

ALGORITHMEN UND DATENSTRUKTUREN

TEIL 6: SUCHE IN ZEICHENKETTEN

Martin Hofmann

LFE Theoretische Informatik, Institut für Informatik,
Ludwig-Maximilians Universität, München

16. April 2016



- 1 RABIN-KARP
- 2 DFAs
- 3 KMP
- 4 BOYER-MOORE



- Problemstellung & Anwendungen 32.1
- Rabin-Karp Verfahren (basierend auf hashing) 32.2
- Suche in Zeichenketten mit endlichen Automaten 32.3
- Knuth-Morris-Pratt Verfahren (Verfeinerung) 32.4



SUCHE IN ZEICHENKETTEN: PROBLEMSTELLUNG

Σ ein endliches **Alphabet** (Menge) von **Zeichen** (characters).

Σ^* = Menge der **Zeichenketten (strings)** über Σ . Eine Zeichenkette ist ein Array von Zeichen.

$T \in \Sigma^*$ eine Zeichenkette der Länge n : der **Text**.

$P \in \Sigma^*$ eine Zeichenkette der Länge m : das **Muster**.

Frage: Kommt P in T vor, d.h. existiert $0 \leq s \leq n - m$ sodass $T[s + 1..s + m] = P$?

Z.B. $P = abaa$ kommt in $T = abcabaabcabac$ vor und zwar mit $s = 3$.

Anwendungen: Textverarbeitung, Suchen in Dokumenten oder Dateien, Molekularbiologie.



$x, y, z, w \in \Sigma^*$.

ϵ = leeres Wort.

$|x|$ = **Länge** von x .

$w \sqsubset x \Leftrightarrow \exists y. x = wy$. Man sagt: w ist **Präfix** von x .

$w \sqsupset x \Leftrightarrow \exists y. x = yw$. Man sagt: w ist **Suffix** von x .

Beachte: $\epsilon \sqsubset x, \epsilon \sqsupset x$.

$x \sqsubset y \Leftrightarrow zx \sqsubset zy$.

$x \sqsupset y \Leftrightarrow xz \sqsupset yz$. **Lemma:** Gelte $x \sqsupset z$ und $y \sqsupset z$. Wenn $|x| \leq |y|$ dann $x \sqsupset y$. Wenn $|x| \geq |y|$ dann $y \sqsupset x$. Wenn $|x| = |y|$ dann $x = y$.

Abkürzung: $x_k = x[1..k]$.

P kommt in T vor, wenn s existiert mit $P \sqsubset T_{s+m}$.



STRING MATCHING: NAIVES VERFAHREN

Naive-String-Matcher(T, P)

1 $n \leftarrow \text{length}[T]$

2 $m \leftarrow \text{length}[P]$

3 **for** $s \leftarrow 0$ **to** $n - m$ **do**

4 **if** $P[1..m] = T[s + 1..s + m]$

5 **then** print „Muster gefunden bei“ s

Worst-case Laufzeit: $\Theta(nm)$. Genauer: $(n - m + 1)m$
Zeichenvergleiche.



RABIN-KARP VERFAHREN

Berechne Hashwert des Musters $h(P)$ und vergleiche sukzessive mit Hashwert $h(T[s + 1..s + m])$. Bei Übereinstimmung genau vergleichen.

Die Hashfunktion wird so gewählt werden, dass $h(T[s + 2..s + 1 + m])$ aus $h(T[s + 1..s + m])$ in Zeit $O(1)$ berechnet werden kann.

Die Laufzeit ist dann $\Theta(n + pm)$ wobei p die Zahl der Hashwertübereinstimmungen ist.

Schlimmstenfalls ist $p = n - m$ also keine Verbesserung. Meist ist aber p klein.



HASHFUNKTION BEI RABIN-KARP

Es bietet sich die schon bekannte Hashfunktion an:

$$h(x) = \sum_{i=1}^{|x|} d^{i-1} \lceil x_{|x|+1-i} \rceil \text{ mod } q$$

Hier ist $d = |\Sigma|$.

Und $\lceil c \rceil$ die Kodierung des Zeichens c als Zahl $1..|\Sigma|$ ist. Z.B.

$\Sigma = \text{ASCII Zeichen}$, $d = 256$. Diese „Kodierungsecken“ $\lceil \cdot \rceil$ lassen wir meist weg.

Und q ist irgendeine Zahl, günstigerweise eine Primzahl nicht nahe bei einer Zweierpotenz, derart dass $dq < 2^{\text{Wortlänge}}$.

Beachte:

$$h(T[s + 2..s + 1 + m]) = d \cdot (h(T[s + 1..s + m]) - d^{m-1} T[s + 1]) + T[s + 1 + m] \text{ mod } q$$

Natürlich reduziert man schon während der Berechnung ständig mod q .



PSEUDOCODE: INITIALISIERUNGSPHASE

Rabin-Karp-Matcher(T, P)

1 $n \leftarrow \text{length}[T]$

2 $m \leftarrow \text{length}[P]$

3 $D \leftarrow d^{m-1} \bmod q$

4 $p \leftarrow 0$

5 $t \leftarrow 0$

6 **for** $i \leftarrow 1$ **to** m **do**

7 $p \leftarrow (dp + P[i]) \bmod q$

8 $t \leftarrow (dt + T[i]) \bmod q$

Es ist $p = h(P)$ und $t = h(T[1..m])$ mit **Hornerschema**.



PSEUDOCODE: HAUPTTEIL

```
9  for  $s \leftarrow 0$  to  $m - n$  do
10     if  $p = t$ 
11         then if  $P = T[s + 1..s + m]$ 
12             print „Muster gefunden bei“  $s$ 
13     if  $s < n - m$ 
14         then  $t \leftarrow (d(t - T[s + 1]D) + T[s + m + 1]) \bmod q$ 
```

Invariante in Zeile 10: $t = h(T[s + 1..s + m])$.



ENDLICHE AUTOMATEN

Ein endlicher Automat M ist ein 5-Tupel $M = (Q, q_0, A, \Sigma, \delta)$, wobei

- Q eine endliche Menge ist, die **Zustände**,
- $q_0 \in Q$, der **Anfangszustand**,
- $A \subseteq Q$, die Menge der **akzeptierenden Zustände**,
- Σ eine endliche Menge ist, das **Eingabealphabet**
- δ eine Funktion von $Q \times \Sigma$ nach Q ist, die **Zustandsübergangsfunktion**.

Beispiel: $\Sigma = \{a, b\}$, $Q = \{0, 1\}$, $q_0 = 0$, $A = \{1\}$,
 $\delta(0, a) = 1$, $\delta(0, b) = 0$, $\delta(1, x) = 0$.



Der Automat arbeitet eine Zeichenkette w ausgehend von q_0 gemäß δ ab und erreicht dadurch einen Endzustand $\phi(w)$.

Formal:

$$\phi(\epsilon) = q_0$$

$$\phi(wa) = \delta(\phi(w), a), \text{ für } w \in \Sigma^* \text{ und } a \in \Sigma$$

Die **akzeptierte Sprache** $L(M) \subseteq \Sigma^*$ ist definiert durch

$$L(M) = \{w \mid \phi(w) \in A\}.$$

Im Beispiel: $L(M) = \{w \mid w = uba^k, k \text{ ungerade}\}.$



STRING-MATCHING MIT AUTOMATEN

Zu gegebenem Muster $P \in \Sigma^*$ definiere Automaten $M(P)$ wie folgt:

- $Q = \{0, 1, \dots, m\}$ (wie immer $m = |P|$)
- $q_0 = 0$
- $A = \{m\}$
- $\delta(q, x) =$ das größte k sodass $P_k \sqsupseteq P[1..q]x$

Suffixfunktion: Wir schreiben $\sigma(x) = \max\{k \mid P_k \sqsupseteq x\}$, also $\delta(q, x) = \sigma(P_q x)$.

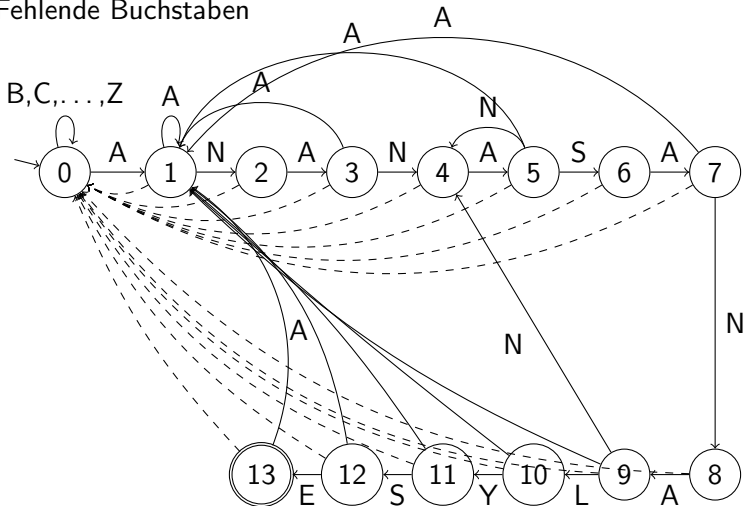
Theorem: $\phi(T_i) = \sigma(T_i)$. (Wird später bewiesen.)

D.h. arbeitet man den **Text** gemäß $M(P)$ ab, so gibt der gerade erreichte Zustand an, welches Anfangsstück des Musters man gerade „gematcht“ hat. Wird dieser Zustand gleich m , so hat man ganz P „gematcht“ und das Muster kommt vor.



BEISPIEL-DFA ZUR WORTSUCHE

DFA für das Wort ANANASANALYSE. Gestrichelte Linien:
Fehlende Buchstaben



Finite-Automaton-Matcher(T, δ, m)

1 $n \leftarrow \text{length}[T]$

2 $q \rightarrow 0$

3 **for** $i \rightarrow 1$ **to** n **do**

4 $q \rightarrow \delta(q, T[i])$

5 **if** $q = m$

6 **then** $s \rightarrow i - m$

7 print „Muster gefunden bei“ s



KORREKTHEIT UND LAUFZEIT

Lemma: Für $x \in \Sigma^*$ und $a \in \Sigma$ gilt $\sigma(xa) \leq \sigma(x) + 1$.

Lemma: Für $x \in \Sigma^*$ und $a \in \Sigma$ gilt: Wenn $q = \sigma(x)$ dass $\sigma(xa) = \sigma(P_q a)$.

Das Theorem $\phi(T_i) = \sigma(T_i)$ folgt nun mit Induktion über Textlänge.

Theorem: $\phi(T_i) = \sigma(T_i)$.

Laufzeit: $O(n)$ + „Zeit für die Konstruktion von $M(P)$ “



BERECHNUNG VON $M(P)$

Die Zustandsübergangsfunktion wird zu Beginn in einem 2D-Array der Größe $(m + 1) \times |\Sigma|$ abgespeichert.

Die Berechnung jedes dieser Einträge gemäß der Definition

$$\delta(q, x) = \text{„das größte } k \text{ sodass } P_k \sqsupseteq P[1..q]x\text{“}$$

erfordert Zeit $O(m^2)$.

Daher Gesamtlaufzeit : $O(m^3|\Sigma|)$.

Das beste Verfahren zur Berechnung von $M(P)$ braucht nur $O(m|\Sigma|)$.



KNUTH-MORRIS-PRATT (KMP) VERFAHREN

Beim Finite-Automaton-Matcher verwenden wir

$$\delta : \{0, \dots, m\} \times \Sigma \rightarrow \{0, \dots, m\}$$

um aus dem alten „gematchten“ Präfix ein neues, d.h., welches das nächste Zeichen einschließt, zu berechnen.

Redundanzen der Funktion δ :

- Ist $q \neq m \wedge P[q + 1] = a$ so ist $\delta(q, a) = q + 1$,
- Ist $k := \delta(q, a) > 0$, so ist $P[k] = a$

Idee von KMP: Nutze diese Redundanzen um die Abhängigkeit der Übergangsfunktion von a ganz zu vermeiden.



Finde $\pi : \{0, \dots, m\} \rightarrow \{0, \dots, m\}$ sodass

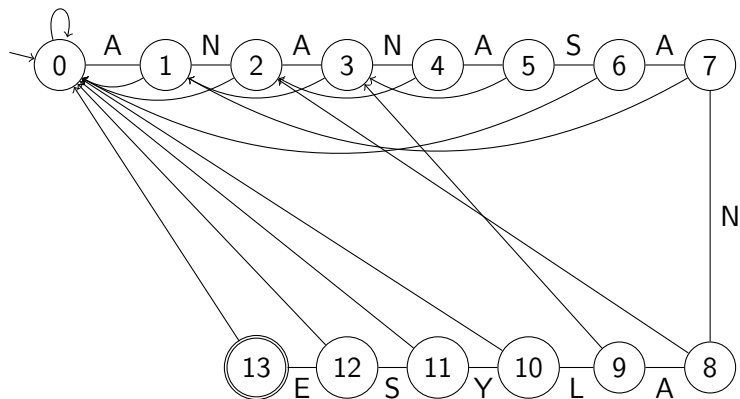
- $\pi(q) < q$ oder $\pi(q) = 0$,
- $q = m \vee P[q + 1] \neq a \Rightarrow \delta(q, a) = \delta(\pi(q), a)$

Mit so einem π (wenn wir es haben) können wir δ wie folgt rekonstruieren.

Delta-From-Pi(q, a)

- 1 **if** $q = m$ **then** $q \leftarrow \pi(q)$
- 2 **while** $q > 0 \wedge P[q + 1] \neq a$ **do**
- 3 $q \leftarrow \pi(q)$
- 4 **if** $P[q + 1] = a$ **then return** $q + 1$ **else return** 0



BEISPIEL π FÜR KMP

DIE PRÄFIXFUNKTION

Definiere die **Präfixfunktion** als

$$\pi(q) = \max\{k \mid k < q \wedge P_k \sqsupseteq P_q\}$$

Klar gilt $\pi(q) < q$ oder $\pi(q) = 0$ (beachte $\max\{\} = 0$).

Theorem: $q = m \vee P[q + 1] \neq a \Rightarrow \delta(q, a) = \delta(\pi(q), a)$.



BEWEIS DES THEOREMS

Lemma: Wenn $k < q$ und $P_k \sqsupset P_q$, dann $P_k \sqsupset P_{\pi(q)}$.

Beweis: Es gilt nach Definition $\pi(q) \geq k$. Die Behauptung folgt mit dem Lemma auf Folie 5.

Beweis des Theorems: Sei $q = m$ oder $P[q + 1] \neq a$. Sei $0 \leq k \leq m$ beliebig. Wir zeigen $P_k \sqsupset P_q a \Leftrightarrow P_k \sqsupset P_{\pi(q)} a$: Wenn $P_k \sqsupset P_q a$, so kann man unter den Voraussetzungen $k \leq q$ sein. Entweder ist $k = 0$ und trivialerweise $P_k = \epsilon \sqsupset P_{\pi(q)} a$ oder $P[k] = a$ und $P_{k-1} \sqsupset P_q$. Mit dem Lemma gilt $P_{k-1} \sqsupset P_{\pi(q)}$ und $P_k \sqsupset P_{\pi(q)} a$. Die umgekehrte Richtung ist klar, da $P_{\pi(q)} \sqsupset P_q$.



DAS KMP-VERFAHREN

```

KMP-Matcher( $T, P, \pi$ )
1   $n \leftarrow \text{length}[T]$  ;  $m \leftarrow \text{length}[P]$  ;  $q \leftarrow 0$ 
4  for  $i \leftarrow 1$  to  $n$  do
5      while  $q > 0 \wedge P[q + 1] \neq T[i]$  do  $q \leftarrow \pi(q)$ 
6      if  $P[q + 1] = T[i]$  then  $q \leftarrow q + 1$ 
7      if  $q = m$  then
8          print „Muster gefunden bei“  $i - m + 1$ 
9           $q \leftarrow \pi(q)$ 

```

Invariante nach Zeile 4: $q < m$.

Invariante nach Zeile 6: $q = \delta(q_0, T[i])$, wobei q_0 Wert von q nach Zeile 6 beim vorherigen Durchlauf.

KMP-Matcher ist äquivalent zu Finite-Automaton-Matcher.



BERECHNUNG DER PRÄFIXFUNKTION

Es ist $\pi(0) = \pi(1) = 0$ und für $q > 1$ gilt

$$\pi(q) = \delta(\pi(q-1), P[q])$$

Aber dieses δ lässt sich mit Delta-From-Pi aus $\pi(k)$ für $k < q$ berechnen.

Zur Erleichterung der Analyse expandieren wir den Code von Delta-From-Pi und lassen die erste Zeile weg, die ja nicht gebraucht wird, da $k < q \leq m$.



Compute-Prefix-Function(P)

```

1   $m \leftarrow \text{length}[P]$  ;  $\pi[1] \leftarrow 0$  ;  $k \leftarrow 0$ 
2  for  $q \leftarrow 2$  to  $m$  do
3      while  $k > 0 \wedge P[k + 1] \neq P[q]$  do
4           $k \leftarrow \pi[k]$ 
5      if  $P[k + 1] = P[q]$  then  $k \leftarrow k + 1$ 
6       $\pi[q] \leftarrow k$ 
7  return  $\pi$ 

```

Berechnung der Zahl der Ausführungen von Zeilen 4 und 5 mit Potentialmethode.

Potential: k .

Amortisierte Kosten von Zeile 4: ≤ 0 , von Zeile 5: ≤ 2 .



LAUFZEIT

Das Potential liegt zwischen 0 und m :

Die tatsächlichen Kosten sind beschränkt durch die Summe der amortisierten Kosten:

Die Zeilen 4 und 5 werden zusammen $\leq 2m$ Mal durchlaufen.

Die Laufzeit von Compute-Prefix-Function ist also $O(m)$.



LAUFZEIT VON KMP-MATCHER

Wir zählen die Durchläufe von Zeilen 7 und 8.

Als Potential nehmen wir q .

Zeile 7 hat amortisierte Kosten ≤ 0

Zeile 8 hat amortisierte Kosten ≤ 2

Also sind die tatsächlichen Kosten $\leq 2n$

Die Laufzeit ist $O(n)$.

Rechnet man noch die Zeit für die Berechnung der Präfixfunktion hinzu, so ergibt sich eine Laufzeit von $O(m + n)$ für das KMP-Verfahren.



BOYER-MOORE VERFAHREN

```

Boyer-Moore-Matcher( $T, P$ )
1   $n \leftarrow \text{length}[T]$  ;  $m \leftarrow \text{length}[P]$  ;  $s \leftarrow 0$ 
4  while  $s \leq n - m$  do
5       $j \leftarrow m$ 
6      while  $j > 0 \wedge P[j] = T[s + j]$  do  $j \leftarrow j - 1$ 
7      if  $j = 0$ 
8          print „Muster gefunden bei“  $s$  ; Advance
10     else Advance

```

Wählen wir Advance zu $s \leftarrow s + 1$ so haben wir den Naive-String-Matcher.

Beim Boyer-Moore-Verfahren wird der Index s gemäß zweier **Heuristiken** im allgemeinen um **mehr als eins** heraufgesetzt.



HEURISTIK „FALSCHES ZEICHEN“

Nehmen wir an, dass der Vergleich von P mit $T[s + 1..s + m]$ abgebrochen wurde, da $P[j] \neq T[s + j]$.

$T[s + j]$ ist das „**falsche Zeichen**“.

Kommt das falsche Zeichen überhaupt nicht in P vor, so können wir s um j erhöhen.

Kommt es (von rechts gelesen) erstmals an der Stelle i in P vor, so können wir immerhin um $j - i$ erhöhen (falls das positiv ist).



HEURISTIK „GUTES SUFFIX“

Nehmen wir an, dass der Vergleich von P mit $T[s + 1..s + m]$ abgebrochen wurde, da $P[j] \neq T[s + j]$.

Wenn $j < m$, so wissen wir immerhin, dass

$P[j + 1..m] \sqsupseteq T[s + 1..s + m]$. Das ist das „gute Suffix“.

Wir schreiben $x \sim y$ für $x \sqsupseteq y \vee y \sqsupseteq x$.

Wir können s um $m - \max\{k \mid 0 \leq k \leq m \wedge P[j + 1..m] \sim P_k\}$ erhöhen.



DAS BOYER-MOORE VERFAHREN

Man kann die möglichen Shifts bei den beiden Heuristiken im Vorhinein ähnlich wie π berechnen.

Advance wird als das Maximum der beiden Heuristiken implementiert.

In Zeile 8 bei erfolgreichem Match kommt nur die Heuristik „Gutes Suffix“ zum Einsatz, da es kein „falsches Zeichen“ gibt.

Es gibt m.W. keine rigorose Laufzeitanalyse des Boyer-Moore Verfahrens.

Bei grossen Alphabeten und langen Mustern verhält es sich in der Praxis sehr gut.

Emacs Ctrl-S arbeitet mit dem Boyer-Moore Verfahren

