Texte durchsuchen – aber schnell

Der Boyer-Moore-Horspool Algorithmus

Markus E. Nebel

Fachbereich Informatik, TU Kaiserslautern

Texte durchsuchen – aber schnell

Motivation: Viele von einem Computer zu verarbeitende Objekte werden als Text repräsentiert:

- Briefe etc. aus der Textverarbeitung
- HTML-Dokumente
- Bilddateien in Formaten wie Postscript
- Einträge aus DNA-Datenbanken
- ...

$\Rightarrow$ Die Suche nach Wörtern in Texten wird oft benötigt (finde Textstelle in einem Brief oder einen Eintrag in einer Datenbank).

String-Matching Problem: Gegeben sei ein Text t und ein Wort w. Finde alle Vorkommen von w innerhalb t.

Beispiel: Das Wort w=*Nadel* kommt im Text t =*Wir suchen eine Nadel im Heu.* offensichtlich genau einmal vor.

Ein Algorithmus, der das String-Matching Problem löst, muß also für beliebige Eingaben t und w möglichst schnell alle diese Vorkommen finden.

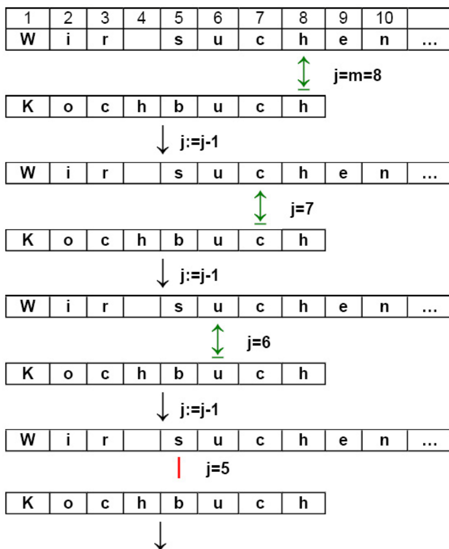
Vorüberlegung: Ein Computer speichert t und w als Folge einzelner Symbole.
$\Rightarrow$ Wir müssen Text und Wort symbolweise vergleichen, um ein Vorkommen zu entdecken. Ein Vergleich an erster Position von t gelingt mit folgendem Programm:

```
j := m;
while (j > 0) and (w[j] = t[j]) do
   j := j - 1;
if (j = 0) then print('Vorkommen an Position 1');
```

Hierbei steht m für die Anzahl Symbole in w, $t[i]$ bzw. $w[j]$ greift auf das i-te bzw. j-te Symbol des Textes bzw. Wortes zu.

Untypisch: Vergleich von rechts nach links (`j:=j-1`).
(Warum? Siehe später!)

Beispiel: (Doppelpfeil erfolgreicher, Linie erfolgloser Vergleich)

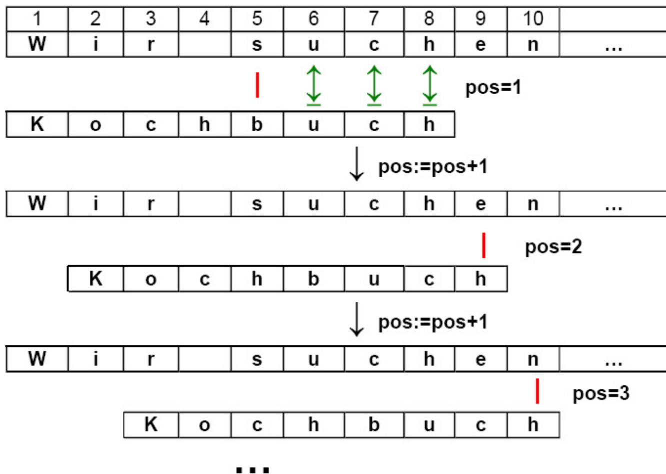


Naive Lösung

Idee: Suche durch symbolweisen Vergleich *w* an allen möglichen Positionen von *t* (also auch beginnend an zweiter, dritter, ... Position):

```
pos := 1;
while pos <= n - m + 1 do begin //alle Positionen
     j := m;
     while (j > 0) and (w[j] = t[pos + j - 1]) do
       j := j - 1;
     if (j = 0) then print('Vorkommen an Pos.', pos);
     pos := pos + 1;
end; //while
```

Beispiel:



Das ist sicher eine korrekte Lösung unseres Problems!

Aber: Viel zu hoher Aufwand. Im schlechtesten Fall vergleichen wir in etwa $m \cdot n$ viele Symbole (n die Anzahl Symbole des Textes).

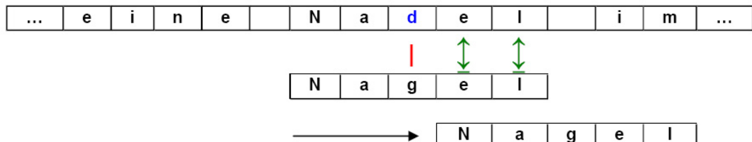
Für große Datenbestände wie etwa die von *Google* (n sehr groß) ist das viel zu langsam.

Doch geht es überhaupt schneller, alle Vorkommen zu finden?

JA!

Problemanalyse

Betrachte folgendes Beispiel:

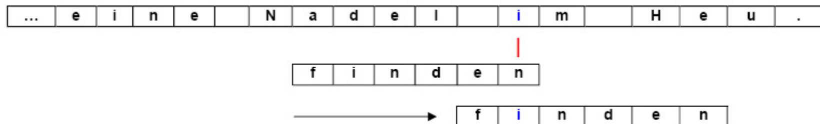


Beobachtung: Das Symbol d des Textes kommt nirgends in w vor $\leadsto$ solange versucht wird, w mit Symbol d in Übereinstimmung zu bringen, **müssen** wir scheitern.

$\Rightarrow$ Wir dürfen w um 3 Positionen nach rechts verschieben (in unserem Programm `pos := pos + 3` setzen), ohne ein Vorkommen von w zu verpassen.

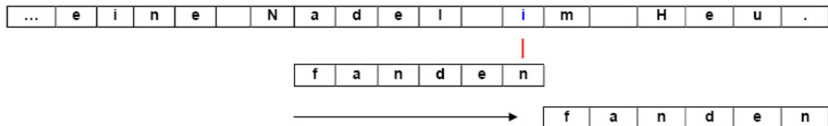
Ab jetzt: Versatz wird immer am bisher rechtesten von t betrachteten Symbol ausgerichtet!

Ein weiteres **Beispiel**:



⇒ Wir können w soweit nach links verschieben, bis sein Symbol i mit dem des Textes übereinstimmt, ohne ein Vorkommen zu verpassen.

Im Extremfall können wir w um seine Länge viele Positionen nach rechts verschieben:



Beobachtung: Die Anzahl an Positionen, die wir überspringen können, hängt nur von w ab und kann vorab berechnet werden. Für alle möglichen Symbole x setzen wir dazu

$$D[x] := \begin{cases} m, \text{ falls } x \text{ keines der ersten } m-1 \text{ Symbole von } w \text{ ist,} \\ m-i, \text{ falls } i \text{ die rechteste Position } \neq m \text{ mit } w[i] = x \text{ ist.} \end{cases}$$

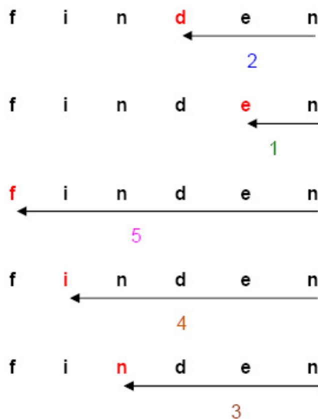
Der erste Fall signalisiert gemäß obigem Beispiel, daß wir w um seine ganze Länge verschieben können, tritt das betrachtete Symbol nirgends (oder nur an letzter Position) in w auf.

Beispiel: $w = \textit{finden}$ (Einträge für $x \notin \{f, i, n, d, e\}$ ausgelassen)

Tabelle D=

d	e	f	i	n
2	1	5	4	3

Begründung:



$D[x]$ ist anschaulich also *der minimale Abstand* von einem Vorkommen des Symbols x zum rechten Rand von w.

Berechnung von D:

```
Für alle Symbole x do
     D[x] := m;
     //D[x] = m für ein nicht vorkommendes Symbol x
Für i := 1 bis m - 1 do
     D[ w[i] ] := m-i;
     // Für die gesehenen Symbole überschreibe
     // die Initialisierung mit m
```

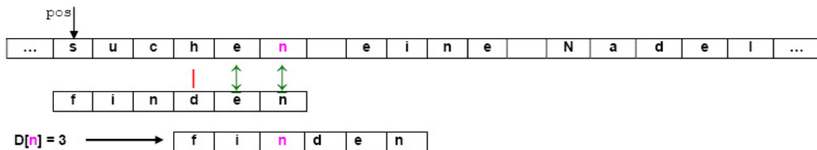
Verbesserter Algorithmus

Wir erhalten aus unserem naiven Verfahren den sog. Boyer-Moore-Horspool Algorithmus, indem wir

1. vor der Suche einmalig `D` berechnen
2. die Zeile `pos := pos + 1` durch `pos := pos + D[ t[pos + m - 1] ]` ersetzen.

```
pos := 1;
while pos <= n - m + 1 do begin //alle Positionen
     j := m;
     while (j > 0) and (w[j] = t[pos + j - 1]) do
        j := j - 1;
     if (j = 0) then print('Vorkommen an Pos.', pos);
     pos := pos + D[ t[pos + m - 1] ];
end; //while
```

Auswirkung:



Die Übereinstimmung der beiden n ist ohne Vergleich bekannt.

Ist dieser Algorithmus wirklich besser?

- Für die denkbar schlechteste Eingabe nicht (hier ist $D[x] = 1$ für alle Symbole x des Textes).
  Beispiel: $t =$ *aaaaaaaaaaaaaaaaaaaaaaaaaaaaa*, $w =$ *baaaa*
  ⇝ 125 Vergleiche zwischen Symbolen.
- In typischen Fällen aus der Anwendung wird aber noch nicht einmal jedes Symbol von t betrachtet, um alle Vorkommen von w zu finden.
  Beispiel: $t =$*Wir suchen eine Nadel im Heu.*, $w =$*Nadel*
  ⇝ 10 Vergleiche zwischen Symbolen im Wort und im Text (zuzüglich dem Aufwand zur Berechnung von D).

Also: Die Idee, Text und Wort jeweils von rechts zu vergleichen, ermöglicht eine drastische Verbesserung des Algorithmus.