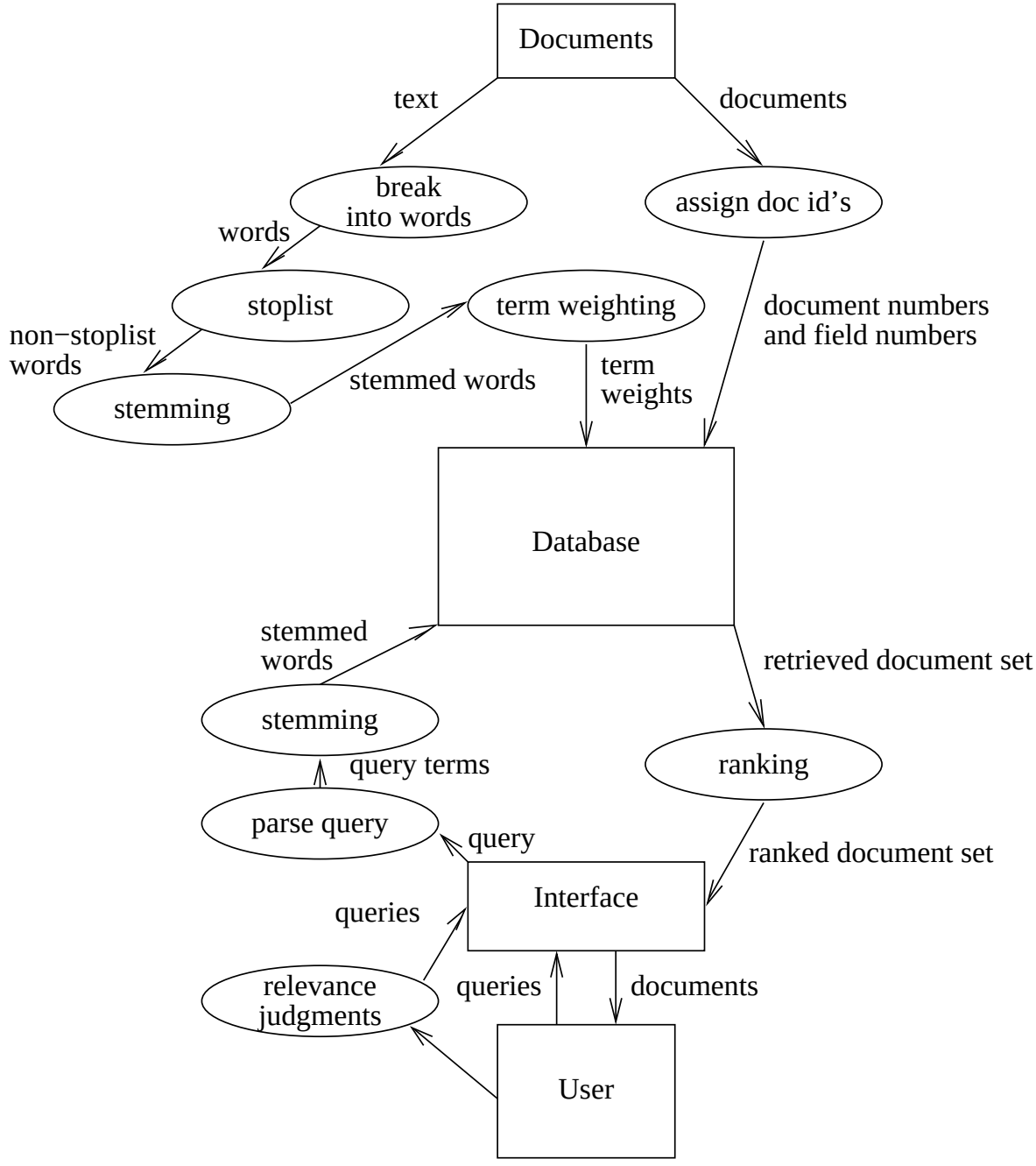
Implementierung von IR-Systemen

Norbert Fuhr

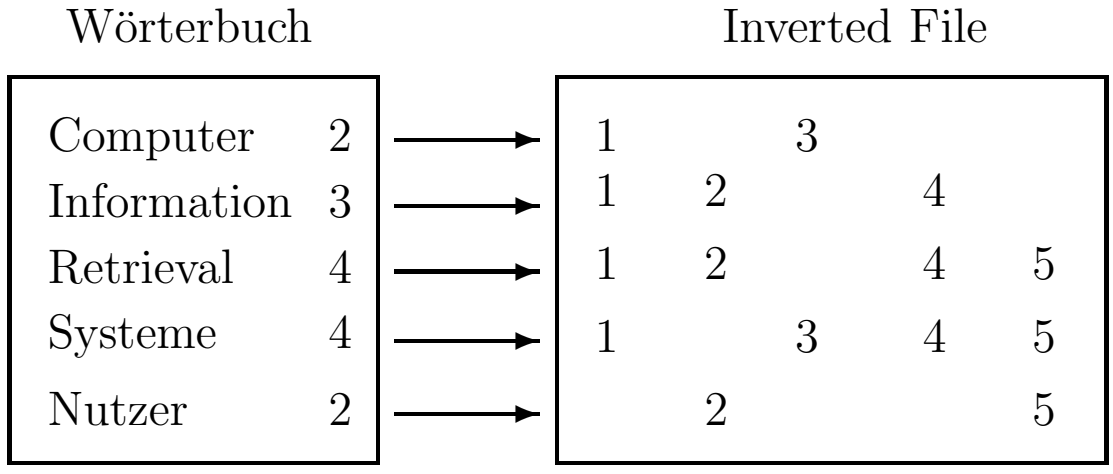
Aufbau von IRS

Funktionale Sicht



Dateistruktur eines IRS

Dokument-Datei / Wörterbuch / inverted File



Dokumentnummer	1	2	3	4	5
Autor	Ash	Brown	Jones	Reynolds	Smith
Titel	Aspekte computer gestutzter Information Retrieval-systeme	Eine Umfrage bei Nutzern von Information Retrieval-systemen	Geschichte der Computer-systeme	Zum Stand der Entwicklung von Retrieval-systemen	Benutzer neuer Retrieval-systeme
Indexierung	Computer Information Retrieval-Systeme	Information Retrieval Nutzer	Computer Systeme	Information Retrieval-Systeme	Retrieval Systeme Nutzer

Dialogfunktionen herkömmlicher IRS

- Zugangskontrolle
- Auswahl der Datenbasis
- Anzeige des Wörterbuchs / Thesaurus
- Formulierung von Anfragen
- Anzeige von Antworten
- Verwaltung von Suchprofilen (einschließlich SDI-Läufe/Downloading)
- Drucken von Antworten

Scanning
Generelle Überlegungen

Verzicht auf Anlegen eines gesonderten Zugriffspfades,
stattdessen möglichst effiziente sequentielle Suche
$\longrightarrow$ erspart den Overhead für das Anlegen des Zugriffspfades

Probleme

Aufwand wächst linear mit dem Datenvolumen, daher nur für kleinere Datenbestände geeignet
(insbesondere auch in Texteditoren eingesetzt)

Berücksichtigung von Flexions- und Derivationsendungen erhöht die Komplexität und den Berechnungsaufwand der Algorithmen

Wortreihenfolge und Stoppwortelimination: dito
information retrieval — retrieval of information

Rankingalgorithmen: schlecht kombinierbar
(inverse Dokumenthäufigkeit steht erst nach dem Durchlaufen aller Dokumente fest)

Anwendungsbereiche:

- hardwaremäßig implementiert:
  Verarbeitungsgeschwindigkeit $\geq$ Transferrate der Plattenlaufwerke
- Highlighting von Suchbegriffen bei der Anzeige von gefundenen Dokumenten
- Vergleichskomponente in Signatur-Systemen
  (Signaturen wirken nur als Filter)

Vorbemerkungen zu Scanning-Algorithmen

im folgenden nur Patterns bestehend aus einer festen Zeichenfolge betrachtet
(keine Alternativen, keine “don’t care’s”)
Notationen:

- n Länge des Textes
- m Länge des Patterns (sei stets $m \leq$n)
- c Größe des zugrundeliegenden Alphabets Σ
- $\bar{C}_n$ Erwartungswert für die Anzahl der zeichenweisen Vergleiche in einem Algorithmus für einen Text der Länge n

Analyse

Analyse basiert auf Annahme einer zufälligen Zeichenkette (Zeichenkette der Länge l besteht aus der Konkatenation von l Zeichen, die unabhängig und gleichverteilt zufällig aus Σ entnommen werden)

Wahrscheinlichkeit für die Gleichheit von zwei zufällig ausgewählten Zeichen: $1/c$

Wahrscheinlichkeit für den match zweier Zeichenfolge der Länge m: $1/c^m$

Erwartungswert der Anzahl Treffer t für ein Pattern der Länge m in einem String der Länge n:

$$E(t) = \frac{n - m + 1}{c^m}$$

Der naive Algorithmus

Pattern: `abracadabra`

```
aababcabcdabracadabra

ab
 abr
  a
   abr
    a
     a
      abr
       a
        a
         a
          abracadabra
```

Algorithmus:

```
public void naivesearch(char[] text, int n, char[] pat, int m) {
/* Search pat[1..m] */
   int i, j, k, lim;

   lim = n-m+1
      for(i = 1; i ≤ lim; i++) {                    /* Search */
         k = i;
         for(j = 1; j ≤ m && text[k] == pat[j]; j++)
            k++;
         if(j > m) Report_match_at_position(i);
      }
}
```

Abschätzung des Aufwands:

Erwartungswert für die Anzahl Vergleiche bis zum ersten Treffer:

$$\bar{C}_{firstmatch} = \frac{c^{m+1}}{c-1} - \frac{c}{c-1}$$

Erwartungswert für die Gesamtzahl der Vergleiche:

$$\bar{C}_n = \frac{c}{c-1}\left(1 - \frac{1}{c^m}\right)(n-m+1) + O(1)$$

(worst case erfordert $m \cdot n$ Vergleiche)

Verbesserung des naiven Algorithmus':

bestimmte Rechnerarchitekturen bieten speziellen Maschinenbefehl zur Suche nach dem ersten Auftreten eines bestimmten Zeichens (bzw. aus einer Menge von Zeichen)
(IBM/360-Architektur: Befehl "Translate and Test")

$\rightarrow$ Einsatz für die Suche nach dem ersten Zeichen des Patterns

Der Knuth-Morris-Pratt-Algorithmus

Grundidee:
wenn bereits eine teilweise Übereinstimmung zwischen Pattern und String gefunden wurde, bevor das erste verschiedene Zeichen auftritt, kann diese Information zur Wahl eines besseren Aufsetzpunktes gewählt werden

```
aababrabrabracadabra

ab
 abr
   abrac
       brac
          bracadabra
```

Beobachtungen:

- weniger Aufsetzpunkte als beim naiven Algorithums
- Zeiger im String muss nie zurückgesetzt werden

Vorprozessierung des Patterns notwendig:
Tabelle `next[1..m]` gibt die naechste Position im Pattern an, mit der bei Ungleichheit verglichen werden muss:

$$
\begin{aligned}
next[j] \quad = \quad & \max\{i | (pattern[k] = pattern[j - i + k] \\
& \text{for } k = 1, \ldots, i - 1) \\
& \text{and } pattern[i] \neq pattern[j]\}
\end{aligned}
$$

(Suche nach dem längsten uebereinstimmenden Praefix, so dass das naechste Zeichen im Pattern verschieden ist von dem Zeichen, bei dem die Ungleichheit auftrat)

Tabelle `next` *fuer den Pattern* `abracadabra`*:*

```
          a b r a c a d a b r a
next[j]   0 1 1 0 2 0 2 0 1 1 0 5
```

`next[i]`=0 $\rightarrow$ Zeiger im Text um eins vorruecken und wieder mit dem Anfang des Patterns vergleichen
`next[m+1]` definiert Aufsetzpunkt im Fall eines Matchs

Knuth-Morris-Pratt-Algorithmus

```
kmpsearch(char[] text, int n, char[] pat, int m) {
/* Search pat[1..m] in text[1..n] */
   int j, k, resume, matches;
   int next[MAX_PATTERN_SIZE];

   pat[m+1] = CHARACTER_NOT_IN_THE_TEXT;
   initnext(pat, m+1, next);              /* Preprocess pattern */
   resume = next[m+1];
   next[m+1] = -1;
   j = k = 1;
```

```
while (k ≤ n) {                                          /*Search*/
if(j == 0 || text[k]==pat[j]) {
k++; j++;
}
else j = next[j];
if(j > m) {
Report_match_at_position(k-j+1);
j = resume;
}
}
pat[m+1] = END_OF_STRING;
}
```

obere Schranke für den Erwartungswert der Gesamtzahl der Vergleiche:
(bei großen Alphabeten)

$$\frac{\bar{C}_n}{n} \leq 1 + \frac{1}{c} - \frac{1}{c^m}$$

Verringerung des Aufwands beim KMP-Algorithmus im Verhältnis zum naiven:

$$\frac{KMP}{naive} \approx 1 - \frac{2}{c^2}$$

Der Boyer-Moore-Algorithmus

Grundideen:

- Vergleich des Patterns von rechts nach links
- Match-Heuristiken (ähnlich wie bei KMP)
- Vorkommensheuristik

Match-Heuristik:

Shift, so dass an der neuen Vergleichsposition

1. Pattern alle vorher übereinstimmenden Zeichen matcht
2. ein anderes Zeichen als vorher an der Vergleichsposition steht

Beispiel zur Match-Heuristik

```
..xaxraxxxxxxabracadabra

  a
  ra
    bra
            dabra
             abracadabra
```

Implementierung der Match-Heuristik als Tabelle `dd`
(gibt den Shift im Text an, Vergleich jeweils beginnend mit dem letzten Pattern-Zeichen)

$$
\begin{aligned}
dd[j] \;=\; & \min\{s + m - j \mid s \geq 1 \text{ and} \\
& ((s \geq j \text{ or } pattern[j - s] \neq pattern[j]) \text{ and} \\
& ((s \geq i \text{ or } pattern[i - s] = pattern[i]) \\
& \text{for } j < i \leq m)\}
\end{aligned}
$$

Beispiel: Tabelle `dd` *für den Pattern* `abracadabra`*:*

	a	b	r	a	c	a	d	a	b	r	a
	10	9	8	7	6	5	4	3	2	1	0
dd[j]	17	16	15	14	13	12	11	13	12	4	1

Vorkommensheuristik

Ausrichten der Textposition, an der Ungleichheit auftrat, mit dem ersten übereinstimmenden Zeichen im Pattern
Definition einer über das Textzeichen indizierten Tabelle `d` (gibt ebenfalls Shift im Text an, Vergleich jeweils beginnend mit dem letzten Pattern-Zeichen)

$$d[x] = \min\{s | s = m \text{ or } (0 \leq s < m \text{ and } pattern[m-s] = x)\}$$

Beispiel: Tabelle `d` *für den Pattern* `abracadabra`*:*

$$d['a'] = 0 \quad d['b'] = 2 \quad d['c'] = 6 \quad d['d'] = 4 \quad d['r'] = 1$$

(für alle anderen Zeichen `x` *ist* `d[x] = 11`*)*

Algorithmus wählt jeweils den größeren Shift von Match- und Vorkommensheuristik
(gleiche Shift-Strategie nach einem Treffer)

Boyer-Moore-Algorithmus

```
public void bmsearch(char[] text, int n, char [] pat, int m) {
/* Search pat[1..m] in text[1..n] */
   int k, j, skip;
   int dd[MAX_PATTERN_SIZE], d[MAX_ALPHABET_SIZE];
   initd(pat, m, d);                          /* Preprocess the pattern */
   initdd(pat, m, dd);
   k = m; skip = dd[1] + 1;
   while(k ≤ n) {                                          /* Search */
      j = m;
      while(j > 0 && text[k] == pat[j]) {
         j--; k--; }
      if(j == 0) {
         Report_match_at_position(k+1);
         k += skip; }
      else k += max(d[text[k]], dd[j]);  } }
```

Aufwandsabschätzungen

- worst case: $O(n + rm)$ mit $r =$ Anzahl Treffer
  (im ungünstigsten Fall wie naiver Algorithmus)
- untere Schranke für große Alphabete und $m \ll n$:

$$\frac{\bar{C}_n}{n} \geq \frac{1}{m} + \frac{m(m+1)}{2m^2 c} + O(c^{-2})$$

- bei ungleicher Auftretenswahrscheinlichkeit der Zeichen gilt $\bar{C}_n/n < 1$ unter der Voraussetzung

$$c\left(1 - \sum_{i=1}^{c} p_i^2\right) > 1$$

Beispiele:

- $c = 3,\ p_i = 1/3,\ \rightarrow 1$
- $c = 3,\ p_1 = 1/2,\ p_2 = 1/4,\ p_3 = 1/4,\ \rightarrow 15/8$

Der Boyer-Moore-Horspool-Algorithmus

vereinfachte, beschleunigte Variante des Boyer-Moore-Algorithmus'

nur Vorkommensheuristik:
berechne Shift mit dem Zeichen im Text, dessen Position momentan mit dem letzten Zeichen des Patterns korrespondiert

Beispiel

```
..xaxxrbxdabracadabra

  ra
      a
       a
         a
        adabra
                 a
          abracadabra
```

Sonderfall:
wenn Text-Zeichen mit dem letzten Zeichen des Patterns übereinstimmt
(aber weiter vorne ist eine Ungleichheit aufgetreten):
Setze zuerst korrespondierendes Zeichen in der Shift-Tabelle auf den Wert m und berechne dann die Shift-Tabelle für die ersten $m-1$ Zeichen des Patterns:

$$\begin{aligned} d[x] \quad = \quad & \min\{s | s = m \\ & \quad \textit{or } (1 \leq s < m \textit{ and } \textit{pattern}[m-s] = x)\} \end{aligned}$$

Beispiel: Tabelle `d` *für den Pattern* `abracadabra`*:*

$$d['a'] = 3 \quad d['b'] = 2 \quad d['c'] = 6 \quad d['d'] = 4 \quad d['r'] = 1$$

(für alle anderen Zeichen `x` *ist* `d[x] = 11`*)*

Boyer-Moore-Horespool-Algorithmus

```
bmhsearch(char[] text, int n, char[] pat, int m) {
/* Search pat[1..m] in text[1..n] */
  int i, j, k;
  int d[MAX_ALPHABET_SIZE];

  for(j=0; j<MAX_ALPHABET_SIZE; j++)
    d[j] = m;                                   /* Preprocessing */
  for(j=1; j<m; j++)
    d[pat[j]] = m-j;

  pat[0] = CHARACTER_NOT_IN_THE_TEXT;
/* To avoid having code */

  text[0] = CHARACTER_NOT_IN_THE_PATTERN;
/* for special cases */
```

```
i = m;
while(i ≤ n) {                                          /*Search*/
k = i;
for(j=m; text[k] == pat[j]; j--)
k--;
if (j == 0)
Report_match_at_position(k+1);
i += d[text[i]];
}                                  /*restore pat[0] and text[0] if necessary*/
}
```

asymptotischer Aufwand für n und c (mit $c \ll n$ und $m > 4$):

$$\frac{\bar{C}_n}{n} = \frac{1}{m} + \frac{m+1}{2mc} + O(c^{-2})$$

Der Shift-Or-Algorithmus

Echtzeit-Algorithmus, ohne Zwischenspeicherung des Textes
$\rightarrow$ für hardwaremäßige Implementierung geeignet

basiert auf der Theorie der endlichen Automaten:
Vektor von m verschiedene Zustandsvariablen,
$i$te Variable gibt den Zustand des Vergleichs zwischen den Positionen $1, \ldots, i$ des Patterns und den Positionen $(j - i + 1), \ldots, j$ des Textes an
($j =$ aktuelle Textposition)

Zustandsvektor

$i$te binäre Zustandsvariable s_i:
=0, falls letzte i Zeichen übereinstimmen
=1, sonst
Repräsentation des Zustandsvektors *state* als Binärzahl:

$$state = \sum_{i=0}^{m-1} s_{i+1} \cdot 2^i$$

Match endend an der aktuellen Position, wenn $s_m = 0$ (bzw. $state < 2^{m-1}$)

Update des Zustandsvektors

beim Lesen eines neuen Zeichens aus dem Text:

- Statusvektor um 1 nach links shiften und $s_1 = 0$ setzen
- Aktualisieren des Statusvektors entsprechend dem nächsten eingelesenen Zeichen
  (mit Hilfe einer Tabelle T mit Einträgen für jedes Zeichen des Alphabets)

  neuer Statusvektor ergibt sich aus Oder-Verknüpfung von altem Vektor mit Tabelleneintrag

Formal:

$$state = (state << 1) \text{ or } T[currchar]$$

($<<$ = Linksshift)

Beispiel für Tabelle T:
Alphabet: $\{a, b, c, d\}$ Pattern: *ababc*

`T[a]=11010` `T[b]=10101` `T[c]=01111` `T[d]=11111`

Definition der Tabelle T: $$T_x = \sum_{i=0}^{m-1} \delta(pat_{i+1} \neq x) \cdot 2^i$$

mit $\delta(C) = 1$, falls die Bedingung C erfüllt ist (sonst 0)

Beispiel für die Suche nach `ababc` im Text `abdababababc`:

Text	:	a	b	d	a	b	a	b	a	b	c
T[x]	:	11010	10101	11111	11010	10101	11010	10101	11010	10101	01111
shift	:	11110	11100	11010	11110	11100	11010	10100	01010	10100	01010
shift-or	:	11110	11101	11111	11110	11101	11010	10101	11010	10101	01111

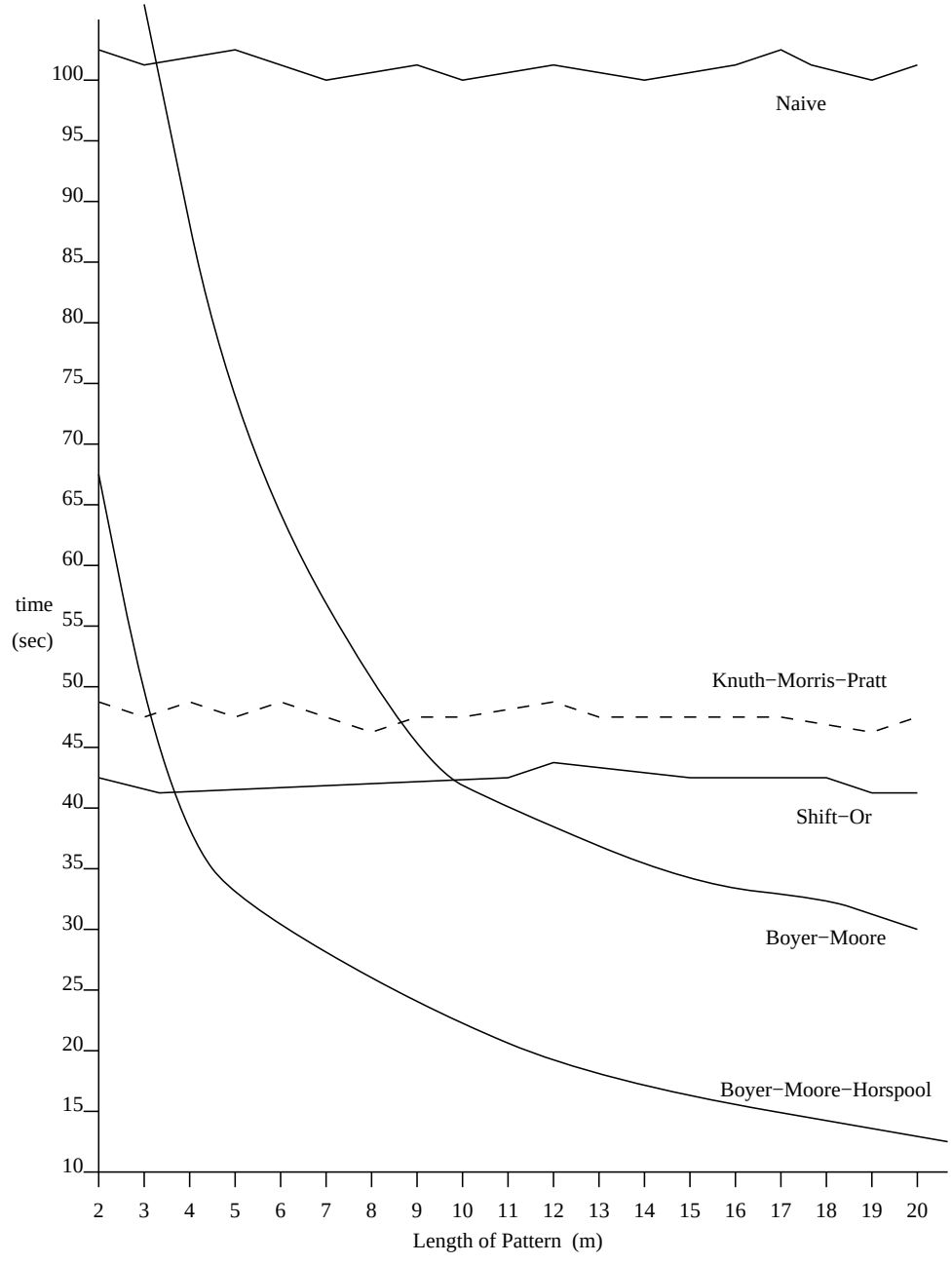
Shift-Or-Algorithmus

```
public static void sosearch(char[] text, int n, char[] pat, int m) {
/* Search pat[1..m] in text[1..n] */
   int start, end, current;
   long state, lim, mbits, hit;
   long MAXSYM = 65535;                                  /*  2^16 - 1 */
   long WORD = 62;
   long T[] = new long[MAXSYM];
   int i, j;

   if(m > WORD) System.out.println("abort - use pat size <=
word size");

   state = 1; for(j=m; j>0; j--) state *= 2; state -= 1; /*  state
= 2^m - 1 */
   for(i=0; i<MAXSYM; i++) T[i] = state;      /* Initialize T[] */
```

```
/*for pat[j] in T[pat[j]] set bit j to 0*/
mbits = state;          /* Only the first m bits of state may be 1 */
lim = (state ≪ 1) & mbits;

/*state vector for the characters of the pattern*/
for(j=0; j<m; j++, lim = ((lim ≪ 1) & mbits) + 1) T[pat[j]] &=
lim;

hit = (state + 1)/2;          /* comparison value for hit = 2^(m-1) */

/*pattern search*/
for(i=0; i<=n-m+1; i++){
state = ((state ≪ 1) | T[text[i]]) & mbits;
if(state < hit) System.out.println("match at position " + (i-m+1));

}
}
```

Komplexität: $O(\lceil \frac{m}{w} \rceil n)$
(dabei ist $\lceil \frac{m}{w} \rceil$ der Aufwand zur Berechnung eines Shifts bzw. zur Oder-Verknüpfung von Bitstrings der Länge m bei einer Wortlänge von w)

Experimentelle Ergebnisse für englischsprachigen Text



Erweiterungen des Shift-Or-Algorithmus'

Zeichenklassen:

- x bestimmtes Zeichen
- $.$ beliebiges Zeichen
- $[Z]$ Zeichen aus der Menge Z
- $\overline{C}$ Komplementmenge der Klasse C

Beispiel:
$M[ae][ij].\overline{[g-ot-z]}$ *matcht Meier, Majer, Meise, aber nicht Maler oder Maien*

Behandlung durch Änderung der Definition der Tabelle T:

$$T_x = \sum_{i=0}^{m-1} \delta(pat_{i+1} \notin \text{Class}_{i+1}) \cdot 2^i$$

$\rightarrow$ Modifikation der Präprozessierung des Patterns
Algorithmus sonst unverändert!

Beispiel: T zum Pattern $a\overline{b}[ab]b\overline{[a-c]}$:

$$\begin{aligned} T[a] &= 11000 \\ T[b] &= 10011 \\ T[c] &= 11101 \\ T[d] &= 01101 \end{aligned}$$

Zeichenkettensuche mit erlaubten Fehlern

Maximalzahl erlaubter Fehler vorgegeben
$\longrightarrow$ ersetze Bits im Statusvektor durch Zähler für Anzahl der Fehler, Addition der Einträge aus T statt OR-Verknüpfung

Beispiel für die Suche nach `ababc` *mit höchstens 2 Fehlern im String* `abdababababc`*:*

text	:	a	b	d	a	b	a	b	a	b	c
$T[x]$	:	11010	10101	11111	11010	10101	11010	10101	11010	10101	01111
shift	:	99990	99900	99010	91210	22200	23010	40200	03010	40200	03010
shift-or	:	99990	99901	99121	92220	32301	34020	50301	14020	50301	04121
									*		*

Alternative Patterns

Suche nach $p_1 \vee \ldots \vee p_l$

a) mit eigenem Statusvektor für jedes Pattern
   Sei $m_{max} = \max_i(|p_i|)$.
   Aufwand: $O(\lceil \frac{m_{max}}{w} \rceil ln)$

b) Verkettung aller Statusvektoren
   Sei $m_{sum} = \sum_i |p_i|$.
   Aufwand: $O(\lceil \frac{m_{sum}}{w} \rceil n)$

Ähnlichkeit von Zeichenketten

(insbesondere für die Suche nach Eigennamen und zur Korrektur von Tippfehlern,
ersetzt nicht die linguistische Grundform- bzw. Stammformreduktion)

Phonetische Gleichheit

Wörter werden durch einen Phonetisierungsalgorithmus auf einen internen Code abgebildet, phonetisch gleiche Wörter dabei auf den gleichen Code

(z.B. SOUNDEX-Algorithmus bildet den gleichen Code für die englischsprachigen Wörter „Dixon“, „Diksen“ und „Dickson“)

aber:
ähnlich geschriebene Wörter werden häufig auf unterschiedliche Codes abgebildet
z.B. „Rodgers“ und „Rogers“

Damerau-Levenstein-Metrik

Ähnlichkeitsmaß für Zeichenketten, soll Zahl der Tippfehler annähern

vier mögliche Fehler: Einfügung, Löschung, Substitution, Transposition

DL-Metrik berechnet für zwei Zeichenketten die minimale Anzahl Fehler, mit der diese ineinander überführt werden können

Zeichen		Operation	Kosten
M	C	Substitution	1
O	E	Substitution	1
N	N	=	0
S	–	Einfügung	1
T	T	=	0
E	R	halbe Transpos.	1/2
R	E	halbe Transpos.	1/2
		Summe	4

Nachteile der DL-Metrik

- relativ aufwendig zu berechnen
- Beschleunigung der best-match-Suche nur durch Clustering möglich

Ähnlichkeitssuche über Trigrams

(Trigram = Zeichenfolge der Länge 3)

einfaches, aber wirkungsvolles Ähnlichkeitsmaß

Wörter werden auf die Menge der enthaltenen Trigrams abgebildet
'MENGE' $\rightarrow$ *{'_ME', 'MEN', 'ENG', 'NGE', 'GE_'}*
'MENAGE' $\rightarrow$ *{'_ME', 'MEN', 'ENA', 'NAG','AGE', 'GE_'}*

Ähnlichkeitssuche = Suche nach Wörtern, die in möglichst vielen Trigrams mit dem gegebenen Wort übereinstimmen

$$\text{z.B.} \quad \varrho(q,d) = \frac{|q^T \cap d^T|}{|q^T|}$$

ϱ('MENGE','MENAGE') $= 3/6$

Beschleunigung durch spezielle Zugriffspfade:
invertierte Listen oder Signaturen

Invertierte Listen

Prinzipieller Aufbau

aufsteigend sortierte Listen von Dokumentnummern, in denen ein Term vorkommt:

t_1	d_2	d_{15}	d_{23}	d_{89}	...
t_2	d_5	d_{15}	d_{89}	...	

Speicherplatzbedarf:

3 Bytes für d und 1 Byte für $f_{d,t} \rightarrow$ 4 Bytes/Eintrag

Beispiel: 2 GB TREC-Kollektion $\rightarrow$ 733 MB inv. Liste

Anwendung für Boolesches Retrieval:

$\vee$ — Vereinigen der Listen
$\wedge$ — Schneiden der Listen
$\wedge\neg$ — Differenzbildung

Ergebnisliste für $t_1 \vee t_2$

d_2	d_5	d_{15}	d_{23}	d_{89}	...

Ergebnisliste für $t_1 \wedge t_2$

d_{15}	d_{89}	...

Erweiterung der Einträge für die Wortabstandssuche:
Angaben über alle Vorkommen in einem Dokument werden mit abgelegt
(z.B. Feldkennung, Satznummer, Wortnummer)

führt aber zu hohem Speicherplatzbedarf
(bis zu 100% der Primärdaten)

Ranking mit invertierten Listen

Aufgabenstellung:
Bestimmung der k Dokumente mit dem höchsten Retrievalgewicht
Annahmen:

- Skalarprodukt als Retrievalfunktion
- Einträge in der invertierten Liste enthalten zusätzlich das Indexierungsgewicht des Terms

Ziel:
Anzahl der Plattenzugriffe soll minimiert werden
(daher Berechnung nur über die invertierten Listen)

Naiver Algorithmus

Prinzipielle Vorgehensweise:
Mischen der invertierten Listen wie bei ODER-Verknüpfung,
dabei zusätzlich Berechnung der Retrievalgewichte

t_1	d_2, u_{1_2}	$d_{15}, u_{1_{15}}$	$d_{23}, u_{1_{23}}$	$d_{89}, u_{1_{89}}$	...
t_2	d_5, u_{2_5}	$d_{15}, u_{2_{15}}$	$d_{89}, u_{2_{89}}$	...	

Ergebnis:
d_2: $w_1 \cdot u_{1_2}$, d_5: $w_2 \cdot u_{2_5}$, d_{15}: $w_1 \cdot u_{1_{15}} + w_2 \cdot u_{2_{15}}$, d_{23}: $w_1 \cdot u_{1_{23}}$,
d_{89}: $w_1 \cdot u_{1_{89}} + w_2 \cdot u_{2_{89}}$

Algorithmus für Skalarprodukt

1. Für jedes Dokument der Kollektion: Setze Akkumulator A_d auf 0
2. Für jeden Term der Anfrage:
   - 2.1 Hole I_t, die invertierte Liste für t.
   - 2.2 Für jedes Paar $\langle$ Dokumentnummer d, Indexierungsgewicht $u_{d,t}$ $\rangle$ in I_t setze $A_d \leftarrow A_d + w_{q,t} \cdot u_{d,t}$.
3. Bestimme die k höchsten Werte A_d
4. Für jedes dieser k Dokumente d:
   - a) Hole die Adresse von Dokument d.
   - b) Hole Dokument d and präsentiere es dem Benutzer.

Komprimierung invertierter Listen

Idee: Lauflängencodierung
Beispiel:

5, 8, 12, 13, 15, 18, 23, 28, 29, 40, 60

Lauflängen:

5, 3, 4, 1, 2, 3, 5, 5, 1, 11, 20

Codes für Lauflängen

Codierung einer natürlichen Zahl x:

γ-Code:

1. $\lfloor \log_2 x \rfloor + 1$ im 1er-Code
   (d.h. $\lfloor \log_2 x \rfloor$ 1-Bits gefolgt von einem 0-Bit)
2. $x - 2^{\lfloor \log_2 x \rfloor}$ im Binärcode

δ-Code:

1. γ-Codierung von $\lfloor \log_2 x \rfloor + 1$
2. $x - 2^{\lfloor \log_2 x \rfloor}$ im Binärcode

Beispiel:

	Codierungsmethode		
x	γ	δ	Golomb, $b = 3$
1	0,	0,	0,0
2	10,0	100,0	0,10
3	10,1	100,1	0,11
4	110,00	101,00	10,0
5	110,01	101,01	10,10
6	110,10	101,10	10,11
7	110,11	101,11	110,0
8	1110,000	11000,000	110,10

- δ-Code benötigt $\lfloor \log_2 x \rfloor + O(\log \log x)$ Bits
- für $x < 15$ γ-Code meist besser, danach δ-Code nie schlechter
- γ- und δ-Code sind Präfix-frei (keine zusätzlichen Bits, kein Backtracking bei Decodierung)

Generelles Codierungsschema

N Anzahl Dokumente der Kollektion

$V = (v_1, v_2, v_3, \ldots)$ Vektor natürlicher Zahlen mit $v_j \leq N$

Codierung von Lauflänge $x \geq 1$:

1. finde $k \geq 1$ mit $\sum_{j=1}^{k-1} v_j < x \leq \sum_{j=1}^{k} v_j$
2. codiere k in geeigneter Repräsentation
3. berechne Rest $r = x - \sum_{j=1}^{k-1} v_j - 1$

4. Codiere r binär:
   - mit $\lfloor \log_2 v_k \rfloor$ Bits für $r < 2^{\lceil \log_2 v_k \rceil} - v_k$,
   - mit $\lceil \log_2 v_k \rceil$ Bits sonst.

(γ-Code entspricht Codierung mit $V = (1, 2, 4, 8, 16, \ldots)$)

Golomb-Code

benutzt Vektor $V_G = (b, b, b, \ldots)$,
k im 1er Code

Codierung ist optimal für $b = \left\lceil \frac{log(2-p)}{-log(1-p)} \right\rceil$

Annahme: geometrische Verteilung mit p=Wahrscheinlichkeit für das Auftreten eines Terms in einem Dokument
$\rightarrow$ Wahrscheinlichkeit für Lücke der Länge x: $(1-p)^{x-1}p$

Effektive Komprimierung:

- Golomb-Code für Lauflängen
- γ-Code für Vorkommenshäufigkeiten $f_{d,t}$

Boolesches Retrieval

1. For each query term t,
   1.1 Search the vocabulary for t.
   1.2 Record f_t and the address if I_t, the inverted list for t.
2. Identify the query term t with the smallest f_t.
3. Read the corresponding inverted list. Use it to initialize C, the list of candidates.
4. For each remaining term t, in increasing order of f_t,
   4.1 Read the inverted list, I_t.
   4.2 For each $d \in C$, if $d \notin I_t$, then set $C \leftarrow C - \{d\}$.
   4.3 If $|C| = 0$, return, since there are no answers.
5. For each $d \in C$,
   5.1 Look up the address of document d.
   5.2 Retrieve document d and present it to the user.

Berechnungsaufwand zur Dekodierung

Prozessierung konjunktiver Anfragen:

- k Anzahl Dokumente im Zwischenergebnis
- p Häufigkeit des nächsten zu berücksichtigenden Terms (Anzahl Einträge in der invertierten Liste)
- t_d Rechenzeit zur Decodierung eines Eintrags
- T_d Rechenzeit zur Decodierung der invertierten Liste:

$$T_d = t_d p$$

Verbesserung: Zweistufige Struktur invertierter Listen

invertierte Liste:
$\langle 5,1\rangle\langle 8,1\rangle\langle 12,2\rangle\langle 13,3\rangle\langle 15,1\rangle\langle 18,1\rangle\langle 23,2\rangle\langle 28,1\rangle\langle 29,1\rangle....$

Lauflängencodierung:
$\langle 5,1\rangle\langle 3,1\rangle\langle 4,2\rangle\langle 1,3\rangle\langle 2,1\rangle\langle 3,1\rangle\langle 5,2\rangle\langle 5,1\rangle\langle 1,1\rangle....$

Sprünge über je 3 Dokumente:
$\langle\langle 5,a_2\rangle\rangle\langle 5,1\rangle\langle 3,1\rangle\langle 4,2\rangle\langle\langle 13,a_3\rangle\rangle\langle 1,3\rangle\langle 2,1\rangle\langle 3,1\rangle$
$\langle\langle 23,a_4\rangle\rangle\langle 5,2\rangle\langle 5,1\rangle\langle 1,1\rangle\langle\langle 40,a_5\rangle\rangle....$

Codierung der Adressen als Differenzen, Weglassen der Nummer des ersten Dokumentes jeder Gruppe:
$\langle\langle 5,a_2\rangle\rangle\langle 1\rangle\langle 3,1\rangle\langle 4,2\rangle\langle\langle 8,a_3-a_2\rangle\rangle\langle 3\rangle\langle 2,1\rangle\langle 3,1\rangle$
$\langle\langle 10,a_4-a_3\rangle\rangle\langle 2\rangle\langle 5,1\rangle\langle 1,1\rangle\langle\langle 17,a_5-a_4\rangle\rangle....$

Aufwandsabschätzung

- k Anzahl Dokumente im Zwischenergebnis
- p Häufigkeit des nächsten zu berücksichtigenden Terms (Anzahl Einträge in der invertierten Liste)
- L Anzahl Einträge pro Gruppe
- p_1 Anzahl Sprungeinträge in der Liste: $p_1 = \lceil p/L \rceil$
- t_d Rechenzeit zur Decodierung eines Eintrags der inv. Liste
- T_d Rechenzeit zur Decodierung der invertierten Liste:

Annahmen:

1. Anzahl zu decodierender Gruppen: $k/2$
2. Aufwand zur Decodierung eines Sprung-Eintrags: $2t_d$

Gesamtaufwand zur Decodierung:

$$T_d = p_1 \cdot 2t_d + \frac{k}{2} L t_d = t_d \left(p_1 \cdot 2 + \frac{k}{2} p \frac{L}{p} \right) = t_d \left(2p_1 + \frac{kp}{2p_1} \right)$$

wird minimal für

$$p_1 = \frac{\sqrt{kp}}{2}$$

Beispiel:
$k = 60$, $p = 60000$, $L = 63$, $t_d = 2.5\mu s$
ohne Sprungliste: 0.150 s
mit Sprungliste: 0.009 s

Berücksichtigung der zusätzlichen Einlesezeit

(für $L = 63$ wächst invertierte Liste um ca. 3 %)

t_r Zeit zum Einlesen eines Eintrages der invertierten Liste

T Verweilzeit zum Einlesen und Decodieren einer Liste:

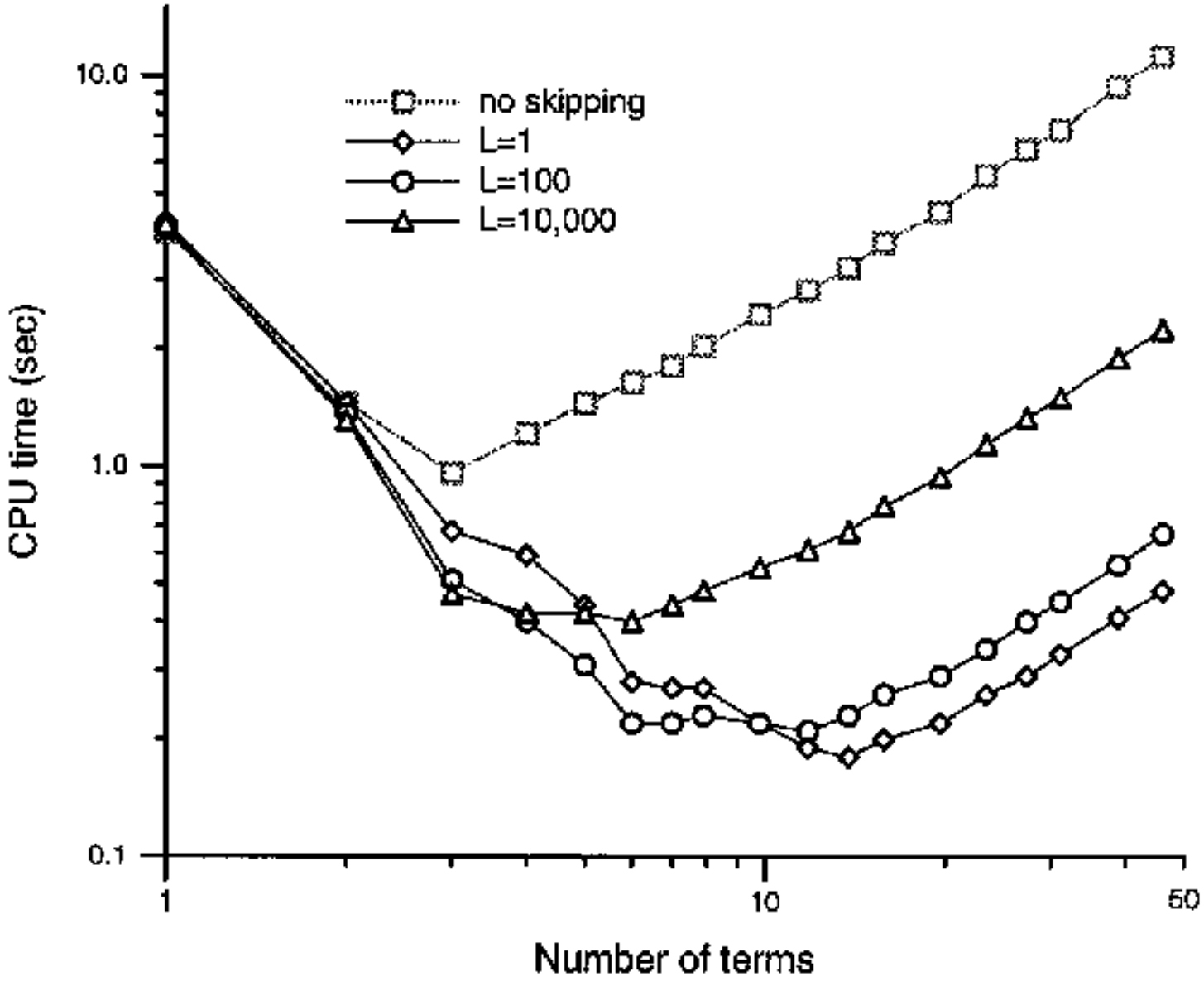
$$T = t_d \left(2p_1 + \frac{kp}{2p_1} \right) + t_r(p + 2p_1)$$

wird minimal für $\quad p_1 = \frac{\sqrt{kp/(1 + t_r/t_d)}}{2}$

Beispiel:
$k = 60$, $p = 60000$, $L = 63$, $t_d = 2.5\mu s$, $t_r = 0.5\mu s$
ohne Sprungliste: 0.180 s, mit Sprungliste: 0.040 s
unkomprimierte Liste: 0.120 s

Speicherplatzbedarf:
komprimierte invertierte Liste: 10% des Textes
mit Sprungliste: 11-12% des Textes

Boolesches Retrieval mit Sprunglisten



Ranking mit invertierten Listen

Naiver Algorithmus

(Algorithmus für Cosinusmaß)

1. For each document d in the collection, set accumulator A_d to zero.
2. For each term t in the query,
   - 2.1 Retrieve I_t, the inverted list for t.
   - 2.2 For each $\langle$document number d, word frequency $f_{d,t}$ $\rangle$ pointer in I_t set $A \leftarrow A_d + w_{q,t} \cdot w_{d,t}$.
3. For each document d, calculate $C_d \leftarrow A_d / W_d$, where W_d is the length of document d, and C_d is the final value of $cosine(d, q)$.
4. Identify the r highest values of C_d, where r is the number of records to be presented to the user.
5. For each document d so selected,
   - 5.1 Look up the address of document d.
   - 5.2 Retrieve document d and present it to the user.

Ranking mit Sprunglisten

a) Quit-Algorithmus

Idee: Häufige Terme (mit niedrigem idf-Gewicht) ignorieren
K: Maximalzahl zu berücksichtigende Dokumente

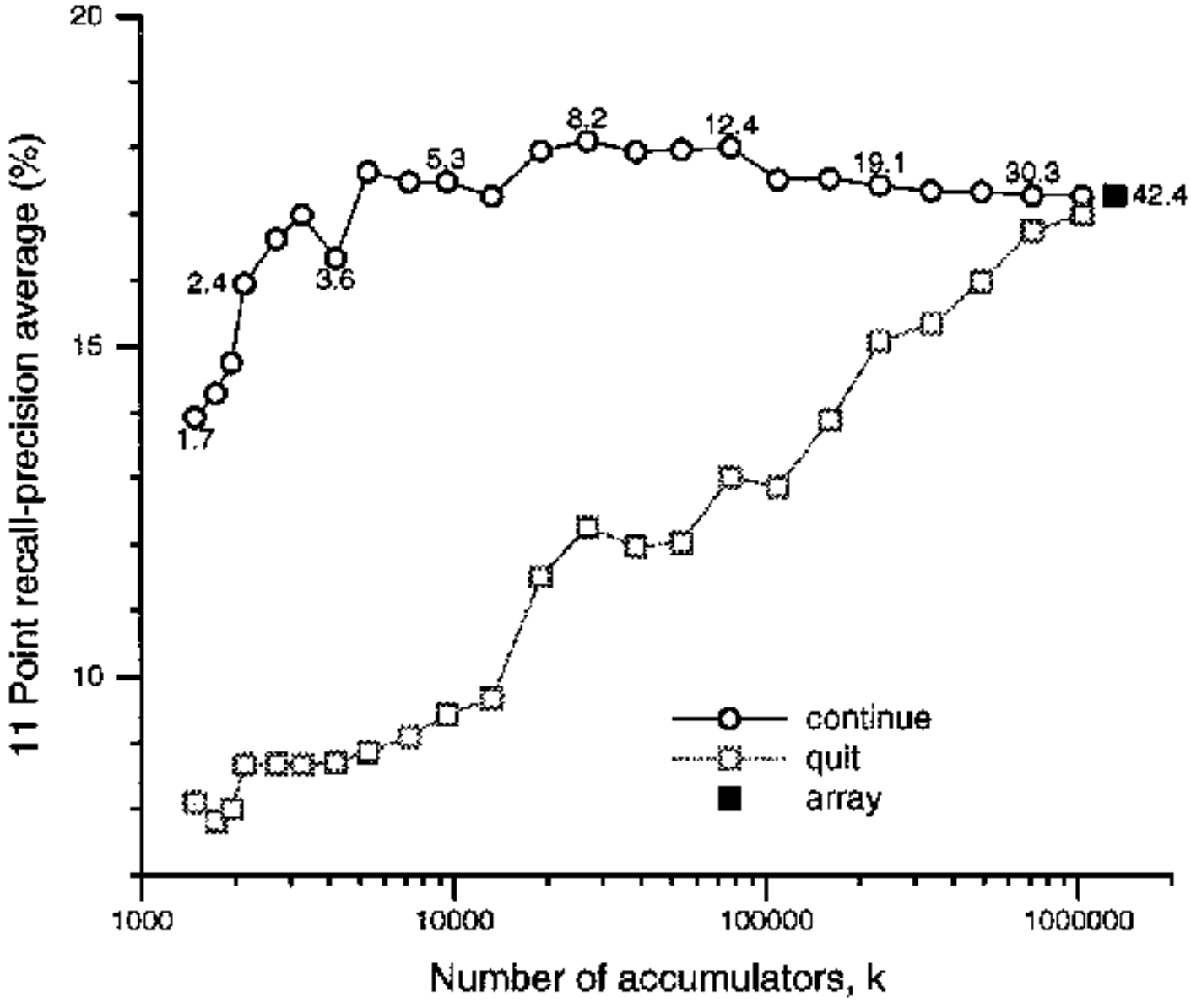
1. Order the words in the query from highest to lowest.
2. Set $A \leftarrow \emptyset$ A is the current set of accumulators.
3. For each term t in the query,
   - 3.1 Retrieve I_t, the inverted list for t.
   - 3.2 For each $\langle d, f_{d,t} \rangle$ pointer in I_t,
     - 3.2.1 If $A_d \in A$, calculate $A_d \leftarrow A_d + w_{q,t} \cdot w_{d,t}$.
     - 3.2.2 Otherwise, set $A \leftarrow A + \{A_d\}$, calculate $A_d \leftarrow w_{q,t} \cdot w_{d,t}$.
   - 3.3 If $|A| > K$, go to step 4
4. For each document d such that $A_d \in A$, calculate $C_d \leftarrow A_d / W_d$.
5. Identify the r highest values of C_d.

b) Continue-Algorithmus

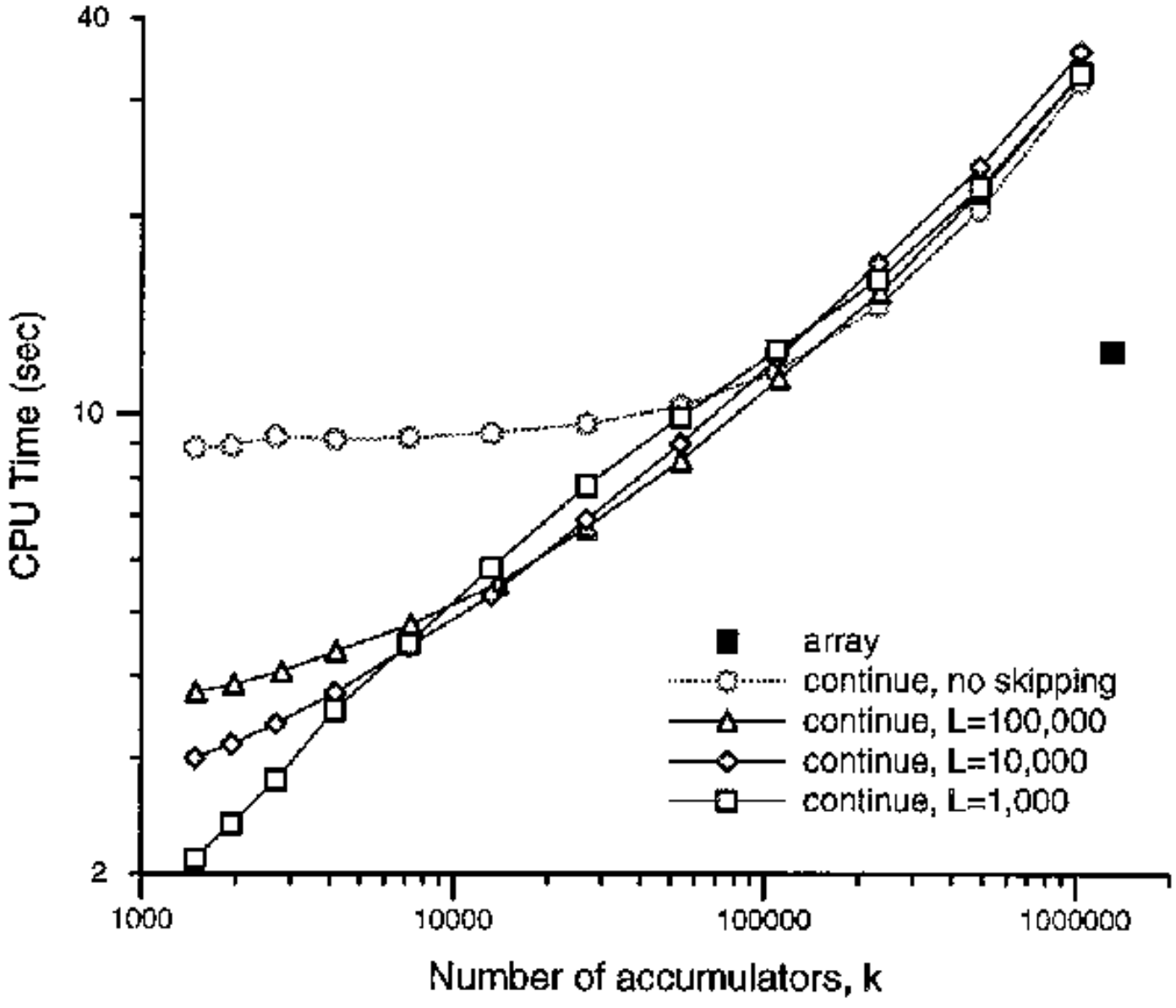
Idee: Häufige Terme nur zur Retrievalwertberechnung berücksichtigen, aber nicht zur Dokumentselektion

1. Order the words in the query from highest to lowest.
2. Set $A \leftarrow \emptyset$.
3. For each term t in the query,
   - 3.1 Retrieve I_t.
   - 3.2 For each $\langle d, f_{d,t} \rangle$ pointer in I_t,
     - 3.2.1 If $A_d \in A$, calculate $A_d \leftarrow A + w_{q,t} \cdot w_{d,t}$.
     - 3.2.2 Otherwise, set $A \leftarrow A + \{A_d\}$, calculate $A_d \leftarrow w_{q,t} \cdot w_{d,t}$.
   - 3.3 If $|A| > K$, go to step 4
4. For each remaining term t in the query,
   - 4.1 Retrieve I_t.
   - 4.2 For each d such that $A_d \in A$,
     if $\langle d, f_{d,t} \rangle \in I_d$, calculate $A_d \leftarrow A_d + w_{q,t} \cdot w_{q,t}$.
5. For each document d such that $A_d \in A$, calculate $C_d \leftarrow A_d / W_d$.
6. Identify the r highest values of C_d.

Retrievalqualität



Effizienz


40
10
2
CPU Time (sec)
1000
10000
100000
1000000
Number of accumulators, k
array
continue, no skipping
continue, L=100,000
continue, L=10,000
continue, L=1,000

PAT-Trees
Grundkonzepte

- Dokumentkollektion als ein String
  `Doc1() Doc2() Doc3( Ch1() Ch2() )`
  `Doc4( Tit() Abstr()`
  `Sec1( Subs1() Subs2() ) Sec2() ) Doc5()`
- Berücksichtigung der Dokumentstruktur bei der Suche möglich
  Suche Section, in der "PAT" vorkommt
- Position = sistring (semi-infinite string)

Definitionen

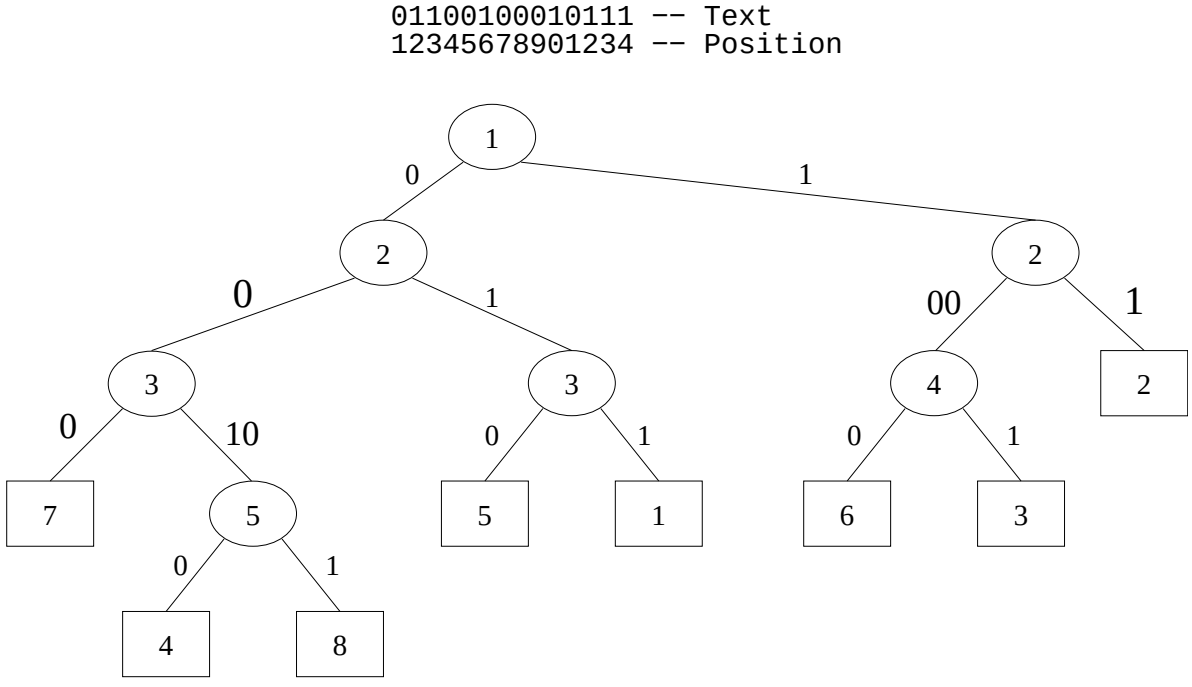
- sistring = String ab Position bis Ende des Gesamtstrings, ID=Position
  String: `THIS IS A SAMPLE STRING`
  sistrings:
  `01 - THIS IS A SAMPLE STRING`
  `02 - HIS IS A SAMPLE STRING`
  `03 - IS IS A SAMPLE STRING`
  `04 - S IS A SAMPLE STRING`
  `05 - IS A SAMPLE STRING`
  `06 - IS A SAMPLE STRING`
  `07 - S A SAMPLE STRING`
  `08 - A SAMPLE STRING`
  ...
- lexikographische Ordnung auf den sistrings
  "A SA..." $<$ *"AMP"* $<$ *"E ST"*

PAT-Tree = Patricia-Tree aller sistrings eines Textes

PAT-Tree = Patricia-Tree aller sistrings eines Textes

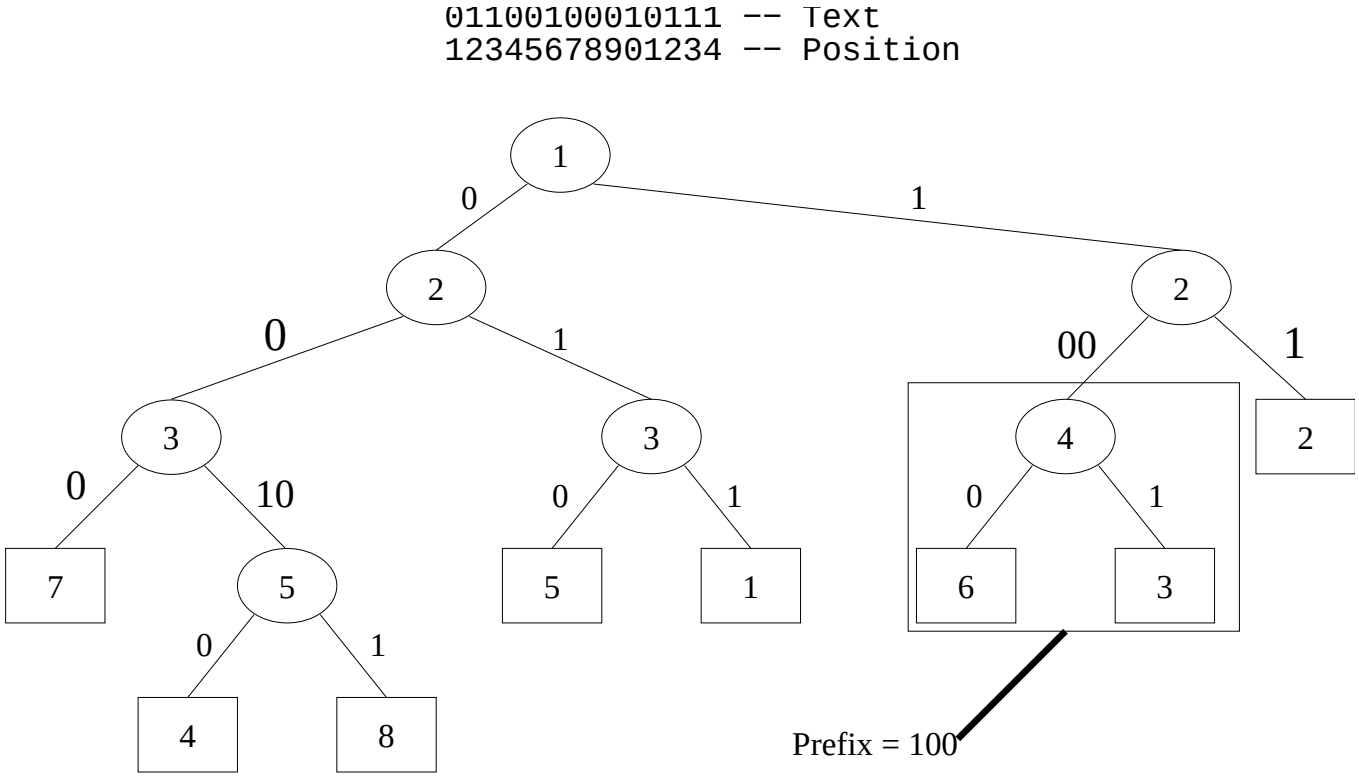
Patricia-Tree:

- Binärer DigitalBaum
- n Blattknoten mit Schlüsselwerten (IDs)
- $n - 1$ interne Knoten
  (Wert = absolute / relative Position im sistring)



Algorithmen auf PAT-Trees

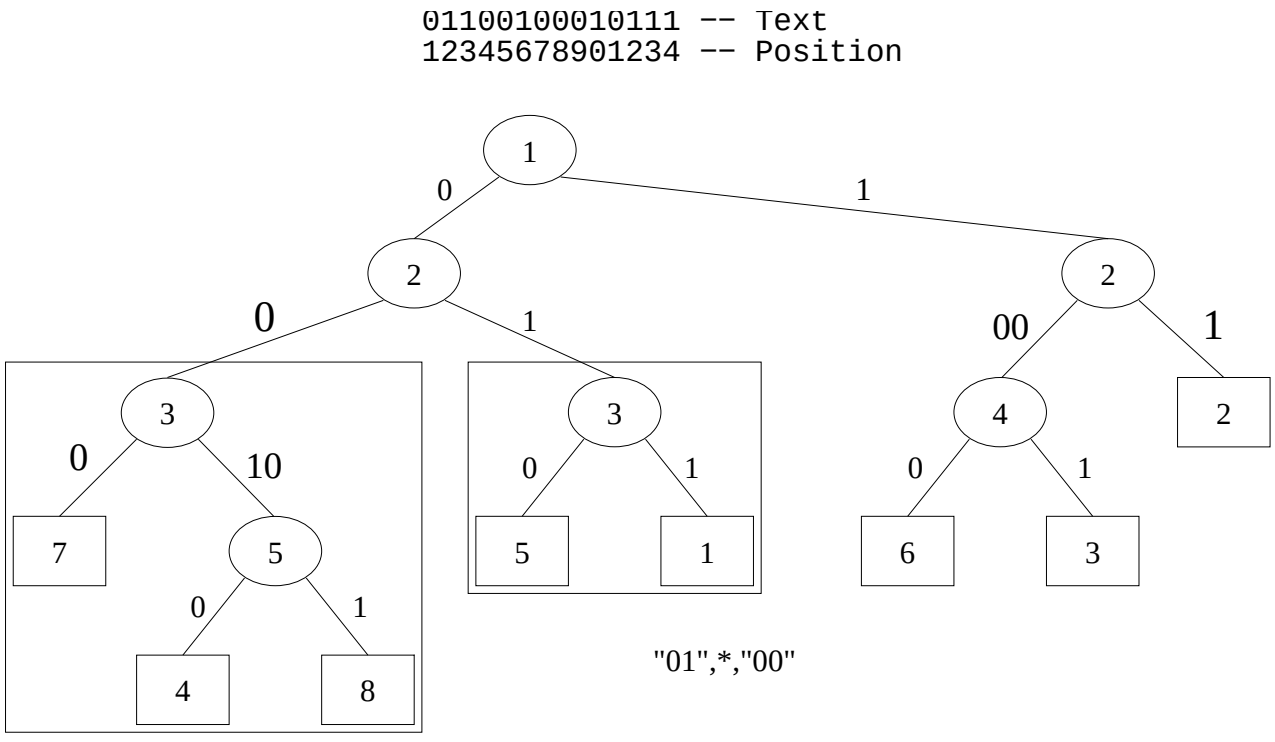
Präfix-Suche



Suche nach 100 liefert Teilbaum mit 3 und 6*

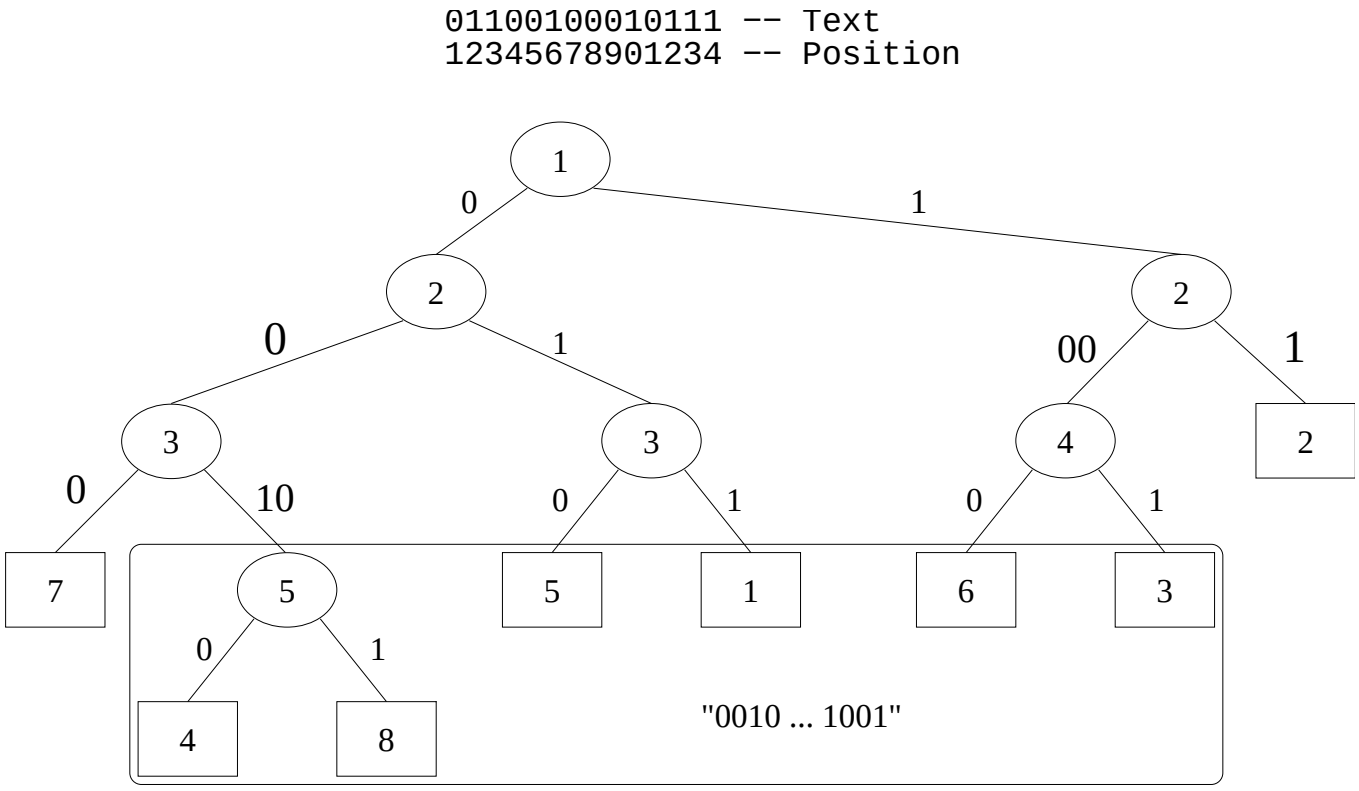
(Suche muß übersprungene Bits kontrollieren)

Reihenfolge

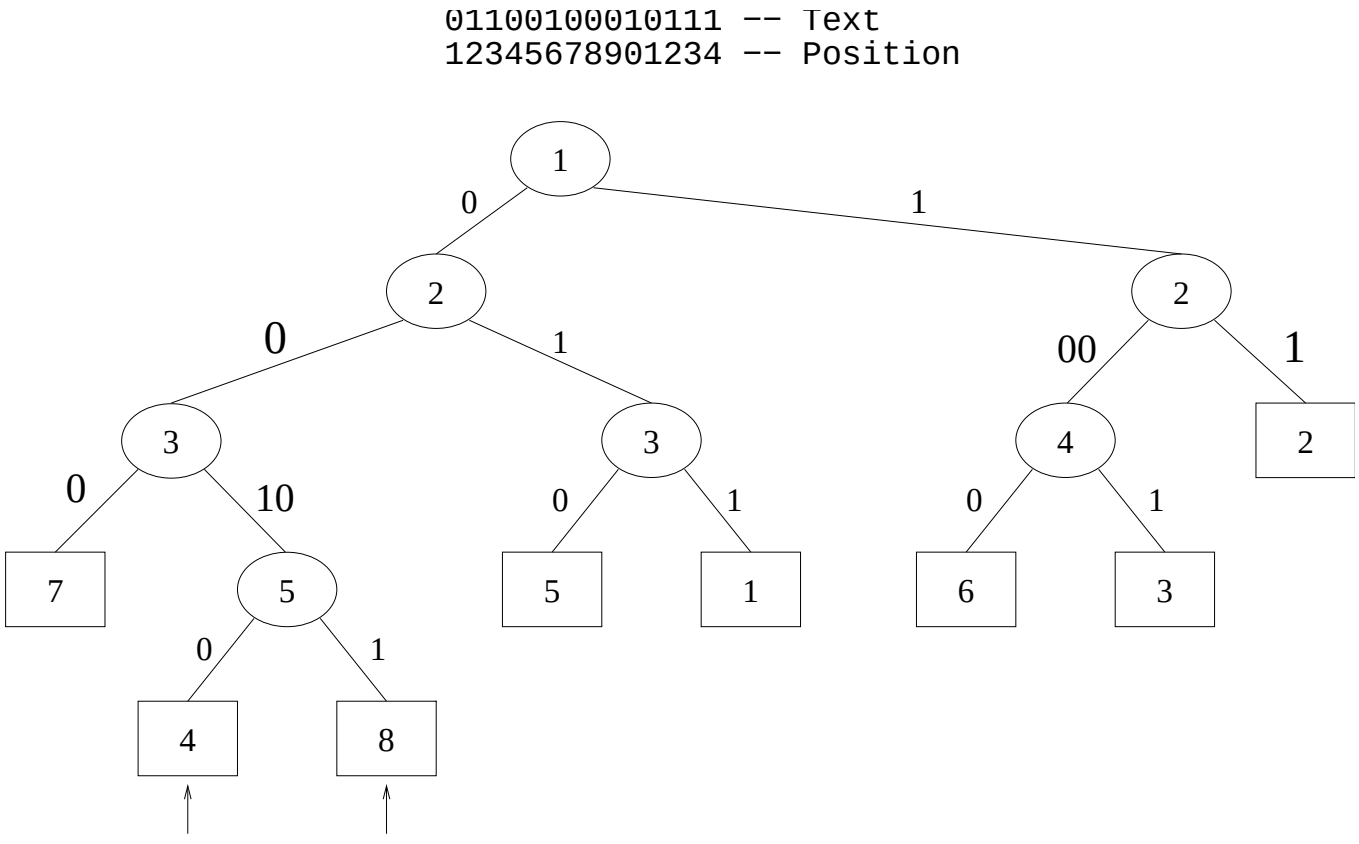


1. Suche der einzelnen Wörter
   (liefert Teilbäume)
2. Bildung aller korrekten Kombinationen von externen Knoten

Bereichssuche

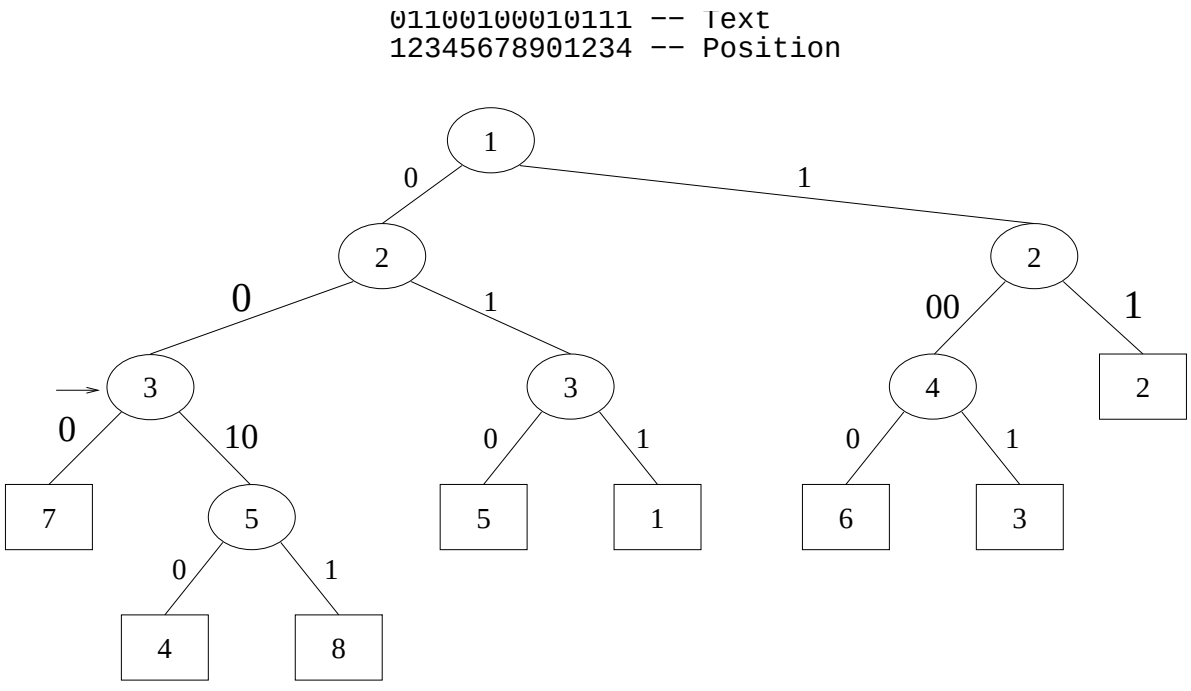


Längste Wiederholung



Suche nach internem Knoten mit dem größten Abstand zur Wurzel

Häufigkeitssuche

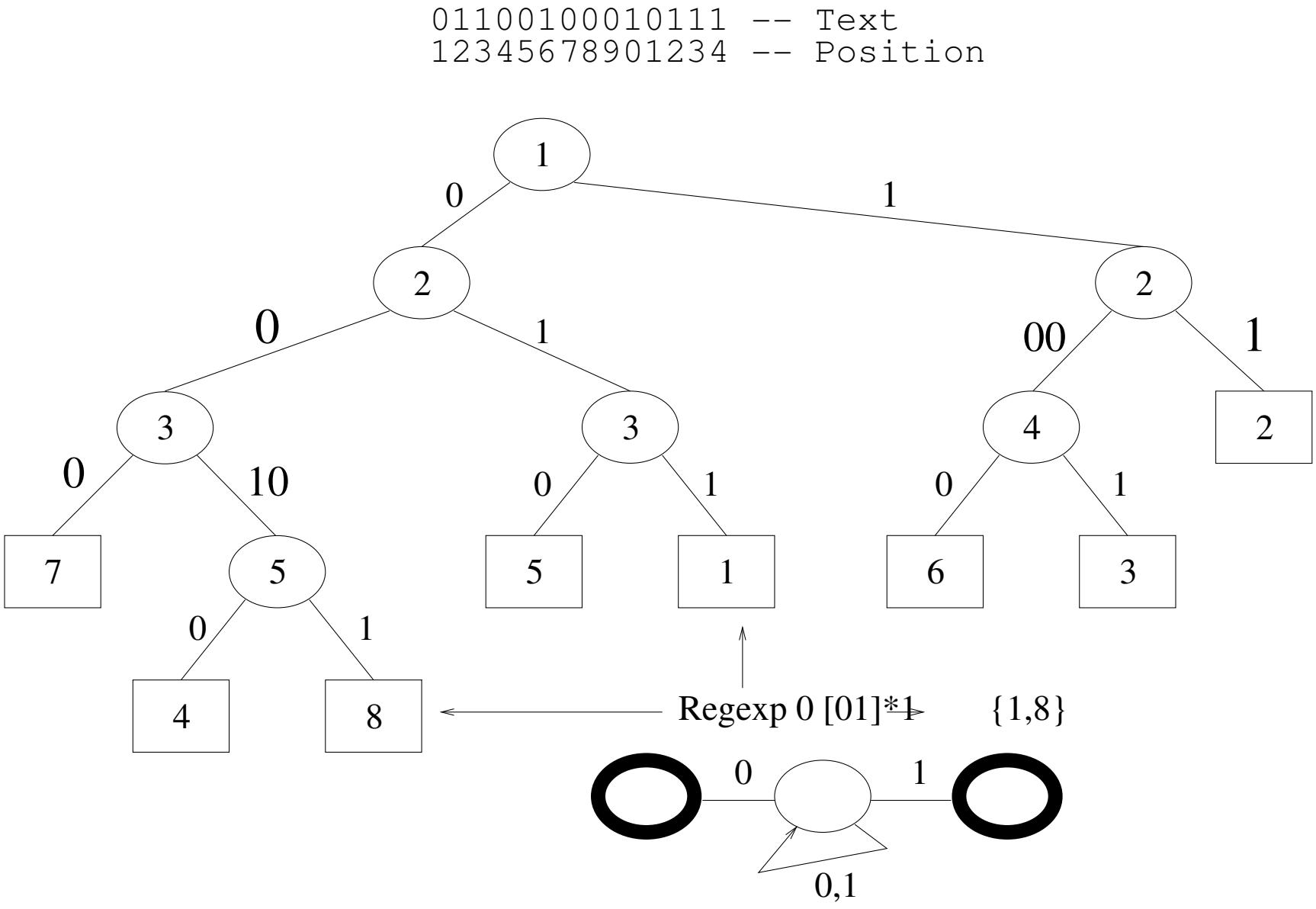


Suche nach internem Knoten mit den meisten Knoten im zugehörigen Teilbaum

häufigstes Bigram = 00

kommt 3mal vor

Reguläre Ausdrücke



PAT Arrays

- Sortierte sistrings $\longrightarrow$ Array mit IDs
- Baumstruktur im Array $\longrightarrow$ $O(n \log n)$ Zugriffe
- Schnelle Präfix- und Bereichssuchen

Signaturen

Das Signaturkonzept

Grundidee

Abbildung von Wörtern und Texten auf Bitstrings fester Länge (=Signaturen)

Suchoperationen auf Signaturen effizienter als auf den Texten, weitere Beschleunigung durch spezielle Speicherungsformen für die Signaturen möglich

Signatur:

$$S :=< b_1, b_2, \ldots, b_L > \quad \text{mit} \quad b_i \in \{0,1\}, L \in \mathcal{N}$$

Erzeugung von Signaturen durch surjektive Abbildung von Wörtern auf Bitstrings (i.a. durch Hashing)

Homonyme: gleiche Signaturen für verschiedene Wörter

Arten von Signaturen:

a) Binärsignaturen:
Abbildung von Wörtern auf alle 2^L möglichen Signaturen

Signaturoperator $=_S$
prüft die Gleichheit von Anfrage- und Satzsignatur

b) überlagerungsfähige Signaturen:
Wert einer Signatur wird nur durch die gesetzten Bits bestimmt

g = Signaturgewicht = # gesetzter Bits
(für alle Wörter gleich)

$\rightarrow$ Abbildung von Wörtern auf $\binom{L}{g}$ verschiedene Signaturen

Überlagerung :

durch ODER-Verknüpfung der Signaturen

text	010010001000	S_1
search	010000100100	S_2
methods	100100000001	S_3
	110110101101	$S_1 \vee S_2 \vee S_3$

Vor- und Nachteile überlagerungsfähiger Signaturen:

- – Entstehung von **Phantomen** (gesetzte Bits sind nicht mehr eindeutig den Ausgangssignaturen zuzuordnen)
- \+ Bildung von Indexstrukturen möglich
- \+ Block Superimposed Coding zur Abbildung einer Menge von Wörtern auf eine einzige Signatur

Signaturoperator $\supseteq_S$
prüft das Enthaltensein der Anfragesignatur in einer Satzsignatur:

$$S \supseteq_S S^Q \quad \Leftrightarrow \quad (\forall)((1 \leq i \leq L) \wedge ((b_i^Q = 1) \Rightarrow (b_i = 1))),$$
$$S, S^Q \in S_L.$$

Zurückführung auf effiziente Bitoperationen:

$$S \supseteq_S S^Q \Leftrightarrow S \wedge S^Q = S^Q \Leftrightarrow (\neg S) \wedge S^Q = 0_S$$

text search methods	110110101101
in search of knowledge-based IR	010110101110
an optical system for full text search	010110101100
the lexicon and IR	101001001001

Anfrage:

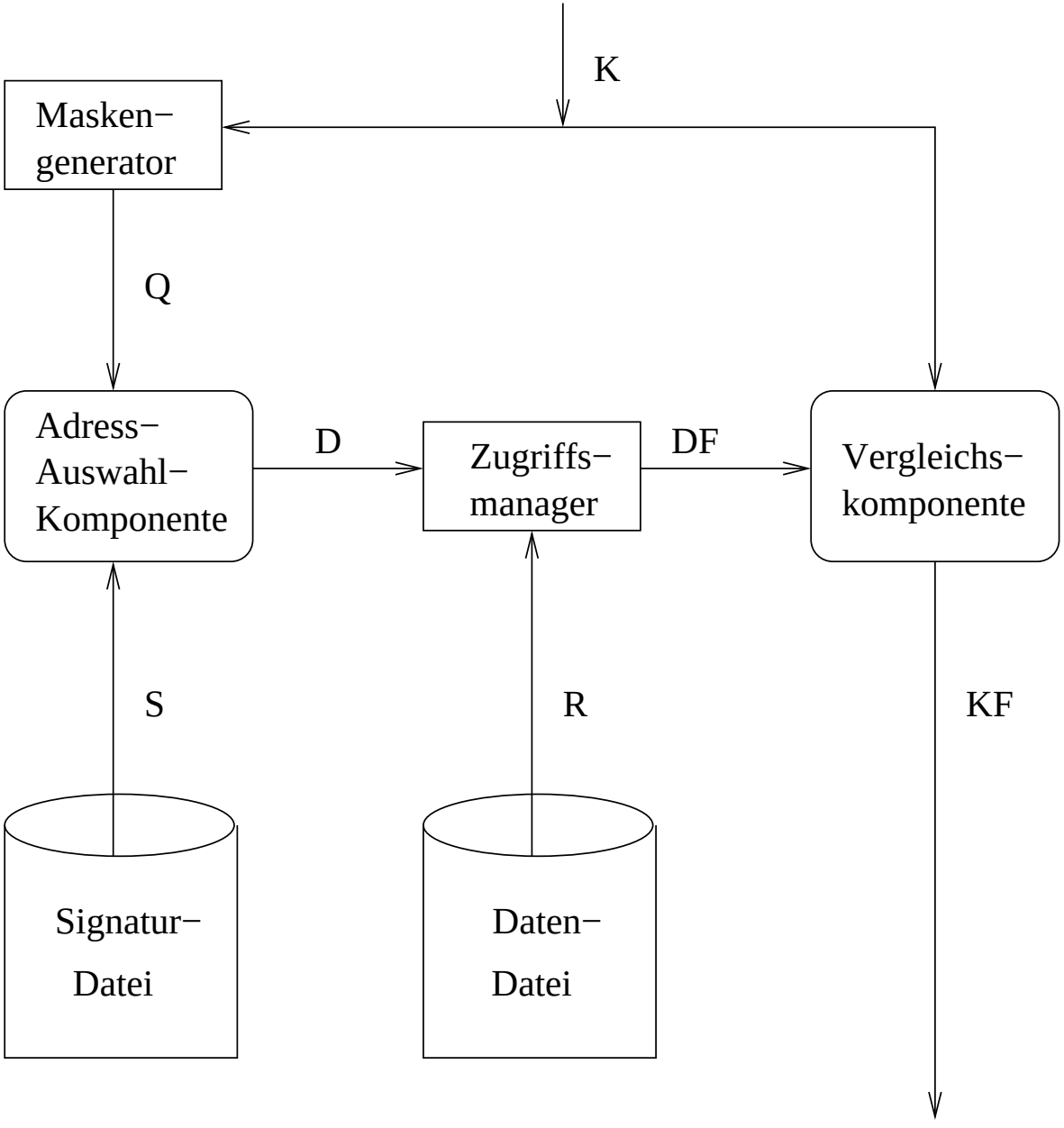
text search	010010101100

false drops:

fehlerhafte Antworten (entstehen durch Homonyme und Phantome)

Im Folgenden nur überlagerungsfähiger Signaturen

Prinzipielle Organisation eines Signatur-Systems



Codierungsmethoden
Disjoint Coding

(auch word coding genannt, wenn auf Wörter angewendet)
Jedes Wort wird einzeln auf eine Signatur abgebildet, die in dieser Form gespeichert wird (abgesehen von einer möglichen anschließenden Komprimierung, hier nicht betrachtet)

Notationen:

- L Länge der Signatur
- g Signaturgewicht (Anzahl gesetzter 1-Bits)
- SP $= SP(L, g)$: Signaturpotential
  $=$ # verschiedener erzeugbarer Kodierungen

Maximierung des Signaturpotentials

für vorgegebene Signaturlänge L:

$$SP = \binom{L}{g} = \frac{L!}{g!(L-g)!} \qquad \text{maximal für } g = \tfrac{L}{2}$$

Beweisskizze:
Da $\binom{L}{g} = \binom{L}{L-g}$, nur Betrachtung von $g \leq \lfloor \frac{L}{2} \rfloor$ notwendig.

Annahme, dass $\quad SP_1 = SP(L, \lfloor \frac{L}{2} \rfloor)$ und

$$SP_2 = SP(L, \lfloor \tfrac{L}{2} \rfloor - 1) = SP_1 \cdot \frac{g}{L-(g-1)}.$$

Wegen $g \leq \frac{L}{2}$ folgt $\quad SP_2 \leq SP_1 \cdot \left(\frac{\lfloor \frac{L}{2} \rfloor}{\lfloor \frac{L}{2} \rfloor + 1} \right) < SP_1$

Anschließend Induktionsbeweis über g und über L

Fehlerrate

F Fehlerrate = # zu erwartende Fehler
(fälschlicherweise gefundene Signaturen)

W Wörterbuchgröße
(# Types= #verschiedener Wörter)

N # Satzsignaturen

Abbildung der Wörter auf Signaturen:
W verschiedene Wörter auf $SP = \binom{L}{\lfloor \frac{L}{2} \rfloor}$ verschiedene Signaturen
$\rightarrow$ einer Signatur sind im Mittel $\frac{W}{SP}$ Types zugeordnet
Retrieval für ein bestimmtes Wort liefert $\frac{W}{SP} - 1$ Signaturen zu anderen Wörtern
$\rightarrow$ Erwartete Fehlerrate:

$$F = \left(\frac{W}{SP} - 1 \right) \frac{N}{W} \tag{1}$$

Festlegung der Signaturlänge für eine bestimmte Anwendung

Signaturpotential als Funktion der Fehlerrate, der Wörterbuchgröße und des Datenvolumens:

$$SP = \frac{W \cdot N}{F \cdot W + N}$$

daraus Berechnung der Signaturlänge möglich

L	g	SP
8	4	70
16	8	12 870
24	12	2 704 156
32	16	601 080 390

Blockweise Codierung von Wörtern

Abbildung der Menge der Wörter, die in einem Textblock auftreten, auf eine Folge von Signaturen

- B Anzahl der Blöcke
- w Anzahl (verschiedener) Wörter pro Block

Fehlerrate bei blockweiser Codierung
bei zufälliger Verteilung von x Token eines Wortes über B Blöcke:
Erwartungswert für die Anzahl Blöcke, in denen das Wort auftritt:

$$B\left(1-\left(1-\frac{1}{B}\right)^{x}\right)$$

Im Mittel $B\frac{w}{SP}$ Token pro Signatur
mit $x = B\frac{w}{SP}$ folgt für erfolglose Anfragen

$$\begin{aligned} F &\approx B\left(1-\left(1-\frac{1}{B}\right)^{B\frac{w}{SP}}\right) \\ &\approx B\left(1-\exp\left(-\frac{w}{SP}\right)\right) \\ &\approx B\frac{w}{SP} \end{aligned}$$

Fehlerwahrscheinlichkeit:

$$\begin{aligned} f &\approx 1-\exp\left(-\frac{w}{SP}\right) \\ &\approx \frac{w}{SP} \end{aligned}$$

Block Superimposed Coding

Überlagerung mehrerer Signaturen (durch ODER-Verknüpfung) erlaubt Abbildung einer Menge von Wörtern (z.B. eines Textblocks) auf eine einzige Signatur

- L Länge der Signatur
- g Gewicht (= Anzahl gesetzter Bits) für ein einzelnes Wort
- λ Anzahl überlagerte Wortsignaturen
- t Anzahl gesetzter Bits in der überlagerten Signatur

Wahrscheinlichkeit, dass durch Überlagerung von λ Wortsignaturen der Länge L mit Gewicht g eine Signatur entsteht, die an t bestimmten Stellen eine 1 enthält:

$$p(L, g, \lambda, t) = \sum_{j=1}^{t} (-1)^j \binom{t}{j} \left(\frac{\binom{L-j}{g}}{\binom{L}{g}} \right)^{\lambda}$$

Für kleine t, λ: $\quad p(L, g, \lambda, t) \approx [p(L, g, \lambda, 1)]^t = (1 - (1 - \frac{g}{L})^{\lambda})^t$

Abschätzung der Fehlerwahrscheinlichkeit

F Anzahl Fehler

N Anzahl Datensätze

$f(t)$ Fehlerwahrscheinlichkeit, $f(t) = F/N$

Annahme dass sich kein Treffer unter den Datensätzen befindet
$\rightarrow$. $f(t) = p(L, g, \lambda, t)$

Speicherungsstrukturen
Sequentielle Signaturen

	b_1^*	b_2^*	b_3^*	b_4^*	b_5^*	b_6^*	b_7^*	b_8^*	@R
S^1	0	0	1	0	1	0	1	1	@r_1
S^2	1	0	1	1	1	0	0	0	@r_2
S^3	0	1	1	0	0	1	1	0	@r_3
S^4	1	0	0	1	0	1	1	1	@r_4
S^5	1	1	1	0	0	1	0	0	@r_5
S^6	0	1	1	0	0	1	0	1	@r_6
S^7	1	0	0	0	1	0	1	0	@r_7
S^8	0	0	0	1	1	1	0	1	@r_8

Sequentielle Speicherung der Signaturen zusammen mit den Adressen der Datensätze

L Länge der Signatur (in Bits)

$size_{@}$ Länge einer Adresse

$size_p$ Seitengröße

$size_r$ Größe eines Datensatzes

N Anzahl Datensätze

$$M \text{ Anzahl Datenseiten} = \left\lceil \frac{N}{\left\lfloor \frac{size_p}{size_r} \right\rfloor} \right\rceil$$

Platzbedarf für eine Signatur mit Adresse: $\lceil \frac{L}{8} \rceil + size_{@}$

$$\text{Anzahl Einträge pro Seite:} \quad K = \left\lfloor \frac{size_P}{\lceil \frac{L}{8} \rceil + size_{@}} \right\rfloor$$

Speicherplatzbedarf (in Seiten)

$$Seq^S = \left\lceil \frac{N}{K} \right\rceil + M$$

Anzahl Seitenzugriffe für Datenbank-Operationen

F Anzahl false drops

D Anzahl echter Treffer

Retrieve:

$$Seq^R = \left\lceil \frac{N}{K} \right\rceil + F + D$$

Insert: Bei Freispeicherverwaltung in Listenform je ein Lese- und Schreibzugriff für Signatur- und Datenseite

$$Seq^I = 2 + 2 = 4$$

Delete: Annahme: Adresse des Datensatzes bekannt $\rightarrow$ sequentielle Suche in den Signaturseiten

$$Seq^D = \left\lceil \frac{N}{2 \cdot K} \right\rceil + 1 + 2 = \left\lceil \frac{N}{2 \cdot K} \right\rceil + 3$$

Bitscheibenorganisation

	b_1^*	b_2^*	b_3^*	b_4^*	b_5^*	b_6^*	b_7^*	b_8^*	@R
S^1	0	0	1	0	1	0	1	1	@r_1
S^2	1	0	1	1	1	0	0	0	@r_2
S^3	0	1	1	0	0	1	1	0	@r_3
S^4	1	0	0	1	0	1	1	1	@r_4
S^5	1	1	1	0	0	1	0	0	@r_5
S^6	0	1	1	0	0	1	0	1	@r_6
S^7	1	0	0	0	1	0	1	0	@r_7
S^8	0	0	0	1	1	1	0	1	@r_8

Speicherung jeder Bitscheibe allein auf einer Seite,
Vektor mit Datensatzadressen getrennt

Anfrage: $S^i \supseteq_S < 10100000 >$

$$b_r^* \quad = \quad b_1^* \wedge b_3^* = \begin{pmatrix} 0 \\ 1 \\ 0 \\ 1 \\ 1 \\ 0 \\ 1 \\ 0 \end{pmatrix} \wedge \begin{pmatrix} 1 \\ 1 \\ 1 \\ 0 \\ 1 \\ 1 \\ 0 \\ 0 \end{pmatrix} = \begin{pmatrix} 0 \\ 1 \\ 0 \\ 0 \\ 1 \\ 0 \\ 0 \\ 0 \end{pmatrix}$$

Ergebnisbitliste: $b_r^* = \bigwedge_{j=q_1}^{q\gamma(S_Q)} b_j, \quad q_i \in \{q | (1 \leq q \leq L) \wedge b_q^Q = 1\}$

Adresse der Trefferkandidaten: $R = \{i | b_r^i = 1\}$

Disjunktive Anfragen

$S(Q_1) \vee S(Q_2) \vee \ldots S(Q_d)$:

Im Prinzip getrennte Prozessierung,
nur Einsparung bei übereinstimmenden 1-Bits in den $S(Q_i)$ möglich

Anzahl Seitenzugriffe für Datenbank-Operationen

Speicherbedarf für eine Bitscheibe: $\lceil \frac{N}{8 \cdot size_p} \rceil$

Retrieve:

$\gamma(Q)$: Anfragegewicht

Zugriffe auf die angesprochenen Bitscheiben +

Zuordnungstabelle und Datenseiten für alle Treffer

$$BS^R = T \cdot \left(\gamma(Q) \left\lceil \frac{N}{8 \cdot size_p} \right\rceil + Z + F + D \right)$$

Z: Anzahl Seitenzugriffe auf die Zuordnungstabelle

R: # Seiten der Zuordnungstabelle

$$R = \left\lceil \frac{N}{\left\lfloor \frac{size_p}{size_@} \right\rfloor} \right\rceil \qquad Z = R \cdot \left(1 - \left(1 - \frac{1}{R} \right)^{(F+D)} \right)$$

$\rightarrow$ ineffizient bei hohen Anfragegewichten

Insert:

$\gamma(S)$: Signaturgewicht des Datensatzes

Bitscheibenblöcke vorher mit 0 initialisiert $\rightarrow$

Zu ändern:

$\gamma(S)$ Bitscheibenseiten + Zuordnungstabelle + Datenseite

$$BS^I = 2 \cdot \gamma(S) + 2 + 2.$$

Delete:

Suche des Eintrages (mit bekannter Satzadresse) über die Signatur

Zu ändern:

1er-Bitscheibenseiten + Zuordnungstabelle + Datenseite

$$BS^D = \gamma \cdot \left(\left\lceil \frac{N}{8 \cdot size_p} \right\rceil + 1 \right) + Z + 2$$

Speicherplatzbedarf (in Seiten):

$$BS^S = L \cdot \left\lceil \frac{N}{8 \cdot size_p} \right\rceil + \left\lceil N \cdot \frac{size_@}{size_p} \right\rceil + M.$$

Zweistufiges Signaturverfahren

Kombination von Bitscheiben- und sequentieller Organisation
Zwei Signaturen für jeden Datensatz:
(mit unabhängigen Signaturfunktionen berechnet)

1. Signatur wie bei sequentieller Organisation als Bitstring in Seiten gespeichert.
   Signatur-Seiten werden in Segmente unterteilt
2. Signaturen werden segmentweise überlagert, bilden Segmentsignatur.
   Segmentsignaturen werden in Bitscheibenorganisation verwaltet

Anfrageprozessierung:

1. Berechnung der beiden Anfragesignaturen
2. Bestimmung der zu durchsuchenden Segmente über die Segmentsignatur
3. Sequentielles Durchsuchen der Segmente mit der 1. Signatur

Quick Filter

Kombination von Signaturen mit Hashing

- Signaturen sind in Seiten organisiert
- Zuordnung der Signaturen zu den Seiten über Hashing

Lineares Hashing

lineare Hash-Funktion g bildet Schlüssel auf den Adressraum $(0, 1, \ldots, n-1)$ ab, wobei $2^{h-1} < n \leq 2^h$ für ein $h \in N$
h: # Anzahl Stufen der Signaturdatei

g muß Split-Funktion sein, die für jeden Schlüssel K die Bedingung erfüllt:

$$g(K, h, n) = \begin{cases} g(K, h-1, n) & \text{oder} \\ g(K, h-1, n) + 2^h \end{cases}$$

n Primärseiten
jede Primärseite hat 0 oder mehr Überlaufseiten (mit der Primärseite verkettet)

Einfügen eines neuen Schlüssels:

1. Berechnung der Seitennummer $p = g(K, h, n)$.
2. Wenn möglich, Einfügen des Schlüssels in die Seite p.
3. Sonst Abspeichern in einer Überlaufseite zu p.
4. Bei Überlauf wird der Adreßraum von n auf $n + 1$ vergrößert

Vergrößerung des Adreßraums:

SP: Zeiger auf die nächste zu splittende Seite

1. Anlegen einer neuen Primärseite n
2. Verteilung des Inhalts der Seite SP und der zugehörigen Überlaufseiten durch neue Hashfunktion auf die Seiten SP und n.
3. $n := n + 1$
4. h wird erhöht, wenn die Seite 0 gesplittet werden soll.
5. $SP := (SP + 1) \bmod 2^{h-1}$

Hash-Funktion für Signaturen

N Signaturen $S_i =< b_1, \ldots, b_L >$

$$g(S_i, h, n) = \begin{cases} \sum_{r=0}^{h-1} b_{L-r} 2^r, & \text{falls } \sum_{r=0}^{h-1} b_{L-r} 2^r < n \\ \sum_{r=0}^{h-2} b_{L-r} 2^r, & \text{sonst} \end{cases} \qquad (2)$$

Quick Filter: Beispiel

S1: 00011110 S2: 11010001 S3: 00111100
S4: 11000011 S5: 00110110 S6: 11001001

Zu Beginn sei $h = 0$, $n = 1$ und $g(S_i, 0, 1) = 0$

Step 0.	**P0**: empty				$SP = 0, h = 0, n = 1$
Step 1.	**P0**: S1				$SP = 0, h = 0, n = 1$
Step 2.	**P0**: S1 S2				$SP = 0, h = 0, n = 1$
Step 3.	**P0**: S1 S3	**P1**: S2			$SP = 0, h = 1, n = 2$
Step 4.	**P0**: S1 S3	**P1**: S2 S4			$SP = 0, h = 1, n = 2$
Step 5.	**P0**: S3	**P1**: S2 S4	**P2**: S1 S5		$SP = 1, h = 2, n = 3$
Step 6.	**P0**: S3	**P1**: S2 S6	**P2**: S1 S5	**P3**: S4	$SP = 0, h = 2, n = 4$

P0: 00111100	**P1**: 11010001 11001001
P2: 00011110 00110110	**P3**: 11000011

Retrieval

Bestimmung der möglichen Signaturseiten aus der Anfragesignatur:
Anzahl zu lesender Seiten hängt vom Gewicht der Anfragesignatur Q ab, genauer:
enthält Q j Einsen im h-Bit-Suffix $h(Q)$, dann müssen höchstens 2^{h-j} Primärseiten und die zugehörigen Überlaufseiten gelesen werden

Algorithmus:

1. $P := g(Q, h, n)$
2. if $h(Q) \cap P = h(Q)$ then {Signaturseite P lesen}
3. $P := P + 1$
4. if $P < n$ then goto 2
5. end

Beispiele:
Q=xxxxxx11 $\rightarrow$ P3 lesen
Q=xxxxxx01 $\rightarrow$ P1, P3 lesen

Vergleich der Speicherungsstrukturen

	Profil			
Struktur	Retrieve	Insert	Delete	Speicher
sequentiell	selten	dominant!	selten	dominant
Bitscheiben	dominant	wenig	wenig	wenig
zweistufig	dominant	wenig	wenig	irrelevant
Quick Filter	wenig	viel	viel	wenig