Textalgorithmen

Ralf Stiebe
Fakultät für Informatik
Otto-von-Guericke-Universität Magdeburg

Vorlesung im Wintersemester 2006/07

Motivation

- Verarbeitung von Zeichenketten ist eine Grundaufgabe in der Informatik
- Anwendungen: Dokumentenverarbeitung, Suche in großen Datenbanken, *Information Retrieval*, Bioinformatik
- Einige Aufgabenstellungen:
  - Suche nach einzelnen Wörtern oder Suchmustern,
  - Vergleich von Zeichenketten (Ähnlichkeiten),
  - Kompression von Daten

Theoretische Aspekte

- Anwendung grundlegender Datenstrukturen und algorithmischer Techniken
- Beweise der Korrektheit von Algorithmen und von Laufzeitabschätzungen
- enge Beziehungen zu endlichen Automaten
- Spannungsverhältnis zwischen Theorie und Praxis:
  Theorie: Oft komplizierte Algorithmen mit optimalem asymptotischen Verhalten
  Praxis: Robuste Algorithmen mit schnellen praktischen Laufzeiten

Inhalt der Vorlesung

1. Suche nach einem Wort in einem Text
2. Suche nach mehreren Wörtern in einem Text
3. Ähnlichkeiten von Zeichenketten und inexakte Suche
4. Indexstrukturen für Texte (Suffix-Bäume und Suffix-Arrays)

Organisatorisches

WWW-Seite:	`http://theo.cs.uni-magdeburg.de/lehre.html`
Schein:	mündliche Prüfung mit oder ohne Note (20 Minuten)
Übungen:	nach Vereinbarung (ca. 14-tägig)

Lehrbeauftragter:	Ralf Stiebe
email:	`stiebe@iws.cs.uni-magdeburg.de`
Telefon:	(0391) 67-12457
Büro:	G29-026

Literatur

[1] Alberto Apostolico and Zvi Galil, Pattern Matching Algorithms. Oxford University Press, New York, 1997.

[2] Dan Gusfield, Algorithms on Strings, Trees, and Sequences. Cambridge University Press, New York, 1997.

[3] Gonzalo Navarro and Mathieu Raffinot, Flexible Pattern Matching in Strings, Cambridge University Press, 2002.

[4] G.A. Stephen, String Searching Algorithms. World Scientific, Singapore, 1994.

Weitere Literatur: siehe Skript

Kapitel 0

Grundlagen

Alphabete und Wörter

Alphabet: endliche Menge von Zeichen (Buchstaben)
Beispiele: $\{a, b, c\}$, $\{A, C, G, T\}$, ASCII-Zeichen,
RGB-Farbcodes $\{\#000000, \#000001, \ldots, \#\texttt{FFFFFF}\}$

Wort über einem Alphabet Σ: endliche Folge von Zeichen aus Σ

$|w|$: Länge des Wortes w

ε: leeres Wort (Wort der Länge 0)

Σ^k: Menge der Wörter der Länge k über Σ
Σ^*: Menge aller Wörter über Σ
Σ^+: Menge aller Wörter über Σ außer ε

Teilwörter

Es seien u und v Wörter.

Gilt $u = u_1 v u_2$, so nennt man v Teilwort oder Infix oder Faktor von u.

Gilt $u = v u_2$, so nennt man v ein Präfix von u.

Gilt $u = u_1 v$, so nennt man v ein Suffix von u.

Ist v ein Teilwort (bzw. Präfix bzw. Suffix) von u mit $v \neq u$, so nennt man v ein echtes Teilwort (bzw. echtes Präfix bzw. echtes Suffix) von u.

Für $1 \leq i \leq |u|$ ist $u[i]$ das Zeichen an der Stelle i von u.

Das Teilwort von der Stelle i bis zur Stelle j von u ist $u[i \ldots j]$.

Gilt $i > j$ oder $i > |u|$, so ist $u[i \ldots j]$ per definitionem das leere Wort.

Einheitskostenmodell (Unit Cost Model)

Sei n die Länge eines von uns betrachteten Textes.

Im Einheitskostenmodell können Zahlen der Größenordnung n mit konstantem Aufwand gespeichert und bearbeitet werden.
(Addition, Multiplikation, Subtraktion, Division, Vergleich, Bit-Arithmetik)

Übliche Größe: $n < 2^{32} \approx 4 \cdot 10^9 \rightarrow$ Einheitskostenmodell ist angebracht.

Gegensatz: Logarithmisches Kostenmodell

Eine Zahl n kann mit einem Aufwand von $\log n$ gepeichert werden.
Der Aufwand der Grundoperationen ist polynomiell bezüglich $\log n$.

Logarithmisches Kostenmodell ist notwendig, wenn beliebig große Zahlen möglich sind.

Kapitel 1

Exakte Suche nach einem Wort

Überblick

- **Aufgabenstellung**
  Gegeben: Text $T \in \Sigma^*$, Suchwort $P \in \Sigma^*$ mit $|T| = n$, $|P| = m$, $|\Sigma| = \sigma$
  Gesucht: alle Vorkommen von P in T

- Es gibt zahlreiche Algorithmen mit verschiedenen Lösungsideen.

- Ausführliche Übersicht und Animationen auf der Seite von Thierry Lecroq.

  `http://www-igm.univ-mlv.fr/~lecroq/lec_en.html`

Laufzeitbetrachtungen

Sei $\mathcal{A}$ ein Algorithmus zur Wortsuche über dem Alphabet Σ.

$t_{\mathcal{A}}(P,T)$: Laufzeit von $\mathcal{A}$ für Eingabe (P,T).

$t_{\mathcal{A}}(m,n)$: Laufzeit von $\mathcal{A}$ im schlechtesten Fall für Eingaben der Längen (m,n).
$t_{\mathcal{A}}(m,n) = \max\{t_{\mathcal{A}}(P,T) : |P| = m, |T| = n\}$

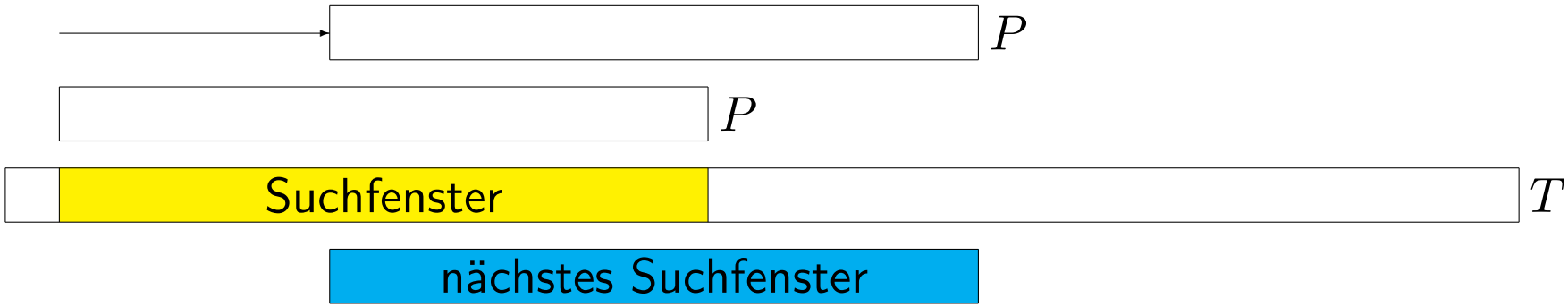
$\overline{t_{\mathcal{A}}}(m,n)$: Laufzeit von $\mathcal{A}$ im mittleren Fall für Eingaben der Längen (m,n).
$\overline{t_{\mathcal{A}}}(m,n) = \sum_{|P|=m,|T|=n} t_{\mathcal{A}}(P,T) \cdot \frac{1}{\sigma^{m+n}}$

Wahrscheinlichkeitstheoretische Interpretation der mittleren Laufzeit:
Erwartungswert, wenn die Buchstaben von P und T unabhängig und gleichverteilt gewählt werden und für diese zufällige Eingabe Algorithmus $\mathcal{A}$ angewendet wird.

Standard-Technik: Suchfenster

In einer Phase wird für einen Textausschnitt der Länge $|P|$ (das Suchfenster) geprüft, ob er mit P identisch ist.

Anschließend verschiebt man das Suchfenster nach rechts, ohne dabei ein Vorkommen von P zu übergehen (sichere Verschiebung).



1.1 Naiver Algorithmus (Brute force)

- Testet Vorkommen von P im Suchfenster durch zeichenweisen Vergleich; anschließend Verschiebung des Fensters um 1
- keine Vorverarbeitung (Präprozessing) des Suchwortes
- Laufzeit im schlechtesten Fall: $\Theta(mn)$
- Laufzeit im mittleren Fall: $\Theta(n)$
  (schlecht nur für Suchwörter mit Präfixen hoher Periodizität)

Algorithmus 1.1 Naiver Algorithmus zur Wortsuche

Eingabe: Wörter P, T mit $|P| = m, |T| = n$
Ausgabe: Menge S der Vorkommen von P in T

(1) $S \leftarrow \emptyset$;
(2) **for** $k \leftarrow 1$ **to** $n - m + 1$
(3) $\quad i \leftarrow 1$; $j \leftarrow k$;
(4) $\quad$ **while** $i \leq m$ **and** $P[i] = T[j]$
(5) $\quad\quad i \leftarrow i + 1; j \leftarrow j + 1$;
(6) $\quad$ **if** $i = m + 1$ **then** $S \leftarrow S \cup \{k\}$;
(7) **return** S;

- Korrektheit ist klar.
- Schlechtester Fall: $P = a^m$, $T = a^n \rightarrow m(n - m + 1)$ Vergleiche.

Mittlere Laufzeit des Naiven Algorithmus

$Comp(m)$ – mittlere Zahl der Vergleiche für eine Textstelle.

p_i – Wahrscheinlichkeit, daß genau die ersten i Zeichen übereinstimmen.

$$Comp(m) = \sum_{i=0}^{m-1} p_i \cdot (i+1) + p_m \cdot m.$$

Betrachte einen Vergleich zweier zufälliger Zeichen:
Wahrscheinlichkeit eines positiven Ausgangs (Match): $\frac{1}{\sigma}$
Wahrscheinlichkeit eines negativen Ausgangs (Mismatch): $1 - \frac{1}{\sigma}$

$$p_i = \frac{1}{\sigma^i} \cdot \left(1 - \frac{1}{\sigma}\right) \text{ für } 0 \leq i \leq m-1,\ p_m = \frac{1}{\sigma^m}$$

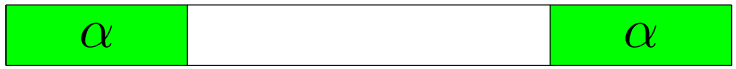
$$\begin{aligned} Comp(m) &= \sum_{i=0}^{m-1} \frac{1}{\sigma^i} \cdot \left(1 - \frac{1}{\sigma}\right) \cdot (i+1) + \frac{1}{\sigma^m} \cdot m \\ &< \frac{\sigma}{\sigma - 1} < 2 \end{aligned}$$

Durchschnittliche Zahl der Vergleiche bei der Suche in einem Text der Länge n:

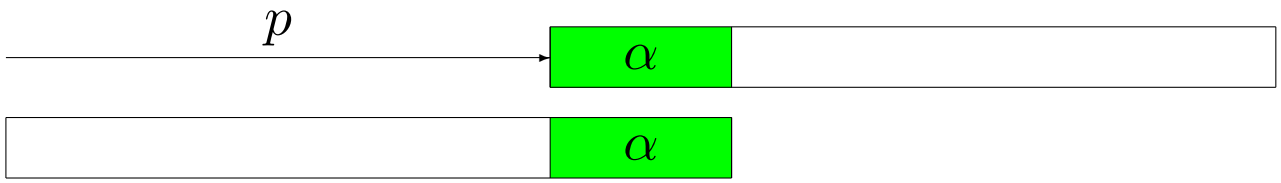
$$(n - m + 1)\, Comp(m) \approx (n - m + 1) \cdot \frac{\sigma}{\sigma - 1}.$$

1.2 Ränder und Perioden

Definition. Es sei P ein Wort. Ist α echtes Präfix und echtes Suffix von P, so nennt man α einen Rand von P. Die Länge des längsten Randes von P wird mit $Border(P)$ bezeichnet.



Definition. Es sei P ein Wort der Länge m. Eine Zahl p mit $1 \leq p \leq m$ heißt Periode von P, wenn $P[i] = P[i+p]$ für alle $1 \leq i \leq m-p$ gilt. Die Länge der kürzesten Periode von P wird mit $Per(P)$ bezeichnet.



Lemma. Es sei P ein Wort der Länge m. Eine Zahl p, $1 \leq p \leq m$, ist genau dann eine Periode von P, wenn P einen Rand der Länge $(m-p)$ besitzt.

Ränder der Präfixe

Definition. Für $1 \leq i \leq |P|$ seien $Border_i(P)$ bzw. $Per_i(P)$ die Länge des längsten Randes bzw. der kürzesten Periode von $P[1 \ldots i]$.

Beispiel. Für $P = abcabba$ erhalten wir folgende Werte für $Border_i$ und Per_i.

i	1	2	3	4	5	6	7
$Border_i(P)$	0	0	0	1	2	0	1
$Per_i(P)$	1	2	3	3	3	6	6

Lemma. Es gelte $T[k \ldots k+i-1] = P[1 \ldots i]$.

1. Das nächste Vorkommen von P in T ist frühestens an der Stelle $k + Per_i(P)$.

2. An der Stelle $k + Per_i(P)$ ist ein Vorkommen von $P[1 \ldots Border_i(P)]$.

Bestimmung der $Border$-Tabelle

Algorithmus 1.2 Bestimmung der längsten Ränder

Eingabe: Wort P mit $|P| = m$

Ausgabe: Längen $Border_i(P)$ der längsten Ränder der Präfixe von P

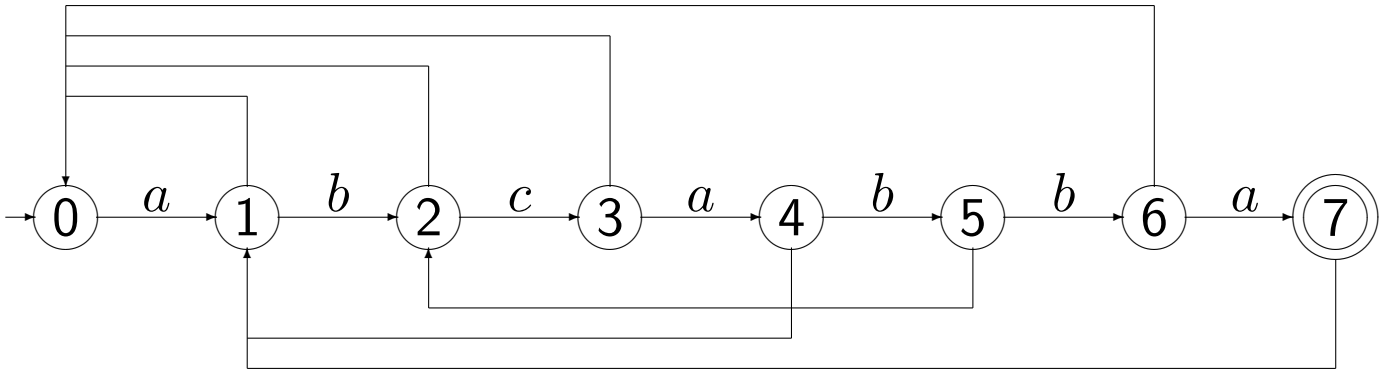
(1) $Border_1 \leftarrow 0$;
(2) **for** $i \leftarrow 1$ **to** $m-1$
(3) $\quad r \leftarrow Border_i$;
(4) $\quad$ **while** $r > 0$ **and** $P[r+1] \neq P[i+1]$
(5) $\quad\quad r \leftarrow Border_r$;
(6) $\quad$ **if** $P[r+1] = P[i+1]$ **then** $Border_{i+1} \leftarrow r+1$;
(7) $\quad\quad\quad\quad\quad\quad\quad\quad$ **else** $Border_{i+1} \leftarrow 0$;
(8) **return** $(Border_1, \ldots, Border_m)$;

Satz. Algorithmus 1.2 bestimmt die Werte $Border_i(P)$, $1 \leq i \leq m$, mit einem Aufwand von $O(m)$.

Ränder – Graphische Darstellung

Für P mit $|P| = m$ konstruiere Graphen mit
den Knoten $0, 1, \ldots, m$,
den beschrifteten Vorwärtskanten $(i-1, P[i], i)$, $1 \leq i \leq m$
und den unbeschrifteten Rückwärtskanten $(i, Border_i)$ (failure links).

Beispiel: Für $P = abcabba$ ergibt sich der folgende Graph:

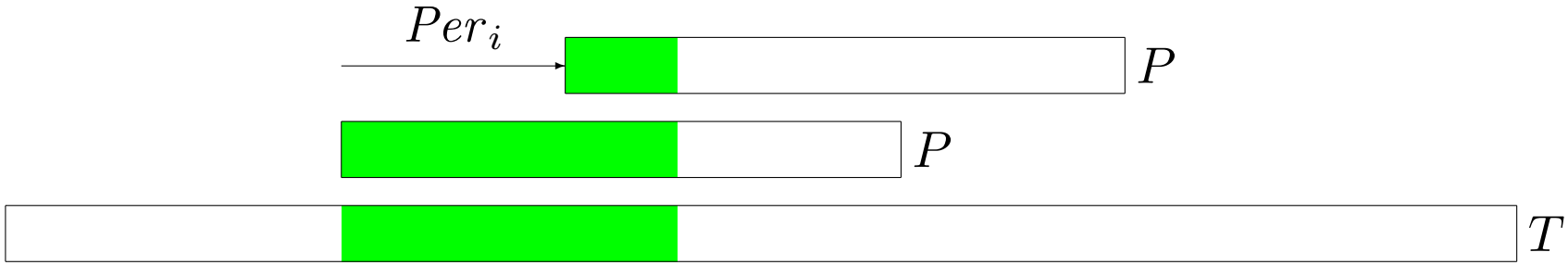


Morris-Pratt-Algorithmus

Im gegenwärtigen Suchfenster sei Übereinstimmung bis zur Position i.

Verschiebe das Fenster um $Per_i(P)$.

Im neuen Fenster ist Übereinstimmung bis zur Position $Border_i(P)$ garantiert.
$\rightarrow$ Keine Vergleiche bis zur Position $Border_i(P)$.

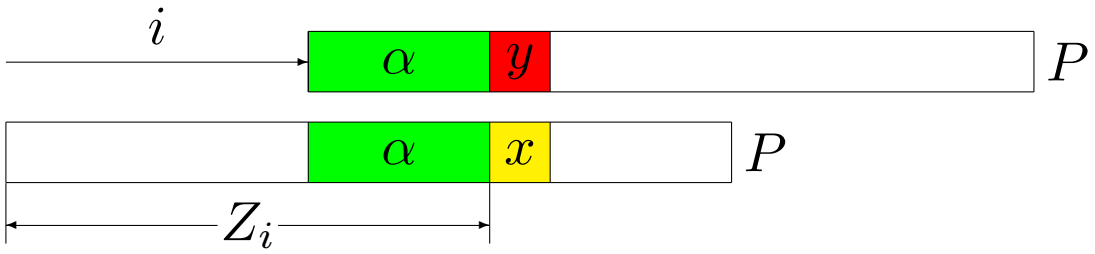


Höchstens 1 positiver Vergleich pro Textposition,
höchstens 1 negativer Vergleich pro Suchfenster (Phase).

$\rightarrow$ lineare Laufzeit

Z-Werte

Definition. Es sei P ein Wort der Länge m. Für $1 \leq i \leq m$ sei $Z_i(P)$ die Länge r des längsten Präfixes von P, so dass i eine Periode von $P[1 \ldots r]$, aber nicht Periode von $P[1 \ldots r+1]$ ist.



Satz. Es sei P ein Wort der Länge m. Eine Zahl i, $1 \leq i \leq m$, ist genau dann eine Periode von P, wenn $Z_i(P) = m$ gilt. Ist i keine Periode, so gilt $P[Z_i(P)+1] \neq P[Z_i(P)+1-i]$; das heißt, die Positionen $Z_i(P)+1$ und $Z_i(P)+1-i$ sind Zeugen dafür, dass i keine Periode ist.

Z-Werte: Beispiel

Beispiel. Für $P = abcabba$ ergibt sich

i	1	2	3	4	5	6	7
Z_i	1	2	5	4	5	7	7

Damit sind 6 und 7 die einzigen Perioden von P.
Für $i = 3$ gilt z.B.

$$P[Z_i(P) + 1] = P[6] = b \text{ und } P[Z_i(P) + 1 - i] = P[3] = c,$$

und damit ist 3 keine Periode von P.

Bestimmung der Z-Werte in Linearzeit

Aktuell höchster Z-Wert wird gespeichert in r, zugehöriger Index in ℓ.
Initialisierung: $\ell = r = 1$.

Für Bestimmung von $Z_i(P)$ mit $\ell < i < r$ benutze $Z_{i-\ell}(P)$.
1. Fall: $Z_{i-\ell}(P) < r - \ell$. Dann ist $Z_i(P) = Z_{i-\ell}(P) + \ell$.
2. Fall: $Z_{i-\ell}(P) \geq r - \ell$. Dann ist $Z_i(P) \geq r$.

Im 2. Fall und falls $i \geq r$ bestimme Z_i durch
explizite Vergleiche zwischen $P[j]$ und $P[j-i]$ ab $j \geq r+1$.
In allen Fällen ist kein expliziter Vergleich bis r nötig.

Abschätzung der Zahl expliziter Vergleiche:
positiv: vergrößert Wert von $r \rightarrow$ insgesamt höchstens m
negativ: höchstens einer für jedes $i \rightarrow$ insgesamt höchstens m

Algorithmus 1.3 Z-Algorithmus

Eingabe: Wort P, $|P| = m$
Ausgabe: $Z_i(P)$, $1 \leq i \leq m$

```
(1)  ℓ ← 1; r ← 1;
(2)  for i ← 1 to m − 1
(3)    if i < r and Z_{i−ℓ} < r − ℓ then Z_i ← Z_{i−ℓ} + ℓ;
(4)    else
(5)      if r < i then r ← i;
(6)      while r < m and P[r + 1] = P[r + 1 − i]
(7)        r ← r + 1;
(8)      Z_i ← r; ℓ ← i;
(9)  Z_m ← m;
(10) return (Z_1, ..., Z_m);
```

Satz. Der Z-Algorithmus berechnet für ein Wort P der Länge m die Werte $Z_i(P)$, $1 \leq i \leq m$, mit einem Aufwand von $O(m)$.

1.3 Suche mit deterministischen endlichen Automaten

- Idee: Konstruiere den minimalen DEA, der die Sprache $\Sigma^* P$ akzeptiert.
- Laufzeit: $\Theta(\sigma \cdot m)$ für das Präprozessing,
  $\Theta(n)$ für die Suche (Realzeit-Algorithmus).
- Nachteil: DEA braucht Speicherplatz von $\Theta(\sigma \cdot m)$.
- Varianten: Algorithmen von Morris-Pratt, Knuth-Morris-Pratt, Simon.
  Jeweils Zeit $\Theta(m)$ für das Präprozessing, $\Theta(n)$ für die Suche

Minimaler DEA für $\Sigma^* P$

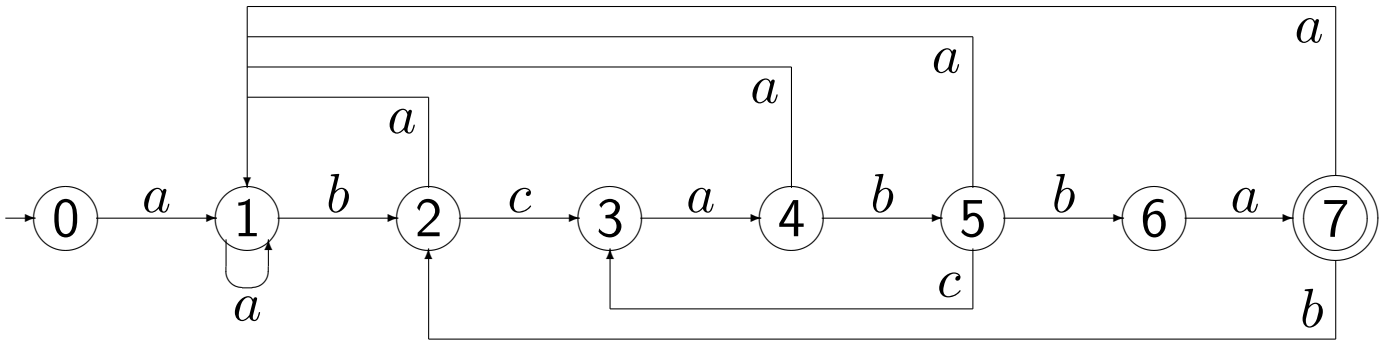
Satz. Es sei $P \in \Sigma^m$. Die Sprache $\Sigma^* P$ wird akzeptiert durch den DEA

$$
\begin{aligned}
A_P &= (\Sigma, \{0, 1, \ldots, m\}, \delta_P, 0, \{m\}) \text{ mit} \\
\delta_P(i, x) &= \begin{cases} i+1 & \text{falls } 0 \leq i < m, x = P[i+1], \\ Border(P[1 \ldots i]x) & \text{sonst.} \end{cases}
\end{aligned}
$$

Bemerkung: Die Zustände entsprechen den Präfixen von P.
Für ein Wort w entspricht der Zustand $\delta_P(0, w)$ dem längsten Präfix von P, das Suffix von w ist.

DEA zur Wortsuche – Beispiel

Für $P = abcabba$ ergibt sich der folgende DEA (Kanten zum Zustand 0 wurden weggelassen):



Konstruktion des DEA zur Wortsuche

Satz. Es sei $|\Sigma| = \sigma$ und $P \in \Sigma^m$. Der Automat A_P kann mit einem Aufwand von $O(m \cdot \sigma)$ konstruiert werden.

Beweis.

- Konstruiere die $Border$-Tabelle für P (in Zeit $O(m)$).
- Zur Bestimmung der Überführungsfunktion benutze die Beziehung

$$Border(P[1 \ldots i]x) = \begin{cases} 0 & \text{falls } i = 0, \\ \delta_P(Border_i(P), x) & \text{falls } 0 < i \leq m. \end{cases}$$

  Aufwand pro Eingabe (i, x) der Überführungsfunktion: $O(1)$.

□

Algorithmus 1.4 DEA-Algorithmus zur Wortsuche

Eingabe: Wörter P, T über Σ mit $|P| = m, |T| = n$
Ausgabe: Menge S der Vorkommen von P in T

(1) Konstruiere den DEA $A_P = (\Sigma, \{0, 1, \ldots, m\}, \delta_P, 0, \{m\})$;
(2) $S \leftarrow \emptyset$; $i \leftarrow 0$;
(3) **for** $j \leftarrow 1$ **to** n
(4) $\quad i \leftarrow \delta_P(i, T[j])$;
(5) $\quad$ **if** $i = m$ **then** $S \leftarrow S \cup \{j - m + 1\}$;
(6) **return** S;

Satz. Algorithmus 1.4 findet alle Vorkommen von P in T mit einer Laufzeit von $\Theta(n)$ für die Suche.

Morris-Pratt-Algorithmus

- Präprozessing bestimmt nur die $Border$-Tabelle
- Nachfolgezustand des DEA ergibt sich durch Rekursion

$$\delta_P(i,x) \quad = \quad \begin{cases} i+1 & \text{falls } 0 \leq i < m, x = P[i+1], \\ 0 & \text{falls } i = 0 \text{ und } x \neq P[i+1] \\ \delta_P(Border_i(P), x) & \text{sonst.} \end{cases}$$

- Laufzeit: $\Theta(m)$ für das Präprozessing, $\Theta(n)$ für die Suche

MP-Algorithmus: Beispiel

Das Wort $P = abcabba$ hat folgende Werte für $Border_i$.

i	1	2	3	4	5	6	7
$Border_i$	0	0	0	1	2	0	1

Damit ergibt sich gemäß der rekursiven Definition von δ_P:

$$\begin{aligned}
\delta_P(5,a) &= \delta_P(Border_5, a) = \delta_P(2,a) \text{ (wegen } P[6] \neq a) \\
&= \delta_P(Border_2, a) = \delta_P(0,a) \text{ (wegen } P[3] \neq a) \\
&= 1 \text{ (wegen } P[1] = a).
\end{aligned}$$

Algorithmus 1.5 Morris-Pratt-Algorithmus

Eingabe: Wörter P, T über Σ mit $|P| = m, |T| = n$
Ausgabe: Menge S der Vorkommen von P in T

(1) Bestimme die Werte $Border_i(P)$;
(2) $S \leftarrow \emptyset$; $i \leftarrow 0$;
(3) **for** $j \leftarrow 1$ **to** n
(4) **while** $i \neq 0$ **and** $P[i+1] \neq T[j]$
(5) $i \leftarrow Border_i(P)$;
(6) **if** $P[i+1] = T[j]$ **then** $i \leftarrow i+1$;
(7) **if** $i = m$ **then** $S \leftarrow S \cup \{j - m + 1\}$;
(8) **return** S;

Satz. Der Morris-Pratt-Algorithmus findet alle Vorkommen von P in T mit einer Laufzeit von $O(n)$.

MP-Algorithmus: Graphische Interpretation

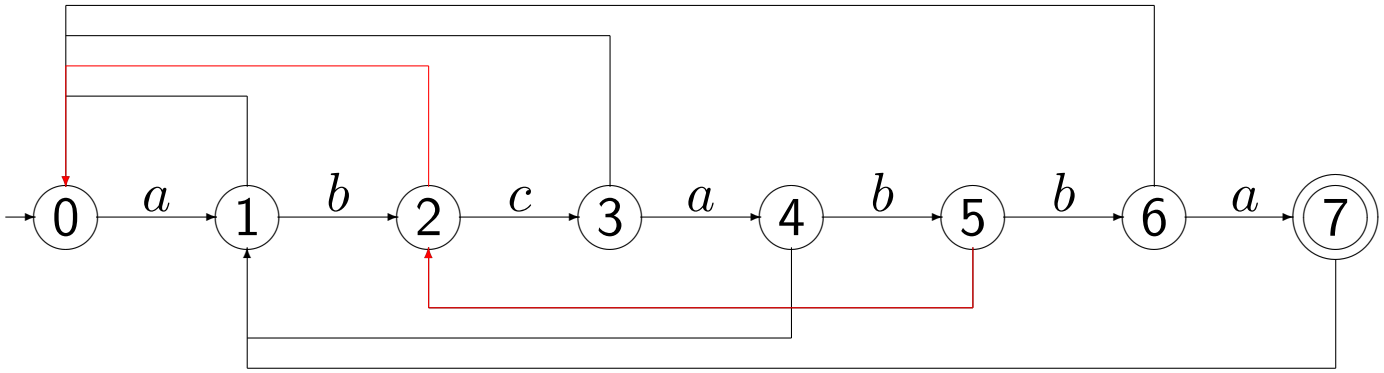
Verwende den Graphen der $Border$-Tabelle von P.

Bestimmung von $\delta_P(i, x)$:

1. Gibt es die Vorwärtskante $(i, x, i+1)$, so ist $\delta_P(i, x) = i + 1$.
2. Ist $i = 0$ und gibt es keine Vorwärtskante $(0, x, 1)$, so ist $\delta_P(i, x) = i + 1$.
3. Anderenfalls folge der Kante zum Knoten $Border_i(P)$ und bestimme rekursiv $\delta_P(i, x) = \delta_P(Border_i(P), x)$.

Graphische Interpretation – Beispiel

Für $P = abcabba$ ergibt sich für $\delta(5, a)$:



1. Gehe zu Knoten $Border_5(P) = 2$ (wegen $P[6] = b \neq a$).
2. Gehe zu Knoten $Border_2(P) = 0$ (wegen $P[3] = c \neq a$).
3. Man erhält $\delta_P(5, a) = 1$ wegen $P[1] = a$.

Knuth-Morris-Pratt-Algorithmus

Definition. Für $1 \leq i \leq |P|$ sei $SBorder_i(P)$ die Länge r des längsten Randes von $P[1 \ldots i]$ mit $P[r+1] \neq P[i+1]$ oder $r = 0$.

Rekursion in MP-Algorithmus lässt sich verfeinern zu

$$\delta_P(i,x) \quad = \quad \begin{cases} i+1 & \text{falls } 0 \leq i < m, x = P[i+1], \\ 0 & \text{falls } i = 0 \text{ und } x \neq P[i+1] \\ \delta_P(SBorder_i(P), x) & \text{sonst.} \end{cases}$$

KMP-Algorithmus macht niemals mehr Vergleiche als der MP-Algorithmus.

Berechnung der $SBorder$-Werte

Satz. Für ein Wort P der Länge m, $1 \leq i \leq m$ und $r = Border_i(P)$ gilt:

$$SBorder_i(P) = \begin{cases} r & \text{falls } r = 0 \text{ oder } P[i+1] \neq P[r+1], \\ SBorder_r(P) & \text{sonst.} \end{cases}$$

Damit ist die $SBorder$-Tabelle in Linearzeit berechenbar,

KMP-Algorithmus: Beispiel

Das Wort $P = abcabba$ hat folgende Werte für $Border_i(P)$ sowie $SBorder_i(P)$.

i	1	2	3	4	5	6	7
$Border_i$	0	0	0	1	2	0	1
$SBorder_i$	0	0	0	0	2	0	1

Damit ergibt sich:

$$\begin{aligned}
\delta_P(4,c) &= \delta_P(Border_4, c) = \delta_P(1,c) \text{ (wegen } P[5] \neq c) \\
&= \delta_P(Border_1, c) = \delta_P(0,c) \text{ (wegen } P[2] \neq c) \\
&= 0 \text{ (wegen } P[1] \neq c) \text{ bzw.} \\
\delta_P(4,c) &= \delta_P(SBorder_4, c) = \delta_P(0,c) \text{ (wegen } P[5] \neq c) \\
&= 0 \text{ (wegen } P[1] \neq c).
\end{aligned}$$

Simon-Algorithmus

Idee: Speichere nur die Kanten in A_P, die nicht zum Zustand 0 führen

Kanten in A_P:

Vorwärtskante: $(k, x, k+1)$
nichttriviale Rückwärtskante: $(k, x, j+1)$ mit $0 \leq j < k$
triviale Rückwärtskante: $(k, x, 0)$

Lemma. Es sei $P \in \Sigma^*$ ein Wort der Länge m. Der Graph des Automaten A_P enthält höchstens m nichttriviale Rückwärtskanten.

Algorithmus 1.6 Simon-Algorithmus

Eingabe: Wörter P, T über Σ mit $|P| = m, |T| = n$

Ausgabe: Menge S der Vorkommen von P in T

(1) Konstruiere den DEA $A_P = (\Sigma, \{0, 1, \ldots, m\}, \delta_P, 0, \{m\})$ ohne triviale Kanten;
(2) $S \leftarrow \emptyset$; $i \leftarrow 0$;
(3) **for** $j \leftarrow 1$ **to** n
(4) **if** $P[i+1] = T[j]$ **then** $i \leftarrow i + 1$;
(5) **else**
(6) $z \leftarrow 0$;
(7) **foreach** Rückwärtskante (i, x, i')
(8) **if** $x = T[j]$ **then** $z \leftarrow i'$; **break**;
(9) $i \leftarrow z$;
(10) **if** $i = m$ **then** $S \leftarrow S \cup \{j - m + 1\}$;
(11) **return** S;

Implementierungen des Simon-Algorithmus

- nichttriviale Rückwärtskanten für jeden Knoten als Liste gespeichert:

  mögliche Verzögerung: $\sigma - 1$
  Sind Zielknoten in fallender Ordnung gespeichert, ist die Verzögerung niemals größer als bei KMP.

- nichttriviale Rückwärtskanten für jeden Knoten als geordnetes Array gespeichert:

  mögliche Verzögerung: $\log_2 \sigma$ (binäre Suche)

Laufzeit des Simon-Algorithmus

Lemma. Es sei $P \in \Sigma^*$ ein Wort der Länge m. Gehen von einem Knoten k im Graphen von A_P r nichttriviale Rückwärtskanten aus, so gilt für jede Rückwärtskante $(k, x, j+1)$: $k - j \geq r$.

Satz. Es seien $P, T \in \Sigma^*$ mit $|P| = m$, $|T| = n$. Jede Implementierung des Simon-Algorithmus, die eine Vorwärtskante in einem Schritt und eine Rückwärtskante von einem Knoten mit r nichttrivialen Rückwärtskanten in höchstens $r + 1$ Schritten findet, findet die Vorkommen von P in T in höchstens $2n$ Schritten.

(Das heißt, für jede "vernünftige" Implementierung hat der Algorithmus lineare Laufzeit (unabhängig von σ).)

Vergleich der Varianten der DEA-Suche

Unterschied bezüglich Anzahl der Schritte zur Berechnung des nächsten Zustandes (Verzögerung, delay).

Algorithmus	max. Verzögerung
DEA-Algorithmus	1 (Realzeit)
MP-Algorithmus	m
KMP-Algorithmus	$\log_\Phi(m)$ mit $\Phi = \frac{1+\sqrt{5}}{2}$
Simon-Algorithmus	$1 + \log_2 \sigma$ (Rückwärtskanten als geordnete Listen)

1.4 Shift-And-Algorithmus

- nutzt durch Bitoperationen mögliche Parallelisierung
- Theoretischer Hintergrund: Nichtdeterministischer endlicher Automat
- Laufzeit: $\Theta(n)$, falls die Länge des Suchwortes nicht größer als die Länge eines Computerwortes ist.
- einfach auf komplexe Suchmuster und inexakte Suche verallgemeinerbar (implementiert z.B. in `agrep` von Manber und Wu)
- NEA und Bitoperationen kommen auch in anderen Suchalgorithmen vor.

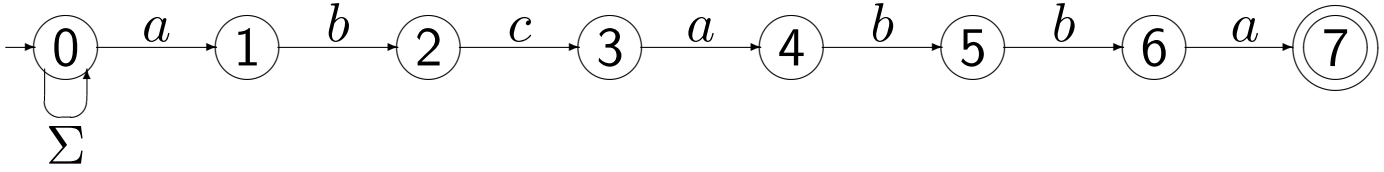
Akzeptierung durch NEA

Die Sprache $\Sigma^* P$, $|P| = m$, wird durch folgenden NEA akzeptiert.

$$A'_P = (\Sigma, \{0, 1, \ldots, m\}, \delta'_P, 0, \{m\}) \text{ mit}$$

$$\delta'_P = \{(i, P[i+1], i+1) : 0 \leq i < m\} \cup \{(0, x, 0) : x \in \Sigma\}.$$

Beispiel. Für $P = abcabba$ ergibt sich der folgende NEA:



Bemerkung: Die Zustände entsprechen den Präfixen von P.
Für ein Wort w entspricht die Menge der erreichbaren Zustände $\delta'_P(0, w)$ der Menge aller Präfixe von P, die Suffix von w sind.

Bitvektoren

Bitvektor der Länge m: Wort aus $\{0,1\}^m$.

Operationen für Bitvektoren:
$\&$ (bitweises AND), $|$ (bitweises OR), $\hat{}$ (bitweises XOR), $\sim$ (bitweise Negation),
<< (Verschiebung (*Shift*) der Bits nach links),
>> (Verschiebung der Bits nach rechts).

Shift-Operationen für Bitvektor $a_m \cdots a_2 a_1$ und $k \in \mathbb{N}$:
$a_m \cdots a_2 a_1 << k = a_{m-k} \cdots a_2 a_1 0^k$
$a_m \cdots a_2 a_1 >> k = 0^k a_m \cdots a_{k+2} a_{k+1}$.

JAVA ermöglicht Arbeit mit Bitvektoren durch den Datentyp **int**.
Für $m \leq 32$ ist eine Operation ein Schritt.

Bitvektoren und Mengen

Sei $M = \{1, 2, \ldots, m\}$.
Darstellung von $M' \subseteq M$ durch den Bitvektor $b_m \cdots b_2 b_1$ mit $b_i = 1 \iff i \in M'$.

Realisierung mengentheoretischer Operationen:

- Einermenge $\{i\}$ erzeugen: $1 << (i-1)$.
- Komplement der Menge A: $\sim A$.
- Vereinigung der Mengen A und B: $A \mid B$.
- Durchschnitt der Mengen A und B: $A \,\&\, B$.
- Test, ob $i \in A$: $A \,\&\, (1 << (i-1)) \neq 0$.

Idee des Shift-And-Algorithmus

Menge der erreichbaren Zustände wird durch Bitvektor $Z = z_m \dots z_2 z_1$ kodiert.
Zustand $i \in \{1, 2, \dots, m\}$ durch $T[1 \dots j]$ erreichbar $\iff$ $z_i = 1$.

Präprozessing: Jedem Buchstaben $x \in \Sigma$ wird ein Bitvektor $B[x]$ zugewiesen.
In $B[x]$ ist das i-te Bit von hinten 1 $\iff$ $P[i] = x$

Initialisierung: $Z \leftarrow 0$.

Aktualisierung für Z an der Textstelle j:

$Z \leftarrow Z \mathbin{<<} 1;$	Das Bit z_i erhält den bisherigen Wert von z_{i-1}, $i \geq 2$.
$Z \leftarrow Z \mid 1;$	Setze das Bit z_1 auf 1.
$Z \leftarrow Z \mathbin{\&} B[x];$	Das Bit z_i behält den Wert 1 $\iff$ $P[i] = x$.

Algorithmus 1.7 Shift-And-Algorithmus

Eingabe: Wörter P, T mit $|P| = m, |T| = n$
Ausgabe: Menge S der Vorkommen von P in T

(1) **foreach** $x \in \Sigma$
(2) $\quad B[x] \leftarrow 0$;
(3) **for** $i \leftarrow 1$ **to** m
(4) $\quad B[P[i]] \leftarrow B[P[i]] \,|\, (1 \texttt{<<} (i-1))$;
(5) $S \leftarrow \emptyset$; $Z \leftarrow 0$;
(6) **for** $j \leftarrow 1$ **to** n
(7) $\quad Z \leftarrow ((Z \texttt{<<} 1) \,|\, 1) \,\&\, B[T[j]]$;
(8) $\quad$ **if** $(Z \,\&\, (1 \texttt{<<} (m-1))) \neq 0$ **then** $S \leftarrow S \cup \{j - m + 1\}$;
(9) **return** S;

Satz. Es seien P und T Wörter über Σ mit $|P| = m$, $|T| = n$ und $|\Sigma| = \sigma$. Der Shift-And-Algorithmus findet alle Vorkommen von P in T in einer Zeit von $O(n \cdot \lceil m/w \rceil)$ und benötigt für das Präprozessing eine Zeit von $O(m + \sigma \cdot \lceil m/w \rceil)$, wobei w die Länge eines Computerwortes ist.

Beispiel

Für $\Sigma = \{a, b, c\}$, $P = abcabba$ und $T = abaabcabbab$ ergibt sich folgender Ablauf des Algorithmus (das rechteste Bit von Z ist unten). Das Ende eines Vorkommens erkennt man am 7. Bit von rechts.

B_a	B_b	B_c
0	0	0
1	0	0
0	1	0
0	1	0
1	0	0
0	0	1
0	1	0
1	0	0

	a	b	a	a	b	c	a	b	b	a	b
0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	1	0
0	0	0	0	0	0	0	0	0	1	0	0
0	0	0	0	0	0	0	0	1	0	0	0
0	0	0	0	0	0	0	1	0	0	0	0
0	0	0	0	0	0	1	0	0	0	0	0
0	0	1	0	0	1	0	0	1	0	0	1
0	1	0	1	1	0	0	1	0	0	1	0

Shift-Or-Algorithmus

Zustand $i \in \{1, 2, \ldots, m\}$ erreichbar $\iff$ in Z' ist das i-te Bit von rechts 0.

Präprozessing: Jedem Buchstaben $x \in \Sigma$ wird ein Bitvektor $B'[x]$ zugewiesen.
In $B'[x]$ ist das i-te Bit von hinten 0 $\iff$ $P[i] = x$

Initialisierung: $Z' \leftarrow \sim 0$. (alle Bits auf 1)

Aktualisierung für Z an der Textstelle j

$$Z' \leftarrow (Z' << 1) \mid B'[T[j]].$$

- wegen einfacherer Aktualisierungsregel etwas schneller als Shift-And

Erweiterung: Buchstabenklassen

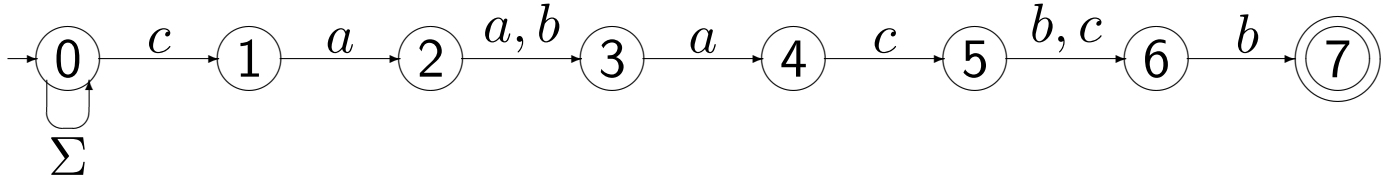
Suchmuster $P = S_1 S_2 \cdots S_m$ mit Teilmengen von $S_i \subseteq \Sigma$.

Vorkommen in T: $T[k \ldots k+m-1]$ mit $T[k+i-1] \in S_i$.

Anpassung des Shift-And-Algorithmus:
In $B[x]$ ist das i-te Bit von rechts 1 $\iff$ $x \in S_i$.

Beispiel. $P = ca\{a,b\}ac\{b,c\}b$. Treffer: $caaacbc$, $caaaccb$, $cabacbb$ und $cabaccb$.

NEA:



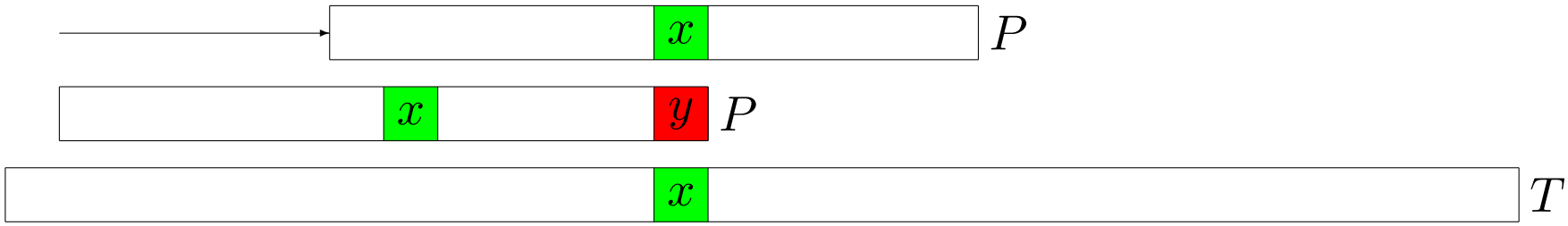
Bitvektoren: $B[a] = 0001110,\ B[b] = 1100100,\ B[c] = 0110001.$

1.5 Algorithmen von Boyer-Moore und Horspool

- Vergleiche im Suchfenster von rechts nach links.
- Verschiebungsheuristiken: Bad Character und Good Suffix.
- Laufzeit: $O(mn)$ im schlechtesten Fall, $O(n/\sigma)$ im durchschnittlichen Fall.
- Es gibt Varianten mit Laufzeit $O(n)$ im schlechtesten Fall, $O(\frac{n \cdot \log m}{m})$ im durchschnittlichen Fall.

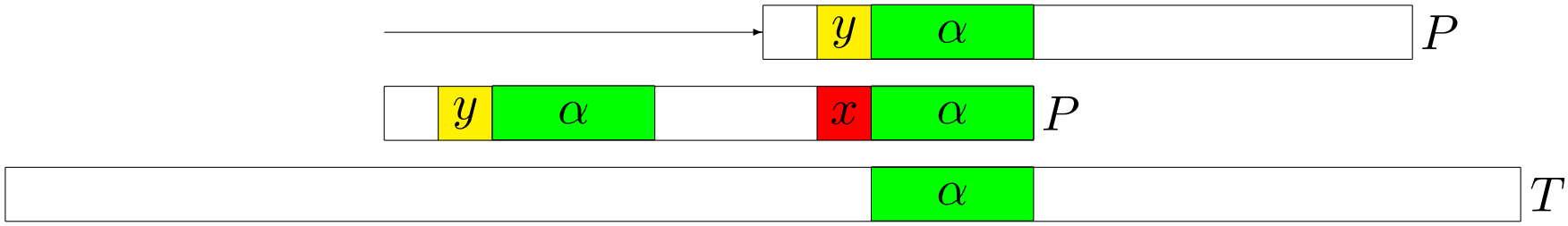
Bad Character Regel

Findet man beim ersten Vergleich das Textsymbol $x \neq P[m]$, so darf man P so weit verschieben, dass das rechteste Vorkommen von x in P auf diese Textposition trifft.



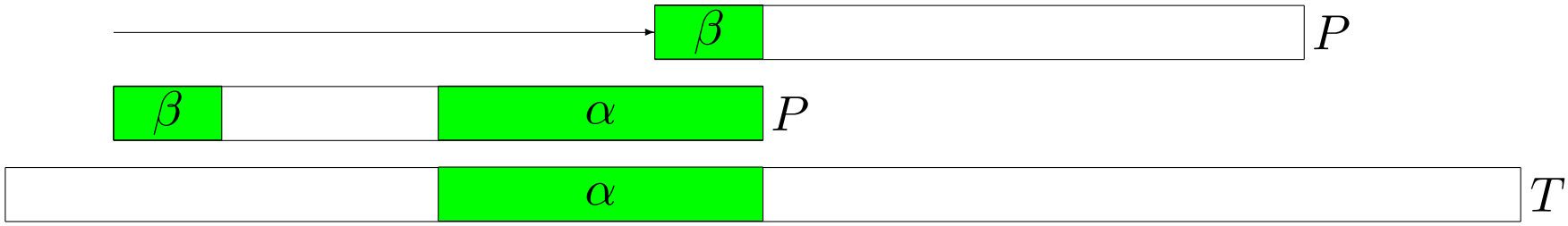
(Starke) Good Suffix Regel – Teil 1

Stimmt ein Suffix α von P mit dem aktuellen Text überein, und stimmt das Suffix $x\alpha$ nicht mit dem Text überein, so darf man P so weit verschieben, dass das letzte Vorkommen von α in P, das kein Vorkommen von $x\alpha$ ist, gegenüber dem entsprechenden Text steht.



Good Suffix Regel – Teil 2

Sollte ein solches Vorkommen des Suffixes α nicht existieren, so darf man P so weit verschieben, dass das längste Suffix β von α, das ein Präfix von P' ist, gegenüber dem Ende des aktuellen Textstückes steht.



Formalisierung der Bad Character Regel

Definition. Für $P \in \Sigma^*$ mit $|P| = m$ und $x \in \Sigma$ ist

$$R_x(P) := \max\left(\{1 \leq i \leq m : P[i] = x\} \cup \{0\}\right),$$

d.h., $R_x(P)$ ist das letzte Vorkommen von x in P.
Außerdem sei $\mathit{Shift}_x(P) = m - R_x(P)$.

Ist der erste Vergleich ein Mismatch mit dem Textzeichen x,
so verschiebe um $\mathit{Shift}_x(P)$.

Formalisierung der Good Suffix Regel

Definition. Sei $P \in \Sigma^*$ mit $|P| = m$. Für $0 \leq i \leq m-1$ sei $L_i(P)$
das Ende des letzten Vorkommens von $P[i+1 \ldots m]$ in $P[1 \ldots m-1]$, das nicht das Ende eines Vorkommens von $P[i \ldots m]$ ist bzw.
die Länge des längsten Präfixes von P, das ein echtes Suffix von $P[i+1 \ldots m]$ ist, falls ein solches Vorkommen von $P[i+1 \ldots m]$ nicht existiert.

Stimmt $P[i+1 \ldots m]$ mit dem Text überein, $P[i \ldots m]$ aber nicht,
so verschiebe um $m - L_i(P)$.

Algorithmus 1.8 Boyer-Moore-Algorithmus

Eingabe: Wörter P, T mit $|P| = m, |T| = n$
Ausgabe: Menge S der Vorkommen von P in T

(1) *Präprozessing:* Bestimme $R_x(P)$, $x \in \Sigma$ und $L_i(P)$, $0 \leq i < m$.
(2) $S \leftarrow \emptyset; k \leftarrow m;$
(3) **while** $k \leq n$
(4) **if** $P[m] \neq T[k]$ **then** $k \leftarrow k + m - R_{T[k]}(P);$
(5) **else**
(6) $i \leftarrow m - 1;\ j \leftarrow k - 1;$
(7) **while** $i > 0$ **and** $P[i] = T[j]$
(8) $i \leftarrow i - 1;\ j \leftarrow j - 1;$
(9) **if** $i = 0$ **then** $S \leftarrow S \cup \{k - m + 1\};$
(10) $k \leftarrow k + m - L_i(P);$
(11) **return** $S;$

Satz. Der Boyer-Moore-Algorithmus findet alle Vorkommen von P in T.

Beispiel

Für $P = abcabba$ mit $\Sigma = \{a, b, c, d\}$ erhalten wir:

x	a	b	c	d
R_x	7	6	3	0
$Shift_x$	0	1	4	7

i	0	1	2	3	4	5	6
L_i	1	1	1	1	1	1	4

Dies führt zu folgenden Verschiebungen.

Suchwort: Textausschnitt:	`abcabba` `.....d...`	`abcabba` `.....c...`	`abcabba` `.....aa..`	`abcabba` `....cba...`
Verschiebung:	7	4	3	6

Boyer-Moore-Präprozessing

- Bestimmung der R_x-Werte sehr einfach. Zeit: $O(\sigma + m)$.
- Bestimmung der L_i-Werte in Zeit $O(m)$ möglich (kompliziert).
  Enge Beziehung zu Z-Werten

Satz. Das Boyer-Moore-Präprozessing kann mit einem Aufwand von $O(\sigma + m)$ erfolgen.

Laufzeit im schlechtesten Fall

- $\Theta(mn)$ für $P = a^m, T = a^n$
- Laufzeit beträgt $\Theta(n)$, falls P nicht in T vorkommt (Knuth, Morris, Pratt)
  Beweis: siehe [Gusfield], Abschnitt 3.2
- lineare Laufzeit bei Beachtung der folgenden Regel

Regel von Galil
Nach dem Auffinden eines Vorkommens von P verschiebe um $Per(P)$ und vergleiche in der nächsten Phase höchstens die letzten $Per(P)$ Zeichen.

Horspool-Algorithmus

- Verschiebung nur nach Bad Character Regel
  Falls erster Vergleich positiv, Verschiebung um 1
- Im Mittel etwas mehr Vergleiche als Boyer-Moore-Algorithmus;
  dafür sehr viel einfacheres Präprozessing
- in der Praxis schnellster Algorithmus, falls $m < \sigma$
  (z.B. für natürlichsprachige Texte)
- Verbesserung: Verschiebung um $m - R_x(P[1 \ldots m-1])$ für Textzeichen x

Algorithmus 1.11 Horspool-Algorithmus

Eingabe: Wörter P, T mit $|P| = m, |T| = n$

Ausgabe: Menge S der Vorkommen von P in T

(1) *Präprozessing:* Bestimme $R_x(P)$, $x \in \Sigma$;
(2) $S \leftarrow \emptyset; k \leftarrow m;$
(3) **while** $k \leq n$
(4) **if** $P[m] \neq T[k]$ **then** $k \leftarrow k + m - R_{T[k]}(P);$
(5) **else**
(6) $i \leftarrow m - 1;\ j \leftarrow k - 1;$
(7) **while** $i > 0$ **and** $P[i] = T[j]$
(8) $i \leftarrow i - 1;\ j \leftarrow j - 1;$
(9) **if** $i = 0$ **then** $S \leftarrow S \cup \{k - m + 1\};$
(10) $k \leftarrow k + 1;$
(11) **return** S;

Durchschnittliche Laufzeit des Horspool-Algorithmus

$Comp(m)$ – mittlere Anzahl der Vergleiche in einer Phase
Wie bei Naivem Algorithmus: $Comp(m) \leq \frac{\sigma}{\sigma-1}$.

$Shift(m)$ – mittlere Verschiebung nach einer Phase

Mittlere Laufzeit: $\overline{t}(m,n) \approx \frac{n \cdot Comp(m)}{Shift(m)}$.

Mittlere Verschiebung $Shift(m)$

$$Shift(m) = \sum_{i=0}^{m-1} p_i \cdot (i+1)$$

p_i: Wahrscheinlichkeit, daß $R_x(P) = m - 1 - i$ für zufälliges $x \in \Sigma$ und $P \in \Sigma^m$.

$p_i = \left(\frac{\sigma-1}{\sigma}\right)^i \cdot \frac{1}{\sigma}$ für $0 \le i \le m-2$, $p_{m-1} = \left(\frac{\sigma-1}{\sigma}\right)^{m-1}$

Rechnung analog zum Naiven Algorithmus ergibt:
$Shift(m) \approx \sigma\left(1 - \left(1 - \frac{1}{\sigma}\right)^m\right)$.

Folgerung. $Shift(m) = \Theta(\min\{\sigma, m\})$.

Für $\sigma = 100$ erhalten wir beispielsweise:

m	2	10	50	100	200	1000
$Shift(m)$	1.99	9.6	39.5	63.4	86.6	99.996

Erweiterte Bad Character Regel

- Bestimme letztes Vorkommen in P für jedes Wort aus Σ^q, $q \geq 1$.
- Bestimme in jeder Phase die letzten q Textzeichen α und verschiebe P bis zum letzten Vorkommen von α.
- große Effizienzsteigerung für kleine Alphabete (z.B. DNA)

Erweiterte BCR: Formalisierung

Definition. Für $q \geq 1$, $\alpha \in \Sigma^q$, $P \in \Sigma^*$, $|P| = m \geq q$ sei $R_\alpha(P)$ die rechteste Stelle in P, an der ein Vorkommen von α endet bzw. $(q-1)$, falls α nicht in P auftritt.
Wir definieren $Shift_\alpha(P) := m - R_\alpha(P)$.

Erweiterte Bad Character Regel
Stimmen die q letzten Zeichen von P nicht mit dem entsprechenden Textstück α überein, so verschiebe um den Betrag $Shift_\alpha(P)$.

Erweiterte Bad Character Regel: Beispiel

$\Sigma = \{a, b, c\}$, $P = abcabba$, $q = 2$

α	aa	ab	ac	ba	bb	bc	ca	cb	cc
R_α	1	5	1	7	6	3	4	1	1
$Shift_\alpha$	6	2	6	0	1	4	3	6	6

Mittlere Verschiebung für P

einfache Bad Character Regel: $5/3$.

erweiterte Bad Character Regel: $34/9$.

Erweiterte Bad Character Regel – Laufzeit

- **Zeit für Präprozessing:** $\Theta(m + \sigma^q)$
- **Durchschnittliche Zeit für die Suche**

  $Comp(m) \approx q,\ Shift(m) = \Theta(\min\{m, \sigma^q\})$
  $\bar{t}(m,n) = \Theta(\frac{n \cdot q}{\min\{m, \sigma^q\}})$
- **Optimale Wahl**

  $q = \log_\sigma m$: $\bar{t}(m,n) = \Theta(\frac{n \cdot \log m}{m})$

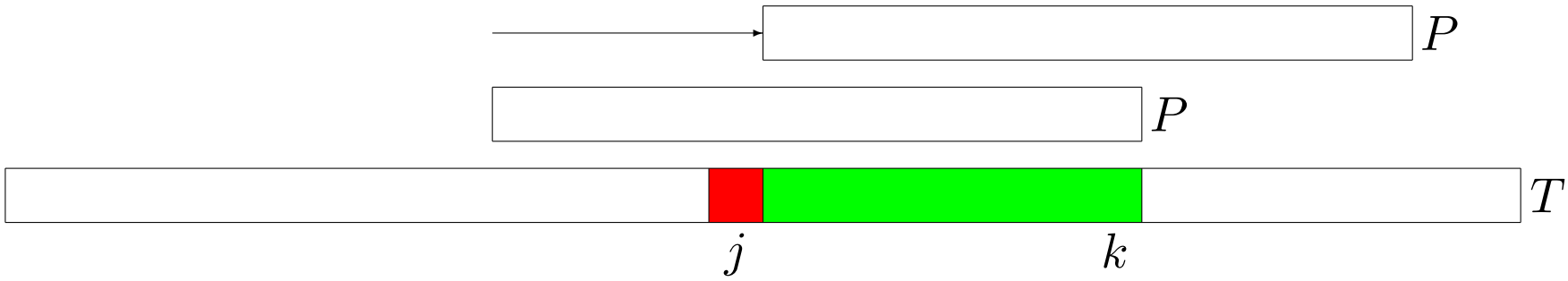
1.6 Algorithmen mit Suffixautomaten (Faktor-Algorithmen)

- wie beim Boyer-Moore-Algorithmus Vergleich von rechts nach links
- Ausführung der Vergleiche, bis Textteil kein Faktor des Suchwortes ist
- mittlere Laufzeit: $\Theta(\frac{n \cdot \log m}{m})$
- Test der Faktor-Eigenschaft durch endlichen Automaten
  - 1. Variante: Verwendung eines DEA (deterministic acyclic word graph)
  - 2. Variante: Verwendung eines NEA (ähnlich zu Shift-And)
  - 3. Variante: Verwendung eines Orakel-DEA

Faktor-Algorithmen – Grundidee

$T[j+1 \ldots k]$ sei ein Faktor von P, $T[j \ldots k]$ sei kein Faktor von P

$\longrightarrow$ ein Vorkommen von P kann frühestens an der Stelle $j+1$ beginnen
$\longrightarrow$ Verschiebung um $m-k+j$ Stellen möglich



Sollte an der Stelle k ein Vorkommen von P enden, so verschiebt man um 1.

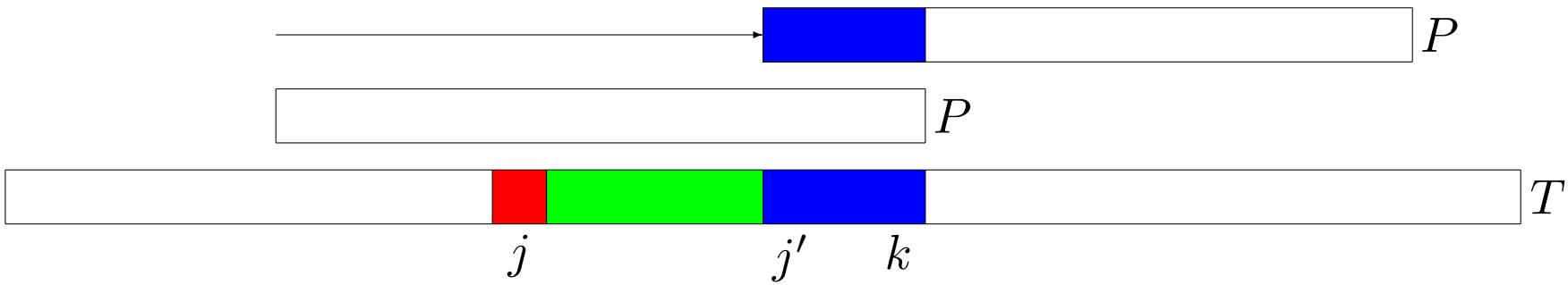
Algorithmus 1.12 Faktor-Algorithmus(Prinzip)

Eingabe: Wörter P, T über Σ mit $|P| = m, |T| = n$
Ausgabe: Menge S der Vorkommen von P in T

(1) $k \leftarrow m$;
(2) **while** $k \leq n$
(3) $\quad j \leftarrow k$;
(4) $\quad$ **while** $T[j \ldots k]$ ist Faktor von P
(5) $\quad\quad j \leftarrow j - 1$;
(6) $\quad$ **if** $j = k - m$ **then** $S \leftarrow S \cup \{k - m + 1\}$; $k \leftarrow k + 1$;
(7) $\quad\quad\quad\quad$ **else** $k \leftarrow j + m$;
(8) **return** S;

Verbesserte Verschiebungsregel

Speichere kleinste Zahl j', für die $T[j' \ldots k]$ ein echtes Präfix von P ist.
Nach der Suchphase Verschiebung bis zur Stelle j'.



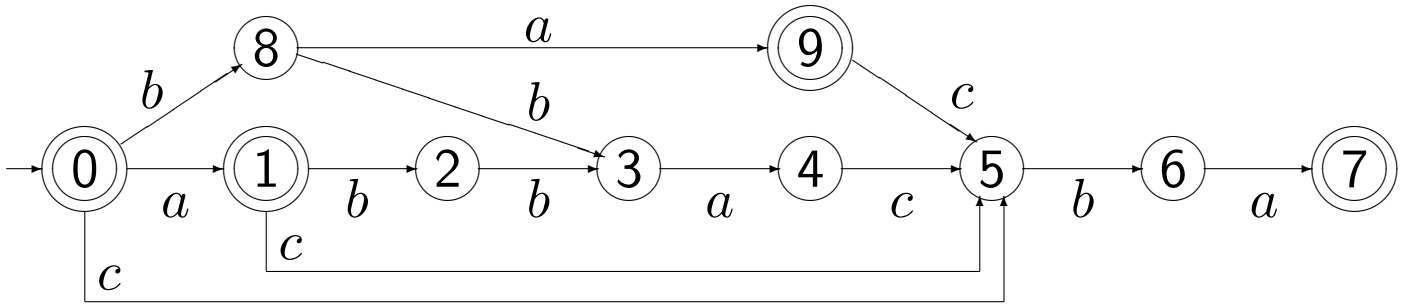
Vorteil: Weitere Verschiebung
Nachteil: Präfix-Test kostet Zeit.

Suffixautomaten

Definition. Es sei w ein Wort. Der Suffixautomat (auch DAWG für Directed Acyclic Word Graph) ist der minimale partielle deterministische endliche Automat, dessen akzeptierte Sprache die Menge der Suffixe von w ist.

Bemerkung: Für die Faktorsuche von rechts nach links benötigt man den DAWG von P^r.

Beispiel. Für $P = abcabba$ ergibt sich der folgende DAWG von $P^r = abbacba$:



Algorithmus 1.13 Backward DAWG Matching (BDM-Algorithmus)

Eingabe: Wörter P, T über Σ mit $|P| = m, |T| = n$
Ausgabe: Menge S der Vorkommen von P in T

(1) Konstruiere den DAWG für P^r $A = (\Sigma, Z, \delta, z_0, F)$;
(2) $S \leftarrow \emptyset$; $z \leftarrow z_0$; $k \leftarrow m$;
(3) **while** $k \leq n$
(4) $\quad j \leftarrow k$;
(5) $\quad$ **while** $\delta(z, T[j])$ existiert
(6) $\quad\quad z \leftarrow \delta(z, T[j])$; $j \leftarrow j - 1$;
(7) $\quad$ **if** $j = k - m$ **then** $S \leftarrow S \cup \{k - m + 1\}$; $k \leftarrow k + 1$;
(8) $\quad\quad\quad\quad$ **else** $k \leftarrow j + m$;
(9) **return** S;

Backward DAWG Matching – Laufzeit

- Aufwand für einen Schritt (Nachfolgezustand bestimmen) sei konstant.
  Bei vollständiger Zustandstabelle korrekt, sonst Aufwand $O(\log_2 \sigma)$.

- im schlechtesten Fall: $\Theta(m \cdot n)$.

- im Durchschnittsfall: $\Theta\left(\frac{n \cdot \ell_m}{m - \ell_m}\right)$

  ℓ_m - mittlere Länge des längsten Faktors von P, der Suffix des Suchfensters ist.

  Es gilt $\ell_m \approx \log_\sigma m$, d.h. mittlere Gesamtlaufzeit: $\Theta\left(\frac{n \cdot \log_\sigma m}{m}\right)$.

- Präprozessing: Konstruktion des DAWG in Zeit $O(m)$ möglich, aber kompliziert.
  (Der DAWG besitzt höchstens $2m$ Knoten und $4m$ Kanten.)

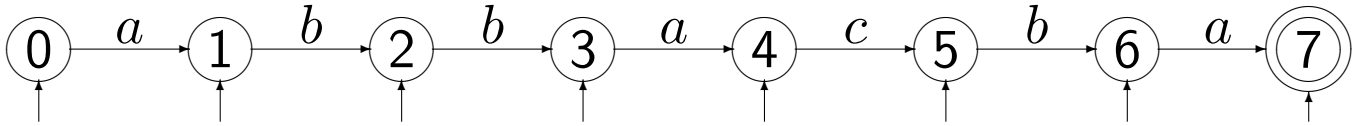
Nichtdeterministische Suffix-Automaten

Definition. Für ein Wort $S \in \Sigma^*$ der Länge m ist der Suffix-NEA der Automat

$$A_S = (\Sigma, \{0, 1, \ldots, m\}, \delta, \{0, 1, \ldots, m\}, \{m\})$$

mit $\delta = \{(i-1, P[i], i)\}$ für $1 \leq i \leq m$.

Beispiel. Für $P = abcabba$ ergibt sich der folgende Suffix-NEA von $P^r = abbacba$:



Implementierung mittels Bit-Arithmetik

Bitvektor Z der Länge $m+1$:
i-tes Bit von rechts ist $1 \iff$ Zustand $(m-i+1)$ erreichbar

Präprozessing: für jedes $x \in \Sigma$ konstruiere Byte $B[x]$
In $B[x]$ ist das i-te Bit von hinten genau dann 1, wenn $P[i] = x$ gilt.

Initialisierung: $Z \leftarrow (1 \mathbin{<<} (m+1)) - 1$ (setze die ersten $m+1$ Bits von rechts auf 1)

Aktualisierung: $Z \leftarrow (Z \mathbin{>>} 1) \,\&\, B[T[j]]$.

Faktor liegt vor, solange $Z \neq 0$.
Präfix liegt vor, wenn letztes Bit von Z gleich 1.

Algorithmus 1.14 Backward NDAWG Matching (BNDM-Algorithmus)

Eingabe: Wörter P, T mit $|P| = m, |T| = n$
Ausgabe: Menge S der Vorkommen von P in T

(1) **foreach** $x \in \Sigma$
(2) $\quad B[x] \leftarrow 0$;
(3) **for** $i \leftarrow 1$ **to** m
(4) $\quad B[P[i]] \leftarrow B[P[i]] \,|\, (1 \texttt{<<} (i-1))$;
(5) $S \leftarrow \emptyset$; $k \leftarrow m$;
(6) **while** $k \leq n$
(7) $\quad Z \leftarrow 1^{m+1}$; $j \leftarrow k$;
(8) $\quad$ **while** $Z \neq 0$
(9) $\quad\quad Z \leftarrow (Z \texttt{>>} 1) \,\&\, B[T[j]]$;
(10) $\quad\quad$ **if** $Z \neq 0$ **then** $j \leftarrow j - 1$;
(11) $\quad$ **if** $j = k - m$ **then** $S \leftarrow S \cup \{k - m + 1\}$; $k \leftarrow k + 1$;
(12) $\quad\quad\quad\quad$ **else** $k \leftarrow j + m$;
(13) **return** S;

BNDM-Algorithmus: Beispiel

Für das Wort $P = abcabba$ erhalten wir im Präprozessing die Bitvektoren $B_a = (01001001), B_b = (00110010), B_c = (00000100)$.

Für das Textfenster $abcacab$ ergibt sich die Bitvektoren-Folge

$$(11111111) \xrightarrow{b} (00110010) \xrightarrow{a} (00001001) \xrightarrow{c} (00000100) \xrightarrow{a} (00000000).$$

Der längste Faktor ist folglich cab, das längste Präfix ist ab.

Orakel-DEA

Orakel-DEA ist ein partieller DEA mit folgenden Eigenschaften:
Einziges akzeptiertes Wort der Länge m ist P.
Ist kein Zustand definiert, so ist die Eingabe garantiert kein Faktor von P.
Umkehrung gilt nicht!

Suche analog zum DAWG, aber etwas kürzere Verschiebungen.
Asymptotisch genauso gut wie DAWG.

Vorteil gegenüber DAWG: einfachere Struktur, einfacher zu konstruieren

Vorteil gegenüber NDAWG: effizient auch für lange Suchwörter

Orakel-DEA: Definition

Induktiver Aufbau für S mit $|S| = m$:

1. Beginne mit Startzustand 0 und ohne Transitionen.

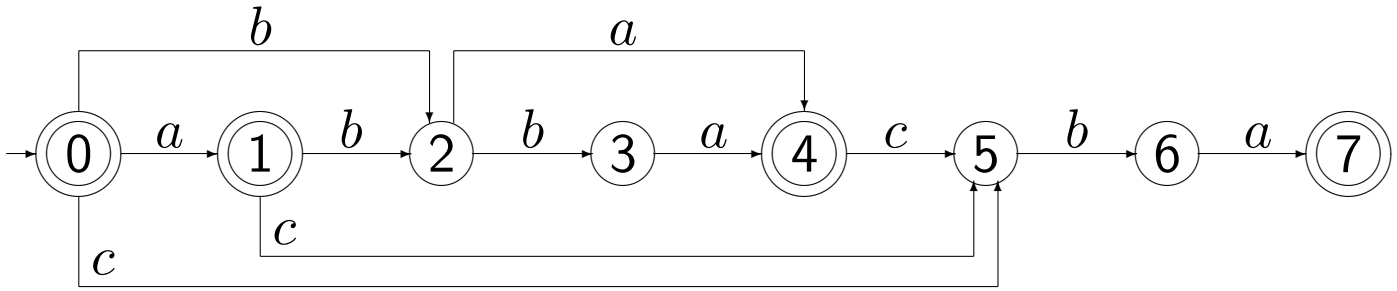
2. Für $0 \leq i \leq m-1$ seien Zustände $0, 1, \ldots, i$ und Transitionenmenge δ konstruiert.
   Füge Zustand $i+1$ hinzu.
   Für jedes Suffix β von $S[1 \ldots i]$ füge Transition $(\delta^*(0, \beta), S[i+1], (i+1))$ hinzu, sofern noch keine Transition $(\delta^*(0, \beta), S[i+1], j)$ mit $j \leq i$ existiert.

Endzustände: $\{\delta^*(0, \beta) : \beta$ ist Suffix von $S\}$.

Bemerkung: Der Orakel-DEA hat höchstens $2m$ Transitionen und kann in Linearzeit konstruiert werden.

Orakel-DEA: Beispiel

Für $P = abcabba$ ergibt sich der folgende Orakel-DEA von $P^r = abbacba$:



1.7 Duell-Algorithmus von Vishkin

- 1. Phase: Ermittlung von "verträglichen" Kandidaten für ein Vorkommen von P
  Ausschluß der meisten unmöglichen Positionen durch Duelle
- 2. Phase: Prüfung der Kandidaten
- Laufzeit: $\Theta(n)$
- Parallelisierung mit paralleler Laufzeit $\Theta(\log m)$ und Arbeit $\Theta(n)$ möglich

Verträglichkeit von Positionen

Definition. Es sei P ein Wort mit $|P| = m$. Zwei natürliche Zahlen j und k mit $j < k$ heißen verträglich bezüglich P, wenn es einen Text T gibt, der sowohl an der Stelle j als auch an der Stelle k ein Vorkommen von P enthält.

Folgerung. Es sei P ein Wort mit $|P| = m$. Zwei Zahlen j und k mit $j < k$ sind genau dann verträglich bezüglich P, wenn $k - j$ eine Periode von P oder größer als m ist.

Lemma. ("Transitivität" der Verträglichkeit)
Sind jeweils $j_1 < j_2$ und $j_2 < j_3$ verträglich bezüglich P, so sind auch j_1 und j_3 verträglich bezüglich P.

Unverträgliche Positionen und Duelle

- Erinnerung Z-Werte:
  Ist $i < |P|$ keine Periode von P, so gilt $P[Z_i(P)+1] \neq P[Z_i(P)+1-i]$.

- Sind $k_1 < k_2$ unverträglich bezüglich P, so ist höchstens eine der Positionen k_1, k_2 mit $T[k_1 + Z_d(P)]$ vereinbar für $d = k_2 - k_1$.

- Der Vergleich von $P[Z_d(P)+1]$ und $T[k_1 + Z_d(P)]$ schließt eine der Positionen k_1, k_2 als Vorkommen von P aus.

Prozedur: $\textsc{Duell}(k_1, k_2)$
Eingabe: Wörter P, T, unverträgliche Positionen $k_1 < k_2$
Ausgabe: nicht ausgeschlossene Position von $\{k_1, k_2\}$
(1) $d \leftarrow k_2 - k_1$;
(2) **if** $T[k_1 + Z_d(P)] = P[1 + Z_d(P)]$ **then return** k_1;
(3) **else return** k_2;

Algorithmus von Vishkin: Phase 1

- Kandidatenmenge C von paarweise verträglichen Positionen wird als Stack von rechts nach links ermittelt.

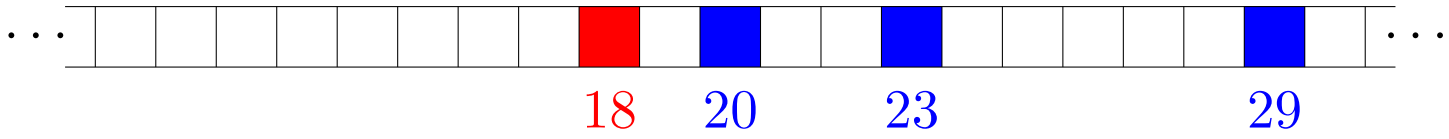
- Ist der neue Kandidat j mit $\textsc{Top}(C)$ verträglich, so ist j mit allen Elementen von C verträglich (wegen der "Transitivität").

- Anderenfalls duelliert sich j mit $\textsc{Top}(C)$.

  Verliert j, so ist der Kandidat j erledigt.
  Gewinnt j, so wird $\textsc{Top}(C)$ aus C entfernt und j mit der neuen Stackspitze verglichen.

Beispiel

Es sei $P = abcabcabcab$. Die kürzeste Periode von P ist 3.
Die obersten Stackpositionen seien $20, 23, 29$; die aktuelle Position 18.



Die Positionen 18 und 20 sind unverträglich; es gilt $Z_2(P) = 2$.
Ist $T[20] = c = P[3]$, so gewinnt die Position 18 das Duell, Position 20 scheidet aus; anderenfalls scheidet Position 18 aus.
Sollte die Position 18 das Duell gewinnen, so sind 18 und 23 ebenfalls unverträglich und müssen sich ebenfalls duellieren.
Gewinnt Position 18 erneut, so wird sie die neue Stackspitze, da 18 und 29 verträglich sind.

Algorithmus 1.15 Algorithmus von Vishkin: 1. Phase

Eingabe: Wörter P,T, $|P| = m$, $|T| = n$
Ausgabe: Menge (Stack) C von Kandidaten für Vorkommen von P in T

(1) $C \leftarrow \{n+1\}$;
(2) **for** $k \leftarrow n-m+1$ **downto** 1
(3) **while** k und $\textsc{Top}(C)$ unverträglich **and** $\textsc{Duell}(k, \textsc{Top}(C)) = k$
(4) $\textsc{Pop}(C)$;
(5) **if** k und $\textsc{Top}(C)$ verträglich **then** $\textsc{Push}(C, k)$;
(6) **return** C;

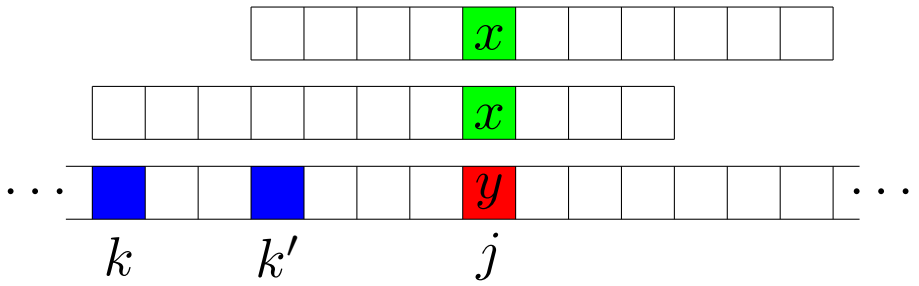
Satz. Nach dem Ablauf von Algorithmus 1.15 enthält die Menge C alle Vorkommen von P in T, wobei die Positionen in C paarweise verträglich bezüglich P sind. Die Laufzeit von Algorithmus 1.15 ist $\Theta(n)$.

Algorithmus von Vishkin: Phase 2

Teste für jede Position in C, ob sie ein Vorkommen ist.

Jede Textstelle muss nur einmal betrachtet werden.

Beispiel. Es sei $P = abcabcabcab$; k und $k' = k + 3$ seien mögliche Kandidaten. Stimmt $T[k \cdots k + 10]$ mit P überein, so gibt es auch eine Übereinstimmung von $T[k' \cdots k' + 7]$ mit $P[1 \cdots 8]$.
Gibt es dagegen ein Mismatch zwischen $P[i]$ und $T[j]$, $j = k + i - 1$, so existiert auch ein Mismatch zwischen $P[i - 3]$ und $T[j]$.



Algorithmus 1.16 Algorithmus von Vishkin: 2. Phase

Eingabe: Wörter P,T, $|P| = m$, $|T| = n$, Stack C (aus der 1. Phase)
Ausgabe: Menge S der Vorkommen von P in T

```
(1)  k ← Pop(C); i ← 1; j ← k;
(2)  while k ≤ n
(3)    while i ≤ m and P(i) = T(j)
(4)      i ← i + 1; j ← j + 1;
(5)    if i > m then
(6)      S ← S ∪ {k}; k ← Pop(C);
(7)      if k ≤ j then i ← j − k + 1;
(8)                else j ← k; i ← 1;
(9)    else
(10)     while k ≤ j
(11)       k ← Pop(C);
(12)     j ← k; i ← 1;
(13) return S;
```

Satz. Algorithmus 1.16 bestimmt die Vorkommen von P in T mit einem Aufwand von $O(n)$.

Modell des Parallelrechners: CREW PRAM

- PRAM: Parallel Random Access Machine
  - (Unbegrenzt viele) Prozessoren haben gemeinsamen Speicher.
  - Jeder Prozessor arbeitet sein eigenes Programm ab.
    Die Arbeit der Prozessoren ist synchronisiert.

- CREW: Concurrent Read Exclusive Write
  - Jeder Prozessor darf an jeder Stelle des gemeinsamen Speichers lesen.
  - Keine zwei Prozessoren dürfen an der gleichen Stelle des gemeinsamen Speichers schreiben.

- Performanzmaße für parallele Algorithmen
  - Zeit: Anzahl der parallelen Schritte
  - Arbeit: Summe der Anzahl der Schritte, in denen die Prozessoren aktiv waren

Parallelisierung des Naiven Algorithmus

- Für jede Textposition k sind m Prozessoren $\Pi_{k,1}, \Pi_{k,2}, \ldots \Pi_{k,m}$ zuständig.
- Prozessor $\Pi_{k,i}$ bestimmt, ob $T[k+i-1] = P[i]$ gilt, setzt Bit $B_{k,i}$.
- P kommt an der Stelle k vor, wenn $\bigwedge_{i=1}^{m} B_{k,i} = 1$ gilt.
- Berechnung des logischen UND für m Bits in $\lceil \log_2 m \rceil$ parallelen Schritten und mit $\Theta(m)$ Arbeit möglich.
- Insgesamt: $\Theta(\log m)$ Zeit und $\Theta(mn)$ Arbeit.

Parallelisierung des Duell-Algorithmus

Kürzeste Periode von P sei p.

Teile den Text in Intervalle der Länge p.
In jedem Intervall sind die Positionen paarweise unverträglich.

1. Bestimme durch Duelle für jedes Intervall einen Kandidaten.
Zeit: $O(\log p)$, Arbeit: $O(p)$ je Intervall, $O(n)$ gesamt.

2. Ermittle für jeden Kandidaten durch den parallelen Naiven Algorithmus, ob tatsächlich ein Vorkommen vorliegt.
Zeit: $O(\log m)$, Arbeit: $O(m)$ je Intervall, $O(n \cdot m/p)$ gesamt.
Damit Arbeit von $O(n)$ im nichtperiodischen Fall ($p > m/2$).
Auch im periodischen Fall kann man Zeit $O(\log m)$ und Arbeit $O(n)$ erreichen.

Algorithmus 1.17 Paralleler Duell-Algorithmus: 1. Phase

Eingabe: Wörter P,T, $Per(P) = p$, Intervall $[k, k + p - 1]$
Ausgabe: Kandidat c_k für Vorkommen von P im Intervall $[k, k + p - 1]$

(1) **pardo for** $r \leftarrow 0$ **to** $p - 1$
(2) $\quad c_{k+r} \leftarrow k + r$;
(3) **for** $i \leftarrow 1$ **to** $\lceil \log_2 p \rceil$
(4) $\quad$ **pardo for** $j \leftarrow 0$ **to** $\lfloor \frac{p}{2^i} \rfloor$
(5) $\quad\quad c_{k+j\cdot 2^i} \leftarrow \textsc{Duell}(c_{k+j\cdot 2^i}, c_{k+j\cdot 2^i+2^{i-1}})$
(6) **return** c_k;

Algorithmus 1.18 Paralleler Duell-Algorithmus: 2. Phase

Eingabe: Wörter P,T, $|P| = m$, Position c

Ausgabe: **true** , falls $T[c \ldots c+m-1] = P$; **false** sonst.

(1) **pardo for** $r \leftarrow 0$ **to** $m-1$

(2) $\quad B_{c,r} \leftarrow (T[c+r] = P[1+r])$;

(3) **for** $i \leftarrow 1$ **to** $\lceil \log_2 m \rceil$

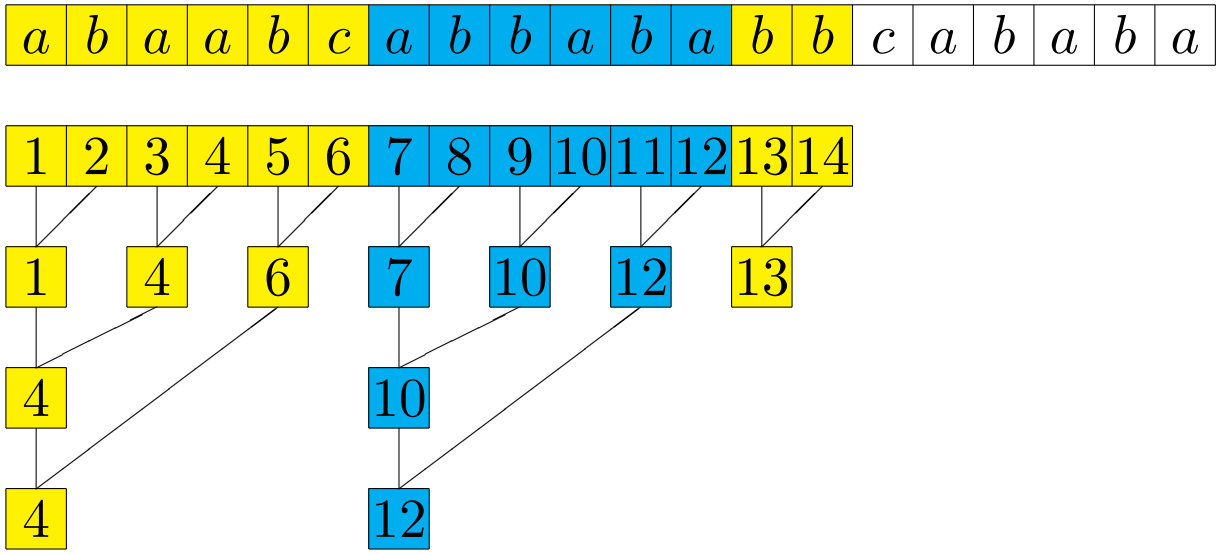
(4) $\quad$ **pardo for** $j \leftarrow 0$ **to** $\lfloor \frac{m}{2^i} \rfloor$

(5) $\quad\quad B_{c,j \cdot 2^i} \leftarrow B_{c,j \cdot 2^i} \wedge B_{c,j \cdot 2^i + 2^{i-1}}$

(6) **return** $B_{c,0}$;

Paralleler Algorithmus von Vishkin – Beispiel

Es seien $P = abcabba$, $T = abaabcabbababbcababa$. Die Zeugentafel von P ist $Z(P) = (1, 2, 5, 4, 5, 7, 7)$; die kürzeste Periode von P ist 6. Für die Duell-Phase ergibt sich folgender Ablauf.



Aufteilung in Intervalle

Initialisierung: $c_k \leftarrow k$

Duelle: 1. Runde

Duelle: 2. Runde

Duelle: 3. Runde

1.8 Karp-Rabin-Algorithmus

- benutzt Hash-Funktion $Hash : \Sigma^m \to \mathbb{N}$
- bestimmt mit einem Aufwand $\Theta(n)$ Positionen mit korrektem $Hash$-Wert
- Überprüfung der Kandidaten mit Aufwand $O(mn)$ oder
  Implementierung als probabilistischer Algorithmus
- Verallgemeinerung auf zweidimensionale Bilder leicht möglich

Wahl der Hash-Funktion

- Anforderungen an die Hash-Funktion
  - gute Differenzierung (unterschiedliche Werte für ähnliche Wörter)
  - $Hash(T_{k+1})$ in konstanter Zeit aus $Hash(T_k)$ berechenbar, wobei $T_j = T[j \ldots j+m-1]$

- O.B.d.A.: $\Sigma = \{0, 1, \ldots, \sigma - 1\} \rightarrow$ Wörter als Zahlen interpretiert.
  $w = x_1 x_2 \ldots x_{m-1} x_m \leftrightarrow H(w) = \sum_{i=1}^{m} x_i \cdot \sigma^{m-i}$

- Als Hash-Funktion eignet sich

$$Hash_q(w) = H(w) \, mod \, q,$$

  wobei q eine Primzahl mit $O(\log n)$ Bits ist.

Für $a, b \in \Sigma$ und $\alpha \in \Sigma^{m-1}$ gilt $H(\alpha b) = H(a\alpha) \cdot \sigma - a \cdot \sigma^m + b$ und folglich

$Hash_q(\alpha b) = (Hash_q(a\alpha) \cdot \sigma - a \cdot (\sigma^m \, mod \, q) + b) \, mod \, q.$

Also: $Hash_q(T_{k+1}) = (Hash_q(T_k) \cdot \sigma - T[k] \cdot s + T[k+m]) \, mod \, q$ mit $s = \sigma^m \, mod \, q$.

$Hash_q(T_{k+1})$ kann mit konstanten Aufwand aus $Hash_q(T_k)$ berechnet werden.

Analog können s, $Hash_q(P)$ und $Hash_q(T_1)$ mit einem Aufwand von $\Theta(m)$ berechnet werden.

Algorithmus 1.19 Karp-Rabin-Algorithmus

Eingabe: Wörter P, T über $\Sigma = \{0, 1, \ldots, \sigma - 1\}$ mit $|P| = m, |T| = n$

Ausgabe: Menge C möglicher Vorkommen von P in T

(1) $C \leftarrow \emptyset$;
(2) Wähle eine Primzahl q;
(3) $s \leftarrow \sigma^m \, mod \, q$; $h \leftarrow Hash_q(P)$; $H \leftarrow Hash_q(T[1 \ldots m])$;
(4) **if** $H = h$ **then** $C \leftarrow C \cup \{1\}$;
(5) **for** $k \leftarrow 1$ **to** $n - m$
(6) $\quad H \leftarrow (H \cdot \sigma - T[k] \cdot s + T[k + m]) \, mod \, q$;
(7) $\quad$ **if** $H = h$ **then** $C \leftarrow C \cup \{k + 1\}$;
(8) **return** C;

Karp-Rabin-Algorithmus: Beispiel

Es seien $\Sigma = \{0, 1, 2, 3\}$, $P = 30303$, $T = 10130303123231011203$.
Für $q = 11$ bzw. $q = 17$ erhalten wir folgenden Ablauf.
(Die Werte T_k sind an die Stelle $k + m - 1 = k + 4$ geschrieben.)

q	$Hash_q(P)$	$\sigma^m \bmod q$
11	5	1
17	3	4

q	1	0	1	3	0	3	0	3	1	2	3	2	3	1	0	1	1	2	0	3
11					9	5	9	5	7	8	10	9	3	1	2	6	1	3	0	3
17					12	13	1	3	1	6	15	11	1	1	13	7	4	6	3	15

Es bleibt nach diesen beiden Läufen nur die Stelle 4 als mögliches Vorkommen.

Test der Kandidaten – Varianten der Implementierung

1. Teste für jeden Kandidaten, ob ein Vorkommen von P vorliegt.
   Deterministischer Algorithmus, Laufzeit $O(mn)$ im schlechtesten, $O(n)$ im mittleren Fall.

2. Wiederhole den Algorithmus k-mal mit unterschiedlichen Primzahlen q.
   Gib alle Kandidaten aus, die bei jedem Durchlauf gefunden wurden.
   Monte-Carlo-Algorithmus, Laufzeit $O(kn)$.
   Fehlerwahrscheinlichkeit (für jede Eingabe):
   $O(1/n^k)$ bei zufälliger Wahl von q aus dem Intervall $[1, \ldots, mn^2]$

3. Teste, ob die Kandidatenliste C einen Fehler enthält (mit Aufwand $O(n)$).
   Wiederhole den Algorithmus solange, bis C keinen Fehler mehr enthält.
   Las-Vegas-Algorithmus, erwartete Laufzeit (für jede Eingabe): $O(n)$

Test der Kandidatenliste im Las-Vegas-Algorithmus

- Teile die geordnete Kandidatenliste C in geordnete Teillisten $C_1, C_2, \ldots, C_r$ auf.
  - Der Abstand zwischen benachbarten Elementen einer Liste C_i ist höchstens $m/2$.
  - Der Abstand vom letzten Element von C_i zum ersten Element von C_{i+1} ist größer als $m/2$.

- Ist C korrekt, so haben benachbarte Elemente in allen Listen die kleinste Periode von P als Abstand.

  Prüfe, ob in allen Listen alle Nachbarn den gleichen Abstand d haben.
  Wenn nicht, so liegt ein Fehler vor und brich ab.

  Vergleiche in den Teillisten die ersten zwei Kandidaten komplett mit dem Text und die restlichen Kandidaten mit den letzten d Stellen, bis ein Fehler gefunden ist oder alle Kandidaten geprüft sind.

1.9 Untere Schranken für die exakte Suche

$|P| = m$, $|T| = n$, $P, T \in \Sigma^*$ mit $|\Sigma| = \sigma$

- Komplexität des schlechtesten Falls: $\Theta(n)$
  Für $P = a^m$ und $T = a^n$ muss jedes Zeichen von T betrachtet werden.
  Optimale Komplexität wird erreicht z.B. durch KMP-Algorithmus.

- Komplexität des besten Falls: $\Theta(n/m)$
  In jedem Intervall der Länge m von T muss mindestens ein Zeichen betrachtet werden.
  Optimale Komplexität wird erreicht z.B. durch Boyer-Moore-Algorithmus für $P = a^m$, $T = b^n$.

- Komplexität im mittleren Falls: $\Theta(n \cdot \log_\sigma m/m)$
  Beweis von Yao (1979) wird im folgenden skizziert.
  Optimale Komplexität wird erreicht z.B. durch BDM-Algorithmus.

Beweis von Yao I

Definition. Es sei $\bullet \notin \Sigma$ ein Symbol. $S_n(l)$ ist die Menge aller Wörter in $(\Sigma \cup \{\bullet\})^n$ mit genau l Symbolen aus Σ. Für $z \in S_n(l)$ ist $I(z)$ die Menge aller Wörter $w \in \Sigma^n$ mit $w[i] = z[i]$ für alle $1 \leq i \leq n$ mit $z[i] \in \Sigma$.

Interpretation von $S_n(l)$: $z \in S_n(l)$ ist ein Wort aus Σ^n, von dem l Zeichen bekannt sind.

Definition. Ein Wort $z \in S_n(l)$ heißt Zertifikat für $\alpha \in \Sigma^*$, falls α für alle Wörter aus $I(z)$ an den gleichen Stellen vorkommt.

Beispiel. Sei $\Sigma = \{a, b\}$. Das Wort $abba$ besitzt das Zertifikat $z = \bullet\bullet a \bullet a \bullet\bullet$. Für jeden Text $T \in I(z)$ kann durch zwei Vergleiche an den Stellen Stellen 3 und 5 festgestellt werden, dass $abba$ nicht in T vorkommt.
Dagegen besitzt $abba$ kein Zertifikat aus $S_7(1)$, d.h. für jeden Text der Länge 7 sind mindestens 2 Vergleiche notwendig, um alle Vorkommen von $abba$ zu bestimmen.

Beweis von Yao II

Lemma. Es sei $\alpha \in \Sigma^*$. Gibt es einen Text $T \in \Sigma^n$, für den man alle Vorkommen von α mit höchstens l Vergleichen finden kann, so gibt es für α ein Zertifikat aus $S_n(l)$.

Lemma. Es seien $n \geq 2m-1$ und $\alpha \in \Sigma^m$. Besitzt α ein Zertifikat aus $S_n(l)$, so gibt es für α ein Zertifikat aus $S_{2m-1}(\lfloor l/q \rfloor)$ mit $q = \lfloor \frac{n}{2m-1} \rfloor$).

Lemma. (Counting Lemma)
Ein Wort $z \in S_{2m-1}(l)$ mit $l < m$ ist für höchstens $(1-1/\sigma^l)^{m/l^2} \cdot \sigma^m$ Wörter aus Σ^m ein Zertifikat.

Lemma. Es sei $m > 2^{44}$ und $l = \frac{1}{2}\log_\sigma m$. Die Anzahl $Z_{2m-1}(l)$ der Wörter aus Σ^m, die ein Zertifikat aus $S_{2m-1}(l)$ besitzen, ist kleiner als $e^{-1}\sigma^m$.

Satz. Für die Suche nach einem Wort der Länge $m > 2^{44}$ in einem Text der Länge $n \geq 2m-1$ benötigt man im Mittel mindestens $\frac{1-1/e}{2} \cdot \frac{n \log_\sigma m}{m}$ Vergleiche.