Algorithmen und Datenstrukturen 2

Sommersemester 2006
9. Vorlesung

Peter Stadler

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

Invertierte Listen

Nutzung vor allem zur Textsuche in Dokumentkollektionen

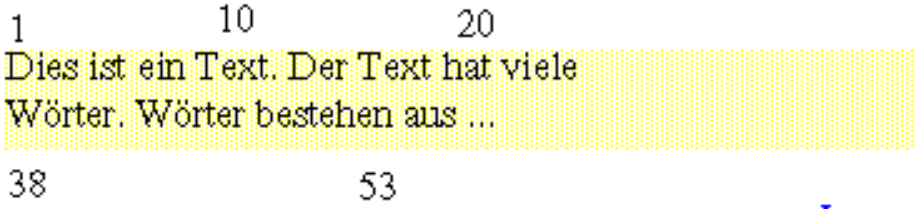
- nicht nur ein Text/Sequenz, sondern beliebig viele Texte / Dokumente
- Suche nach bestimmten Wörtern/Schlüsselbegriffen/Deskriptoren, nicht nach beliebigen Zeichenketten
- Begriffe werden ggf. auf Stammform reduziert; Elimination so genannter "Stopp-Wörter" (der, die, das, ist, er ...)
- klassische 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: Invertierte Listen

Beispiel 1: Invertierung eines Textes



Invertierter Index

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

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

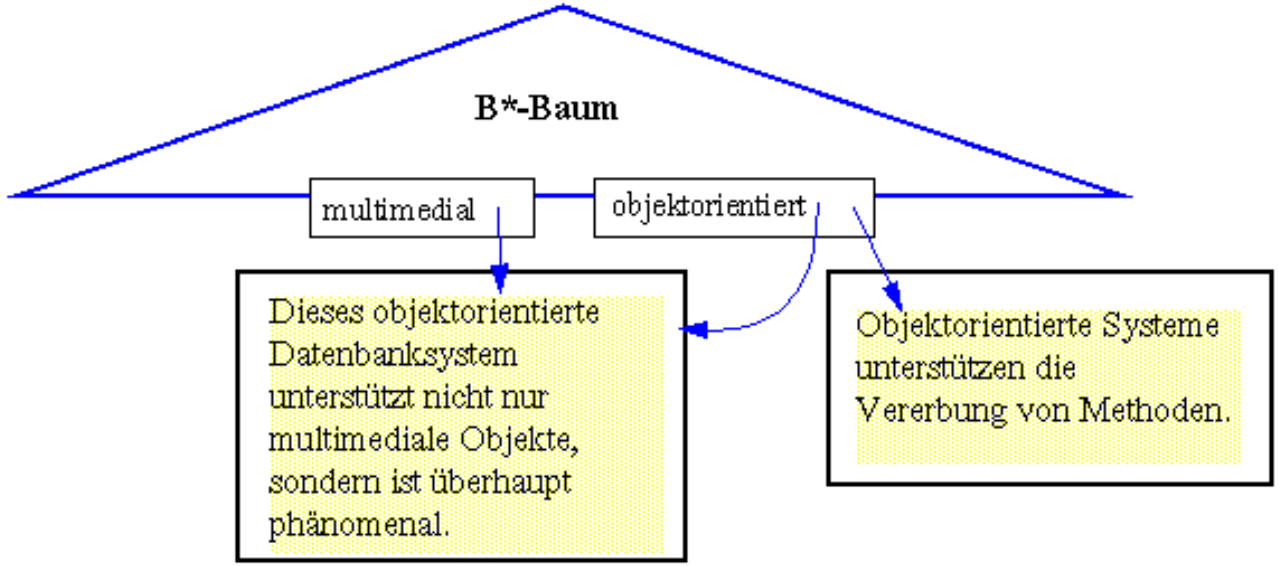
Zugriffskosten mittels invertierter Liste

Zugriffskosten werden durch Datenstruktur zur Verwaltung der invertierten Liste bestimmt

- B*-Baum
- Hash-Verfahren ...

Effiziente Realisierung über (indirekten) B*-Baum

- variabel lange Verweis/Zeigerlisten pro Schlüssel auf Blattebene



Boolesche Operationen: Verknüpfung von Zeigerlisten

- Beispiel: Suche nach Dokumenten mit "multimedial" UND "objektorientiert"

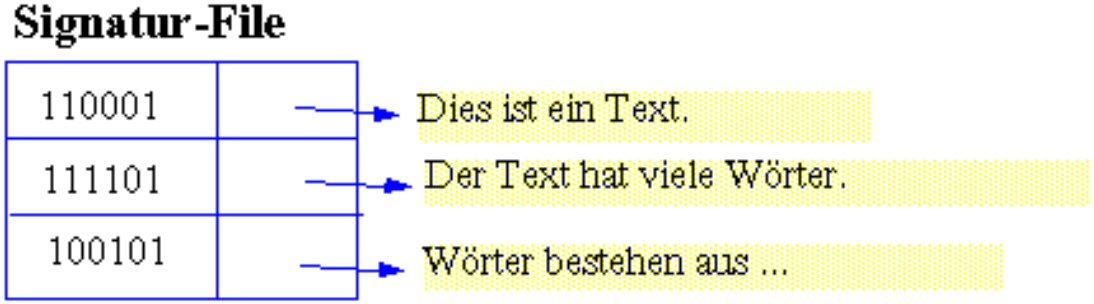
Signatur-Dateien

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 (Dies)	= 010001
s (Text)	= 110000
s (bestehen)	= 100100
s (viele)	= 001100
s (Wörter)	= 100001



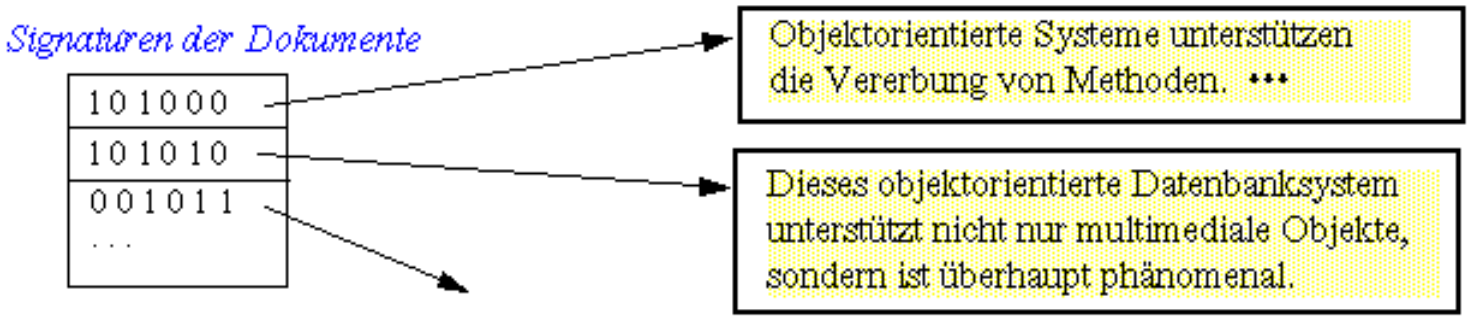
Suchen mit Signaturen

Suchbegriff wird über dieselbe Signaturfunktion 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

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 I

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<= k
- exakte Stringsuche ergibt sich als Spezialfall mit k=0

Beispiel (k=2)

Text: erster testtext
Muster: test
k=2

Approximative Suche II

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: $\text{abs}(n - m) \leq D(s1,s2) \leq \max(m,n)$

Beispiel: Editier-Distanz zwischen "Auto" und „Anton" ?

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:	AGCACACA	AGCACAC-A	AG-CACACA
s2:	ACACACTA	A-CACACTA	ACACACT-A
	Match (A,A)	Match (A,A)	Match (A,A)
	Replace (G,C)	Delete (G, -)	Replace (G,C)
	Replace (C,A)	Match (C,C)	Insert (-, A)
	Replace (A,C)	Match (A,A)	Match (C, C)
	Replace (C,A)	Match (C.C)	Match (A,A)
	Replace (A,C)	Match (A,A)	Match (C,C)
	Replace (C,T)	Match (C,C)	Replace (A,T)
	Match (A,A)	Insert (-,T)	Delete (C,-)
		Match (A,A)	Replace (A,A)

Editierdistanz (2)

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 I

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

Lösung des Optimierungsproblems durch Ansatz der dynamischen Programmierung

- Konstruktion der optimalen Gesamtlösung durch rekursive Kombination von Lösungen für Teilprobleme

Berechnung der Editierdistanz II

Sei s1 = (a_1, ... a_n), s2=(b_1, ...b_m).

D_{ij} sei Editierdistanz für Präfixe (a_1, ... a_i) und (b_1, ...b_j); $0 \le i \le n$; $0 \le j \le m$

- 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

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
- $D_{i,0} = D((a_1,...,a_j),-) = i$ // i Löschungen

Berechnung der Editierdistanz III

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:

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

Beispiel zur Editierdistanz

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)

Stringähnlichkeit für Texte

Die Fragestellung lässt sich mit verschiedener Granularität stellen:

- Satzähnlichkeit: Wann sind sich zwei Sätze sehr ähnlich?
  - Wenn sie durch beinahe den gleichen String repräsentiert werden? Dann ist eine Operation mit Zeichenketten sinnvoll.
  - Wenn sie beinahe die gleiche Aussage enthalten? Hier ist vielleicht eine Operation mit Wörtern statt Zeichenketten sinnvoll.
- Ähnlichkeit von Dokumenten, z.B. HTML-Seiten. Wichtig sind vielleicht nicht alle Wörter, sondern nur inhaltlich wichtige Wörter.
- Ähnlichkeit von Dokumentenkollektionen oder Websites

Anwendung für Stringähnlichkeit: Translation Memories

Teilgebiet beim Maschinellen Übersetzen, bei dem bereits korrekt übersetzte Sätze ggf. wiederverwendet werden sollen.

- Algorithmus:
  - Gegeben ist ein zu übersetzender Satz A.
  - Im Translation Memory wird nach einem Satz A‘ gesucht, der möglichst ähnlich zu A ist.
  - Falls die Ähnlichkeit zwischen A und A‘ einen Schwellwert übersteigt, wird A‘ und seine Übersetzung B‘ ausgegeben. Abweichungen von A werden bei A‘ markiert. Falls möglich, werden auch die entsprechenden Stellen in B‘ markiert.

Zum Einsatz kommt Stringähnlichkeit. Wünschenswert ist eine Toleranz gegenüber

- Wortreihenfolge (freie Satzstellung im Deutschen) und
- Flexion (z.B. Einzahl / Mehrzahl).