Einführung in die Informatik 2

– Suchen in Texten –

Sven Kosub

AG Algorithmik/Theorie komplexer Systeme
Universität Konstanz

E 202 | Sven.Kosub@uni-konstanz.de | Sprechstunde: Freitag, 12:30-14:00 Uhr, o.n.V.

Sommersemester 2009

Lineare Suche

Das elementare Suchproblem:

- Gegeben: abstrakter Datentyp Reihung (Array) F von Elementen
- Methode: Suche Element a in F, d.h. bestimme eine Position $P(F, a)$ eines Elementes a in der Reihung F

Generische lineare Suche:

- „Idee“: Teste jedes Element der Reihung in natürlicher Reihenfolge
- Reihung f von Elementen des Types `Object`
- Gleichheitstest: virtuelle Methode `equals`
- Implementierung von `linearSearch` in Klasse `SearchClass`
- `linearSearch` als `static` deklariert

Lineare Suche in Java

```
public class SearchClass{
  public static int linearSearch(Object[] f, Object a){
    int i=0;
    while (i<f.length && !(f[i].equals(a))) i++;
    if (i == f.length) return(-1);
    else return(i);
  }
}
```

Beurteilung:

- Anzahl der Vergleiche linear im schlechtesten Fall (a nicht in f enthalten), d.h. $O(n)$ Vergleiche, wobei n Anzahl der Elemente in f
- in sortierten Reihungen binäre Suche mit $O(\log n)$ Vergleichen

$\longrightarrow$ demnächst!

Texte

- Alphabet ist eine endliche Menge von Symbolen (Buchstaben)
- Wort (String, Zeichenkette) ist eine endliche Folge von Symbolen aus dem Alphabet
- Länge eines Wortes s wird mit $|s|$ bezeichnet, d.h. es gilt $|s| = n$ für $s = s_0 s_1 \cdots s_{n-1}$, wobei die Buchstaben s_i zum Alphabet gehören
- Wort der Länge 0 heißt leeres Wort und wird mit ε bezeichnet

- Binäralphabet besteht aus zwei Buchstaben 0 und 1
- Umgangssprachliches Alphabet mit 26 Buchstaben ohne Umlaute, Leerzeichen, Kleinbuchstaben etc.
- „DNA“-Alphabet besteht aus vier Buchstaben A,C,G,T

Das Wort EINSTEINSTREIFTEEINSTEINSTEIN besteht aus Buchstaben aus dem Alphabet E, I, N, S, T, R, F und hat die Länge 29.

Affixe

Wir betrachten Wörter s und t der Länge m und n

- s ist Teilwort von t, falls s ab einer bestimmten Position in t vorkommt, d.h. für eine Position i gilt $s = t_i t_{i+1} \cdots t_{i+m-1}$
- s ist Präfix von t, falls $s = t_0 t_1 \cdots t_{m-1}$ gilt
- s ist Suffix von t, falls $s = t_{n-m} t_{n-m+1} \cdots t_{n-1}$ gilt

Es seien t =EINSTEINSTREIFTEEINSTEINSTEIN und s =EINSTEIN

1. EINSTEINSTREIFTEEINSTEINSTEIN $\longrightarrow$ s ist Präfix von t
2. EINSTEINSTREIFTEEINSTEINSTEIN $\longrightarrow$ s ist Infix von t
3. EINSTEINSTREIFTEEINSTEINSTEIN $\longrightarrow$ s ist Suffix von t

Suchwort s =STREIFE kein Teilwort von t

Exakte Suche in Texten

Das (exakte) Suchproblem in Texten (im Englischen *pattern matching*):

- Gegeben: abstrakter Datentyp `String`
- Methode: Entscheide, ob ein Wort s als Teilwort vorkommt, d.h., bestimme Position, ab der s vorkommt

Idee für einen einfachen Algorithmus

- Schiebe ein Fenster der Größe m von links beginnend nach rechts über das Wort t
- Innerhalb des Fensters vergleiche die Buchstaben von s und t von links nach rechts

Einfacher Algorithmus in Java

```
public class StringSearchClass{
  public static int searchStringFromLeftToRight
                                        (String s, String t){
    int m=s.length();
    int n=t.length();
    int i=0,j=0;
    while (i<=(n-m)){
      while ((j<m) && (s.charAt(j)==t.charAt(i+j))) j++;
      if (j==m) return(i);
      i++;
      j=0;
    }
    return(-1);
  }
}
```

Einfacher Algorithmus: Beurteilung

Anzahl der Vergleiche des einfachen Algorithmus im schlechtesten Fall:

- Frage: Mit wievielen Positionen in t wird jede Position in s maximal verglichen?
- Antwort: $n - m$

- Damit: einfacher Algorithmus benötigt maximal $m \cdot (n - m)$ Vergleiche, d.h. $O(n \cdot m)$
- falls s halb so lang wie t ist, bedeutet das: $O(n^2)$ Vergleiche

Variation des einfachen Algorithmus

wirkungsvolle Idee zur Verbesserung des einfachen Algorithmus:

- Schiebe ein Fenster der Größe m von links beginnend nach rechts über das Wort t
- Innerhalb des Fensters vergleiche die Buchstaben von s und t von rechts nach links

Einfacher Algorithms mit Rechts-Links-Test in Java

```
public class StringSearchClass{
  public static int searchStringFromRightToLeft
                                        (String s, String t){
    int m=s.length();
    int n=t.length();
    int i=0,j=m-1;
    while (i<=(n-m)){
      while ( (j>=0) && (s.charAt(j)==t.charAt(i+j))) j--;
      if (j==-1) return(i);
      i++;
      j=m-1;
    }
    return(-1);
  }
}
```

Der Algorithmus von Boyer und Moore

Wieso ist Idee mit Rechts-Links-Test interessant?

- Positionenpaar (i, j) heißt Unverträglichkeit von s und t, falls Vergleich „`s.charAt(i)==t.charAt(i+j)`“ negativ ausfällt
- kommt Symbol an Position $i + j$ von t gar nicht in s bis Position j vor, dann Verschiebung des Fensters nicht um 1 sondern gleich um $j + 1$ (Bad-Character-Rule)
- Problem: Zu testen, ob x in $s_0 \dots s_{j-1}$ vorkommt, dauert genauso lange, wie durch größeren Sprung eingespart wird
- Ausweg: Vorberechnung einer Tabelle, die für jedes Präfix von s und für jeden Buchstaben x die Position des rechtesten Vorkommens im Präfix angibt, d.h.

$$S[\ell, x] =_{\text{def}} \begin{cases} \ell + 1 & \text{falls } x \text{ nicht in } s_0 \dots s_{\ell-1} \text{ vorkommt} \\ \ell - k & \text{falls } k < \ell \text{ maximal für } s_k = x \text{ ist} \end{cases}$$

Der Algorithmus von Boyer und Moore in Java

```
public class StringSearchClass{
  public static int searchStringWithBoyerMoore
                                        (String s, String t){
    int m=s.length();
    int n=t.length();
    int i=0,j=m-1;
    int[][] S;
    ...      // Hier wird S berechnet
    while (i<=(n-m)){
      while ( (j>=0) && (s.charAt(j)==t.charAt(i+j))) j--;
      if (j==-1) return(i);
      i=i+S[j,(int) t.charAt[i+j]];
      j=m-1;
    }
    return(-1);
  }
}
```

Der Algorithmus von Boyer und Moore: Beurteilung

- mit Bad-Character-Rule im besten Fall $O(n/m)$ Vergleiche
- ohne Bad-Character-Rule im besten Fall $O(n)$ Vergleiche
- im schlechtesten Fall $O(n \cdot m)$ Vergleiche (z.B. für $t = ba^{m-1}$ und $s = a^n$)
- gibt Versionen unter Verwendung weiterer Regeln (z.B. Good-Suffix-Rule), die Anzahl der Vergleiche auf $O(n + m)$ senken
- Bad-Character-Rule besonders gut auf großen Alphabeten und kleinen Suchwörtern (z.B. Suche in Web-Dokumenten); nicht so gut bei Suche in DNA-Sequenzen
- Algorithmus von Boyer und Moore deshalb in vielen Texteditoren implementiert