Skriptausschnitt zur Vorlesung

Algorithmen und Datenstrukturen

Prof. Dr. Petra Mutzel

Wintersemester 2008/09
Lehrstuhl für Algorithm Engineering
Universität Dortmund

Kapitel 3

Suchen in Texten / String Matching

3.1 Einführung

3.1.1 Motivation und Definitionen

Dieser Abschnitt beschäftigt sich mit dem Suchen einer gegebenen Zeichenfolge P (Muster bzw. "pattern") in einem Text T. Dieses Problem (Pattern Matching, String Matching) tritt häufig in der Textverarbeitung auf, etwa in Form der "Suchfunktion" in Texteditoren. Auch in der Molekularbiologie wird häufig nach Mustern in einer gegebenen DNS-Sequenz gesucht. Meist ist dabei die Länge M des Musters relativ klein im Vergleich zur Länge N des zu durchsuchenden Textes.

Wir betrachten in diesem Abschnitt das folgende Problem: Gegeben ist eine Text-Zeichenfolge T der Länge N und eine Muster-Zeichenfolge P ("Pattern") der Länge M jeweils aus einem endlichen Alphabet Σ. Gesucht sind alle Vorkommen von P innerhalb von T, d. h. jeweils die Anfangs-Indizes i ($1 \leq i \leq N - M + 1$). Wir nehmen dabei an, dass der Text und das Muster in jeweils einem Feld von $T[1 \ldots N]$ bzw. $P[1 \ldots M]$ gespeichert ist. Formal ist das Problem folgendermaßen definiert.

Pattern Matching Problem	
Gegeben:	eine Zeichenkette (Text) $T_1 \ldots T_N$ von Zeichen aus einem endlichen Alphabet Σ und eine Zeichenkette (Muster/Pattern) $P_1 \ldots P_M$, ebenfalls mit $P_i \in \Sigma, 1 \leq i \leq M$.
Gesucht:	ein oder alle Vorkommen von $P_1 \ldots P_M$ in $T_1 \ldots T_N$, d. h. jene Indizes i mit $1 \leq i \leq N - M + 1$, so dass $\forall j = 1, \ldots, M : T_{i+j-1} = P_j$.

In der Regel gilt: $N \gg M > 1$. Dies wird im allgemeinen gegeben sein. Ein Beispiel ist das *Oxford English Dictionary* mit > 600.000 Definitionen, in dem man normalerweise nach einem Wort der Länge bis maximal 20 sucht.

3.1.2 Naives Verfahren

Die offensichtliche Methode besteht darin, das Muster der Reihe nach (von links nach rechts) an jeden Teilstring des Textes mit Länge M anzulegen und dann zu prüfen, ob tatsächlich Übereinstimmung an der momentanen Stelle vorliegt.

Algorithmus 1 Naive-Search (T, P)

Input: Text $T = T[1..N]$ und Muster $P = P[1..M]$
Output: Ausgabe aller Vorkommen von P in T

```
1:  i = 0;  // Position vor der Anlegestelle von P
2:  solange (i ≤ N − M) {
3:      j = 1;
4:      solange ((j ≤ M)&&(P[j] == T[i + j])) {
5:          j = j + 1;
6:      }
7:      falls (j == M + 1) dann {
8:          Ausgabe: "P an Stelle i + 1 im Text gefunden";
9:      }
10:     i = i + 1;
11: }
```

Das Programm verwendet einen Textzeiger i, der jeweils auf den Index vor der Anlegestelle des Musters zeigt. Stimmen die Zeichen an der Stelle $i + j$ überein ("match") wird j hochgesetzt und das nächste Zeichen wird verglichen. Andernfalls ("mismatch") wird das Muster um eine Stelle weiter rechts angesetzt. Wurde j M-Mal hintereinander hochgezählt, wurde das Muster gefunden.

Analyse der Laufzeit: Das Muster wird genau $N - M + 1$ Mal an den Text angelegt. Im schlimmsten Fall müssen bei jedem solchen Anlegen $\Theta(M)$ Vergleiche durchgeführt werden (was in der Praxis nur selten vorkommt). Insgesamt ergibt dies eine Worst-Case Laufzeit von $O(NM)$. Das Problem des naiven Algorithmus ist es, dass die Information, die während der Vergleiche erhalten wird, nicht benutzt wird, sondern vergessen wird. Man nennt dies auch einen Algorithmus "ohne Gedächtnis".

In diesem Kapitel werden wir einige Algorithmen zur Textsuche kennenlernen, die deutlich effizienter sind. Wir unterscheiden grundsätzlich zwei Szenarien:

- Das Pattern ist vorgegeben, und es sollen beliebige Texte durchsucht werden, die vorher nicht gegeben sind. In diesem Fall benutzt man am besten die String Matching Algorithmen in 3.2 - 3.5.

- Der Text ist vorgegeben, und es sollen beliebige Suchmasken im Text gefunden werden. Dann ist es am besten, den Text vorher aufzubereiten (Indizierung),

sodass man hinterher besonders schnell darin suchen kann. Hierzu verwendet man Suffix Tree oder Suffix Arrays (s. entsprechenden Vorlesungsteil).

3.2 Verfahren von Knuth-Morris-Pratt

Die Idee hierbei ist es, die Informationen, die wir jeweils bis zu einem "Mismatch" über den Text an den jeweiligen Stellen erhalten haben, auszunutzen. Dabei wird versucht, das Muster nach jedem Mismatch um mehr als nur eine Position nach rechts zu rücken. Ziel ist es, zu verhindern, dass die Vergleiche noch einmal über bereits bekannte Zeichen geführt werden müssen.

Beispiel 3.1

Position:	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	.	.	.	.	.
Text 1:	*N*	*A*	*N*	*A*	*N*	*A*		*D*	*A*	*S*		*I*	*S*	*T*
Muster (erstes Anlegen):	*A*	*N*	*A*	*N*	*A*	*S*								
Muster (zweites Anlegen):		*A*	*N*	*A*	*N*	*A*	*S*							

Beim ersten Anlegen des Musters "ANANAS" an den Text "NANANA DAS IST" erfolgt sogleich ein Mismatch. Jedoch beim zweiten Anlegen (an Position 2) erfolgen fünf Übereinstimmungen bevor der erste Mismatch an Position 7 auftaucht. Wegen der Übereinstimmungen kennen wir bereits die fünf Zeichen des Textes vor der aktuellen Position, d. h. diese brauchen wir nicht noch einmal anzuschauen. Doch wie weit müssen wir das Muster nach rechts schieben? Es ist folgendes zu beachten:

(1) Nach der Verschiebung muss garantiert sein, dass links von der aktuellen Position im Muster nur Zeichen stehen, die alle mit dem jeweiligen Zeichen im Text übereinstimmen.

(2) Dabei müssen wir beachten, dass wir das Muster keinesfalls zu weit nach rechts schieben dürfen.

Die erste Eigenschaft (1) ist in unserem Beispiel bei Position 4 im Text sowie bei Position 6 im Text erfüllt. Würden wir das Muster auf Anlegeposition 6 legen, dann könnte uns eventuell ein Vorkommen des Musters im Text (Pattern Matching) verloren gehen. Es wäre also in diesem Fall richtig, das Muster um 2 Positionen nach rechts zu schieben.

Wir nehmen an:

- Die letzten q gelesenen Zeichen im Text stimmen mit den ersten q Zeichen des Musters überein. Es sei P_q das Teilmuster von $P[1]$ bis $P[q]$.

- Das gerade gelesene i-te Zeichen im Text ist verschieden vom $q+1$-ten Zeichen im Muster.

Wir bestimmen vom Anfangsstück des Musters mit Länge q (dem gematchten Teil des Musters) ein Endstück maximaler Länge $l < q$ (Suffix(P_q)), das ebenfalls Anfangsstück des Musters (Prefix(P)) ist.

Das Muster kann dann um $q-l$ Stellen nach rechts geschoben werden. Position $l+1$ ist dann im Muster die erste Stelle, die man mit dem i-ten Zeichen im Text als nächstes vergleichen muss.

Beispiel 3.2

q	`abababababca`	l =*next[q]*	*Shift nach rechts* ($= q - l$)
1	`a#`	*0*	*1*
2	`ab#`	*0*	*2*
3	`aba#`	*1*	*2*
4	`abab#`	*2*	*2*
5	`ababa#`	*3*	*2*
6	`abababab#`	*4*	*2*
7	`abababa#`	*5*	*2*
8	`abababab#`	*6*	*2*
9	`ababababc#`	*0*	*9*
10	`abababababca`	*1*	*9 (kompletter Match)*

Algorithmus 2 zeigt eine Realisierung der Knuth-Morris-Pratt Idee. Die Werte von l für $j = 1, \ldots, M$ werden dabei vom Algorithmus InitNext(P) (s. Algorithmus 3) vorausberechnet und in dem Feld $next[1..M]$ abgespeichert. Im Wesentlichen geschieht die Berechnung von InitNext(P) sehr ähnlich wie die eigentliche Idee des Knuth-Morris-Pratt Algorithmus: das Muster wird mit sich selbst verglichen.

3.2.1 Analyse von Knuth-Morris-Pratt

Die Korrektheit des Knuth-Morris-Pratt Algorithmus ergibt sich aus der obigen Argumentation und der korrekten Berechnung des Feldes $next[]$ in InitNext(P), die wir näher diskutieren wollen.

Fall 1: $P[l+1] == P[q]$ in Zeile (3). In diesem Fall ist das Endstück des Musters P_q, das Anfangsstück von P_q ist, das bisherige (von P_{q-1}) vereinigt mit $P[q]$.

Algorithmus 2 Knuth-Morris-Pratt(T, P)

Input: Text $T = T[1..N]$ und Muster $P = P[1..M]$
Output: Ausgabe aller Vorkommen von P in T

```
1: InitNext(P); // Initialisiere das Feld next[]
2: j = 0;
3: für i = 1, ..., N {
4:    solange ((j > 0)&&(P[j + 1] ≠ T[i])) {
5:       j = next[j];
6:    }
7:    falls (P[j + 1] == T[i]) dann {
8:       j = j + 1;
9:    }
10:   falls (j == M) dann {
11:      P gefunden: Ausgabe; j = next[j];
12:   }
13: }
```

Algorithmus 3 Prozedur InitNext(P)

Input: Muster $P = P[1..M]$
Output: Initialisiert das Feld $next[]$ für Muster P

```
1: next[1] = 0; l = 0;
2: für q = 2, ..., M {
3:    solange ((l > 0)&&(P[l + 1] ≠ P[q])) {
4:       l = next[l];
5:    }
6:    falls (P[l + 1] == P[q]) dann {
7:       l = l + 1;
8:    }
9:    next[q] = l;
10: }
```

Fall 2: $P[l+1] \neq P[q]$ in Zeile (3). Dann geht man am besten genauso vor wie im Knuth-Morris-Pratt Algorithmus, nämlich man vergleicht der Reihe nach $P[q]$ mit $next[l]$, $next[next[l]]$, ..., bis man bei 0 angekommen ist.

Analyse der Laufzeit: Im Algorithmus wird der Index i genau N Mal und der Index j maximal N Mal erhöht. Der Index j kann allerdings höchstens so oft zurückgesetzt werden, wie er erhöht wurde, also insgesamt höchstens N Mal. Das gleiche Argument gilt auch für Algorithmus InitNext(P). Dort kann l höchstens so oft zurückgesetzt werden, wie es insgesamt erhöht wurde, und das ist maximal M. Wir erhalten also als Gesamtlaufzeit des Knuth-Morris-Pratt Algorithmus $\Theta(N+M)$.

Anmerkung 3.1 *Bei Knuth-Morris-Pratt wird der Text ausschließlich sequentiell durchlaufen, da der Index i niemals zurückgesetzt wird. Dies ist in manchen Anwendungen von Vorteil, z.B., wenn der Text auf externen Medien gespeichert ist, die nur sequentiell in einer Richtung durchlaufen werden können (z.B. Bänder).*

3.3 Verfahren von Boyer-Moore

Die Grundidee von Boyer-Moore ist die folgende: Eine Verschiebung des Musters bei Mismatch ist hier davon abhängig, welches Zeichen im Text für den Mismatch verantwortlich ist und wo dieses im Muster auftritt. Dazu wird das Muster von links nach rechts angelegt, aber die Zeichen werden von rechts nach links gelesen. Die Motivation hierfür liegt in der Beobachtung, dass die meisten Textzeichen im Muster nicht auftauchen (da die Musterlänge im Verhältnis zum Alphabet bzw. Text sehr klein ist), und so das Muster sehr oft um die ganze Länge verschoben werden kann.

Wenn man allerdings nur diese Idee (*Last-Verschiebung*) implementiert, dann kann es im schlimmsten Fall zu einer Laufzeit von $O(NM)$ kommen. Deswegen verwendet man noch ein zweites Verschiebungskriterium, das so weit verschiebt, bis die letzten gematchten Zeichen im Text mit den ersten Zeichen in P übereinstimmen (*Suffix-Verschiebung*).

Es gibt also zwei verschiedene Verschiebungskriterien, die jeweils unabhängig voneinander berechnet werden und von denen dann die größere Verschiebung durchgeführt wird. Algorithmus Boyer-Moore(T, P) (s. Algorithmus 4) gibt den Algorithmus an. Dabei wird zunächst die Berechnung der Verschiebungstabellen $last[]$ und $suffix[]$ aufgerufen.

Anmerkung 3.2 *Die Ersetzung von Zeile (11) und (13) durch den Befehl $i = i + 1$ ergibt einen naiven Algorithmus mit Laufzeit $\Theta(NM)$.*

Algorithmus 4 Boyer-Moore (T, P)

Input: Text $T = T[1..N]$ und Muster $P = P[1..M]$
Output: Ausgabe aller Vorkommen von P in T

```
 1: InitLast(P);
 2: InitSuffix(P);
 3: i = 0; // Position vor der Anlegestelle von P
 4: solange (i ≤ N − M) {
 5:    j = M;
 6:    solange ((j > 0)&&(P[j] == T[i + j])) {
 7:       j = j − 1;
 8:    }
 9:    falls (j == 0) dann {
10:       P gefunden: Ausgabe;
11:       i = i + suffix[M − 1];
12:    } sonst {
13:       i = i + max(suffix[j], j − last[T[i + j]]);
14:    }
15: }
```

3.3.1 Berechnung von *last*[]

Wir betrachten die Situation, dass die erste Nichtübereinstimmung bei j auftrat, d.h. $P[j] \neq T[i + j]$ für ein $j, 1 \leq j \leq M$. Das Zeichen an dieser Stelle im Text sei c. In diesem Fall verschieben wir P so weit nach rechts, bis c im Text über dem rechtesten Zeichen gleich c in P ist. Falls c nicht in P vorkommt, dann kann P bis hinter die Position $i + j$ verschoben werden.

Lemma 3.1 *Sei k der größte Index aus $1 \leq k \leq M$, für den gilt $T[i + j] = P[k]$, falls vorhanden, sonst $k = 0$. Dann ist es korrekt, i um $j - k$ zu erhöhen.*

Beweis: Fall 1: $k = 0$. Dann sind alle anderen Zeichen links von j ungleich $T[i + j]$ $\longrightarrow i = i + j\surd$
Fall 2: $k < j$. Das rechteste Erscheinen des Mismatch-Zeichens c in T liegt links von j in P. In diesem Fall ist eine Verschiebung um $j - k > 0$ nach rechts korrekt.
Fall 3: $k > j$. In diesem Fall ist $j - k < 0$ und es wird keine Verschiebung (wegen *last*) durchgeführt. □

Der Algorithmus InitLast(P) (s. Algorithmus 5) berechnet jeweils die Position des rechtesten Vorkommens für alle Zeichen im Alphabet Σ.

Algorithmus 5 Prozedur InitLast(P)

```
1: für alle c aus dem Alphabet Σ {
2:     last[c] = 0;
3: }
4: für j = 1, ..., M {
5:     last[P[j]] = j;
6: }
```

3.3.2 Berechnung von *suffix*[]

Verschiebe P soweit nach rechts, bis die bisher untersuchten gematchten Zeichen im Text mit den ersten Zeichen im Muster übereinstimmen. Falls keinerlei Übereinstimmung herrscht, dann kann das Muster bis ganz hinter die Position $i + j$ geschoben werden.
Wir betrachten wieder die Situation, dass der erste Mismatch bei j aufgetaucht ist, d.h. $P[j] \neq T[i + j]$.

Wir definieren $\mathit{suffix}[j]$ als die kleinste Verschiebung s, so dass

$$\begin{aligned} P[j+1-s] &= P[j+1] \\ \vdots &\quad \vdots \\ P[M-s] &= P[M] \end{aligned}$$

Eine alternative Sichtweise ist die folgende: $\mathit{suffix}[j] = M - k$, $0 \leq k < M$, so dass $P[j+1], \ldots, P[M]$ Suffix von P_k ist, wobei $P_k = P[1] \ldots P[k]$.

Zur Berechnung unterscheiden wir zwei Fälle.

1. Fall: Das Suffix befindet sich nur am Anfang von P.
Dann gilt $\mathit{suffix}[j] = M - \pi[M]$ $\forall j$, wobei $\pi[M]$ die Länge des längsten Präfixes von P ist, das echtes Suffix von $P_M = P$ ist.

2. Fall: Das Suffix befindet sich nicht nur am Anfang von P.
In diesem Fall kann das Suffix im invertierten Muster P^{-1} mit Hilfe von $\mathit{next}[]$ (s. Abschnitt 3.2) ausgerechnet werden. Es gilt $\mathit{suffix}[j] \leq q - \mathit{next}[q]$ für alle $1 \leq q \leq M$ und $j = M - \mathit{next}[q]$. Intuitiv suchen wir alle diejenigen Längen q, deren Endungen mit dem gesuchten Präfix übereinstimmen. Und das sind genau diejenigen, für die gilt $\mathit{next}[q] = M - j$. Das Suffix kann berechnet werden als

$$\mathit{suffix}[j] = \min(\{M - \pi[M]\} \cup \{q - \mathit{next}[q] : 1 \leq q \leq M \text{ und } j = M - \mathit{next}[q]\})$$

Der Algorithmus InitSuffix(P) (s. Algorithmus 6) bestimmt zunächst im gegebenen Muster das Suffix, das Fall 1 entspricht (d. h. am Anfang des Musters). Danach

bestimmt er im invertierten Muster P^{-1} das Suffix, das Fall 2 entspricht. Dazu berechnet er im invertierten Muster das Feld $\overline{next}[]$.

Algorithmus 6 Prozedur InitSuffix(P)

```
1: Berechne InitNext(P) → next[]
2: Berechne InitNext(P^{-1}) → next_rev[]
3: für j = 0, ..., M {
4:    suffix[j] = M − next[M];
5: }
6: für q = 1, ..., M {
7:    j = M − next_rev[q];
8:    falls suffix[j] > q − next_rev[q] dann {
9:       suffix[j] = q − next_rev[q];
10:   }
11: }
```

3.3.3 Analyse von Boyer-Moore

Man kann zeigen, dass die Laufzeit des Boyer-Moore Verfahrens in der Ordnung von $\Theta(M+N)$ liegt. Dies gilt aber nur, wenn beide Verschiebeverfahren verwendet werden.

Anmerkung 3.3 *Die Verschiebung mittels* last *ist sehr einfach, die Suffix-Verschiebung etwas aufwändiger zu programmieren. Aus diesem Grund wird der Algorithmus in der Praxis oft ohne die Suffix-Verschiebung implementiert. Das Verfahren ist in der Praxis in beiden Fällen sehr schnell.*

3.4 Algorithmus von Aho-Corasick

Hierzu gibt es leider kein Skript. Bitte studieren Sie die Vorlesungsfolien sowie bei Unklarheiten die dort angegebene Literatur.

3.5 Bit-Parallelismus (Shift-And)

Hierzu gibt es leider kein Skript. Bitte studieren Sie die Vorlesungsfolien sowie bei Unklarheiten die dort angegebene Literatur.

3.6 Weiterführende Literatur

Suchen in Texten wird z.B. in den Büchern von Sedgewick und Cormen, Leiserson und Rivest ausführlich beschrieben.