
Anfrage-Sprachen

Teil 1

Viele Folien in diesem Abschnitt sind eine deutsche Übersetzung der Folien von Raymond J. Mooney (<http://www.cs.utexas.edu/users/mooney/ir-course/>). Ein weiterer großer Anteil wurde mit freundlicher Genehmigung von Peter Becker übernommen (<http://www2.inf.fh-rhein-sieg.de/~pbecke2m/retrieval/>).

1

Boolesche Anfragen

- Schlüsselworte kombiniert mit Booleschen Operatoren:
 - ODER: (e_1 OR e_2)
 - UND: (e_1 AND e_2)
 - ABER: (e_1 BUT e_2) erfüllt e_1 aber **nicht** e_2
- Negation nur eingeschränkt als ABER erlaubt, um den invertierten Index nutzen zu können: effektives Bearbeiten von e_1 durch inv. Index, dann Filtern der Ergebnisse.
- Problem: Untrainierte Anwender haben Probleme mit Boolescher Logik.

2

Boolesche Retrieval mit invertierten Indizes

- **Einzelnes Schlüsselwort**: Finde die gesuchten Dokumente durch Verwendung des invertierten Index.
- **ODER**: Finde auf rekursive Weise e_1 und e_2 bilde die Vereinigung der Ergebnisse.
- **UND**: Finde auf rekursive Weise e_1 und e_2 und bilde den Durchschnitt der Ergebnisse.
- **ABER**: Finde auf rekursive Weise e_1 und e_2 und bilde die Mengendifferenz der Ergebnisse.

3

“Natürlichsprachige” Anfragen

- Volltext-Anfragen als beliebige Strings.
- Typischerweise werden sie nur wie ein “bag of words” im Vektorraum-Modell behandelt und mit Standard-Vektor-Retrieval-Methoden verarbeitet.

4

Phrasen-Anfragen

- Findet Dokumente, die spezifische Phrasen (geordnete Listen zusammenhängender Wörter) enthalten
 - “information theory”
- Kann intervenierende Stopworte und/oder Stemming zulassen.
 - “buy camera” matches:
“buy a camera”
“buying the cameras”
etc.

5

Phrasen-Retrieval mit invertierten Indizes

- Muss einen invertierten Index verwenden, der auch die *Positionen* der Schlüsselworte in den Dokumenten speichert.
- Finde Dokumente und Positionen für jedes individuelle Wort, bilde Durchschnitt der Treffermengen, und prüfe abschließend auf den richtigen Zusammenhang der Schlüsselwort-Positionen.
- Am besten beginnt man die Prüfung der Zusammenhänge mit dem am wenigsten häufigen Wort in der Phrase.

6

Phrasensuche

Aufgabe: Finde die Menge D von Dokumenten, bei denen alle Schlüsselworte $(k_1 \dots k_m)$ in einer Phrase vorkommen (verwende UND-Anfrageverarbeitung).

Initialisiere eine leere Menge, R , von gefundenen Dokumenten.

Für jedes Dokument d in D :

Suche für jedes k_i die Folge der Positionen P_i der Vorkommen in d

Bestimme die kürzeste Folge P_s der P_i 's

Für jede Position p des Schlüsselworts k_s in P_s

Für jedes Schlüsselwort k_i außer k_s

Verwende die binäre Suche, um eine Position $(p - s + i)$ in der Folge P_i zu finden.

Falls für jedes Schlüsselwort die korrekte Position gefunden wurde, füge d zu R hinzu

Gib R aus.

7

Näherungsanfragen

- Liste von Termen mit zusätzlichen maximalen Abstandsbeschränkungen zwischen den Termen.
- Beispiel: “dogs” und “race” in 4 Worten passt zu “...dogs will begin the race...”
- Man kann zusätzlich Stemming verwenden und/oder Stopwörter ignorieren.

8

Näherungs-Retrieval mit invertiertem Index

- Benutze Ansatz ähnlich wie bei Phrasensuche, um alle Dokumente zu finden, die die Schlüsselwörter enthalten.
- Finde bei der binären Suche nach den Positionen der verbleibenden Schlüsselwörter die nächste Position von k_i nach p und überprüfe, dass diese innerhalb des maximal zulässigen Abstands ist.

9

String-Matching

- Erlaubt allgemeinere Stringvergleiche in Anfragen, während der vorher diskutierte Ansatz die Worte als atomare Einheiten (Tokens) betrachtet.
- Erfordert für effiziente Bearbeitung komplexere Datenstrukturen und Algorithmen als invertierte Indizes.

10

String Matching

- Die Suche von einem Muster in einem Text wird auch als *String Matching* oder *Pattern Matching* bezeichnet.
- Generell besteht die Aufgabe darin, einen String (das *Muster*, *Pattern*) der Länge m in einem Text der Länge n zu finden, wobei $n > m$ gilt.
- Je nach Freiheitsgraden bei der Suche unterscheidet man verschiedene Arten von String-Matching-Problemen.

11

Arten von String-Matching-Problemen

- **Exaktes String-Matching:** Wo tritt ein String pat in einem Text $text$ auf? Beispiel: fgrep
- **Matching von Wortmengen:** Gegeben sei eine Menge S von Strings. Wo tritt in einem Text ein String aus S auf? Beispiel: agrep
- **Matching regulärer Ausdrücke:** Welche Stellen in einem Text passen auf einen regulären Ausdruck? Beispiel: grep, egrep
- **Approximatives String-Matching:** Welche Stellen in einem Text passen am besten auf ein Muster (Best-Match-Anfrage)?

12

Arten von String-Matching-Problemen

- Welche Stellen in einem Text stimmen mit einem Muster bis auf d Fehler überein (Distance-Match-Anfrage)? Beispiel: agrep
- **Editierdistanz:** Wie kann man am “günstigsten” einen String s in einen String t überführen? Beispiel: diff

13

Einfache Strukturen

- **Prefix:** Struktur, die zum Wortanfang passt.
 - “anti” passt zu “antiquity”, “antibody”, etc.
- **Suffix:** Struktur, die zum Wortende passt:
 - “ix” passt zu “fix”, “matrix”, etc.
- **Substring:** Struktur, die zu einer willkürlichen Teilfolge von Zeichen passt.
 - “rapt” passt zu “enrapture”, “velociraptor” etc.
- **Range:** Stringpaar, das zu jedem Wort passt, das lexikographisch (alphabetisch) dazwischen liegt.
 - “tin” to “tix” passt zu “tip”, “tire”, “title”, etc.

14

Anwendungen

- Wo braucht man String-Matching-Verfahren? Z.B.:
 - Volltextdatenbanken
 - Retrievalsysteme
 - Suchmaschinen
 - Bioinformatik
- In diesem Abschnitt lernen wir effiziente Algorithmen für String-Matching kennen.

15

Bezeichnungen

- Ein *Alphabet* ist eine endliche Menge Σ von Symbolen. $|\Sigma|$ bezeichnet die Kardinalität von Σ .
- Ein *String* (*Zeichenkette*, *Wort*) s über einem Alphabet ist eine endliche Folge von Symbolen aus Σ . $|s|$ bezeichnet die *Länge* von s .
- ε bezeichnet den *leeren String*.
- Wenn x und y Strings sind, dann bezeichnet xy die *Konkatenation* von x und y .

16

Bezeichnungen

- $s[i]$ bezeichnet das i -te Element eines Strings s ($1 \leq i \leq |s|$).
- $s[i..j]$ bezeichnet den String $s[i]s[i+1]...s[j]$. Für $i > j$ gelte
- $s[i..j] = \varepsilon$.
- Für einen String s (mit $m = |s|$) bezeichnet s^{-1} die *Umkehrung*
- $s[m]s[m-1]...s[1]$ von s .
- Für zwei Strings x und y gilt $x = y$ genau dann, wenn $|x| = |y| = m$
- und $x[i] = y[i]$ für alle $1 \leq i \leq m$ gilt.
- Wenn $w = xyz$ ein String ist, dann ist x ein *Präfix* und z ein *Suffix*
- von w .
- Gilt $w \neq x$ ($w \neq z$), dann ist x (z) ein *echter Präfix* (*echter Suffix*) von w .

17

Exaktes String-Matching

- **Problem 6.1. [Exaktes String-Matching]**
Gegeben sind die Strings pat und $text$.
 - (a) Man bestimme, ob pat ein Substring von $text$ ist.
 - (b) Man bestimme die Menge aller Positionen, an denen pat in $text$ auftritt. Diese Menge wird mit $MATCH(pat; text)$ bezeichnet.
- Im folgenden wird nur die Variante (a) von Problem 6.1 betrachtet.
- Algorithmen für die Variante (b) erhält man durch einfache Modifikationen der Algorithmen für (a).
- Im folgenden sei $m = |pat|$ und $n = |text|$.

19

Naiver Ansatz

- Ein String x (mit $m = |x|$) heißt *Substring* (*Faktor*) von y , wenn ein i existiert mit $x = y[i..i+m-1]$. Andere Sprechweisen: x tritt in y an Position i auf bzw. Position i ist ein *Match* für x in y .
- x (mit $m = |x|$) heißt *Subsequenz* von y , wenn Positionen $i_1 < i_2 < \dots < i_m$ existieren mit $x = y[i_1]y[i_2]...y[i_m]$.

18

- Der naive Ansatz besteht darin, für jede Position von $text$ (bzw. solange pat ab der aktuellen Position in $text$ passt) von neuem zu testen, ob pat an dieser Position auftritt.
- Das allgemeine Schema für solch einen naiven Algorithmus lautet:
 - **for** $i := 1$ **to** $n-m+1$ **do**
man prüfe, ob $pat = text[i..i+m-1]$ gilt
- Die Prüfung kann nun “von links nach rechts” oder “von rechts nach links” erfolgen.
- Dies führt zu unterschiedlichen naiven Algorithmen und darauf aufbauend zu unterschiedlichen Ansätzen der Verbesserung:
 - „von links nach rechts“ → Algorithmus von Morris und Pratt
 - „von rechts nach links“ → Algorithmus von Boyer und Moore
- Wir betrachten hier nur die zweite Variante.

20

Der Algorithmus von Boyer und Moore

- Der Algorithmus von Boyer und Moore kann als eine verbesserte Variante eines naiven String-Matching-Algorithmus angesehen werden, bei dem pat mit $text$ von rechts nach links verglichen wird.
- **Algorithmus 6.1 [naives String-Matching von rechts nach links]**

```
i := 1
while i ≤ n-m+1 do
  j := m
  while j ≥ 1 and pat[j] = text[i+j-1] do j := j-1 end
  if j = 0 then return true
  i := i + 1
end
return false
```

21

Analyse des naiven Algorithmus

- **Satz 6.1.** *Der naive Algorithmus 6.1 löst Problem 6.1 in Zeit $O(nm)$ und Platz $O(m)$.*
- Für $pat = ba^{m-1}$ und $text = a^n$ benötigt Algorithmus 6.1 $(n-m+1)m = nm - m^2 + m$ Zeichenvergleiche (Worst Case).
- Bei einem binären Alphabet und zufällig erzeugten pat und $text$ (jedes Zeichen unabhängig und jedes Symbol mit Wahrscheinlichkeit $1/2$) ergibt sich für die durchschnittliche Anzahl an Zeichenvergleichen: $(2-2^{-m})n + O(1)$

22

Analyse des naiven Algorithmus

- Trotz der im Durchschnitt linearen Laufzeit lohnt sich der Einsatz von “besseren” String-Matching-Algorithmen, denn:
 - die nachfolgenden String-Matching-Algorithmen haben sich nicht nur in der Theorie, sondern auch in der Praxis als erheblich effizienter erwiesen, und
 - die Realität gehorcht nicht immer den Gesetzen der Wahrscheinlichkeitstheorie.

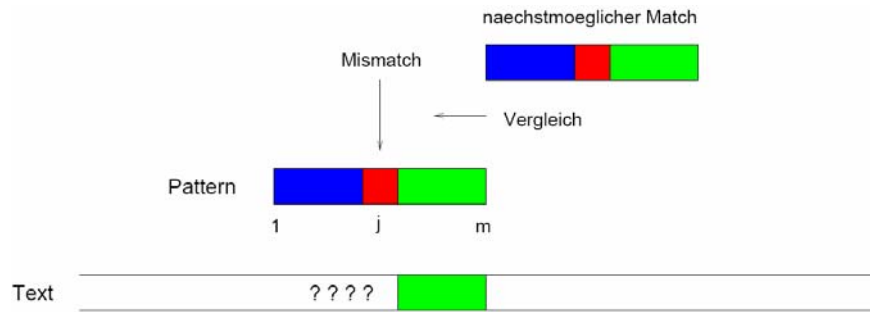
23

Der Algorithmus von Boyer und Moore

- Der Algorithmus von Boyer und Moore basiert auf der folgenden Überlegung:
- Tritt in Algorithmus 1.6 an Stelle j ein Mismatch auf und kommt $pat[j+1..m]$ nicht ein weiteres mal in pat als Substring vor, so kann pat gleich um m Zeichen nach rechts verschoben werden.
- Vergleicht man dagegen von links nach rechts, kann pat nach einem Mismatch an Position j nie um mehr als j Positionen nach rechts verschoben werden. (Dies ist das Prinzip des Algorithmus von Morris und Pratt).

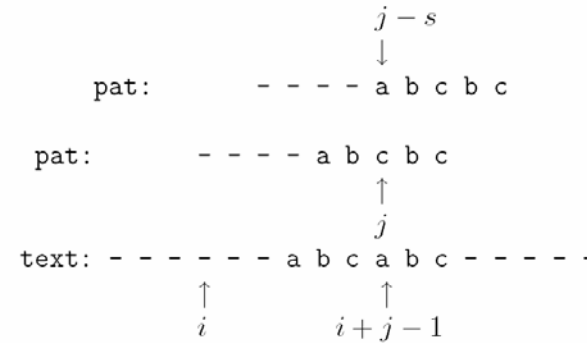
24

Veranschaulichung



25

Veranschaulichung



Damit man keinen Match verpasst, muss s möglichst klein gewählt werden.

27

- Kommt es in Algorithmus 6.1 an Stelle j von pat zu einem Mismatch, so gilt
 $pat[j+1\dots m] = text[i+j \dots i+m-1]$ und
 $pat[j] \neq text[i+j-1]$.

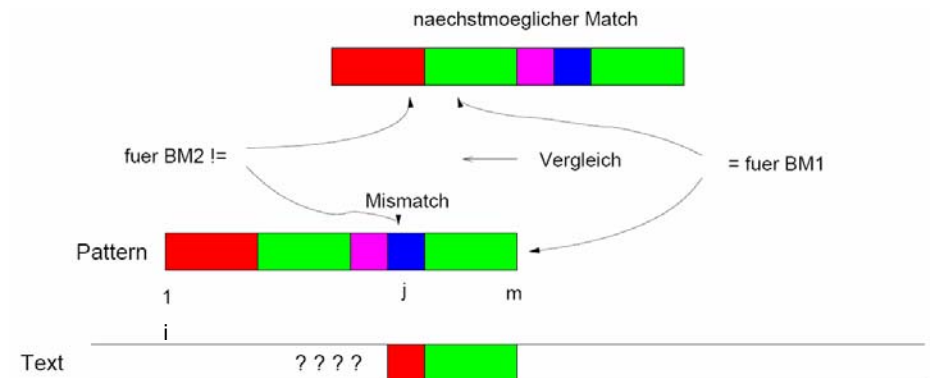
- Dies kann wie folgt ausgenutzt werden:
 Angenommen, pat tritt in $text$ an einer Position $i+s$ mit $i < i+s < i+m$ auf. Dann müssen die beiden folgenden Bedingungen gelten:

$$(BM1) \quad \forall j < k \leq m: \quad k \leq s \vee pat[k-s] = pat[k]$$

$$(BM2) \quad s < j \Rightarrow pat[j-s] \neq pat[j]$$

26

Veranschaulichung



Wdh.: Angenommen, es gibt einen Mismatch an Stelle j von pat , und pat tritt in $text$ an einer Position $i+s$ mit $i < i+s < i+m$ auf. Dann gelten die beiden folgenden Bedingungen:

$$(BM1) \quad \forall j < k \leq m: \quad k \leq s \vee pat[k-s] = pat[k]$$

$$(BM2) \quad s < j \rightarrow pat[j-s] \neq pat[j]$$

28