Bestimmung des Next-Arrays im KMP-Algorithmus

Klaus Messner

Das allgemeine Problem

Gegeben sei ein Text T der Länge n und ein Muster oder Suchwort P der Länge m. Wir suchen einen Algorithmus der die Anfangsposition eines jeden Vorkommens von P in T liefert.

Roter Faden

- Vorstellung des naiven Algorithmus.
- Kurze Besprechung des verbesserten Algorithmus von Knuth, Morris und Pratt.
- Bestimmung des Next-Arrays im KMP-Algorithmus.

Der naive Algorithmus

Für jede mögliche Startposition j in T lege man P an T an und vergleiche. Bei Übereinstimmung gebe man j aus.

Beispiel:
T = xxxabababababxxx, P = abababa. Wir vergleichen wie folgt:

```
1234567890123456
xxxabababababxxx
abababa.........  keine Übereinstimmung
.abababa........  keine Übereinstimmung
usw.
...abababa......  Übereinstimmung ab Position 4
usw.
.....abababa....  Übereinstimmung ab Position 6
usw.
.........abababa  keine Übereinstimmung
```

Der naive Algorithmus

Die erste Übereinstimmung von P mit T wird also an Position 4, die zweite bei Position 7 gefunden.

Das Problem beim naiven Verfahren ist die schlechte Laufzeit in einer Größenordnung von $\Omega(nm)$ im schlimmsten Fall.

Die schlechte Laufzeit kommt dadurch zustande, dass das Muster P immer genau um eine Position weitergerückt wird.

An dieser Stelle setzt der Algorithmus von Knuth, Morris und Pratt an, kurz KMP-Algorithmus.

Der KMP-Algorithmus

Nach einem Vergleich von P mit T liegen Informationen sowohl über P als auch über T vor.

Dabei ist es gleichgültig, ob der Vergleich nun positiv (Match) oder negativ (Mismatch) ausging.

Der KMP-Algorithmus zieht aus den gelesenen Informationen dahingehend einen Nutzen, dass er berechnet, um wieviele Positionen P im nächsten Schritt weitergeschoben werden kann.

Der KMP-Algorithmus wurde bereits detailliert in der Vorlesung besprochen, so dass ich hier nur noch einmal das Wesentliche wiederhole.

Der KMP-Algorithmus

```
 1: long nTextLen, nPatternLen;
 2: long i, j;
 3: List L;
 4: Array next;
 5:
 6: nTextLen = Len(T);
 7: mPatternLen = Len(P);
 8: next = KMPnext(P); {Berechne next-Array anhand des Musters}
 9: L = [];
10: j=0;
11:
12: for i=1 to nTextLen do
13:    while  j>0 and T[i] <> P[j+1]  do {Mismatch}
14:       j = next[j];
15:    end while
16:
17:    if T[i] == P[j+1] then {Muster und Text stimmen bis zur Musterposition j+1 überein}
18:       j++;
19:    end if
20:
21:    if j == mPatternLen then {Match}
22:       L = [L, i-mPatternLen];
23:       j = next[j];
24:    end if
25: end for
```

Der KMP-Algorithmus

Zeile 9: L soll eine Liste sein, die alle Positionen speichert, an denen P in T vorkommt. Sie wird zu Beginn als leere Liste initialisiert.

Zeile 12: Der Text wird zeichenweise durchlaufen.

Zeilen 17 - 19: Der Text wird zeichenweise mit dem Muster verglichen. Bei Übereinstimmung wird der Index des Musters erhöht.

Zeilen 21, 22: Falls die Länge des Musters erreicht ist, d.h. das Muster wurde im Text gefunden, so wird der Liste L die Findeposition hinzugefügt und im nächsten Schleifendurchlauf der Textindex erhöht.

Zeilen 13,14: Falls aber zwischen Textposition i und Musterposition j+1 ein Mismatch auftritt, so muss P weitergeschoben werden. Das Muster wird nach rechts verschoben, was in Zeile 24 dem Heruntersetzen des Musterindex j gleichkommt. Auf welchen Wert j gesetzt wird, wird dem next-Array entnommen.

Zeile 23: Auch im Falle eines Matches, muss P weitergeschoben werden.

Wie berechnet der KMP-Algorithmus die Verschiebungen des Musters?

Zunächst könnte man glauben, dass, im Falle einer Übereinstimmung zwischen Text und Muster, man das Muster einfach um dessen Länge weiterschieben muss.

Wie das Eingangsbeispiel oben zeigt, kann es dann aber sein, dass wir dabei Treffer übersehen.

Würden wir nach dem ersten Treffer (Position 4) das Muster um 7 Zeichen nach rechts verschieben, so würden wir den Treffer auf Position 6 übersehen.

Besonderheiten des Musters.

Die Besonderheit des Musters liegt darin, dass es Wiederholungen enthält und diese müssen wir beim Verschieben berücksichtigen.

Die Frage ist also: Wenn ein Muster Wiederholungen enthält, um wieviele Positionen muss es dann verschoben werden, damit wir im Text keinen Treffer verpassen?

Sehen wir uns hierzu drei kleine Beispiele an:

Besonderheiten des Musters (Beispiele).

```
1234567890123456
xxabcyabcyabcxx
..abcyabc......   Für nächsten Treffer Verschiebung um:
......abcyabc..   4 = 7 − 3 = m − Len(abc)

xxabyabyabyabxx
..abyabyab.....   Für nächsten Treffer Verschiebung um:
.....abyabyab..   3 = 8 − 5 = m − Len(abyab)

xxababababxx
..ababa.....      Für nächsten Treffer Verschiebung um:
....ababa...      2 = 5 − 3 = m − Len(aba)
```

Im ersten Beispiel ist „abc“ die längstmögliche echte Wiederholungsgruppe, die sowohl am Beginn als auch am Ende des Musters vorkommt. Oder mit den Worten aus der Vorlesung: „abc“ ist das längste echte Präfix in P, das auch Suffix ist. Im zweiten Beispiel ist es „abyab“ und im dritten „aba“.

Echte Präfixe

Besondere Aufmerksamkeit verdient hier noch einmal der Begriff echtes Präfix. Im dritten Beispiel ist das längstmögliche Präfix das Muster selbst. Würden wir aber dessen Länge einsetzen (nämlich 5), dann bekämen wir eine Verschiebung von 0.

Dieser Fall muss also noch ausgeschlossen werden, indem wir eben das längste echte Präfix verwenden, also eines, das nicht gerade das Muster selbst ist. Diese Länge bezeichnen wir zukünftig mit lp.

Der KMP-Algorithmus berechnet nun in Zeile 8 für j=1 bis j=m den Wert von lp des Teilwortes P[1..j] und legt diese Werte im next-Array ab.

Beispiel zur Berechnung des next-Arrays

Wir betrachten das Muster P = abyabyab. Dann gilt:

$$
\begin{array}{llll}
\texttt{next}[1] & = \texttt{lp}(\texttt{P}[1..1]) & = \texttt{lp}(\texttt{a}) & = \texttt{len}() = 0 \\
\texttt{next}[2] & = \texttt{lp}(\texttt{P}[1..2]) & = \texttt{lp}(\texttt{ab}) & = \texttt{len}() = 0 \\
\texttt{next}[3] & = \texttt{lp}(\texttt{P}[1..3]) & = \texttt{lp}(\texttt{aby}) & = \texttt{len}() = 0 \\
\texttt{next}[4] & = \texttt{lp}(\texttt{P}[1..4]) & = \texttt{lp}(\texttt{abya}) & = \texttt{len}(\texttt{a}) = 1 \\
\texttt{next}[5] & = \texttt{lp}(\texttt{P}[1..5]) & = \texttt{lp}(\texttt{abyab}) & = \texttt{len}(\texttt{ab}) = 2 \\
\texttt{next}[6] & = \texttt{lp}(\texttt{P}[1..6]) & = \texttt{lp}(\texttt{abyaby}) & = \texttt{len}(\texttt{aby}) = 3 \\
\texttt{next}[7] & = \texttt{lp}(\texttt{P}[1..7]) & = \texttt{lp}(\texttt{abyabya}) & = \texttt{len}(\texttt{abya}) = 4 \\
\texttt{next}[8] & = \texttt{lp}(\texttt{P}[1..8]) & = \texttt{lp}(\texttt{abyabyab}) & = \texttt{len}(\texttt{abyab}) = 5
\end{array}
$$

Ideen für einen Algorithmus zur Berechnung des next-Arrays

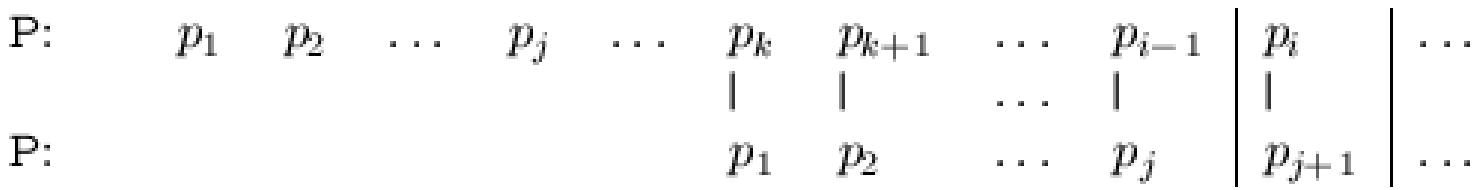
Beim KMP-Algorithmus haben wir gesehen, dass das Muster P am Text angelegt, auf Match oder Mismatch geprüft und anschließend P für die nächste Prüfung weitergeschoben wird. Der Berechnung des next-Arrays liegen zwei wichtige Ideen zugrunde:

1. Das next-Array soll, wenn möglich, induktiv berechnet werden. Es sollen also nachfolgende Elemente aus vorangehenden bestimmt werden können.
2. P wird als Text angesehen und wie beim KMP-Algorithmus gegen sich selbst verschoben.

Ideen für einen Algorithmus zur Berechnung des next-Arrays

Den Induktionsanfang legen wir fest mit next[1]=0, denn in P[1..1] gibt es kein Präfix, das gleichzeitig auch echtes Suffix ist. Weiterhin gelte next[i-1] = j, d.h. das längste Präfix in P[1..i-1], das gleichzeitig auch echtes Suffix ist, habe die Länge j.

Die folgende Abbildung verdeutlicht die Situation:



Beim Vergleich von P mit sich selbst haben wir also bereits festgestellt, dass die ersten j Zeichen von P (untere Zeile in der Abbildung) mit den letzten Zeichen von P[1..i-1] übereinstimmen (obere Zeile in der Abbildung).

Ideen für einen Algorithmus zur Berechnung des next-Arrays

Wenn wir jetzt next[i] bestimmen (Induktionsschritt), fragen wir uns erneut: Wie lange ist das längste Präfix von P[1..i], welches auch echtes Suffix ist

Wir wissen aber bereits, wie die Situation in P[1..i-1] aussieht, dass nämlich next[i-1]=j gilt, und müssen jetzt also nur noch die Zeichen p_i und p_{j+1} miteinander vergleichen. Dabei gibt es zwei Fälle, entweder stimmen die Zeichen überein oder nicht.

1. $p_i=p_{j+1}$:
   Hier ist sofort ersichtlich, dass next[i] = j+1 gilt. (Das längste Präfix in P[1..i], das echtes Suffix ist, hat Länge j+1, siehe Abbildung).

Ideen für einen Algorithmus zur Berechnung des next-Arrays

2. $p_i <> p_{j+1}$:

   Stimmen die Zeichen nicht überein, dann können wir next[i] noch nicht bestimmen. Zuerst müssen wir P wieder weiterschieben, d.h. den Musterindex j neu bestimmen.

   Um wieviele Positionen müssen wir weiterschieben? Doch gerade um so viele Positionen, wie die Länge des längsten Präfixes in P[1..j] angibt, welches auch echtes Suffix ist. Diese Länge ist next[j].

   Wegen j<i ist next[j] bereits in einem der vorangegangenen Schritte berechnet worden, hier also bekannt. Demnach setzen wir j_{neu} := next[j] und vergleichen erneut p_i mit p_{jneu+1}. Jetzt gibt es wieder die zwei Fälle, entweder stimmen die Zeichen überein oder nicht.

Ideen für einen Algorithmus zur Berechnung des next-Arrays

a) $p_i = p_{jneu+1}$:
Jetzt ist $next[i] = j_{neu} + 1$.

b) $p_i <> p_{jneu+1}$:
Weiter mit Fall 2. Somit gelangen wir in eine Schleife, die erst endet, wenn irgendwann eine Übereinstimmung gefunden wurde (Fall 2(a)) oder $j_{neu} = 1$ ist, d.h. keine Übereinstimmung gefunden wurde. Dann ist $next[i] = 0$.

Mit diesen Betrachtungen ergibt sich der Algorithmus (in Anlehnung an KMP) wie folgt:

Algorithmus zur Berechnung des next-Arrays

```
 1: long nPatternLen;
 2: long i, j;
 3: Array next;
 4:
 5: mPatternLen = Len(P);
 6: next = Array(nPatternLen); {Anlegen des next-Arrays mit nPatternlen Einträgen}
 7: next[1] = 0; {Induktionsbeginn}
 8: j=0;
 9:
10: for i=2 to nPatternLen do
11:    while  j>0 and P[i] <> P[j+1]  do {Mismatch}
12:       j = next[j];
13:    end while
14:
15:    if P[i] == P[j+1] then {Stimmen P(1..j+1) und P(k..i) überein? (Siehe Abbildung)}
16:       j++;
17:    end if
18:    next[i] = j;
19: end for
20: Return next;
```

Beispiel zum KMPNext-Algorithmus

Es sei P = ababaa. In Zeile 7 wird next[1]=0 gesetzt. Die folgenden Abbildungen verdeutlichen den weiteren Verlauf. Der Musterzeiger j+1 wurde der Einfachheit halber mit t bezeichnet.

```
i=2, j=0
.|i|....
1|2|3456
a|b|abaa
.|a|babaa
.|1|23456
.|t|.....
```

$P[2] \neq P[1]$
$\Rightarrow next[2] = 0$

```
i=3, j=0
..|i|...
12|3|456
ab|a|baa
..|a|babaa
..|1|23456
..|t|.....
```

$P[3] = P[1]$
$\Rightarrow j = 1, next[3] = 1$

```
i=4, j=1
...|i|..
123|4|56
aba|b|aa
..a|b|abaa
..1|2|3456
...|t|....
```

$P[4] = P[2]$
$\Rightarrow j = 2, next[4] = 2$

```
i=5, j=2
....|i|.
1234|5|6
abab|a|a
..ab|a|baa
..12|3|456
....|t|...
```

$P[5] = P[3]$
$\Rightarrow j = 3, next[5] = 3$

```
i=6, j=3
.....|i|
12345|6|
ababa|a|
..aba|b|aa
..123|4|56
.....|t|..
```

Beispiel zum KMPNext-Algorithmus

Hier kommt erstmals die While-Schleife in Zeile 11 zum Zuge, denn j=3>0 und P[6]<>P[4], => j=next[3]=1. Jetzt haben wir diese Situation:

```
i=6, j=1
.....|i|
12345|6|
ababa|a|
....a|b|abaa
....1|2|3456
.....|t|..
```

Nun ist j>0,P[6]<>P[2] => j=next[1]=0. Damit wird die While-Schleife verlassen und wir landen bei der If-Abfrage in Zeile 15.

Dort gilt dann schließlich P[6]=P[1] und es folgt j=1 und next[6]=j=1. Damit ist das next-Array vollständig bestimmt und der Algorithmus terminiert, genau wie mein Vortrag.