Algorithmen auf Strings

0. Motivation

Algorithmen auf Strings spielen u.a. eine große Rolle bei

- der Textverarbeitung,
- der Untersuchung von biologischen Sequenzen
- der Internetsuche.

Typische Fragestellungen sind u.a.

- exaktes Finden eines Musterstrings in einem Textstring (Stringmatchingproblem)
- Finden von zu einem gegebenen Musterstring "ähnliche" Teilstrings in einem Textstring (approximatives Stringmatchingproblem).
- Komprimierung von Textstrings.
- Finden von übergeordneten Strukturen in Strings

Wir werden uns mit derartigen Fragestellungen beschäftigen.

Literatur:

Maxime Crochemore, Wojciech Rytter: Text Algorithms, Oxford University Press, 1994.

Maxime Crochemore, Wojciech Rytter: Jewels of Stringology, World Scientific, 2003.

Maxime Crochemore, Christophe Hancart, Thierry Lecroq: Algorithms on Strings, Cambridge University Press, 2007.

Bill Smyth: Computing Patterns in Strings, Pearson - Addison Wesley, 2003.

Dan Gusfield: Algorithms on Strings, Trees, and Sequences, Cambridge University Press, 1997.

Michael S. Waterman: Introduction to Computational Biology: Maps, Sequences and Genomes, Chapman & Hall, 1995.

Pavel A. Pevzner: Computational Molecular Biology: An Algorithmic Approach, MIT Press, 2000.

James A. Storer: Data Compression: methods and theory, Computer Science Press, 1988.

Timothy C. Bell, John G. Cleary, Ian H. Witten: Text Compression, Prentice Hall 1990.

David Sankoff, Joseph B. Kruskal (eds.): Time Warps, String Edits, and Macromolecules: The Theory and Practice of Sequence Comparison, Addison-Wesley, 1983.

M. Lothaire: Applied Combinatorics on Words
Cambridge University Press, 2005.

1. Das Stringmatchingproblem

gegeben: Textstring x über einem endlichen Alphabet Σ, $|x| = n$.
Musterstring y über Σ, $|y| = m$.

gesucht: das erste (alle) Vorkommen von y in x.

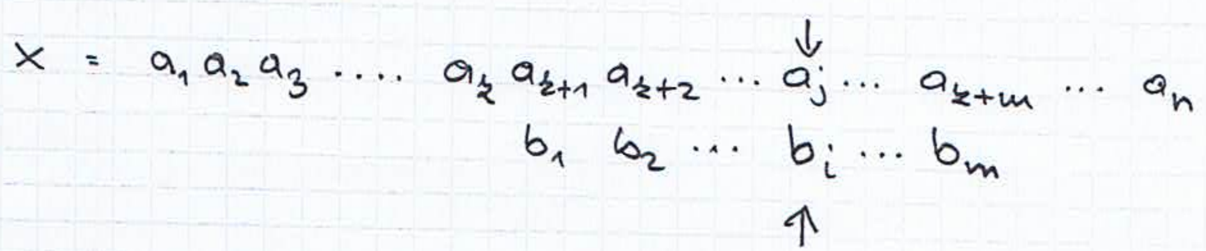
1.1 Der Algorithmus von Knuth, Morris und Pratt

Das Stringmatchingproblem kann leicht in $O(n \cdot m)$ Zeit gelöst werden. Hierzu testet man alle möglichen $n-m+1$ Positionen in x, ob dort ein Teilstring z mit $z = y$ beginnt.

Aufwand:
pro Test $O(m)$ $\Rightarrow$ insgesamt $O(n \cdot m)$

Frage:
Kann der obige naive Ansatz verbessert werden?

Zur Beantwortung dieser Frage analysieren wir den naiven Ansatz genauer. Wir schieben synchron zwei Zeiger über den Text- und über den Musterstring.



Wir unterscheiden zwei Fälle:

<u>1. Fall</u>: $a_j = b_i$

$\Rightarrow a_{j-i+1} a_{j-i+2} \dots a_j = b_1 b_2 \dots b_i$

Beide Zeiger werden um eine Position weitergeschoben.

<u>2. Fall</u>: $a_j \neq b_i$

$\Rightarrow a_{j-i+1} a_{j-i+2} \dots a_{j-1} = b_1 b_2 \dots b_{i-1}$

und

$a_j \neq b_i$.

Der naive Ansatz schiebt den Musterstring um eine Position nach rechts (d.h., erhöht k um 1) und setzt beide Zeiger um $i-1$ Positionen zurück.

$\Rightarrow$

Das Wissen $a_{j-i+1} \dots a_{j-1} = b_1 \dots b_{i-1}$ wird nicht berücksichtigt und auch vergessen.

<u>Frage</u>:
Mit welchem Gewinn können wir dieses Wissen verwerten?

<u>Idee</u>:
Setze den Textstringzeiger überhaupt nicht zurück. D.h., jedes Textstringsymbol wird nur einmal gelesen

$\Longrightarrow$

Der Musterstring muss derart nach rechts verschoben werden, dass

1) links vom Textstringzeiger Muster und Text übereinstimmen und

2) jede geringere Verschiebung des Musterstrings 1) verletzen würde.

<u>Bemerkung:</u>
Den Musterstring nach rechts verschieben bedeutet den Musterstringzeiger nach links zurück setzen. Demzufolge müsste der Musterstringzeiger so wenig wie möglich zurückgesetzt werden, so dass links vom Muster- und vom Textstringzeiger beide Strings übereinstimmen.

<u>Ziel:</u>
Ermittlung der gemäß obigen Überlegungen neue Position des Musterstringzeigers.

Hierzu definieren wir für $1 \leq r \leq m$

$$H(r) := \max_{e<r} \{ e \mid b_1 b_2 \ldots b_{e-1} \text{ ist Suffix von } b_1 b_2 \cdots b_{r-1} \}.$$

Dies bedeutet, dass $H(r) - 1$ die Länge des maximalen Präfixes von $b_1 b_2 \ldots b_{r-2}$, der auch Suffix von $b_1 b_2 \ldots b_{r-1}$ ist, bezeichnet.

Nehmen wir an, dass $H(r)$, $1 \leq r \leq m$, bereits berechnet ist. Dann können wir die naive Methode wie folgt verfeinern:

- Verschiebe den Musterstring derart, dass $b_{H(i)}$ unter a_j steht und verändere den Textstringzeiger nicht. Der Musterstringzeiger zeigt danach auf $b_{H(i)}$.

Wir erhalten dann folgenden Algorithmus:

<u>Algorithmus KMP</u>

<u>Eingabe</u>: Textstring $x = a_1 a_2 \ldots a_n$,
Musterstring $y = b_1 b_2 \ldots b_m$,
Tabelle H.

<u>Ausgabe</u>: k minimal, so dass

$$a_{k+1} a_{k+2} \ldots a_{k+m} = b_1 b_2 \ldots b_m,$$

falls solches k existiert und "Fehlanzeige" sonst.

<u>Methode</u>:

```
i := 1 ; j := 1 ;
while i ≤ m and j ≤ n
   do
      while i > 0 cand b_i ≠ a_j
         do
            i := H(i)   (* Beachte H(1) = 0 *)
         od ;
      i := i+1 ; j := j+1
   od .
```

```
if i > m
  then
    k := j - (m+1)
  else
    "Fehlanzeige"
fi.
```

Das cand in der inneren while - Schleife ist ein "bedingtes and" und vergleicht b_i mit a_j nur, falls $i > 0$. Jede Iteration der inneren while - Schleife schiebt den Musterstring so lange um $(i - H(i))$ Zeichen nach rechts, bis $i = 0$ oder $b_i = a_j$. Im ersten Fall ist kein nichtleerer Präfix von y Suffix von $a_1 a_2 \ldots a_j$. Im zweiten Fall gilt $b_1 b_2 \ldots b_i = a_{j-i+1} a_{j-i+2} \ldots a_j$. Die äußere while - Schleife erhöht synchron den Textstring - und den Musterstringzeiger.

Lemma 1.1

Der Algorithmus KMP ist korrekt.

Beweis:

Zu zeigen ist, dass der Algorithmus KMP das kleinste k mit

$$a_{k+1} a_{k+2} \ldots a_{k+m} = b_1 b_2 \ldots b_m$$

berechnet, falls solches existiert und ansonsten "Fehlanzeige" ausgibt.

Falls KMP die äußere while-Schleife wegen $i = m+1$ verlässt, dann gilt gemäß Konstruktion

a) $j \leq n+1$ und

b) $a_{k+1} a_{k+2} \dots a_{k+m} = b_1 b_2 \dots b_m$.

Dies kann leicht mittels Induktion bewiesen werden.

Wir müssen uns noch davon überzeugen, dass kein $k' \leq n-m$ mit

i) $a_{k'+1} a_{k'+2} \dots a_{k'+m} = b_1 b_2 \dots b_m$ und

ii) $k' < k$, falls KMP k ausgibt,

existiert.

Annahme: Es existiert solches k'.

Zur Herbeiführung eines Widerspruches betrachten wir den Ablauf des Algorithmus KMP.

Da die Ausgabe von KMP $k > k'$ oder "Fehlanzeige" ist, gibt es folgende Situation:

$x = a_1 a_2 \dots a_e a_{e+1} \dots a_{k'} \dots a_q \dots a_j \dots a_{e+m} \dots a_n$

vor Shift $\quad b_1 \; b_2 \dots \quad b_i \dots b_m$ (mit $a_j \neq b_i$)

nach Shift $\quad b_1 b_2 \dots \quad b_m$

- Unmittelbar vor der Abarbeitung des Blockes der inneren while-Schleife steht der Anfang des Musterstrings echt links von $a_{k'+1}$ und

danach echt rechts von $a_{k'+1}$.

$\Rightarrow \quad l < k' < j-1$

Wegen

$$a_{k'+1} a_{k'+2} \cdots a_j \cdots a_{k'+m} = b_1 b_2 \cdots b_m$$

und

$$a_{l+1} a_{l+2} \cdots a_{l+(i-1)} = b_1 b_2 \cdots b_{i-1}$$

gilt:

$$b_1 b_2 \ldots b_{i-k'-1} \quad \text{ist Suffix von } b_1 b_2 \cdots b_{i-1}$$

Gemäß obiger Situation gilt aber

$$i - k' > H(i),$$

was der Definition von $H(i)$ widerspricht. Also ist unsere Annahme falsch. ■

<u>Aufwandsanalyse</u>:

<u>benötigte Zeit</u>:

- Der Textstringzeiger j wird niemals vermindert.
- Eine Erhöhung des Musterstringzeigers um eins impliziert auch die Erhöhung des Textstringzeigers um eins.

$\Rightarrow$

Der Musterstringzeiger i wird maximal n-mal um eins erhöht.

- Eine Verminderung von i in der inneren <u>while</u>-Schleife setzt eine entsprechende Erhöhung von i voraus.

$\Longrightarrow$

Die benötigte Zeit ist $O(n)$.

<u>benötigter Platz</u>:

Der zusätzliche Platzbedarf setzt sich aus dem Platzbedarf für die beiden Zeiger und für die Tabelle H zusammen, ist also $O(m)$.

<u>Berechnung der Tabelle H</u>:

<u>Idee</u>: Induktive Berechnung von H.

1) Gemäß Definition gilt: $H(1) = 0$ und $H(2) = 1$.

2) Seien $H(1), H(2), \ldots, H(i)$, $i \geq 2$ berechnet. Wir werden nun $H(i+1)$ berechnen. Hierzu unterscheiden wir zwei Fälle:

   a) $b_i = b_{H(i)}$

      Dann folgt direkt $H(i+1) = H(i) + 1$

   b) $b_i \neq b_{H(i)}$

      Dann liegt folgende Situation vor:

$b_1 b_2 \ldots b_{H(i)-1}$ ist Suffix von $b_1 b_2 \ldots b_{i-1}$, aber $b_i \neq b_{H(i)}$.

D.h., $b_1 b_2 \ldots b_{H(i)}$ ist kein Suffix von $b_1 b_2 \ldots b_i$.

Wir iterieren nun obige Betrachtungsweise und überprüfen $H(H(i))$, welches gemäß Voraus=setzung bereits berechnet ist.

- Falls nun $b_i = b_{H(H(i))}$, dann gilt $H(i+1) = H(H(i)) + 1$. Andernfalls wieder=holen wir obige Vorgehensweise.

Dies tun wir so lange, bis wir entweder $H(i+1)$ berechnet haben oder zum Anfang des Musterstrings y gelangt sind.

Somit ergibt sich folgender Algorithmus:

<u>Algorithmus</u> BERECHNUNG VON H

<u>Eingabe</u>: $y = b_1 b_2 \ldots b_m$
<u>Ausgabe</u>: Tabelle H
<u>Methode</u>:

```
H(1) := 0 ; H(2) := 1 ;
for i from 2 to m-1
  do
     j := H(i) ;
     while j > 0 cand b_i ≠ b_j
       do
          j := H(j)
       od;
```

$H(i+1) := j+1$

<u>od</u>.

Analog zur Analyse des Algorithmus KMP kann man zeigen, dass der Algorithmus BERECHNUNG VON H $O(m)$ Zeit benötigt.

<u>Übung</u>:

a) Beweisen Sie die Korrektheit des Algorithmus BERECHNUNG VON H. Führen Sie für diesen Algorithmus formal eine Aufwandsanalyse durch.

b) Analysieren Sie exakt die Anzahl der von dem Algorithmus KMP durchgeführten Vergleiche.

c) Übungsaufgabe 5.4.6) aus "Algorithmen und Datenstrukturen".

Der Algorithmus KMP berechnet das <u>erste</u> Vorkommen des Musterstrings y im Textstring x. Es stellt sich nun die Frage, wie wir <u>alle</u> Vorkommen des Musterstrings y im Textstring x in $O(n+m)$ Zeit berechnen können. Es verbietet sich, nach jedem Finden von y in x den Musterstring einfach um eine Position nach rechts zu verschieben und KMP erneut zu starten. Wie folgendes Beispiel zeigt, könnte dies zu einer Laufzeit von $\Omega(m \cdot n)$ führen.

Beispiel:

$$x = \underbrace{aaa \ldots a}_{n\text{-mal}} \quad , \quad y = \underbrace{aa \ldots a}_{m\text{-mal}}$$

Für jedes der $n-m+1$ Vorkommen von y in x würden bei obiger Vorgehensweise m Vergleiche durchgeführt werden. Die Gesamtanzahl der durchgeführten Vergleiche wäre somit

$$m(n-m+1) = mn - m^2 + m$$

Ziel:

Erweiterung des Algorithmus KMP, so dass alle Vorkommen von y in x in Zeit $O(n+m)$ berechnet werden.

Idee:

Erweitere die Tabelle H um den Wert

$$H(m+1) := \max_{\ell < m+1} \{ \ell \mid b_1 b_2 \ldots b_{\ell-1} \text{ ist Suffix von } b_1 b_2 \ldots b_m \}.$$

Wir können dann mit Hilfe von $H(m+1)$ den minimalen Rechtsshift berechnen und den Algorithmus KMP im Textstring dort fortsetzen, wo er mit der letzten Ausgabe terminiert ist.

Übung:

a) Modifizieren Sie den Algorithmus KMP, so dass alle Vorkommen von y in x in Zeit $O(n+m)$ berechnet werden.

b) Erweitern Sie den Algorithmus BERECHNUNG VON H, so dass auch $H(m+1)$ berechnet wird.

Literatur:

D.E. Knuth, J.H. Morris, V.R. Pratt, Fast Pattern Matching in Strings, SIAM J. Comput. 6 (1977), 323 – 350.

Insgesamt erhalten wir folgenden Satz:

Satz 1.1

Unter Verwendung des Algorithmus KMP kann das Stringmatchingproblem in $O(n+m)$ Zeit gelöst werden. Der benötigte zusätzliche Speicherplatz ist $O(m)$.

Bemerkung:

Anstatt der Tabelle H, welche die Größe m hat, kann eine Tabelle F der Größe $|\Sigma| \cdot m$ berechnet werden, mittels der erreicht werden kann, dass in der inneren while-Schleife stets der Musterstringzeiger auf die richtige Position gesetzt wird.
(siehe Vorlesung Alg. auf Strings, WS 06/07 S. 14 – 18.)

Insgesamt erhalten wir folgenden Satz:

Satz 1.1

Unter Verwendung des Algorithmus KMP kann das Stringmatchingproblem in $O(n+m)$ Zeit gelöst werden. Der benötigte zusätzliche Speicherplatz ist $O(m)$.

Unser Ziel ist nun, den Algorithmus KMP zu verbessern. Betrachten wir hierzu nochmals diesen Algorithmus. In der inneren while-Schleife vermindern wir eventuell mehrmals die Laufvariable i, ohne dass der Textstringzeiger j nach rechts geschoben wird. Der Grund hierfür ist, dass stets $b_i \neq a_j$. Diese Beobachtung wirft folgende Frage auf:

Frage:

Können wir eine Tabelle F berechnen, so dass direkt der Musterstringzeiger auf die richtige Position gesetzt wird und danach sowohl Text- als auch Musterstringzeiger um eine Position nach rechts geschoben werden?

Hierzu benötigen wir für $1 \leq r \leq m$, $c \in \Sigma$

$$F(r,c) := \max_{\ell \leq r} \{ \ell \mid b_1 b_2 \ldots b_\ell \text{ ist Suffix von } b_1 b_2 \ldots b_{r-1} c \}.$$

Wir verfeinern zunächst KMP unter der Annahme, dass die Tabelle F bereits

berechnet ist.

Algorithmus KMP*

Eingabe: $x = a_1 a_2 \dots a_n$, $y = b_1 b_2 \dots b_m$, Tabelle F

Ausgabe: k minimal, so dass

$a_{k+1} a_{k+2} \dots a_{k+m} = b_1 b_2 \dots b_m$,

falls solches k existiert und "Fehlanzeige" sonst.

Methode:

```
(1) i := 1 ; j := 1;
(2) while i ≤ m and j ≤ n
      do
        if i > 0 cand b_i ≠ a_j
          then
            i := F(i, a_j)
        fi;
        i := i+1 ; j := j+1
      od;
(3) if i > m
      then
        k := j - (m+1)
      else
        "Fehlanzeige"
    fi.
```

Bemerkung:

$$F(i, a_j) > 0 \Rightarrow b_{F(i,a_j)} = a_j.$$

Übung:

a) Beweisen Sie die Korrektheit des Algorithmus STRINGMATCHING*.

b) Analysieren Sie exakt die Anzahl der durchgeführten Vergleiche.

Offen ist noch die effiziente Berechnung der Tabelle F. Hierzu nehmen wir an, dass die Tabelle H bereits berechnet ist.

Aus der Definition von F folgt direkt

- $F(0, c) = 0$ für alle $c \in \Sigma$,
- $F(1, b_1) = 1$ und
- $F(1, c) = 0$ für alle $c \in \Sigma \setminus \{b_1\}$.

Des Weiteren folgt aus der Definition von F für $r > 1$:

$$F(r, c) = \begin{cases} H(r) & \text{falls } b_{H(r)} = c \\ F(H(r), c) & \text{sonst} \end{cases}$$

Somit ergibt sich folgender Algorithmus zur Berechnung der Tabelle F:

Algorithmus BERECHNUNG VON F

Eingabe: $y = b_1 b_2 \dots b_m$, Tabelle H

Ausgabe: Tabelle F.

Methode:

```
(1)  F(0,b_1) := 0 ; F(1,b_1) := 1 ;
(2)  for alle c ∈ Σ \ {b_1}
        do
           F(0,c) := 0 ; F(1,c) := 0
        od ;
(3)  for i from 2 to m
        do
           for alle c ∈ Σ
              do
                 if b_H(i) = c
                   then
                     F(i,c) := H(i)
                   else
                     F(i,c) := F(H(i),c)
                 fi
              od
        od.
```

Laufzeitanalyse:

Schritt (1) und (2)	$O(\lvert\Sigma\rvert)$
Schritt (3)	$O(\lvert\Sigma\rvert \cdot m)$
insgesamt	$O(\lvert\Sigma\rvert \cdot m)$

Der Algorithmus STRINGMATCHING* benötigt weniger Vergleiche als der Algorithmus STRING=MATCHING. Dafür ist seine Vorbereitungszeit um den Faktor $|\Sigma|$ größer.

<u>offenes Problem</u>: (Diplomarbeit ?)
Kann die Vorbereitungszeit für den Algorithmus STRINGMATCHING* verbessert werden?

Da die Tabelle F bereits die Größe $|\Sigma| \cdot m$ hat, wäre hierzu eine komprimierte Darstellung der Tabelle F notwendig. Diese muss folgende Eigenschaften besitzen:

i) Sie kann in Zeit $\ll |\Sigma| \cdot m$ berechnet werden.
ii) Sie ermöglicht konstante Zugriffszeit.

Unser Ziel ist nun, noch effizientere Algorithmen für das Stringmatchingproblem zu entwickeln.

1.2 Der Algorithmus von Boyer und Moore.

~~Obige Algorithmen betrachten stets den gesamten Textstring. Wir möchten einen Algorithmus entwickeln, der häufig nicht den gesamten Textstring betrachtet.~~