

Last-Tabelle:

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

Boyer-Moore (4)

- weitere Verbesserung durch 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:    C B A B B C B B C A B A . . .
Muster:  A B B A B C
```

```
C B A B B C B B C A B A . . .
A B C C B C
```

```
Text:    B A A B B C A B C A B A . . .
Muster:  C B A A B C
```

- lineare Worst-Case-Komplexität O (n+m)

Signaturen

- 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
- Kosten O (n) falls Signaturen effizient bestimmt werden können
  - inkrementelle Berechnung von s_i aus s_{i-1}
  - unterschiedliche Vorschläge mit konstantem Berechnungaufwand pro Fenster

Signaturen (2)

- Beispiel: Ziffernalphabet; Quersumme als Signaturfunktion

Text: 7 6 2 1 3 0 8 7 2 5 0 8 . . .
-1 6-

Muster: 1 3 0 8
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)

- Alternative Signaturfunktion (Karp-Rabin)
  - Abbildung des Musters / Fensters in Dezimalzahl von max. 9 Stellen (mit 32 Bits repräsentierbar)
  - Signatur des Musters: $s\,(p_1, \ldots p_m) = \Sigma_{j=1..m}\,(10^{j-1} \cdot p_{m+1-j})\bmod 10^9$
  - Signatur s_{i+1} des neuen Fensters ($t_{i+1} \ldots t_{i+m}$) abgeleitet aus Signatur s_i des vorherigen Fensters ($t_i \ldots t_{i+m-1}$):
$$s_{i+1} = ((s_i - t_i \cdot 10^{m-1}) \cdot 10 + t_{i+m}) \bmod 10^9$$
  - Signaturfunktion ist auch für größere Alphabete anwendbar

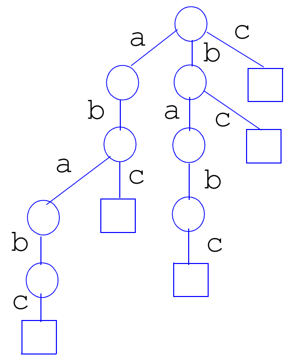
Statische Suchverfahren

- 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

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

Text: `ababc`
5 Suffixe: `ababc`, `babc`, `abc`, `bc`, `c`

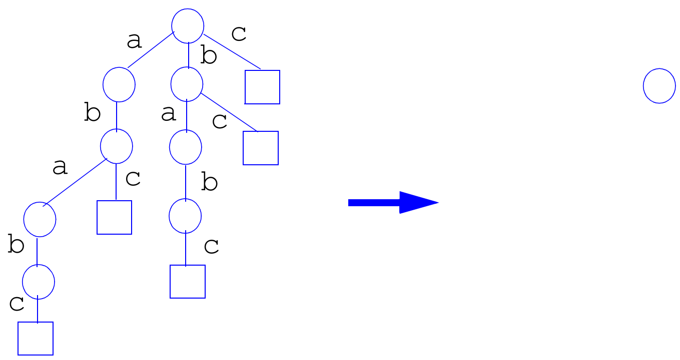


  - hoher Platzbedarf für Suffix-Tries $O(n^2)$ -> Kompaktierung durch Suffix-Bäume

Suffix-Bäume (2)

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

Text: `ababc`
Suffixe: `ababc`, `babc`, `abc`, `bc`, `c`



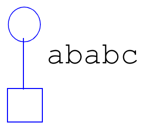
- Eigenschaften für Suffix-Baum S
  - jede Kante in S repräsentiert nicht-leeres Teilwort des Eingabetextes T
  - die Teilworte von T, die benachbarten Kanten in S zugeordnet sind, beginnen mit *verschiedenen* Buchstaben
  - jeder innerer Knoten von S (außer der Wurzel) hat wenigstens zwei Söhne
  - jedes Blatt repräsentiert ein nicht-leeres Suffix von T
- linearer Platzbedarf O(n): n Blätter und höchsten n-1 innere Knoten

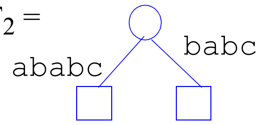
Suffix-Bäume (3)

- 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`

$T_0 =$ 

$T_1 =$ 

$T_2 =$ 

$T_3 =$ 

$suff_3 =$ `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)

Invertierte Listen

- Nutzung vor allem zur Textsuche in Dokumentkollektionen
  - nicht nur 1 Text/Sequenz, sondern beliebig viele Texte / Dokumente
  - Suche nach bestimmten Wörtern/Schlüsselbegriffen/Deskriptoren, nicht nach beliebigen Zeichenketten
  - Begriffe werden ggf. auf Stammform reduziert; Elimination sogenannnter „Stop-Wörter“ (der, die, das, ist, er ...)
  - klassische Aufgabenstellung des Information Retrieval
- Invertierung: Verzeichnis (Index) aller Vorkommen von Schlüsselbegriffen
  - lexikographisch sortierte Liste der vorkommenden Schlüsselbegriffe
  - pro Eintrag (Begriff) Liste der Dokumente (Verweise/Zeiger), die Begriff enthalten
  - eventuell zusätzliche Information pro Dokument wie Häufigkeit des Auftretens oder Position der Vorkommen
- Beispiel 1: Invertierung eines Textes

```
1            10              20
Dies ist ein Text. Der Text hat viele
Wörter. Wörter bestehen aus ...
38                    53
```

Invertierter Index

Begriff	Vorkommen
bestehen	53
Dies	1
Text	14, 24
viele	33
Wörter	38, 46

Invertierte Listen (2)

- Beispiel 2: Invertierung mehrerer Texte / Dokumente

d1

Dieses objektorientierte Datenbanksystem unterstützt nicht nur multimediale Objekte, sondern ist überhaupt phänomenal.

d2

Objektorientierte Systeme unterstützen die Vererbung von Methoden.

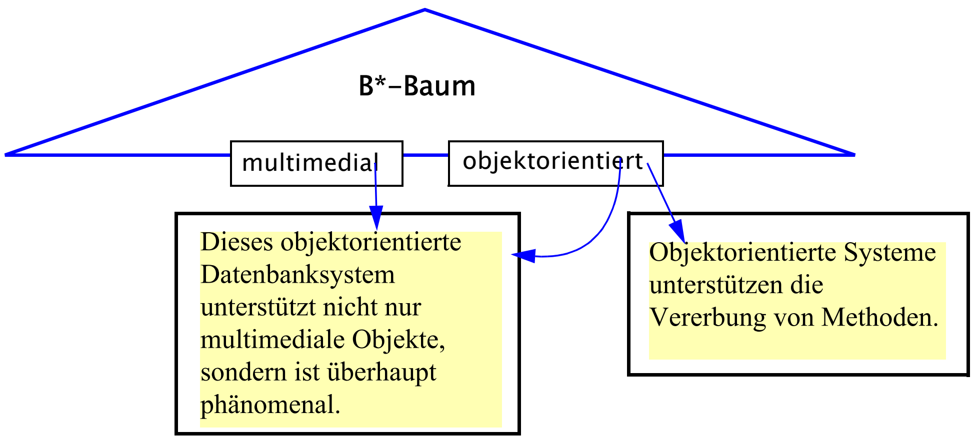
Invertierter Index

Begriff	Vorkommen
Datenbanksystem	d1
Methode	d2
multimedial	d1
objektorientiert	d1, d2
phänomenal	d1
System	d2
überhaupt	d1
unterstützen	d2
Vererbung	d2

- Zugriffskosten werden durch Datenstruktur zur Verwaltung der invertierten Liste bestimmt
  - B*-Baum
  - Hash-Verfahren ...

Invertierte Listen (3)

- effiziente Realisierung über (indirekten) B*-Baum
  - variabel lange Verweis/Zeigerlisten pro Schlüssel auf Blattebene



- Boole'sche Operationen: Verknüpfung von Zeigerlisten
  - Beispiel: Suche nach Dokumenten mit „multimedial““ UND „objektorientiert“

Signatur-Dateien

- Alternative zu invertierten Listen: Einsatz von ***Signaturen***
  - zu jedem Dokument bzw. Textfragment wird Bitvektor fester Länge (*Signatur*) geführt
  - Begriffe werden über Signaturgenerierungsfunktion (Hash-Funktion) s auf Bitvektor abgebildet
  - OR-Verknüpfung der Bitvektoren aller im Dokument bzw. Textfragment vorkommenden Begriffe ergibt *Dokument- bzw. Fragment*-Signatur

- Signaturen aller Dokumente/Fragmente werden sequentiell gespeichert (bzw. in speziellem Signaturbaum)

```
s (bestehen)   = 000101
s (Text)       = 110000
s (bestehen)   = 100100
s (viele)      = 001100
s (Wörter)     = 100001
```

Signatur-File

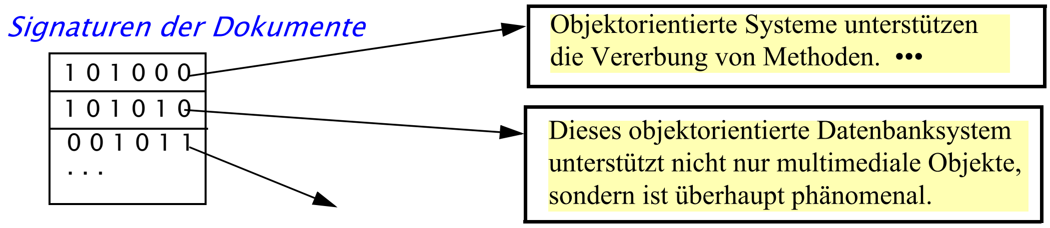
110001	→ Dies ist ein Text.
111101	→ Der Text hat viele Wörter.
100101	→ Wörter bestehen aus ...

- Suchbegriff wird über dieselbe Signaturgenerierungsfunktion s auf eine *Anfragesignatur* abgebildet
  - mehrere Suchbegriffe können einfach zu einer Anfragesignatur kombiniert werden (OR, AND, NOT-Verknüpfung der Bitvektoren)
  - wegen Nichtinjektivität der Signaturgenerierungsfunktion muß bei ermittelten Dokumenten/Fragmenten geprüft werden, ob tatsächlich ein Treffer vorliegt

Signatur-Dateien (2)

- Beispiel bezüglich mehrerer Dokumente
  - Signaturgenerierungsfunktion:

    objektorientiert / multimedial / Datenbanksystem / Vererbung -> Bit 0 / 2 / 4 / 2



  - Anfrage: Dokumente mit Begriffen "objektorientiert" und "multimedial"

    Anfragesignatur:

- Eigenschaften
  - geringer Platzbedarf für Dokumentsignaturen
  - Zugriffskosten aufgrund Nachbearbeitungsaufwand bei False Matches meist höher als bei invertierten Listen

Approximative Suche

- Ähnlichkeitssuche erfordert Maß für die Ähnlichkeit zwischen Zeichenketten s1 und s2, z.B.
  - *Hamming-Distanz:* Anzahl der Mismatches zwischen s1 und s2 (s1 und s2 haben gleiche Länge)
  - *Editierdistanz:* Kosten zum Editieren von s1, um s2 zu erhalten (Einfüge-, Lösch-, Ersetzungsoperationen)

s1:	AGCAA	AGCACACA
s2:	ACCTA	ACACACTA
Hamming-Distanz:		

- *k-Mismatch-Suchproblem*
  - Gesucht werden alle Vorkommen eines Musters in einem Text, so daß höchstens an k der m Stellen des Musters ein Mismatch vorliegt, d.h. Hamming-Distanz $\leq$ k
  - exakte Stringsuche ergibt sich als Spezialfall mit k=0

- Beispiel (k=2)

  Text: `erster testtext`
  Muster: `test`

  k=2

Approximative Suche (2)

- Naiver Such-Algorithmus kann für k-Mismatch-Problem leicht angepasst werden

```
FOR i=1 to n -m+1 DO BEGIN
   z := 1;
   FOR j=1 to m DO IF text[i] ≠ pat [j] THEN z :=z+1;{ Mismatch }
   IF z <= k THEN write („Treffer an Position “, i, „ mit “, z, „ Mismatches“);
END;
RETURN -1;
```

  - analoges Vorgehen, um Sequenz mit geringstem Hamming-Abstand zu bestimmen

- Komplexität O(n*m)
- effzientere Suchalgorithmen (KMP, BM ...) können analog angepaßt werden
- Editierdistanz oft geeigneter als Hamming-Distanz
  - anwendbar für Sequenzen unterschiedlicher Länge
  - Hamming-Distanz ist Spezialfall ohne Einfüge-/Löschoperationen (Anzahl der Ersetzungen)
  - Bioinformatik: Vergleich von DNA-Sequenzen auf Basis der Editier (Evolutions)-Distanz

Editierdistanz

- 3 Arten von Editier-Operationen: *Löschen* eines Zeichens, *Einfügen* eines Zeichens und *Ersetzen* eines Zeichens x durch ein anderes Zeichen y
- Einfügeoperationen korrespondieren zu je einer Mismatch-Situation zwischen s1 und s2, wobei „-“ für leeres Wort bzw. Lücke (gap) steht:
  - (-, y) Einfügung von y in s2 gegenüber s1
  - (x, -) Löschung von x in s1
  - (x, y) Ersetzung von x durch y
  - (x, x) Match-Situation (keine Änderung)
- jeder Operation wird Gewicht bzw. Kosten w (x,y) zugewiesen
- *Einheitskostenmodell*: w (x, y) = w (- , y) = w (x, -) = 1; w (x, x) = 0
- *Editierdistanz D (s1,s2):* Minimale Kosten, die Folge von Editier-Operationen hat, um s1 nach s2 zu überführen
  - bei Einheitskostenmodell spricht man auch von *Levensthein-Distanz*
  - im Einheitskostenmodell gilt *D (s1,s2)=D (s2,s1)* und für Kardinalitäten n und m von s1 und s2: *abs (n - m) ≤ D (s1,s2) ≤ max (m,n)*
- Beispiel: Editier-Distanz zwischen „Auto“ und „Rad“ ?

Editierdistanz in der Bioinformatik†

- Bestimmung eines *Alignments* zweier Sequenzen s1 und s2:
  - Übereinanderstellen von s1 und s2 und durch Einfügen von Gap-Zeichen Sequenzen auf dieselbe Länge bringen: Jedes Zeichenpaar repräsentiert zugehörige Editier-Operation
  - Kosten des Alignment: Summe der Kosten der Editier-Operationen
  - *optimales Alignment*: Alignment mit minimalen Kosten (= Editierdistanz)

s1:	A G C A C A C A	A G C A C A C - A	A G - C A C A C A
s2:	A C A C A C T A	A - C A C A C T A	A C A C A C T - A
	Match (A,A) Replace (G,C) Replace (C,A) Replace (A,C) Replace (C,A) Replace (A,C) Replace (C,T) Match (A,A)	Match (A,A) Delete (G, -) Match (C,C) Match (A,A) Match (C.C) Match (A,A) Match (C,C) Insert (-,T) Match (A,A)	Match (A,A) Replace (G,C) Insert (-, A) Match (C, C) Match (A,A) Match (C,C) Replace (A,T) Delete (C,-) Replace (A,A)

† www.techfak.uni-bielefeld.de/bcd/Curric/PrwAli/node2.html

Editierdistanz (3)

- Problem 1: Berechnung der Editierdistanz
  - berechne für zwei Zeichenketten / Sequenzen s1 und s2 möglichst effizient die Editierdistanz D(s1,s2) und eine kostenminimale Folge von Editier-Operationen, die s1 in s2 überführt
  - entspricht Bestimmung eines optimalen Alignments
- Problem 2: Approximate Suche
  - suche zu einem (kurzen) Muster p alle Vorkommen von Strings p' in einem Text, so daß die Editierdistanz D (p,p') <= k ist, für ein vorgegebenes k
  - Spezialfall 1: exakte Stringsuche (k=0)
  - Spezialfall 2: k-Mismatch-Problem, falls nur Ersetzungen und keine Einfüge- oder Lösch-Operationen zugelassen werden
- Variationen von Problem 2
  - Suche zu Muster/Sequenz das ähnlichste Vorkommen (lokales Alignment)
  - bestimme zwischen 2 Sequenzen s1 und s2 die ähnlichsten Teilsequenzen s1' und s2'

Berechnung der Editierdistanz

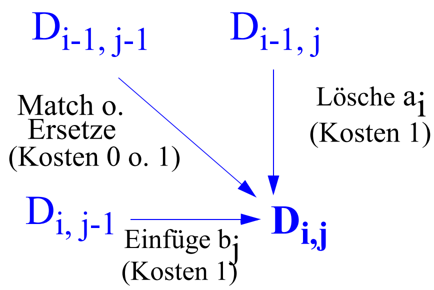
- Nutzung folgender Eigenschaften zur Begrenzung zu prüfender Editier-Operationen
  - optimale Folge von Editier-Operationen ändert jedes Zeichen höchstens einmal
  - jede Zerlegung einer optimalen Anordnung führt zur optimalen Anordnung der entsprechenden Teilsequenzen

```
AGCACAC-A
A-CACACTA
```

- Lösung des Optimierungsproblems durch Ansatz der *dynamischen Programmierung*
  - Konstruktion der optimalen Gesamtlösung durch rekursive Kombination von Lösungen für Teilprobleme
- Sei $s_1 = (a_1, \ldots a_n)$, $s_2=(b_1, \ldots b_m)$.
  D_{ij} sei Editierdistanz für Präfixe $(a_1, \ldots a_i)$ und $(b_1, \ldots b_j)$; $0 \leq i \leq n$; $0 \leq j \leq m$
  - D_{ij} kann ausschließlich aus $D_{i-1,j}$, $D_{i,j-1}$ und $D_{i-1,j-1}$ bestimmt werden
  - es gibt triviale Lösungen für $D_{0,0}$, $D_{0,j}$, $D_{i,0}$
  - Eintragung der $D_{i,j}$ in (n+1, m+1)-Matrix
  - Editierdistanz zwischen s1 und s2 insgesamt ergibt sich für i=n, j=m
  - es wird hier nur das Einheitskostenmodell angenommen

Berechnung der Editierdistanz (2)

- Editierdistanz D_{ij} für i=0 oder j=0
  - $D_{0,0} = D(-,-) = 0$
  - $D_{0,j} = D(-, (b_1,.., b_j)) = j$ // j Einfügungen
  - $D_{i,0} = D((a_1,.., a_i), -) = i$ // i Löschungen
- Editierdistanz D_{ij} für i>0 und j>0 kann aus günstigstem der folgenden Fälle abgeleitet werden:
  - Match oder Ersetze:
    falls $a_i=b_j$ (Match): $D_{i,j} = D_{i-1,j-1}$;
    falls $a_i \neq b_j$: $D_{i,j} = 1 + D_{i-1,j-1}$
  - Lösche a_i: $D_{i,j} = D((a_1,.., a_{i-1}), (b_1,.., b_j)) + 1 = D_{i-1,j} + 1$
  - Einfüge b_j: $D_{i,j} = D((a_1,.., a_i), (b_1,.., b_{j-1})) + 1 = D_{i,j-1} + 1$



Somit ergibt sich:

$$D_{i,j} = \min\left(D_{i-1,j-1} + \begin{cases} 0 & \text{falls } a_i=b_j \\ 1 & \text{falls } a_i \neq b_j \end{cases}, D_{i-1,j} + 1, D_{i,j-1} + 1 \right)$$

Berechnung der Editierdistanz (3)

- Beispiele

i \ j		0	1	2	3
		-	R	A	D
0	-				
1	A				
2	U				
3	T				
4	O				

	-	A	C	A	C	A	C	T	A
-	0	1	2	3	4	5	6	7	8
A	1	0	1	2	3	4	5	6	7
G	2	1	1	2	3	4	5	6	7
C	3	2	1	2	2	3	4	5	6
A	4	3	2	1	2	2	3	4	5
C	5	4	3	2	1	2	2	3	4
A	6	5	4	3	2	1	2	3	3
C	7	6	5	4	3	2	1	2	3
A	8	7	6	5	4	3	2	2	2

- jeder Weg von links oben nach rechts unten entspricht einer Folge von Edit-Operationen, die s1 in s2 transformiert
  - ggf. mehrere Pfade mit minimalen Kosten
- Komplexität: O (n*m)

Zusammenfassung

- naive Textsuche
  - einfache Realisierung ohne vorzuberechnende Hilfsinformationen
  - Worst Case O (n*m), aber oft linearer Aufwand O(n+m)
- schnellere Ansätze zur dynamischen Textsuche
  - Vorverarbeitung des Musters, jedoch nicht des Textes
  - Knuth-Morrison-Pratt: linearer Worst-Case-Aufwand O (n+m), aber oft nur wenig besser als naive Textsuche
  - Boyer-Moore: Worst-Case O(n*m) bzw. O(n+m), aber im Mittel oft sehr schnell O (n/m)
  - Signaturen: O (n)
- Indexierung erlaubt wesentlich schnellere Suchergebnisse
  - Vorverarbeitung des Textes bzw. der Dokumentkollektionen
  - hohe Flexibilität von Suffixbäumen (Probleme: Größe; Externspeicherzuordnung)
  - Suche in Dokumentkollektionen mit invertierten Listen oder Signatur-Dateien
- Approximative Suche
  - erfordert Ähnlichkeitsmaß, z.B. Hamming-Distanz oder Editierdistanz
  - Bestimmung der optimalen Folge von Editier-Operationen sowie Editierdistanz über dynamische Programmierung; O(n*m)