Übung zu „Einführung in die Informatik 2“

– Der Algorithmus von Knuth, Morris und Pratt –

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

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
- s ist Rand von t, falls s sowohl Präfix als auch Suffix von t ist

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

Also ist s ein Rand von t; s ist sogar eigentlicher Rand, da es keinen längeren Rand in t gibt

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:

- 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

Es geht besser!

Algorithmus von Knuth, Morris und Pratt

- 1977 veröffentlicht (SIAM Journal on Computing)
- benannt nach Donald E. Knuth, James H. Morris Jr. und Vaughan R. Pratt
- benötigt $O(n + m)$ Vergleiche

Unverträglichkeitsanalyse

- Idee: Vergleiche sparen durch Wiederverwendung positiver Vergleiche

- Positionenpaar (i, j) ist Unverträglichkeit von s und t, falls der Vergleich „`s.charAt(j)==t.charAt(i+j)`“ in der Methode `searchStringFromLeftToRight` negativ ausfällt, d.h. es gilt

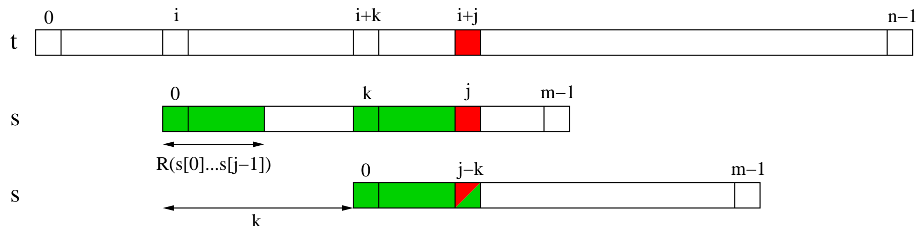
$$s_0 \ldots s_{j-1} = t_i \ldots t_{i+j-1} \quad \text{und} \quad s_j \neq t_{i+j}$$

- Verschiebung von i auf $i + k$ heißt zulässig, falls gilt

$$t_{i+k} \ldots t_{i+j-1} = s_0 \ldots s_{j-1-k},$$

  d.h. es müssen erst wieder Vergleiche ab Position $j - k$ in s durchgeführt werden

Unverträglichkeitsanalyse



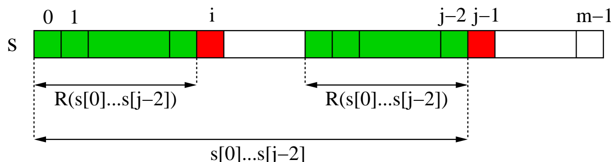
Es sei eine Unverträglichkeit bei (i, j) aufgetreten

- Verschiebung von i auf $i + j$ immer zulässig
- für jede zulässige Verschiebung von i auf $i + k$ ist $s_0 \dots s_{j-1-k}$ ein Rand von $s_0 \dots s_{j-1}$
- Damit: kürzeste zulässige Verschiebung ist die von i auf $i + j - |R(s_0 \dots s_{j-1})|$, wobei $R(s_0 \dots s_{j-1})$ der eigentliche Rand von $s_0 \dots s_{j-1}$ ist

Der Algorithmus von Knut, Morris und Pratt in Java

```
import java.lang.*;
public class StringSearchClass{
  public static int searchStringWithKnuthMorrisPratt(String s, String t){
    int m=s.length();
    int n=t.length();
    int i=0,j=0;
    int[] B=new int[m+1];
    // Hier werden die Einträge von B berechnet:
    // B[0] =def −1 und B[j] =def |R(s0 ... sj−1)| für j = 1, 2, ..., m
    ⋮
    while (i<=(n-m)){
      while ((j<m) && (s.charAt(j)==t.charAt(i+j))) j++;
      if (j==m) return(i);
      i=i+j-B[j];
      j=Math.max(0, B[j]);
    }
    return(-1);
  }
}
```

Bestimmung der Ränderlängen



Berechnen Tabelle B iterativ:

- $B[0] = -1$ und $B[1] = 0$
- angenommen $B[0], \dots, B[j-1]$ bereits berechnet
- bei Berechnung von $B[j] = |R(s_0 \dots s_{j-1})|$ unterscheide zwei Fälle:
  - ist $s_{B[j-1]} = s_{j-1}$, so gilt $B[j] = B[j-1] + 1$
  - ist $s_{B[j-1]} \neq s_{j-1}$, so verfahre wie folgt:
    - suche kürzeres Präfix von $s_0 \dots s_{j-2}$, das auch Suffix von $s_0 \dots s_{j-2}$ ist
    - nächstkürzerer Rand ist $R(R(s_0 \dots s_{j-2}))$
    - teste, ob $R(R(s_0 \dots s_{j-2}))$ zu Rand von $s_0 \dots s_{j-1}$ erweiterbar
    - wenn nicht, dann betrachte $R(R(R(s_0 \dots s_{j-2})))$ usw. usf.

Bestimmung der Ränderlängen

Ränderlängen von EINSTEINSTREIFTEEINSTEINSTEIN:

j	0	1	2	3	4	5	6	7	8	9
$B[j]$	-1	0	0	0	0	0	1	2	3	4

j	10	11	12	13	14	15	16	17	18	19
$B[j]$	5	0	1	2	0	0	1	1	2	3

j	20	21	22	23	24	25	26	27	28	29
$B[j]$	4	5	6	7	8	9	10	6	7	8

- $B[26] = 10$ steht für EINSTEINSTREIFTEEINSTEINSTEIN
- EINSTEINST nicht zum Rand EINSTEINSTE für $B[27]$ erweiterbar
- Randlänge von EINSTEINST ist $B[10] = 5$
- EINST zu Rand von EINSTEINSTE erweiterbar, deshalb $B[27] = 6$

Der Algorithmus von Knut, Morris und Pratt in Java

```
import java.lang.*;
public class StringSearchClass{
  public static int searchStringWithKnuthMorrisPratt(String s, String t){
    int m=s.length();
    int n=t.length();
    int i=0,j=0,k=0;
    int[] B=new int[m+1];
    B[0]=-1; B[1]=0;
    for (int l=2; l<=m; l++){
      while ((k>=0) && !(s.charAt(k)==s.charAt(l-1))) k=B[k];
      B[l]=++k;
    }
    while (i<=(n-m)){
      while ((j<m) && (s.charAt(j)==t.charAt(i+j))) j++;
      if (j==m) return(i);
      i=i+j-B[j];
      j=Math.max(0, B[j]);
    }
    return(-1);
  }
}
```

Der Algorithmus von Knuth, Morris und Pratt: Beurteilung

Anzahl der Vergleiche des Algorithmus von Knuth, Morris und Pratt:

- ohne Berechnung der Ränderlängen maximal $2n - m + 1$ Vergleiche
  - erfolglose Vergleiche „`s.charAt(j)==s.charAt(i+j)`“: $\leq n - m + 1$
  - erfolgreiche Vergleiche „`s.charAt(j)==s.charAt(i+j)`“: $\leq n$
- Berechnung der Ränderlängen maximal $2m - 1$ Vergleiche
- insgesamt $2n + m$ Vergleiche, d.h. $O(n + m)$ Vergleiche

... und der Rest von Boyer und Moore

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

... und der Rest von Boyer und Moore

```
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);
  }
}
```

... und der Rest von Boyer und Moore

Wieso ist Idee mit Rechts-Links-Test interessant?

- 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.

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

Aufgabe 2: Algorithmus von Boyer und Moore

```
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 (bei erfolgloser Suche)
- ohne Bad-Character-Rule im besten Fall $O(n)$ Vergleiche (bei erfolgloser Suche)
- im schlechtesten Fall $O(n \cdot m)$ Vergleiche (z.B. für $s = ba^{m-1}$ und $t = a^n$)
- 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