Algorithmen und Datenstrukturen in der Stringverarbeitung

Hans Leiß

Universität München, CIS
SS 2008

29. April 2008

Suchverfahren

- Naive Suche
- Morris-Pratt
- Knuth-Morris-Pratt

Problem: Suche im Text $t = t[1] \cdots t[n]$ nach Vorkommen des Suchworts $p = p[1] \cdots p[m]$

Naive Suche

- Vergleiche an jeder Stelle i im Text t, ob $t[i+1] \cdots t[i+m] = p$.

```
i := 0;
while i <= n-m do
  if t[i+1]...t[i+m] = p then print i;
  i := i+1
end;
print "done";
```

- Aufwand: $O(n \cdot m)$ Zeichenvergleiche

- Besser: Vergleiche nur bis zum ersten Unterschied zu p:

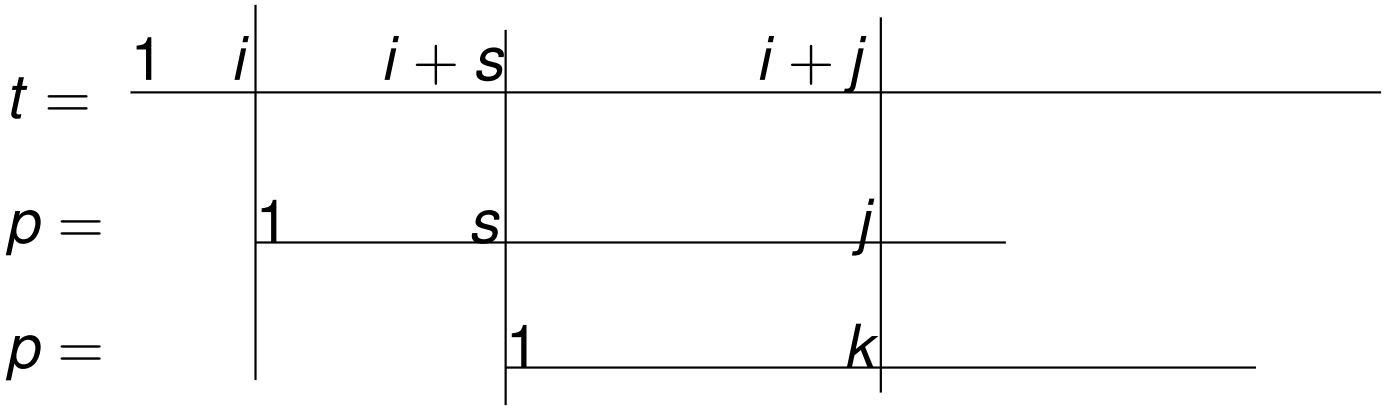
```
i := 0;
while i <= n-m do
  j := 0;
  while j<m and p[j+1]=t[i+j+1]
    do j := j+1 end;  (* { P(i,j) } *)
  if j=m then print i;
  i := i+1
end;
print "done";
```

- Aufwand: bei $p = a^{m-1}b$ und $t = a^{n-1}b$ braucht man m Vergleiche, um $t[i+1]\cdots t[i+m] \neq p$ festzustellen (für $i < n-m-1$), also insgesamt auch $O(n \cdot m)$ Vergleiche

Morris und Pratt (1970) nutzen die Invariante

$$P(i,j) \equiv (p[1..j] = t[i+1..i+j])$$

aus, um im Falle von $t([i+j+1] \neq p[j+1]$ die nächste Stelle $i+s$ in t zu finden, wo p vorkommen könnte:



Ist $t[i+s+1]...t[i+j]$ ein Anfang $p[1]..p[k]$ von p, so ist

$$p[1..k] = p[s+1..j].$$

Man sucht also das längste *echte* Suffix

$$p[s+1]\cdots p[j] \quad (0 < s)$$

von $p[1]\cdots p[j]$, das auch ein Präfix von $p[1]\cdots p[j]$ ist.

Im Text liegt s von $i+j$ um $k = j - s = f_p(j)$ Schritte zurück.
Man berechnet k mit der *failure function*

$$f_p(j) := \begin{cases} \text{größtes } k < j \text{ mit } p[1..k] \text{ ist Suffix von } p[1..j] & \text{falls } 0 < j \\ -1 & \text{falls } 0 = j. \end{cases}$$

unabhängig vom Text.

- Morris-Pratt-Algorithmus:

```
i := 0; j := 0;
while i <= n-m do
  while j<m and p[j+1]=t[i+j+1]
    do j := j+1 end;
  if j=m then print i;
  i := i+(j-fp(j));
  j := max(0,fp(j))
end;
print "done";
```

Aufwand des Morris-Pratt-Algorithmus:

Betrachte die Anzahl der Vergleiche $p[j+1] =^{?} t[i+j+1]$.

1. Für jedes $0 \leq i \leq n-m$ ist höchstens ein Vergleich negativ, also $\leq n-m+1$ viele.
2. Die Summe $i+j$ erfüllt $0 \leq i+j \leq (n-m)+m = n$, wird in der j-Schleife nicht verringert, aber bei positivem Vergleich um 1 erhöht; also gibt es $\leq n$ positive Vergleiche.
3. Insgesamt also $\leq n+(n-m)+1 = 2n-m+1$ Vergleiche.
4. Beh. f_p kann in $O(m)$ Schritten berechnet werden.
5. Wegen 4. kommen pro Zeichenvergleich nur $O(1)$ Schritte hinzu, also braucht man insgesamt $O(n+m)$ Schritte.

Berechnung von f_p:

- Offenbar sind $f_p(0) = -1$ und $f_p(1) = 0$.
- Seien $f_p(0)$ bis $f_p(j)$ bestimmt und im Feld f_p gespeichert. Ist $f_p(j+1) = k' + 1$, so ist $p[1..k'+1]$ ein Suffix von $p[1..j+1]$, also $p[1..k']$ ein Suffix und Präfix von $p[1..j]$.
- Sei $f_p(j) = k \geq 0$, also $p[1..k]$ das längste echte Suffix=Präfix von $p[1..j]$. Ist $p[k+1] = p[j+1]$, so ist $f_p(j+1) := f_p(j) + 1$.
- Sonst ist $p[1..k'+1] = u \cdot p[k'+1]$ für ein kürzeres Suffix=Präfix u von $p[1..j]$. Vergleiche mit dem nächsten Kandidaten $p[1..fp(k)]$. usw.

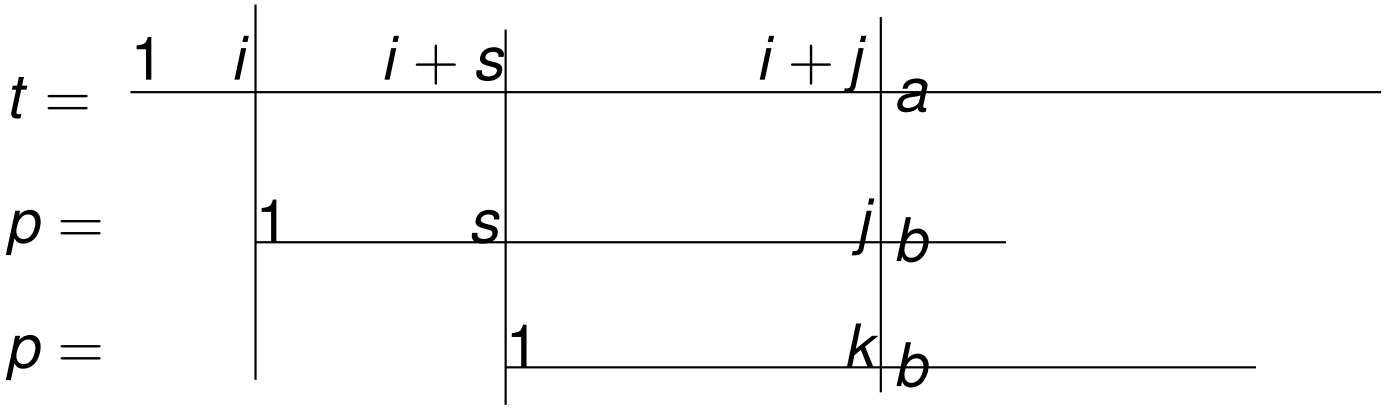
- Berechnung von f_p in $O(m)$ Schritten:

```
k := -1; f_p(0) := k;
for j=0 to m-1 do
  while k>=0 and p[k+1]=/=p[j+1]
    do k:=f_p(k) end;
  k := k+1; f_p(j+1) := k;
end;
```

- Aufwand: Für $j = 0$ wird nur $k := 0 =: f_p(1)$ gesetzt. Für jedes $j = 1, .., m-1$ wird k höchstens einmal um 1 erhöht, kann also durch die $k := f_p(k)$ auch nur $m-1$ mal echt erniedrigt werden. Für jedes $j = 1, .., m-1$ gibt es höchstens einen positiven Vergleich, also insgesamt $\leq (m-1) + (m-1) = 2m-2$ Vergleiche.

Knuth, Morris, Pratt (1977) verbessern das Verfahren:

Ist $p[j+1] \neq t[i+j+1]$ *und* $p[k+1] = p[j+1]$, so kommt p auch bei $s = j - k$ nicht vor:



Also springt man statt zu $j - f_p(j)$ gleich zu $j - f_p(f_p(j))$ usw.

Man berechnet also allein aus dem Muster eine Iteration von f_p:

$$f_p^*(j) := \begin{cases} f_p(j) & \text{falls } j = m, \\ \text{größtes } k < j \text{ mit:} & \\ \ p[1..k] \text{ ist Suffix von } p[1..j] & \\ \text{und } p[k+1] \neq p[j+1] & \text{sonst, falls existent} \\ -1 & \text{sonst.} \end{cases}$$

- Knuth-Morris-Pratt-Algorithmus:

```
i := 0; j := 0;
while i <= n-m do
  while j<m and p[j+1]=t[i+j+1]
    do j := j+1 end;
  if j=m then print i;
  i := i+(j-fp*(j));
  j := max(0,fp*(j))
end;
print "done";
```

Berechnung von f_p^*:

```
k := -1; fp*(0) := k;
for j=0 to m-1 do { k= fp[j] }
  while k>=0 and p[k+1]=/=p[j+1]
    do k:=fp*(k) end;
  k := k+1;
  if j+1=m or p[k+1]=/=p[j+2] then fp*[j+1] := k
  else fp*[j+1] := fp*[k];
end;
```