Algorithm Engineering

„Zeichenkettensuche“

Stefan Edelkamp

Motivation

Schnelle Textsuche ist angesicht der wachsenden Menge von elektronisch verfügbaren Quellen eine Schlüsseltechnologie im 3. Jahrtausend

1. Das Internet verdankt sein Siegeszug den sogenannten *Suchmaschinen*, die Schlagwort-Anfragen mit eine Liste dazu korrespondierender Seiten beantworten.
2. In der Biologie treten Zeichenketten z.B. in der Beschreibung des Genoms auf, da dieses gut durch eine lange Sequenz von Aminosäuren-Basenpaaren beschrieben werden kann.

   Demnach gehören effiziente Datenstrukturen zur Lokalisierung von Zeichenketten zum Erfolgsrezept in der *Genomanalyse*.

Wir studieren das Problem, ein Muster $M[1..m] \in \Sigma^m$ in einem Text $T[1..n] \in \Sigma^n$ zu finden, wobei $n > m$ ist.

Übersicht

1. Brute-Force
2. Rabin-Karp
3. DFA-Matching
4. RA-Matching
5. Knuth-Morris-Pratt
6. KMP-DFA Konvertierung
7. Boyer-Moore
8. Aho-Corasick
9. Wildcard-Matching

Brute-Force

Der Algorithmus prüft für jede Textposition $1 \leq j \leq n - m + 1$, ob

$$M[1..m] \text{ mit } T[j..j + m - 1]$$

übereinstimmt.

Das Verfahren hat offensichtlich eine Laufzeit von $O(nm)$.

Best-, Average- und Worst-Case

Beispiel 1 *Das Verfahren Brute-Force für* $M = \texttt{aab}$ *und* $T = \texttt{acaabc}$*:*

```
acaabc    acaabc    acaabc    acaabc
|#         #          |||        |#
aab       aab        aab       aab
```

Das Verfahren für größere Alphabete in der Praxis schnell und liegt in der Regel nahe bei der optimalen Laufzeit von $O(n + m)$.

Dennoch, im ungünstigen Fall, z.B. bei der Suche von $M = 0^{m-1}1$ in $T = 0^{n-1}1$,

ist die Laufzeit nur durch $O(nm)$ zu beschränken.

Rabin-Karp Algorithmus

Die Idee bei diesem Algorithmus ist, zunächst das Muster M mittels einer Hashfunktion h auf eine Zahl $h(M)$ abzubilden, so dass diese in eine Speicherzelle passt und in $O(1)$ zu verarbeiten ist.

Bei dem Vergleich wird für $1 \leq j \leq n - m + 1$ geprüft, ob

$$h(M) = h(T[j..j + n - 1])$$

gilt.

Durch die Möglichkeit von Kollisionen ist dies zwar kein hinreichendes, doch immerhin notwendiges Kriterium für die Übereinstimmung von M und $T[j..j+n-1]$.

Eine Zeichen-für-Zeichen Vergleich wird nur dann durchgeführt, wenn $h(M) = h(T[j..j+m-1])$ ist.

Bei der Bestimmung $h(T[j+1..j+m])$ nutzt man das Ergebnis $h(T[j..j+m-1])$ aus, um nur konstanten Berechnungsaufwand je Index j zu investieren.

Dies geht wie folgt.

Sei q eine nicht zu kleine Primzahl mit $q > m$. Wir nehmen an, dass Zahlen der Größe $q \cdot |\Sigma|$ in eine Speicherzelle passen.

Um die Notation zu vereinfachen, identifizieren wir Buchstaben in Σ durch ihre Ordnungsnummer.

In der Praxis

Beispiel 2 *Sei* $\Sigma = \{0, \ldots, 9\}$ *und* $q = 13$. *Suche Muster* $M = 31415$ *im Text* $T = 2359023141526739921$

```
 2  3  5  9  0  2  3  1  4  1  5  2  6  7  3  9  9  2  1
                    \___     ___/     \___     ___/
                        \   /   mod 13    \   /
                         \ /               \ /
        8  9  3 11  0  1  7  8  4  5 10 11  7  9 11
```

Inkrementelle Berechnung

$$
\begin{aligned}
14152 &\equiv (31415 - 3 \cdot 10000) \cdot 10 + 2 \pmod{13} \\
&\equiv (7 - 3 \cdot 3) \cdot 10 + 2 \pmod{13} \\
&\equiv 8 \pmod{13}
\end{aligned}
$$

Algorithmus

1. $p \leftarrow t \leftarrow 0; u \leftarrow |\Sigma|^{m-1} \bmod q$
2. Für alle $1 \leq i \leq m : p \leftarrow (|\Sigma| \cdot p + M[i]) \bmod q$
3. Für alle $1 \leq i \leq m : t \leftarrow (|\Sigma| \cdot p + T[i]) \bmod q$
4. Für alle $1 \leq j \leq n - m + 1$
    1. if $p = t$ then return *check* $(M, T[j..j + m - 1])$
    2. if $j \leq n - m$ then $t \leftarrow ((t - T[j] \cdot u) \cdot |\Sigma| + T[j + m]) \bmod q$

Korrektheit

Satz 1 *Zu Beginn des j-ten Schleifendurchlaufs in Schritt 4 des Verfahrens von Rabin-Karp ist der Wert*

$$p = \left(\sum_{i=1}^{m} M[i]|\Sigma|^{m-i} \right) \bmod q$$

und der Wert

$$t_j = \left(\sum_{i=j}^{m+j-1} T[i]|\Sigma|^{m-i+j-1} \right) \bmod q.$$

Induktives Argument

Beweis: Es ist

$$t_1 = \left(\sum_{i=1}^{m} T[i]|\Sigma|^{m-i} \right) \bmod q$$

und induktiv

$$\begin{aligned} t_j &= \left(\left(t_{j-1} - T[j-1] \cdot u \right) \cdot |\Sigma| + T[j+m-1] \right) \bmod q \\ &= \left(\left(\left(\sum_{i=j-1}^{m+j-2} T[i]|\Sigma|^{m-i+j-2} \right) - T[j-1] \cdot u \right) \cdot |\Sigma| + T[j+m-1] \right) \bmod q \\ &= \left(\sum_{i=j}^{m+j-1} T[i]|\Sigma|^{m-i+j-1} \right) \bmod q. \end{aligned}$$

Laufzeit

Die Berechnung aller Hashadressen hat somit eine Laufzeit von $O(n+m)$.

Im schlechtesten Fall hat der Rabin-Karp Algorithmus dennoch die Komplexität $O(nm)$, wie das Beispiel $M = 0^m$ in $T = 0^n$ zeigt.

Nimmt man jedoch an, dass das Muster in dem Text nur konstant oft vorkommt, so erreicht man bei K als mittlere Anzahl von Kollisionen eine mittlere Komplexität von $O(m+n+mK)$.

Für die meisten Texte und Muster liegt die Anzahl der Kollisionen im Rahmen der statistischen Trefferhäufigkeit.

Das heißt, es sind weniger als n/m Kollisionen zu erwarten und die mittlere Laufzeit beträgt $O(m+n)$.

Endliche (Stringerkennungs-) Automaten

Endliche Automaten sind 5-Tupel $A = (Q, q_0, F, \Sigma, \delta)$ mit Zustandsmenge Q, Startzustand q_0, Zielzustandsmenge F, Alphabet Σ und Übergangsfunktion δ.

Ein Stringerkennungsautomat $A(M)$ für das Muster M ist definiert durch:

1. Das Alphabet Σ ist so gewählt, dass $M \in \Sigma^*$ ist,
2. $Q = \{0, \ldots, m\}$, $q_0 = 0$ und $F = \{m\}$.
3. $\delta(q, a) = \max\{k \mid M[1..k]$ ist Suffix von $M[1..q]a\}$.

Maximaler Präfix-Suffix String

Beispiel 3 *Sei* $M = \texttt{ababaca}$, *dann ist* δ *definiert durch*

	0	1	2	3	4	5	6	7
a	1	1	3	1	5	1	7	1
b	0	2	0	4	0	4	0	2
c	0	0	0	0	0	6	0	0

Es ist $\delta(3, b) = 4$, *da* $\texttt{abab}$ *der maximale Präfix* $M[1..k]$ *von* $\texttt{ababaca}$ *ist, der Suffix von* $M[1..3]b = \texttt{abab}$ *ist.*

Die effiziente Berechnung von δ *in Zeit* $O(|\Sigma| \cdot m) = O(m)$ *durch den Algorithmus von Knuth-Morris-Pratt wird im nächsten Kapitel behandelt.*

Beispiellauf

Sei nun $T =$ `abababacaba`*, dann durchläuft der Automat die Zustände:*

i	-	1	2	3	4	5	6	7	8	9	10	11
T[i]	-	a	b	a	b	a	b	a	c	a	b	a
Q[i]	0	1	2	3	4	5	4	5	6	7	2	3

Somit wird ein Auftreten von M *in* T *gefunden, dass an Position 9 endet.*

Bei gegebenen Stringerkennungsautomat P liegt die Texterkennungzeit in $O(n)$.

Reguläre Ausdrücke

Reguläre Ausdrücke sind Namen zur Bezeichnung von Mengen von Worten über einem bestimmten Alphabet.

Ist α ein solcher Ausdruck, so bezeichnet $L(\alpha)$ die Menge von Worten.

Simultan-induktive Definition

1. Für jedes $a \in \Sigma \cup \{\epsilon\}$ ist a ein regulärer Ausdruck und $L(a) = \{a\}$.
2. Sind α und β reguläre Ausdrücke, so auch $\alpha \cup \beta$, $\alpha\beta$, α^* und
   - $L(\alpha \cup \beta) = L(\alpha) \cup L(\beta)$
   - $L(\alpha\beta) = L(\alpha) \circ L(\beta)$
   - $L(\alpha^*) = (L(\alpha))^*$

Äquivalenz Reg. Ausdruck, NFA

Satz 2 *Zu jedem regulären Ausdruck über dem Alphabet Σ kann man einen NFA $N = (S, \Sigma, s_0, F, \delta)$ angeben mit $T(N) = L(\alpha)$.*

Beweis: Durch Induktion über α.

1. $\alpha = a, a \in \Sigma \cup \{\epsilon\}$. Dann ist $L(\alpha) = T(N)$ für den wie folgt definierten NFA: $\rightarrow \bigcirc \xrightarrow{a} \circledcirc$
2. Hat man bereits NFA's N_α und N_β mit $L(\alpha) = T(N_\alpha)$ und $L(\beta) = T(N_\beta)$, so kann man NFA's konstruieren, die $L((\alpha \cup \beta))$, $L((\alpha\beta))$ und $L(\alpha^*)$ akzeptieren, weil die Klasse der regulären Sprachen gegenüber $\cup$, $\circ$ und $*$ abgeschlossen ist.

Der entstehende NFA ist nicht minimal.

Erste Erkenntnisse

Beobachtung 1 *Für den NFA N gilt:*

- *die Anzahl der Zustände von N is kleiner-gleich $2 \cdot |\alpha|$*
- *Jeder Knoten hat ein maximalen Fan-in oder Fan-out von 2.*
- *N läßt sich in Zeit $O(|\alpha|)$ berechnen.*

Das String-Matching Problem läßt sich auf das Wortproblem reduzieren, da

$$\exists M : M \text{ Teilwort von } T \text{ und } M \in L(\alpha) \iff T \in L(\Sigma^* \alpha \Sigma^*)$$

Zum Matching könnte man den zum NFA korrespondierenden DFA konstruieren.

Die Potenzmengen-Konstruktion ist allerdings zu ineffizient.

Matching mit NFAs

Satz 3 *Das Wortproblem für reguläre Sprachen läßt sich in Zeit $O(|\alpha| \cdot |T|)$ lösen.*

Beweis: Mit dynamischem Programmieren wird die Folge $S_0, \ldots, S_n$ ($n = |T|$) der Zustandsmengen berechnet, die die erreichbaren Zustände beim on-line Lesen von $T[1..n]$ enthalten.

Es gilt dann: $T \in L(\alpha) \iff t_\alpha \in S_n$, mit t_α Endzustand. von N_α

Die Mengen S_i werden 2-stufig bestimmt.

Stufe 1 bestimmt ein Zwischenergebnis Z_i unter der Ausblendung von ϵ-Transitionen.

Stufe 2 schließt Z_i mit ϵ-Transitionen ab.

Formal:

1. $Z_0 = \{s_\alpha\}$
2. Für $1 \leq i \leq n$: $Z_i = \cup_{q \in S_{i-1}} \delta(q, T[i])$
3. Für $0 \leq i \leq n$: $S_i = \{q' \mid \exists q \in Z_i : q \rightarrow_\epsilon^* q'\}$

Pseudo-Code

Der folgende Algorithmus realisiert die Berechnung mit Hilfe einer Queue Q und führt für $0 \leq i \leq n$ die folgenden Schritte aus:

1. if $(i = 0)$ then $S_i \leftarrow \{s_\alpha\}$ else $S_i = \cup_{q \in S_{i-1}} \delta(q, T[i])$
2. mark each $q \in S_i$ *considered*
3. mark each $q \notin S \setminus S_i$ *unconsidered*
4. $Q \leftarrow S_i$
5. while $(Q \neq \emptyset)$
    1. $q \leftarrow Q.dequeue()$
    2. for each *unconsidered* $u \in \delta(q, \epsilon)$: $Q.enqueue(u)$

Laufzeit

Zeile 1 realisiert die Berechnung von Z_i (als Zwischenwert von S_i).

Da $|S_{i-1}| \leq |N_\alpha| \leq 2|\alpha|$ und der Ausgangsgrad kleiner-gleich 2 kostet Stufe 1 nur $O(|\alpha|)$ Schritte.

Zeilen 2 bis 4 bereiten die Berechnung von S_i aus Z_i in $O(|\alpha|)$ Schritten vor.

Da jeder Zustand nur einmal in Q aufgenommen ist ist Q nach $O(|\alpha|)$ Iterationen der inneren Schleife leer.

Ein Durchlauf der Schleife kostet wegen des beschränkten Ausgangsgrades nur konstant viele Schritte.

Insgesamt ergeben sich somit $O(|\alpha|n)$ Schritte.

Knuth-Morris-Pratt

Das Verfahren von Knuth, Morris und Pratt sucht nach einem Mustervorkommen von M in Text T.

Das Verfahren nutzt die Definition einer Fehlerfunktion f, einem Selbstähnlichkeitsmaß im Muster, das wie folgt definiert ist

$$f(j) = \max\{1 \leq k < j \mid M[1..k] = M[j-k+1..j]\},$$

wobei $1 \leq j \leq m$ gilt.

Das heißt, $f(j)$ ist die Länge des maximialen Präfixes von $M[1..j]$ der gleichzeitig auch (echter) Suffix von $M[1..j]$ ist.

„Wenn hier einer Anna nass macht“

Beispiel 4 *Für das Wort* `ananas` *gilt*

$f(1) = 0,$

$f(2) = 0,$

$f(3) = 1,$

$f(4) = 2,$

$f(5) = 3,$ *und*

$f(6) = 0.$

Berechnung der Fehlerfunktion

Die Berechnung von f geschieht durch den folgenden Algorithmus:

1. $f(1) \leftarrow j \leftarrow 0$;
2. for $2 \leq i \leq m$
   1. while $(j > 0)$ and $(M[j+1] \neq M[i]) : j \leftarrow f(j)$
   2. if $(M[j+1] = M[i]) : j \leftarrow j + 1$
   3. $f(i) \leftarrow j$

Textsuche

Erstaunlicherweise ist das Verfahren zur Suche von M in T bei gegebener Funktion f von der Algorithmik sehr ähnlich:

1. $j \leftarrow 0$;
2. for $1 \leq i \leq n$
    1. while $(j > 0)$ and $(M[j+1] \neq T[i]) : j \leftarrow f(j)$
    2. if $(M[j+1] = T[i]) : j \leftarrow j + 1$
    3. if $(j = m)$ return $i - m + 1, \ldots, i$

„Wer ‚a‘ sagt muss auch ‚b‘ sagen …“

Beispiel 5 *Sei* $M = \texttt{ababaca}$ *und* $T = \texttt{bacbababaabcbab}$. *Dann ist* $f(5) = 3$ *und* $f(3) = 1$.

```
bacbababaababaca        bacbababaababaca        bacbababaababaca
     |||||#                   |||#                     |#
     ababaca                 ababaca                  ababaca
```

Die Anfangsposition des Musters ist $s = 5$ *mit* $q = 5$ *Übereinstimmungen.*

Beim sechsten Zeichen tritt ein Mismatch auf.

Die neue Anfangsposition des Musters ist $s' = s + (q - f(q)) = 5 + 5 - 3 = 7$.

Beim abermaligen Mismatch an Position 10, wird das Muster auf Startposition $s'' = s' + (q' - f(q')) = 7 + 3 - 1 = 9$ *verschoben.*

Laufzeiten

Satz 4 *Der erste Algorithmus benötigt Zeit* $O(m)$ *und der zweite Algorithmus* $O(n)$.

Beweis: In 2(a) kann der Wert von j nicht stärker herabgesetzt werden kann, als er in 2(b) heraufgesetzt wurde.

Demnach ist die Laufzeit der inneren while-Schleife amortisiert konstant.

Die Laufzeit des gesamten Verfahrens ist somit linear in der Summe aus Anfragestring und Textlänge.

Da zumindest Text und Muster gelesen werden müssen, ist der Algorithms somit worst-case optimal.

Endliche Automaten und KMP

Für eine gegebene Fehlerfunktion f für das Muster M läßt sich die Übergangsfunktion δ im Automat $A(M)$ wie folgt berechnen:

1. Für alle $q \in \{0, \ldots, m-1\}$ ist $\delta(q, M[l+1]) \leftarrow q+1$.
2. Für alle $a \in \Sigma \setminus \{M[1]\}$ ist $\delta(0, a) \leftarrow 0$.
3. Für alle $q \in \{0, \ldots, m-1\}$ und für alle $a \in \Sigma \setminus \{M[l+1]\}$ ist $\delta(q, a) \leftarrow \delta(f(q), a)$.

Offensichtlich läuft der Algorithmus in Zeit $O(m \cdot |\Sigma|) = O(m)$.

Boyer-Moore

Das Verfahren von Boyer-Moore beruht auf der Grundidee, den Fragestring von hinten an die Textstelle anzulegen, um, bei Nichtübereinstimmung, den Fragestring eine größere Distanz zu verschieben.

Dies ist insbesondere bei großen Alphabeten und kurzen Mustern von Vorteil.

Die Verschiebung des Musters entspricht beim Vergleich des Buchstabens $T[i] = a$ dem Maximum folgenden zwei Strategien:

„Schlechte Charaktere“

1. Bei der sogenannten *Bad-Character Heuristik* vermerkt man den am weitest rechts auftretenen auftretenen gleichlautenden Buchstaben a links zur Position i, d.h. man tabelliert

$$B(i, a) = \max\{1 \leq j < i \mid M[j] = a\}.$$

Sollte das Maximum nicht existieren, so gilt $B(i, a) = 0$.
Aus der Tabelle kann die Verschiebungslänge $i - B(i, a)$ für das *schlechte* Zeichen a entnommen werden.
Die Tabelle kann in Zeit $O(m \cdot |\Sigma|) = O(m)$ berechnet werden.

„Happy End“

2. Bei der sogennanten *Good-Suffix Heuristik* sucht man nach dem am weitest rechts stehenden Suffix links von der Mismatchposition, der mit dem tatsächlichen Pattern übereinstimmt, d.h.

$$G(i) = \max\{1 \leq j < m - i \mid M[j..j + m - i - 1] = M[i + 1..m]\}.$$

Diese Tabelle kann in Zeit $O(m)$ berechnet werden.

Zusammenspiel der beiden Heuristiken

Beispiel 6 *Betrachte*

```
written notice that  written notice that  written notice that
           #||                  |                    ||
  reminiscence             reminiscence        reminiscence
```

Das bad character *ist* i *und der gute Suffix ist* ce.

Die Heuristiken sagen $G(10) = 3$ *und* $B(10, i) = 6$.

Demnach ist die Gesamtverschiebung gleich $\max\{3, 10 - 6\} = 4$.

Best-, Average, Worst-Case

Im ungünstigen Fall der Suche von 10^{m-1} in 10^{n-1} ist jedoch die Laufzeit wie beim naiven Algorithmus $O(nm)$.

In günstigen Fall jedoch wird das Muster bei jedem Vergleich um seine volle Länge verschoben.

Dies entspricht einer Laufzeit von $O(n/m)$.

Bei einer mittleren Verschiebungen der Größenordnung $m - o(m)$ läuft das Verfahren incl. der Initialisierungszeit $O(m)$ in einer Laufzeit von $O(m + n/(m - o(m))) = O(n/m)$.

Aho & Corasick

Für eine Menge von Zeichenketten $W_1, \ldots, W_k$ mit $W_i \in \Sigma^*$ für $1 \leq i \leq k$, soll in Zeit $O(D)$ mit $D = \sum_{i=1}^{k} |W_i|$ eine Anfragestruktur konstruiert werden, die Teilwortanfragen in linearer Zeit ermöglichen.

Der Algorithmus von Aho und Corasick verwaltet einen annotierten Vielwegsuchbaum (Σ-Baum, engl. *Trie*) als Repräsentation der gespeicherten Zeichenketten.

Es ist leicht, den reinen Mehrwegsuchbaum T für die Worte $W_1, \ldots, W_k$ in Zeit $O(|\Sigma| \cdot D) = O(D)$ mit $D = \sum_{i=1}^{k} |W_i|$ zu konstruieren.

Fehlerfunktion

Jedem Zustand im Trie wird ein Fehlerfunktionswert zugeordnet, der es erlaubt, von einem Zweig des Vielwegsuchbaumes zum anderen zu springen.

Definition 1 *Sei t ein Knoten im Trie und $v(t)$ der durch t beschriebene String.*

Der Fehlerfunktionswert $f(t)$ ist definiert als das längste (echte) Endstück von $v(t)$, das gleichzeitig Anfangsstück eines im Wörterbuch gespeicherten Strings ist.

Diese Festlegung erweitert in natürlicher Art und Weise die Definition der Fehlerfunktion in dem Algorithmus von Knuth, Morris und Pratt.

Endlich ein Bild

Beispiel 7 *In Abbildung 1 wird ein Trie mit Fehlerfunktion für die Zeichenketten* 0110011, 1001101 *und* 10101 *dargestellt.*

Der berechnete Fehlerfunktionswert von Knoten 4 ist Knoten 9.

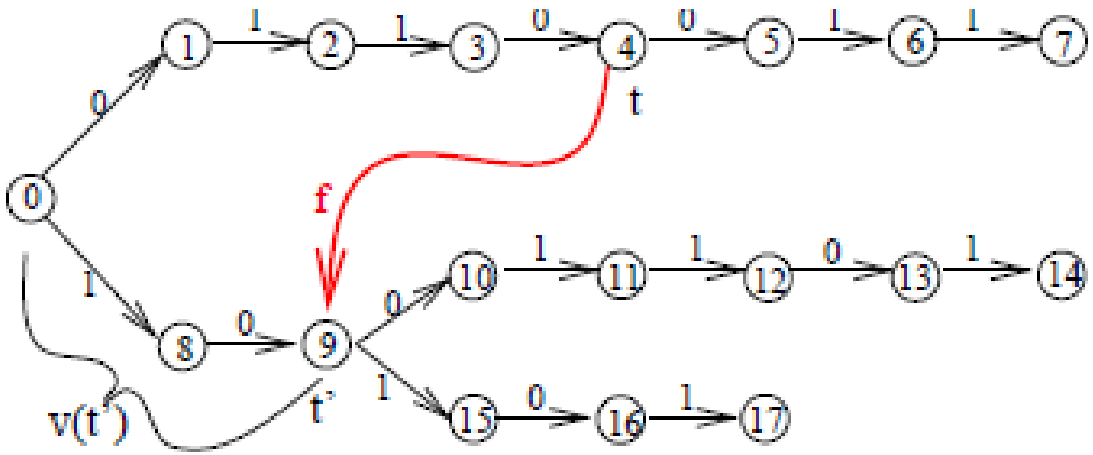

Abbildung 1

Pseudo-Code

Der Algorithmus nutzt eine Queue Q, um die Breitensuchdurchlaufordnung zu garantieren.

In einem ersten Schritt werden die Fehlerfunktionswerte der Wurzel und der Knoten in Level 1 bestimmt.

Letztere werden in Q eingefügt.

Dann wird, solange Q nicht leer ist, das erste Element *top* bestimmt.

Sei a der betreffende Buchstabe von *top* zum Nachfoger *child*.

Um $f(child)$ zu bestimmen, wird derjenige Knoten in der Sequenz $f(top)$, $f^2(top)$, $f^3(top)$, usw. bestimmt, von dem aus ein Übergang mit a möglich ist.

Laufzeit

Zum Glück läßt sich durch ein analoges Amortisierungsargument wie im Algorithmus von Knuth, Morris und Pratt die Laufzeit der Fehlerfunktionskonstruktion auf $O(|T| \cdot |\Sigma|) = O(D)$ beschränken.

Satz 5 *Der Aho-Corasick Ansatz benötigt Zeit $O(D)$ zur Konstruktion der Fehlerfunktion.*

Beweis: Dazu sei $|v(f(u))|$ die Länge des längsten echten Suffix von $v(u)$, der auch Präfix im Wörterbuch ist.

Falls u' und u Knoten auf dem Pfad von der Wurzel zu einem Knoten i in T ist und u' der Vorgänger von u ist, gilt $|v(f(u))| \leq |v(f(u'))| + 1$.

Wähle einen Musterstring W_i mit Pfad P_i aus.

Damit ist Gesamzuwachs in der Länge von der Fehlerfunktionswerte für W_i gleich $\sum_{u \in P_i} |v(f(u))| - |v(f(u'))| \leq |W_i|$.

Auf der anderen Seite schrumpft $|v(f(u))|$ bei jedem Fehlerfunktionsverweis um mindestens eins, wird aber nie negativ.

Damit ist die Gesamtabnahme der Länge der Fehlerfunktionsstrings für u aus P_i maximal $|W_i|$.

Konstruktion eines Automatens

Der Skelettautomat von A ist T selbst.

Die Übergänge von $t \in T$ für die nicht spezifizierten Zeichen a lassen sich aus der Fehlerfunktion leicht bestimmen, indem die entsprechenden Übergange von $f(t')$ via a ausgeführt werden.

Schon wieder ein Bild

Beispiel 8 *In dem Beispiel von Abbildung 1 ist dementsprechend der Automatübergang* $A(4,0)$ *gleich* 5 *und* $A(4,1)$ *gleich* 15.

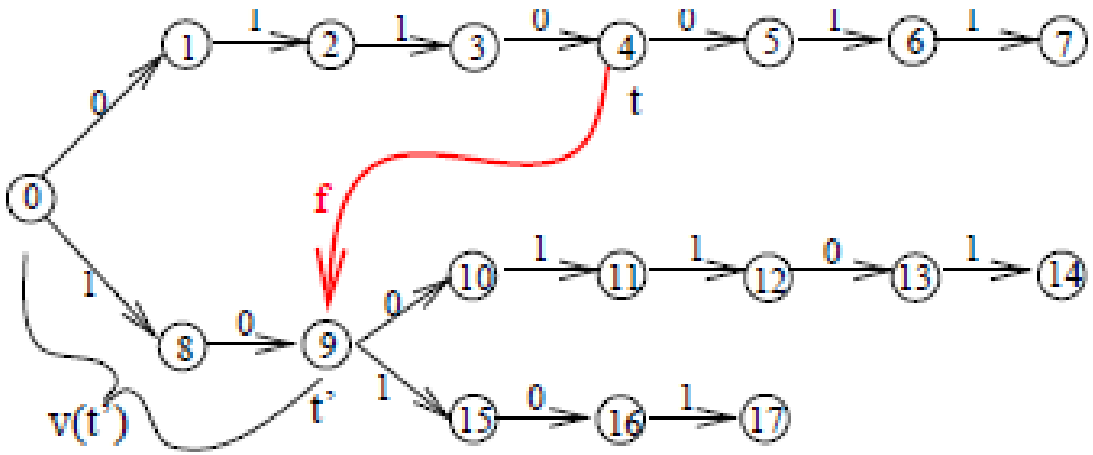

Abbildung 1

Laufzeit Aho-Corasick

Die Berechnung von A läuft offensichtlich in Zeit $O(|T| \cdot |\Sigma|) = O(D)$.

Dies ist optimal, wenn man bedenkt, das man alle Strings lesen muss.

Auch ist die Speicherausbeute mit $O(D)$ optimal.

Die Suche eines bzw. aller Teilstrings in einem Text der Länge n mit dem Aho-Corasick Automaten ist nun in Linearzeit $O(n)$ möglich.

Satz 6 *Das frühste Auftreten (oder alle Auftreten) von k Musterstrings der Gesamtlänge D in einem Textstring der Länge n kann in $O(n + D)$ Schritten lokalisiert werden.*

Wild-Cards

Definition 2 *Ein* Platzhalter $?$ *(engl. wild card) ist ein Buchstabe, der zu einer Übereinstimmung mit jedem anderen Zeichen* $a \in \Sigma$ *führt.*

Beispiel 9 *Das Muster* $M = \texttt{b??b?a??}$ *tritt in* $T = \texttt{baabcabcabb}$ *zweimal (an den Startpositionen 1 und 4) auf.*

Satz 7 *Alle Übereinstimmungen eines Musters mit Platzhaltern können in Zeit* $O(n + km)$ *gefunden werden, wobei* k *die Kardinalität der Multimenge aller maximalen Teilstrings von* M *ist, die keine Wildcards enthalten.*

Beweis durch Pseudo-Code

Beweis: Der folgende Algorithmus löst das Problem.

1. Sei C der Vektor der Länge $|T|$, auf Null initialisiert.
2. Sei $P = \{P_1, \ldots, P_k\}$ die Multimenge aller maximalen Teilstrings von M, die keine Wildcards enthalten. Seien desweiteren $l_1, l_2, \ldots, l_k$ die Startpositionen dieser Teilstrings in P.
3. Nutze den Algorithmus Aho-Corasick, um alle Startpositionen in T für alle P_i in P zu bestimmen. Für jede Startposition j erhöhe $C[j - l_i + 1]$ um eins.
4. Durchlaufe C: Genau dann, wenn $C[p] = k$ beginnt eine Übereinstimmung von M in T an Stelle $T[p]$.

Die Laufzeit ist $O(n + km)$. Der Algorithmus ist korrekt, da nach Aho-Corasick alle maximalen Teilstrings im Text auftauchen und gemäß des Startpositionenvektors C zueinander passen müssen.

Letztes Beispiel

Beispiel 10 *Sei* $M = \texttt{b??b?a??}$, $T = \texttt{baabcabcabb}$, *dann ist* $P = [b, b, a]$.

Startpositionen in P *sind* $l = \{1, 3, 6\}$, *Startpositionen* (j) *in* T *sind* $P[1] = P[2] = \{1, 4, 7, 10, 11\}$, *und* $P[3] = \{2, 3, 6, 9\}$, *d.h. die Werte von* $j - l_i + 1$ *sind*

für $P[1]$ *gleich 1, 4, 7, 10, 11*

für $P[2]$ *gleich (-2), 1, 4, 7, 8*

für $P[3]$ *gleich (-3, -2), 1, 4.*

Damit gilt $C_n = \texttt{30030021011}$

und zwei Übereinstimmungen an den Positionen 1 und 4 wurden gefunden.

Probleme Aho-Corasick

Leider wirft eine dynamische Aktualisierung von f bei einzufügenden oder zu löschenden Zeichenketten Probleme auf, da die Fehlerfunktion Schichtweise berechnet wird.

Das Inverse der Fehlerfunktion besitzt eine Baumstruktur, da für jeden Knoten außer der Wurzel der Fehlerfunktionswert auf einen Knoten geringerer Triefe im Trie verweist.

Die Aktualisierungen der Fehlerfunktion werden durch das Einfügen neuer Strings dadurch zwar prinzipiell ohne komplette Neuberechnung möglich, doch ist es nicht gelungen, aus der Idee einen Algorithmus mit Laufzeit $O(|W|)$ zu formen.

Die benötigten strukturellen Selbstähnlichkeitsanforderungen sind demnach hoch.