ADS: Algorithmen und Datenstrukturen 1

Teil 12

Uwe Quasthoff

Institut für Informatik
Abteilung Automatische Sprachverarbeitung
Universität Leipzig

16. Januar 2018

[Letzte Aktualisierung: 15/01/2018, 15:02]

Textsuche I

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

- String Matching
- Pattern Matching
- Sequence Matching

Häufig benötigte Funktion

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

Textsuche II

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 (eine konkrete Instanz) vs.
- Wörtern/Begriffen (mehrere Instanzen einer abstrakten Entität)

Textsuche: Komplexität

Text-Länge n, Anfrage/Query-Länge m.

- **Naive Suche**
  vergleiche Query mit jeder möglichen Start-Position $0 \leq i \leq n - m$.
  Aufwand offenbar $\mathcal{O}(n \times m)$.
  Etwas schlauer: An jedem i, breche Vergleich beim ersten Mismatch ab
  $\Rightarrow$ immer noch $\mathcal{O}(n \times m)$
- **Viele (kurze) Anfragen im selben Text**
  effiziente Index-Strukuren z.B. Suffix-Bäume
  Relativ grosser Overhead; einmalig für die Index-Strukturen,
  aber dann Suche in $\mathcal{O}(m)$ (!)
- **Wenige Anfragen im selben Text**
  aufwendige Indexstrukturen werden unrentabel
  aber Vorverarbeitung der Anfrage kann sich lohnen!
  $\Rightarrow$ Suche in $\mathcal{O}(n + m)$

Knuth-Morris-Pratt (1974)

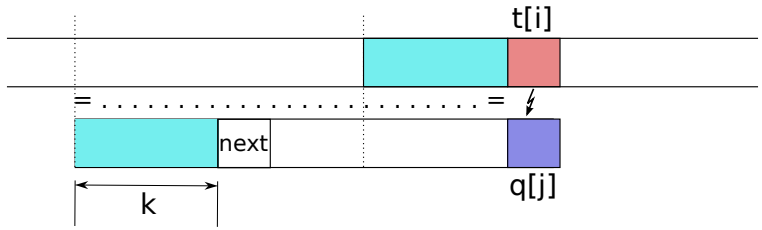
- nutze bereits gelesene Information bei einem Mismatch
  - verschiebe Query/Muster möglichst weit nach rechts
  - gehe im Text nie zurück!
- Allgemeiner Zusammenhang
  - Mismatch an Textposition i mit j-tem Zeichen im Muster
  - $j-1$ vorhergehende Zeichen stimmen überein

Warum kann man nicht einfach ab i wieder "von vorne" das Muster ab seiner ersten Position mit dem Text vergleichen, also immer um die komplette gematchte Länge weiterschieben?

Also:

Mit welchem Muster-Zeichen muss nach einem Mismatch das i-te Textzeichen als nächstes verglichen werden, so dass garantiert kein Muster-Vorkommen übersehen wird?

Knuth-Morris-Pratt: nach Mismatch $t[i] \neq q[j]$



Falls die letzten gematchten Zeichen einem Präfix von q gleichen, kann das nächste Vorkommen von q vor t[i] beginnen!
$\Rightarrow$ für alle $j > 1$, bestimme längstes Präfix des Musters, das auch echtes Suffix von $q[1..j-1]$ ist.
("echt" bedeutet hier: nicht der gesamte Teilstring $q[1..j-1]$.)

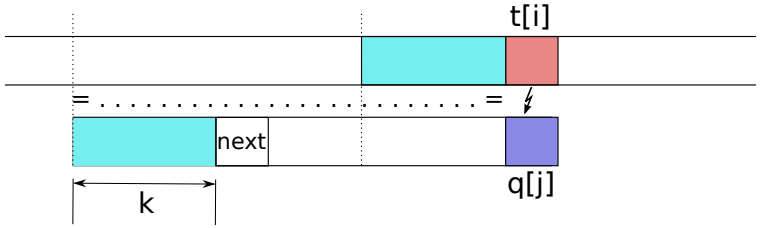
Die maximalen Präfix/Suffixlängen k kann man für q vorberechnen; speichere für $j > 1$: $\texttt{next}[j] = k + 1$; setze $\texttt{next}[0] = 0$.

Beispiel

j	1	2	3	4	5
$q[j]$	A	B	A	B	C
next$[j]$	0	1	1	2	3

```
Text:   ABABABCABABAB
Query:  ABABC
            ABABC
so nicht!
```

Knuth-Morris-Prat: Verwendung von `next[]`



- falls Mismatch bei $j = 1$ ($\texttt{next}[j] = 0$), verschiebe Muster um 1
- sonst $j > 1$, nach Mismatch $t[i] \neq q[j]$ ist als nächstes Position $\texttt{next}[j]$ des Musters mit Textzeichen $t[i]$ zu vergleichen.
  "Verschiebung des Musters" um $j - \texttt{next}[j]$ Positionen.
- Details zu Hilfstabelle next[j] (liefert jeweils nächste zu prüfende Position des Musters nach Mismatch bei j)
  - Spezialfall $j = 1$: next[1]=0
  - $j > 1$: next[j] $= 1 + k$,
    wobei $k =$ "Länge des längsten echten Suffix von $q[1..j-1]$, das Präfix von q ist"

Knuth-Morris-Pratt: Algorithmus

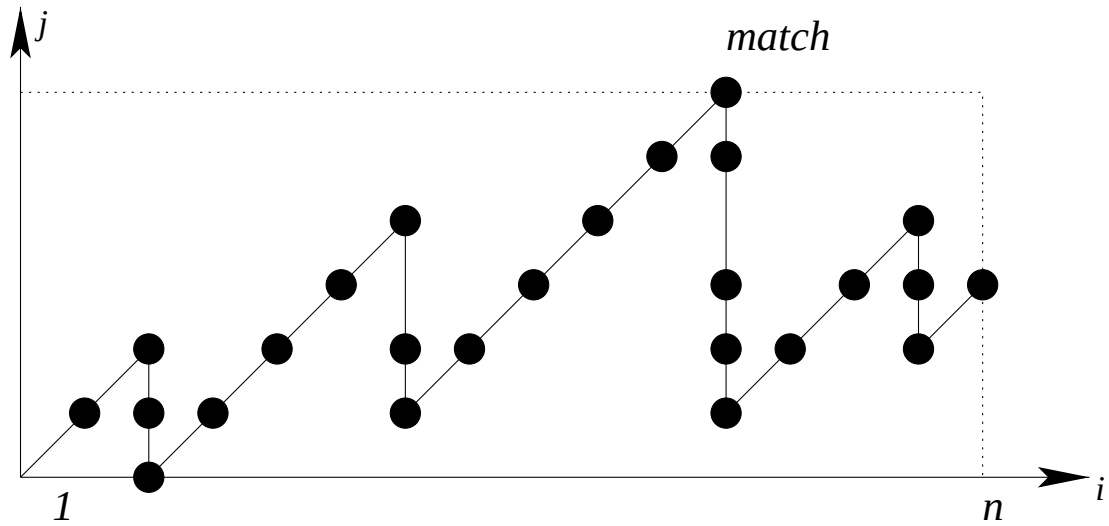
```
j=1; i=1;
while(i<=n) {
   if q[j] == t[i] {
      if (j==m) return i-m+1;  /* match */
      j++; i++;
   }
   else {
      if(j>1) j = next[j];
      else i++;
   }
}
return -1  /* mismatch */
```

Beispiel

```
Text:   ABABABABABC
Query:  ABABC
          abABC
            abABC
              abABC
```

Knuth-Morris-Pratt: Zusammenfassung

Verlauf von i und j (schematisch)



Lineares Worst-Case Verhalten: $O(n + m)$

Suche: $[n..2n]$ Operationen

Vorberechnung der "next"-Tabelle: $O(m)$

Boyer-Moore

- Vergleich 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 c
  - wenn c im Muster nicht vorkommt (v.a. bei kurzen Mustern sehr wahrscheinlich), kann Muster hinter c geschoben werden, also um j Positionen
  - wenn c vorkommt, kann Muster um einen Betrag verschoben werden, der der Position des letzten Vorkommens des Symbols im Suchmuster entspricht
  - Verschiebeumfang kann für jeden Buchstaben des Alphabets vorab auf Muster bestimmt und in einer Tabelle vermerkt werden

Boyer-Moore: Algorithmus

- für jedes Symbol c des Alphabets wird die Position seines letzten Vorkommens im Muster gepeichter $\Rightarrow$ last[c]
- last[c] $:= 0$, falls das Symbol c nicht im Muster vorkommt
- für Mismatch an Musterposition $j >$ last[c], verschiebt sich der Anfang des Musters um $j -$ last[c] Positionen (sonst nur um eine Pos.)

```
i=1;
while(i+m-1<=n) {
    j=m;
    while( (j>=1)&&(q[j]==t[i+j-1]) ) { j--; }
    if( j==0 ) return i;     /* match */
    else { // mismatch an Musterpos. j
        if (last[ t[i+j-1] ] > j) i=i+1;
        else i = i + j-last[ t[i+j-1] ];
    }
}
return -1;    /* mismatch */
```

Boyer-Moore: 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 $O(n \times m)$

Boyer-Moore: Verbesserungen

Match-Heuristik ("good suffix heuristic")

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

Linear 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

Signaturen: *Pessimistische Philosophie*

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

Signaturen

- Kosten $O(n)$ falls Signaturen effizient bestimmt werden können
  inkrementelle Berechnung von s_i aus s_{i-1}
  unterschiedliche Vorschläge mit konstantem Berechnungsaufwand pro Fenster
- Beispiel: Ziffernalphabet; Quersumme als Signaturfunktion
  inkrementelle Berechenbarkeit der Quersumme eines neuen Fensters (Subtraktion der herausfallenden Ziffer, Addition der neuen Ziffer)
- Oft hohe Wahrscheinlichkeit von Kollisionen (false matches)

Signaturen: Karp-Rabin

- Abbildung des Musters / Fensters in Dezimalzahl von max. 9 Stellen (mit 32 Bits repräsentierbar)
- Signatur des Musters: $s(p_1, ..., p_m) = \sum_{j=1}^{m}(10^{j-1} p_{m+1-j}) \mod 10^9$
- Signatur $s_i + 1$ des neuen Fensters $(t_{i+1}, ..., t_{i+m})$ abgeleitet aus Signatur s_i des vorherigen Fensters $(t_i, ..., t_{i+m-1})$:
  $s_{i+1} = (10(s_i - 10^{m-1} t_i) + t_{i+m}) \mod 10^9$
- 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

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

Invertierte Liste: Beispiel

Dies ist ein Text. Der Text hat viele Wörter. Wörter bestehen aus ...

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

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

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

Boolesche Operationen: Verknüpfung von Zeigerlisten

Signatur-Dateien

- Alternative zu invertierten Listen: Einsatz von Signaturen
- zu jedem Dokument/Textfragment: Bitvektor fixer Länge (=Signatur)
- Signatur wird aus Begriffen generiert durch Hash-Funktion s
- OR-Verknüpfung der Bitvektoren aller im Dokument/Fragment vorkommenden Begriffe ergibt Dokument- bzw. Fragment-Signatur
- Signaturen aller Dokumente/Fragmente werden entweder sequentiell oder in einem speziellen Signaturbaum gespeichert.

Suche

- Hashfunktion s angewandt auf *Suchbegriff* liefert *Anfragesignatur*
- mehrere Suchbegriffe können einfach zu einer Anfragesignatur kombiniert werden (OR, AND, NOT-Verknüpfung der Bitvektoren)
- da Signatur nicht eindeutig, muss bei ermittelten Dokumenten / Fragmenten geprüft werden, ob tatsächlich ein Treffer vorliegt

Approximative Suche / Ähnlichkeitssuche

Erfordert **Maß für die Ähnlichkeit zwischen Zeichenketten** s_1 und s_2, z.B.

- Hamming-Distanz: Anzahl der Mismatches zwischen s_1 und s_2 (nur sinnvoll wenn s_1 und s_2 die gleiche Länge haben)
- Editierdistanz: Kosten zum Editieren von s_1, um s_2 zu erhalten (Einfüge-, Lösch-, Ersetzungsoperationen)

Beispiel

s_1	AGCAA	AGCACACA
s_2	ACCTA	ACACACTA
Hamming distance	2	6

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 $\leq k$ ist.

Exakte Stringsuche ergibt sich als Spezialfall mit $k = 0$

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

```
for(i=1 .. n-m+1) {
   z=0;
   for (j=1 .. m)
      if( t[i+j-1]!=q[j] ) z=z+1; /* mismatch */
   if (z<=k)
     print("Treffer in ",i," mit ",z,"Mismatches");
}
```

Auch effizientere Suchalgorithmen (KMP, BM, . . .) können angepasst werden