ADS: Algorithmen und Datenstrukturen 2

Teil 100

Peter F. Stadler & Konstantin Klemm

Bioinformatics Group, Dept. of Computer Science & Interdisciplinary Center for Bioinformatics, **University of Leipzig**

26. Mai 2010

Textsuche — anders rum

Text-Länge n Anfrage-Länge m.

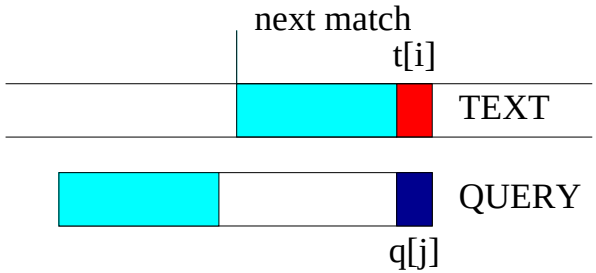
- Naïve Suche
  vergleiche Query mit jeder möglichen Start-Position $0 \leq i \leq n - m$.
  Aufwand offenbar $O(n \times m)$.
  Etwas schlauer: Für jedes i wird der Vergleich beim ersten Mismatch abgebrochen
  immer noch $O(n \times m)$
- Viele (kurze) Anfragen im selben Text
  effiziente Index-Strukuren wie z.B. Suffix-Bäume
  Relative grosser Overhead einmalig fuer die Index-Strukturen, aber dann Suche in $O(m)$
  (siehe letzte Stunde!)
- Wenige Anfragen im selben Text
  aufwendige Indexstrukturen werden unrentabel
  aber Vorverarbeitung der Anfrage kann sich lohnen! $\mathcal{O}(n+m)$ Suche

Knuth-Morris-Pratt (1974)

- 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 Textzeichen verglichen werden, so dass kein Vorkommen des Musters übersehen wird?

Knuth-Morris-Pratt

Betrachte die Position des nächsten Matches:
der Infix von dieser Position an bis exklusive $q[j]$ muss ein Präfix der Query sein. Sonst gäbe es einen Mismatch in diesem Match-Versuch der noch vor $t[i]$ liegt. Also kommt es auf das längste Präfix des Musters (Länge $k < j-1$), das Suffix des übereinstimmenden Bereiches ist, d.h. gleich $q[j-k-1..j-1]$ 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[j] spezifiziert die nächste zu prüfende Position des Musters
  - next[1]=0
  - 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 $q[1..j-1]$, das Präfix von q ist)
- Beispiel

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

Knuth-Morris-Pratt

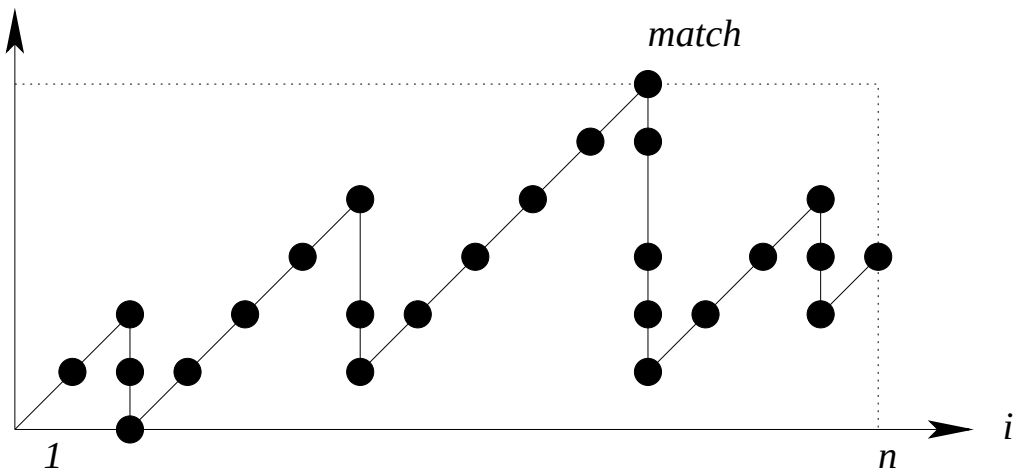
```
j=1; i=1;
while(i<=n) {
   if q[j] = t[i] {
      if(j==m) return i-m+1;  /* match */
      j++; i++;
   }
   else {
      if(j>1) j = next[j];
      else i++;
   }
}
return -1  /* mismatch */
```

Beispiel

```
          3
Text:   ABABABABABC
Query:  ABABC
          abABC
            abABC
              abABC
```

KMP

Verlauf von i und j



Lineares Worst-Case Verhalten: $O(n + m)$
Suche: $[n..2n]$ Operationen
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

Boyer-Moore: Algorithmus

- 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 $j - \text{last}[t] + 1$ Positionen

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

Boyer-Moore: Algorithmus

- 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

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

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, zB. 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 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 entweder sequentiell oder in einem speziellen Signaturbaum gespeichert.

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

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)

s_1	AGCAA	AGCACACA
s_2	ACCTA	ACACACTA
HD	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=1;
   for (j=1 .. m) if( t[i]!=q[j] ) z=z+1; /* mismatch */
   if (z<=k) print("Treffer in ",i," mit ",z,"Mismatches");
}
```

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

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 s_1 und s_2, wobei für leeres Wort bzw. Lücke (gap) steht:

  - $(-, y)$ Einfügung von y in s_2 gegenüber s_1
  - $(x, -)$ Löschung von x in s_1
  - (x, y) Ersetzung von x durch y
  - (x, x) Match-Situation (keine Änderung)
- Jeder Operation wird Gewicht bzw. Kosten $w(x, y)$ zugewiesen
  Einheitskostenmodell: $w(x, y) = w(-, y) = w(x, -) = 1$; $w(x, x) = 0$
  Levensthein-Distanz
- Editierdistanz $D(s1, s2)$: Minimale Kosten, die Folge von Editier-Operationen hat, um s_1 nach s_2 zu überführen
- Für die Levensthein-Distanz gilt: $D(s1, s2) = D(s2, s1)$ und für Kardinalitäten n und m sieht man $|n - m| \leq D(s1, s2) \leq \max(m, n)$
- Beispiel: Editier-Distanz zwischen `Auto` und `Anton`

Editierdistanz in der Bioinformatik

Bestimmung eines Alignments zweier Sequenzen s_1 und s_2:

- Übereinanderstellen von s_1 und s_2 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)

Alignment:
AGCACAC-A
A-CACTCTA
Editier-Sequenz m(A,A) d(G,-) m(C,C) m(A,A) m(C,C) m(A,T) m(C,C) i(-,T) m(A,A)

Editierdistanz

Problem 1: Berechnung der Editierdistanz

- berechne für zwei Zeichenketten / Sequenzen s_1 und s_2 möglichst effizient die Editierdistanz $D(s_1, s_2)$ und eine kostenminimale Folge von Editier-Operationen, die s_1 in s_2 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 $D(p, p') \leq k$ 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

(semi-lokales Alignment)

- bestimme zwischen zwei Sequenzen s_1 und s_2 die ähnlichsten Teilsequenzen s_1' und s_2'

(lokales Alignment)