2 String- und Pattern-Matching

2.1 Definitionen und Notationen

Wir führen zunächst eine Reihe notwendiger Begriffe ein:

- Ein *Alphabet* Σ ist eine endliche Menge von Symbolen (Buchstaben). $|\Sigma|$ bezeichnet die Kardinalität von Σ.
- Ein *Wort* (String, Zeichenkette) s über einem Alphabet Σ ist eine endliche Folge von Symbolen aus Σ. $|s|$ bezeichnet die *Länge* von s, d.h. für $s = s_1 s_2 \dots s_n \in \Sigma^n$ ist $|s| = n$.
- ϵ bezeichnet das *leere Wort*, d.h. $|\epsilon| = 0$.
- Für ein Alphabet Σ und ein $n \geq 0$ bezeichnet Σ^n die Menge aller Worte über Σ der Länge n. Σ^0 ist definiert als $\{\epsilon\}$.
- Für ein Alphabet Σ bezeichnet $\Sigma^* = \bigcup_{n \geq 0} \Sigma^n$ die Menge aller Worte über Σ und $\Sigma^+ = \bigcup_{n \geq 1} \Sigma^n$ die Menge aller Worte außer ϵ.
- Wenn x und y Wörter sind, dann bezeichnet xy die *Konkatenation* von x und y.

Definition 2.1 *Es sei $s = s_1 \dots s_n$ ein Wort über Σ.*

1. *Ein Wort $s' \in \Sigma^*$ der Länge m heißt* Teilwort *von s, falls es ein $i \geq 1$ gibt mit $s_i s_{i+1} \dots s_{i+m-1} = s'$.*
2. *Ein Wort $s' \in \Sigma^*$ der Länge m heißt* Präfix *von s, falls $s' = s_1 s_2 \dots s_m$.*
3. *Ein Wort $s' \in \Sigma^*$ der Länge m heißt* Suffix *von s, falls $s' = s_{n-m+1} s_{n-m+2} \dots s_n$.*

Das exakte String-Matching Problem gibt es in zwei Varianten. Gegeben die Worte s (wie Suchwort) und t (wie Text),

1. bestimme, ob s ein Teilwort von t ist, oder
2. bestimme die Menge aller Positionen, an denen s in t auftritt.

Wir werden hauptsächlich an der Variante (2) interessiert sein. Für den Rest des Kapitels werden wir annehmen, dass $|s| \leq |t|$ ist, da sonst s trivialerweise nicht in t vorkommen kann.

2.2 Ein naiver Algorithmus

Betrachte den Algorithmus in Abb. 1. Die while-Schleife des Algorithmus hat offensichtlich einen Zeitaufwand von $O(m)$, so dass wir den folgenden Satz erhalten:

Satz 2.2 *Der naive Algorithmus löst die exakte Suche in Zeit $O(n \cdot m)$ und (zusätzlichem) Platz $O(1)$.*

Der Platz ist zwar optimal, aber es stellt sich natürlich die Frage, ob wir eine bessere Zeit erreichen können. Hierfür gibt es verschiedene Ansätze. Wir betrachten zunächst einen randomisierten, auf Hashing basierenden Ansatz und dann zwei deterministische Ansätze, die durch ein geeignetes Preprocessing des Suchwortes die Suche deutlich beschleunigen können.

```
Algorithmus SimpleMatching:
    FOR i := 1 TO n − m + 1 DO
        j := 1
        WHILE j ≤ m AND s[j] = t[i + j − 1]
            j := j + 1
            IF j > m THEN gib i aus
```

Figure 1: Ein naiver Algorithmus für die exakte Suche.

2.3 Karp-Rabin Algorithmus

Betrachte ein beliebiges Alphabet Σ der Größe $q-1$. Sei $f : \Sigma^* \to U$ die Arithmetisierung der Worte über $\Sigma = \{c_1, \ldots, c_{q-1}\}$ mit der Eigenschaft, dass

- $f(\epsilon) = 0$,
- $f(c_i) = i$ für alle $i \in \{1, \ldots, q-1\}$ und
- $f(w) = \sum_{i=1}^{n} f(w_i) \cdot q^i$ für alle Worte $w = w_{n-1} \ldots w_1 w_0$ mit $|w| \geq 2$.

Mit dieser Regel gibt es für jedes $x \in \mathbb{N}$ höchstens ein Wort $w \in \Sigma^*$ mit $f(w) = x$.

Für ein exaktes Sting-Matching Problem mit Suchwort s und Text t wählen wir zunächst eine Primzahl $p > |\Sigma| + 1$ und berechnen für die Hashfunktion $h(x) = x \bmod p$ den Wert $h(f(s))$. In der naiven Variante des Rabin-Karp Algorithmus berechnen wir auch $h(f(t_1 \ldots t_m)), h(f(t_2 \ldots t_{m+1}))$, $\ldots, h(f(t_{n-m+1} \ldots t_n))$ und würden jeden dieser Hashwerte mit $h(f(s))$ vergleichen. Für jedes i mit $h(f(s)) = h(f(t_i \ldots t_{i+m-1}))$ überprüfen wir zusätzlich, ob $s = t_i \ldots t_{i+m-1}$ ist, und wenn ja, dann geben wir i aus. Diese extra Überprüfung ist notwendig, da $h(f(s)) = h(f(t_i \ldots t_{i+m-1}))$ sein kann, obwohl $s \neq (t_i \ldots t_{i+m-1})$ ist. Allerdings wäre dann der Karp-Rabin Algorithmus nicht effizienter als der naive Algorithmus oben. Um eine deutliche Laufzeitverbesserung zu erzielen, führen wir eine wesentlich effizientere Berechnung der Hashwerte $y_i = h(f(t_i \ldots t_{i+m-1}))$ nach dem bekannten Horner-Schema durch. Dazu zunächst ein Beispiel.

Seien Σ, s und t so gewählt, dass $|\Sigma| = 9$, $f(s) = 15926$ und $f(t) = 3141592653589793$ ist. Sei $q = 10$ und $h(x) = x \bmod 97$. So gilt für die ersten Teilfolgen der Länge 5 in t:

- $y_1 = h(31415) = 84$
- $y_2 = h(14159) = 94$
- $y_3 = h(41592) = 76$

Um y_{i+1} aus y_i berechnen zu können, stellen wir zunächst fest, dass $10000 \bmod 97 = 9$ ist. Daraus folgt:

$$\begin{aligned} 14159 &= (31415 - 30000) \cdot 10 + 9 \\ &= (84 - 3 \cdot 9) \cdot 10 + 9 \pmod{97} \\ &= 579 = 94 \pmod{97} \end{aligned}$$

und

$$\begin{aligned} 41592 &= (14159 - 10000) \cdot 10 + 2 \\ &= (94 - 1 \cdot 9) + 2 \pmod{97} \\ &= 76 \pmod{97} \end{aligned}$$

Allgemein gilt für $h(x) = x \bmod p$ und $d = q^{|s|-1} \bmod p$, dass

$$y_{i+1} = (y_i - f(t_i) \cdot d) \cdot q + f(t_{i+m}) \pmod{p}$$

für alle $i \in \{1, \ldots, n-m\}$, was nur einen konstanten Zeitaufwand erfordert. Damit ergibt sich der Algorithmus in Abbildung 2.

```
Algorithmus Karp-Rabin:
    // wähle ein geeignetes p (wird unten genauer spezifiziert)
    q := |Σ| + 1; m := |s|; n := |t|;
    x := 0;   // für f(s) mod p
    y := 0;   // für f(t_1 ... t_m) mod p
    d := 1;   // für q^{|s|-1} mod p
    FOR i := 1 TO m - 1 DO
        d := q · d mod p;
    FOR i := 1 TO m DO
        x := q · x + f(s[i]) mod p;
        y := q · y + f(t[i]) mod p;
    FOR i := 1 TO n - m + 1 DO
        IF x = y THEN
            IF s = (t_i ... t_{i+m-1}) THEN gib i aus
        IF i ≤ n - m THEN
        y := (y - f(t[i]) · d) · q + f(t[i + m]) mod p
```

Figure 2: Der Karp-Rabin Algorithmus.

Obwohl die Berechnung der Hashwerte nun effizient ist, kann es bei einer schlechten Wahl von s, t und p zu sehr vielen Situationen kommen, in denen $h(f(s)) = h(f(t_i \ldots t_{i+m-1}))$ ist, obwohl $s \neq (t_i \ldots t_{i+m-1})$ ist, d.h. wir haben ein *falsches Matching*. Das Ziel ist es also, p genügend klein zu wählen, dass die Arithmetik effizient ist, aber auch genügend groß, dass die Wahrscheinlichkeit eines falschen Matchings genügend klein ist. Hierzu müssen wir einige Eigenschaften von Primzahlen ausnützen. Einige dieser Eigenschaften geben wir ohne Beweis an.

Definition 2.3 *Für eine natürliche Zahl u sei $\pi(u)$ die Anzahl der Primzahlen kleiner gleich u.*

Lemma 2.4 (Primzahltheorem) *Für $u \geq 29$ gilt $0,922 \frac{u}{\ln u} \leq \pi(u) \leq 1,105 \frac{u}{\ln u}$.*

Lemma 2.5 *Für $u \geq 29$ ist das Produkt aller Primzahlen, die kleiner gleich u sind, größer als 2^u.*

Korollar 2.6 *Falls $u \geq 29$ und x eine Zahl kleiner gleich 2^u ist, dann hat x weniger als $\pi(u)$ verschiedene Primteiler.*

Beweis. Angenommen, x habe $k > \pi(u)$ verschiedene Primteiler $q_1, q_2, \ldots, q_k$. Dann ist $2^u \geq x \geq q_1 \cdot q_2 \cdot \ldots \cdot q_k$. Aber $q_1 \cdot q_2 \cdot \ldots \cdot q_k$ ist mindestens so groß wie das Produkt der ersten k Primzahlen, was größer ist als das Produkt der ersten $\pi(u)$ Primzahlen (da ja $k > \pi(u)$). Das führt aber aufgrund Lemma 2.5 zu einem Widerspruch. $\square$

Damit sind wir in der Lage, die folgende Aussage zu beweisen.

Lemma 2.7 *Seien s und t Wörter mit $n \cdot m \geq 29$, wobei $m = |s|$ und $n = |t|$ ($|x|$ sei hier die Länge der Binärkodierung von x). Sei P eine beliebige natürliche Zahl. Falls p eine zufällige Primzahl kleiner gleich P ist, dann ist die Wahrscheinlichkeit eines falschen Matchings zwischen s und einer Position in t höchstens $\pi(n \cdot m)/\pi(P)$.*

Beweis. Sei R die Menge der Positionen in t, in denen s *nicht* beginnt. Für jedes $i \in R$ gilt also $f(s) \neq f(t_i \ldots t_{i+m-1})$. Betrachte nun das Produkt $\Pi_{i \in R}|f(s) - f(t_i \ldots t_{i+m-1})|$. Dieses Produkt kann höchstens $2^{n \cdot m}$ sein, da für jedes i gilt $|f(s) - f(t_i \ldots t_{i+m-1})| \leq 2^m$ (unter der Annahme, dass s und t binär kodiert sind). Damit ergibt sich aus Korollar 2.6, dass $\Pi_{i \in R}|f(s) - f(t_i \ldots t_{i+m-1})|$ höchstens $\pi(n \cdot m)$ verschiedene Primteiler hat.

Betrachte nun ein falsches Matching zwischen s und t an Position i in t. Das bedeutet, dass $f(s) \bmod p = f(t_i \ldots t_{i+m-1}) \bmod p$ ist und daher p $|f(s) - f(t_i \ldots t_{i+m-1})|$ teilt. Also teilt p auch $\Pi_{i \in R}|f(s) - f(t_i \ldots t_{i+m-1})|$, d.h. p ist einer der Primteiler dieses Produkts. Falls p ein falsches Matching zulässt, dann muss p einer von höchstens $\pi(n \cdot m)$ vielen Primteilern sein. Da aber p zufällig aus der Menge $\pi(P)$ ausgewählt wird, ist die Wahrscheinlichkeit, dass p ein falsches Matching zulässt, höchstens $\pi(n \cdot m)/\pi(P)$. $\square$

Mit dieser Aussage können wir die erwartete Laufzeit des Karp-Rabin Algorithmus abschätzen. n und m sind wie oben gewählt.

Satz 2.8 *Seien s und t Wörter mit $n \cdot m \geq 29$ und $P = n \cdot m^2$. Falls s k-mal in t vorkommt, dann ist die erwartete Laufzeit des Karp-Rabin Algorithmus $O(n + k \cdot m)$.*

Beweis. Sei R die Menge der Positionen in t, in denen s *nicht* beginnt. Für jede Position $i \in R$ definieren wir eine binäre Zufallsvariable X_i, die 1 ist genau dann, wenn ein falsches Matching an der Position i stattfindet. Von Lemma 2.7 und Lemma 2.4 wissen wir, dass

$$\mathrm{E}[X_i] \leq \frac{\pi(n \cdot m)}{\pi(P)} \leq \frac{1,105nm/\ln(nm)}{0,922nm^2/\ln(nm^2)} \leq \frac{1,2}{m} \cdot \frac{\ln n + 2\ln m}{\ln n + \ln m} \leq \frac{2}{m}$$

Daraus folgt aufgrund der Linearität des Erwartungswerts, dass

$$\mathrm{E}[X] = \sum_{i \in R} \mathrm{E}[X_i] \leq \frac{2|R|}{m}$$

Da für jedes falsche Matching eine Zeit von $O(m)$ gebraucht wird und sonst nur eine Zeit von $O(1)$, ergibt sich eine erwartete Gesamtlaufzeit von $O(|R|) = O(n)$ für die Positionen in R. Für die k

Positionen, in denen s in t vorkommt, haben wir eine Laufzeit von insgesamt $O(k \cdot m)$. Daraus ergibt sich der Satz. □

Sind wir also nur daran interessiert, das erste Vorkommen von s in t (also die erste Variante des exakten Sting-Matching Problems) zu finden, haben wir eine erwartete Laufzeit von $O(n)$.

2.4 Knuth-Morris-Pratt Algorithmus

Das Prinzip hinter dem Knuth-Morris-Pratt Algorithmus ist denkbar einfach. Angenommen, wir benutzen den naiven Vergleichsalgorithmus und wir haben bis Position i ein Matching zwischen s und t festgestellt, aber in Position $i+1$ sind s und t verschieden. Dann können wir zur minimalen Position $j > 1$ über gehen, so dass $(s_1 \dots s_{i-j+2}) = (t_j \dots t_{i+1})$ ist, und machen dort weiter mit der Suche eines Matchings. Positionen zwischen t_1 und t_{j-1} brauchen wir nicht mehr zu betrachten, da für diese nach der Definition von j schon kein Matching mehr zu erzielen ist. Ein schnelles Preprocessing für diese Regel zu entwerfen ist allerdings nicht ganz einfach. Daher werden wir uns damit zufrieden geben, zur minimalen Position j über zu gehen, so dass $(s_1 \dots s_{i-j+1}) = (s_j \dots s_i) = (t_j \dots t_i)$ ist. Kann man diese Position j in konstanter Zeit bestimmen, so werden wir sehen, dass man dann t in $O(n)$ Zeit nach allen Vorkommen von s durchlaufen kann. Um die Positionen j effektiv zu bestimmen, benötigen wir zunächst ein geeignetes Preprocessing von s.

Ziel des Preprocessings von s ist es, für jede Position i in s das minimale $j > 1$ zu finden, so dass $(s_1 \dots s_{i-j+1}) = (s_j \dots s_i)$ ist. Falls es kein solches j gibt, setzen wir es auf $i+1$. Wir bezeichnen dieses minimale j mit d_i. Dann gilt das folgende Lemma. (m sei die Länge von s.)

Lemma 2.9 *Für jedes $i \in \{1, \dots, m-1\}$ gilt $d_i \leq d_{i+1}$.*

Beweis. Betrachte ein beliebiges i und sei d_i wie oben definiert. Dann gibt es kein $1 < j < d_i$ mit $(s_1 \dots s_{i-j+1}) = (s_j \dots s_i)$ und damit auch kein $1 < j < d_i$ mit $(s_1 \dots s_{i-j+2}) = (s_j \dots s_{i+1})$, woraus folgt, dass $d_i \leq d_{i+1}$ sein muss. □

Lemma 2.10 *Für jedes $i \in \{1, \dots, m-1\}$ mit $(s_1 \dots s_{(i+1)-d_i+1}) \neq (s_{d_i} \dots s_{i+1})$ gilt $d_{i+1} \geq d_i + d_{i+1-d_i} - 1$, und $d_{i+1} = d_i + d_{i+1-d_i} - 1$ genau dann, wenn $s_{(i+1)-d_i+1} = s_{(i+1)-(d_i+d_{(i+1)-d_i}-1)+1}$.*

Beweis. Wenn $(s_1 \dots s_{(i+1)-d_i+1}) \neq (s_{d_i} \dots s_{i+1})$, so muss nach der Definition von d_i gelten, dass $(s_1 \dots s_{i-d_i+1}) = (s_{d_i} \dots s_i)$ und $s_{(i+1)-d_i+1} \neq s_{i+1}$. Die Frage ist dann, um wieviel wir d_{i+1} größer als d_i wählen müssen, bis wir das nächstemal die Möglichkeit haben, dass $(s_1 \dots s_{(i+1)-d_{i+1}+1}) = (s_{d_{i+1}} \dots s_{i+1})$ ist. Da wir wissen, dass $(s_1 \dots s_{i-d_i+1}) = (s_{d_i} \dots s_i)$ ist, ist der nächste Kandidat dafür das minimale $j > 1$ mit $(s_1 \dots s_{(i-d_i+1)-j+1}) = (s_j \dots s_{i-d_i+1})$. Dieses j ist gleich d_{i-d_i+1}. Also muss $d_{i+1} \geq d_i + (d_{i+1-d_i} - 1)$ sein, und $d_{i+1} = d_i + (d_{i+1-d_i} - 1)$ wenn $s_{(i+1)-d_i+1} = s_k$ ist mit

$$k = [(i - d_i + 1) - d_{i-d_i+1} + 1] + 1 = (i+1) - (d_i + d_{(i+1)-d_i} - 1) + 1$$

□

Wir sind damit soweit, die d_i-Werte zu berechnen. Dazu betrachten wir Abb. 3. Die Korrektheit des Algorithmus ergibt sich aus den Lemmas oben. Die Laufzeit des Algorithmus ist wie folgt beschränkt.

Satz 2.11 *Die Laufzeit des KMP-Preprocessing ist $O(m)$.*

```
Algorithmus KMP-Preprocessing:
    d[0] := 2; d[1] := 2
    j := 2    // für Berechnung der d_i-Werte
    FOR i := 2 TO m DO
        WHILE j ≤ i AND s[i] ≠ s[i − j + 1] DO
            j := j + d[i − j] − 1
        d[i] := j
```

Figure 3: Der Algorithmus zur Berechnung der d_i-Werte.

Beweis. Da alle $d_i \geq 2$ sind, wird für jede erfüllte WHILE-Bedingung j erhöht. Da diese Bedingung nicht mehr erfüllt werden kann, sobald $j > m$ ist, kann die WHILE-Bedingung also höchstens m-mal erfüllt sein. Desweiteren wird die FOR-Schleife $(m-1)$-mal durchlaufen, so dass sich eine Gesamtlaufzeit von $O(m)$ ergibt. □

```
Knuth-Morris-Pratt Algorithmus:
    führe KMP-Preprocessing durch
    i := 1    // aktuelle Position in t
    j := 1    // aktuelle Anfangsposition von s in t
    WHILE i ≤ n DO
        IF j ≤ i AND t[i] ≠ s[i − j + 1] THEN
            j := j + d[i − j] − 1
        ELSE
            IF i − j + 1 = m THEN
                gib j aus
                j := j + d[m] − 1
            i := i + 1
```

Figure 4: Der Knuth-Morris-Pratt Algorithmus.

Als nächstes stellen wir den Knuth-Morris-Pratt Algorithmus in Abb. 4 vor. Die Korrektheit des Knuth-Morris-Pratt Algorithmus folgt aus der Korrektheit des KMP-Preprocessing. Es reicht also, die Laufzeit zu bestimmen.

Satz 2.12 *Die Laufzeit des Knuth-Morris-Pratt Algorithmus ist $O(n)$.*

Beweis. In jedem Schleifendurchlauf wird entweder i oder j (oder beide) erhöht. Da i nach oben hin durch n beschränkt ist und nur noch i erhöht wird, sobald $j > n$ ist, folgt der Satz. □

Man kann den Knuth-Morris-Pratt Algorithmus durch verschiedene Tricks weiter verbessern. So kann man in das Vorwärtsspringen einbeziehen, dass wir eigentlich zum minimalen j übergehen könnten mit $(s_1 \ldots s_{i-j+1}) = (s_j \ldots s_i)$ und $s_{i-j+2} \neq s_{i+1}$ (weil wir für s_{i+1} ein Mismatch in t hatten). Oder wir könnten darauf testen, ob ein Symbol t_i überhaupt in s vorkommt. Wenn nicht, kann man gleich j und i auf $i+1$ setzen.

2.5 Boyer-Moore Algorithmus

Der Boyer-Moore Algorithmus kann als eine verbesserte Variante eines naiven String-Matching Algorithmus angesehen werden, in dem s mit einer Teilfolge in t von *rechts nach links* verglichen wird. Siehe dazu auch Abb. 5.

```
Naiver Boyer-Moore Algorithmus:
    i := 1
    WHILE i ≤ n − m + 1 DO
        j := m
        WHILE j ≥ 1 AND s[j] = t[i + j − 1] DO
            j := j − 1
        IF j = 0 THEN gib j aus
        i := i + 1
```

Figure 5: Naives String-Matching von rechts nach links.

Der Boyer-Moore Algorithmus basiert auf der folgenden Überlegung:

- Tritt im naiven Boyer-Moore Algorithmus an Stelle j ein Mismatch auf und kommt $(s_{j+1} \dots s_m)$ nicht ein weiteres mal in s als Teilwort vor, so kann s gleich um m Stellen nach rechts verschoben werden.

- Vergleicht man dagegen von links nach rechts, kann s nach einem Mismatch an Position j nie um mehr als j Positionen nach rechts verschoben werden.

Kommt es nun im naiven Boyer-Moore Algorithmus an der Stelle j von s zu einem Mismatch, so gilt $(s_{j+1} \dots s_m) = (t_{i+j} \dots t_{i+m-1})$ und $s_j \neq t_{i+j-1}$. Wir können dann die folgende Suffix-Regel verwenden, um den naiven Boyer-Moore Algorithmus zu beschleunigen:

1. Es gibt ein $k > 0$ mit $s_{j-k} \neq s_j$ und $(s_{j-k+1} \dots s_{m-k}) = (s_{j+1} \dots s_m)$. In diesem Fall wählen wir das minimale k mit dieser Eigenschaft und können i um k erhöhen, ohne Gefahr zu laufen, ein Matching auszulassen.

2. Es gibt kein $k > 0$ mit der Eigenschaft oben. Dann suchen wir das minimale $k \in \{j, \dots, m\}$ mit $(s_1 \dots s_{m-k}) = (s_{k+1} \dots s_m)$. Wir können dann i um k erhöhen, ohne ein Matching auszulassen.

Für jedes $0 \leq j \leq m$ sei $D_j = k$ für das wie oben gewählte k. Haben wir eine korrekte Berechnung der D_j-Werte sichergestellt, so ist der verbesserte Boyer-Moore Algorithmus in Abb. 6 trivialerweise korrekt. Die Berechnung der D_j-Werte werden wir im nächsten Übungsblatt behandeln.

Wir müssen noch die Laufzeit der Boyer-Moor-Algorithmus bestimmen, was im nächsten Satz geschieht.

Satz 2.13 *Unter der Annahme, dass die D-Tabelle in $O(m)$ Zeit berechnet werden kann, ist die Laufzeit des Boyer-Moore Algorithmus, falls s nicht in t vorkommt, $O(n)$.*

Beweis. Wir unterteilen den Algorithmus in Runden von Vergleichs- und Verschiebungs-Operationen und verwenden die folgenden Parameter in dem Beweis:

```
Boyer-Moore Algorithmus:
    führe BM-Preprocessing durch
    i := 1
    WHILE i ≤ n − m + 1 DO
        j := m
        WHILE j ≥ 1 AND s[j] = t[i + j − 1] DO
            j := j − 1
        IF j = 0 THEN gib i aus
        i := i + D[j]
```

Figure 6: Schnelles String-Matching von rechts nach links.

- T_r: Teilstring von t in Runde r, für den $T_r = (s_{m-|T_r|+1} \dots s_m)$ gilt und für den links von T_r ein Mismatch mit $s_{m-|T_r|}$ stattgefunden hat.
- d_r: Anzahl der Positionen, um die die Variable i am Ende der Runde r nach rechts verschoben worden ist.
- v_r: Anzahl Positionen in T_r, die schon in vorigen Runden verglichen worden sind.
- v'_r: Anzahl Position in T_r, die noch nicht in vorigen Runden verglichen worden sind.

Damit ergibt sich eine Laufzeit des Algorithmus von $O(\sum_r (v_r + v'_r))$, und unser Ziel ist es zu zeigen, dass diese Summe gleich $O(n)$ ist.

Offensichtlich ist $\sum_r v'_r \leq n$. Um die v_r-Summe zu beschränken, werden wir zeigen, dass für alle r gilt $d_r \geq v_r/3$. Da $\sum_r d_r \leq n$ ist, würde damit folgen, dass $\sum_r v_r \leq 3n$ ist. Insgesamt hätten wir also $\sum_r (v_r + v'_r) \leq 4n$, und wir wären fertig. Um zu zeigen, dass $d_r \geq v_r/3$ ist, benötigen wir einige Definitionen und Lemmas. Im folgenden sei für ein Wort w das Worst w^i für ein $i \in \mathbb{N}$ das i-mal hintereinander geschriebene Wort w.

Lemma 2.14 *Seien γ und δ zwei nichtleere Worte mit $\gamma\delta = \delta\gamma$. Dann ist $\delta = \rho^i$ und $\gamma = \rho^j$ für ein Wort ρ und $i, j \in \mathbb{N}$.*

Beweis. Wir beweisen das Lemma durch vollständige Induktion über $|\delta| + |\gamma|$. Falls $|\delta| + |\gamma| = 2$, dann muss gelten, dass $\delta = \gamma = \rho$ und $i = j = 1$. Betrachte nun grßere Längen. Falls $|\delta| = |\gamma|$ ist, dann gilt wiederum $\delta = \gamma = \rho$ und $i = j = 1$. Wir nehmen also an, dass $|\delta| < |\gamma|$ ist. Da $\delta\gamma = \gamma\delta$ und $|\delta| < |\gamma|$, muss δ ein Präfix von γ sein, also $\gamma = \delta\delta'$ für ein Wort δ'. Wenn wir das in $\delta\gamma = \gamma\delta$ substituieren, erhalten wir $\delta\delta\delta' = \delta\delta'\delta$. Löschen wir die linke Kopie von δ, so ergibt sich daraus $\delta\delta' = \delta'\delta$. Allerdings ist $|\delta| + |\delta|' = |\gamma| < |\delta| + |\gamma|$, und so folgt aus der Induktionsannahme, dass es ein Wort ρ gibt mit $\delta = \rho^i$ und $\delta' = \rho^j$. Also ist $\gamma = \delta\delta' = \rho^k$ mit $k = i + j$. □

Definition 2.15 *Ein Wort α ist* semiperiodisch *mit* Periode *β, falls α aus einem nichtleeren Suffix von β gefolgt mit einem oder mehreren Kopien von β besteht. Ein Wort α heißt* periodisch *mit* Periode *β, falls α aus einer oder mehreren vollständigen Kopien von β besteht.*

Das Wort $bcabcabc$ ist zum Bespiel semiperiodisch mit Periode abc, aber es ist nicht periodisch. Das Wort $abcabc$ ist periodisch mit Periode abc. Ein Wort kann verschiedene Perioden haben. So hat $abababab$ die Periode $abab$ und die Periode ab. Eine alternative Definition zu semiperiodischen Worten ist die folgende:

Definition 2.16 *Ein Wort α ist* präfix-semiperiodisch *mit Periode γ, falls α aus einer oder mehreren Kopien von γ besteht gefolgt von einem Präfix von γ.*

Das nächste Lemma ist leicht einzusehen und zeigt, dass “semiperiodisch” und “präfix-semiperiodisch” tatsächlich das gleiche bedeuten.

Lemma 2.17 *Ein Wort α ist semiperiodisch mit Periode β genau dann, wenn es auch präfix-semiperiodisch mit einer Periode γ ist.*

Das Wort $abaabaabaab$ ist zum Beispiel semiperiodisch mit Periode aab und präfix-semiperiodisch mit Periode aba. Das folgende Lemma ist einfach zu zeigen.

Lemma 2.18 *Angenommen s kommt in t an den Positionen i und $i' > i$ vor, wobei $i' - i \leq \lfloor m/2 \rfloor$. Dann ist s semiperiodisch mit Periodelänge $i' - i$.*

Kommen wir also zurück zu dem Beweis für $d_r \geq v_r/3$. Falls $d_r \geq (|T_r| + 1)/3$ ist, dann ist $d_r \geq v_r/3$, selbst wenn alle Zeichen in T_r schon in vorigen Runden verglichen worden sind. Wir nehmen also im folgenden an, dass $d_r < (|T_r| + 1)/3$ oder $|T_r| + 1 > 3d_r$ ist.

Wir benötigen noch einige Notationen. Sei α das Suffix von s der Länge $|d_r|$ und β das kleinste Wort, so dass $\alpha = \beta^\ell$ für ein $\ell \in \mathbb{N}$. (Es kann sein, dass $\alpha = \beta$, also $\ell = 1$ ist.) Sei $S = (s_{m-|T_r|} \dots s_m)$ das Suffix von s der Länge $|T_r| + 1$, d.h. der Anteil von s, der in Runde r betrachtet worden ist.

Lemma 2.19 *Falls $|T_r| + 1 > 3d_r$ ist, dann sind T_r und S semiperiodisch mit Periode α und daher auch mit Periode β.*

Beweis. Angefangen vom rechten Ende von S, partitioniere S in Teilworte der Länge d_r, bis keine d_r Zeichen mehr übrigbleiben (siehe Abb. 7). Es gibt mindestens drei volle Teilworte, da $|S| = |T_r|+1 > 3d_r$ ist. Runde r endet damit, dass s um d_r Positionen nach rechts verschoben wird. Betrachte nun, wie S mit t vorher und nachher ausgelegt ist (Abb 7). Aus der Definition von d_r und α und der Suffix-Regel folgt, dass alle entfernten Teilworte Kopien von α sein müssen und die übrigbleibenden Zeichen ein Suffix von α sein müssen. S ist also semiperiodisch mit Periode α. □

Nach der Definition von T_r sind nun alle in Runde r verglichene Zeichen bis auf eines (das den Mismatch verursacht hat) in T_r enthalten, und ein Zeichen von T_r kann nur in den vorigen Runden betrachtet worden sein, in denen sich s mit T_r überschneidet. Um also v_r zu beschränken, müssen wir genau betrachten, wie sich s in vorigen Runden mit T_r überlappt hat.

Lemma 2.20 *Falls $|T_r| + 1 > 3d_r$ ist, dann gilt in jeder Runde $r' < r$, dass das rechte Ende von s nicht mit dem rechten Ende einer vollen Kopie β in T_r gleichauf sein kann.*

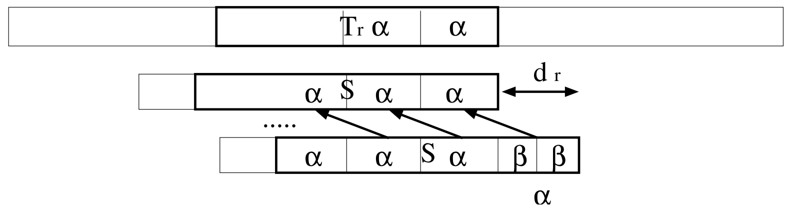


Figure 7: Partitionierung von S in α- und β-Worte.

Beweis. Aus Lemma 2.19 folgt, dass T_r semiperiodisch mit Periode β ist. In Runde $r' < r$ kann das rechte Ende von s nicht gleichauf sein mit dem rechten Ende von T_r, da $d_{r'} \geq 1$ sein muss. Nehmen wir also an, in Runde r' sei das rechte Ende von s gleichauf mit dem rechten Ende einer vollen Kopie von β in T_r. Wir nennen diese Kopie $\bar{\beta}$, und sein rechtes Ende sei $q|\beta|$ Zeichen vom rechten Ende von T_r entfernt für ein $q \geq 1$. Zunächst schauen wir uns an, wie Runde r' geendet haben muss, und dann werden wir das benutzen, um das Lemma zu zeigen.

Sei k' die Position in t zur Linken von T_r, und sei k die Position in s, die gleichauf mit $t_{k'}$ in Runde r' ist. Wir behaupten, dass in Runde r' der Vergleich von s und t Übereinstimmungen bis zum linken Ende von T_r hat, es dann aber zum Mismatch zwischen $t_{k'}$ und s_k kommt. Die Begründung dafür ist die folgende:

Die Worte S und T_r sind semiperiodisch mit Periode β, und in Runde r' ist das rechte Ende von s gleichauf mit dem rechten Ende eines β. Also stimmen s und t offensichtlich bis zum linken Ende von T_r überein. S ist nun semiperiodisch mit Periode β, und in Runde r' ist s exact $q|\beta|$ Positionen weg vom rechten Ende von T_r. Daraus ergibt sich, dass $S_1 = S_{1+|\beta|} = \ldots = S_{1+q|\beta|} = s_k$ ist. Aber in Runde r gibt es ein Mismatch beim Vergleich von $t_{k'}$ mit S_1, also $s_k = S_1 \neq t_{k'}$.

Jetzt betrachten wir die möglichen Verschiebungen von s in Runde r'. Wir werden zeigen, dass jede mögliche Verschiebung zu einem Widerspruch führt. Daher sind keine Verschiebungen möglich und damit auch nicht die angenommene Ausrichtung von s und t in Runde r', was das Lemma beweist.

Wir betrachten zwei Fälle für das rechte Ende von s nach Runde r'. (1) Das rechte Ende von s ist gleichauf mit dem rechten Ende einer Kopie von β (in T_r) oder (2) das rechte Ende von P ist im Inneren einer vollen Kopie von β (in T_r).

Fall 1: Wenn nach der Verschiebung in Runde r' das rechte Ende von P gleichauf mit dem rechten Ende einer vollen Kopie von β ist, dann ist das Zeichen in s gleichauf mit $t_{k'}$ an der Position $s_{k-r|\beta|}$ für ein $r \in \mathbb{N}$. Da aber S semiperiodisch mit Periode β ist, muss s_k gleich $s_{k-r|\beta|}$ sein, ein Widerspruch.

Fall 2: Angenommen, Runde r' verschiebt s so, dass dessen rechtes Ende mit dem Inneren einer vollen Kopie von β ausgerichtet ist. Das bedeutet, dass in dieser Ausrichtung das rechte Ende eines β Wortes in s gleichauf mit einem Zeichen im Inneren von $\bar{\beta}$ ist. Weiterhin folgt aus der Suffix-Regel, dass die Zeichen im verschobenen s unter $\bar{\beta}$ mit $\bar{\beta}$ übereinstimmen (siehe Abb. 8). Sei $\gamma\delta$ das Wort im verschobenen s, das gleichauf mit $\bar{\beta}$ in T_r ist, wobei γ das Wort bis zum Ende von β und δ der Rest ist. Da $\bar{\beta} = \beta$, ist γ ein Suffix von β, δ ein Präfix von β, und $|\gamma| + |\delta| = |\bar{\beta}| = |\beta|$, d.h. $\gamma\delta = \delta\gamma$. Nach Lemma 2.14 folgt damit, dass $\beta = \rho^t$ für ein $t > 1$ ist, ein Widerspruch zur Wahl von β.

Wir erreichen also in beiden Fällen einen Widerspruch, was das Lemma beweist. $\square$

Lemma 2.21 *Falls $|T_r| + 1 > 3d_r$ ist, dann kann s in Runde $r' < r$ mit T_r höchstens $|\beta| - 1$ Zeichen gemeinsam haben.*

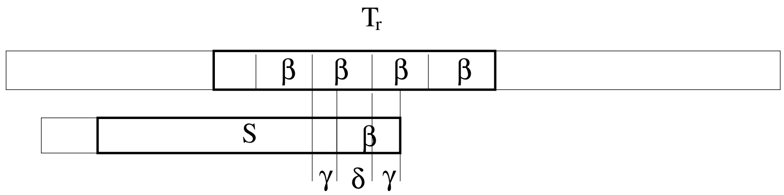


Figure 8: Partitionierung von β in γ und δ.

Beweis. Nach Lemma 2.20 kann s nicht gleichauf mit einem vollen β in T_r in Runde $r' < r$ sein. Falls s und T_r sich also um mindestens $|\beta|$ Zeichen überlappen, dann müsste das Suffix von s der Länge $|\beta|$ aus einem Suffix, γ, von β gefolgt von einem Präfix, δ, von β bestehen. Dann aber wäre $\beta = \gamma\delta = \delta\gamma$, was nach Lemma 2.14 einen Widerspruch zur Wahl von β bedeuten würde. $\square$

Dieses Lemma gilt übrigens selbst dann, wenn es in Runde r' kein Mismatch gegeben hat, also s in t gefunden worden ist. Wir nehmen hier nur an, in Runde r habe es ein Mismatch gegeben.

Lemma 2.22 *Falls $|T_r| + 1 > 3d_r$ ist, dann gilt in Runde $r' < r$, dass wenn das rechte Ende von s gleichauf mit einem Zeichen in T_r ist, es nur gleichauf mit einem der $|\beta| - 1$ Zeichen am linken Ende von T_r oder einem der $|\beta|$ Zeichen am rechten Ende von T_r sein kann.*

Beweis. Angenommen, das rechte Ende von s ist in Runde r' gleichauf mit einem Zeichen in T_r, das nicht zu den $|\beta|-1$ Zeichen am linken Ende oder den $|\beta|$ Zeichen am rechten Ende von T_r gehört. Dann ist nach Lemma 2.20 das rechte Ende von s im Inneren einer Kopie β' von β, und nach Lemma 2.21 würde ein Mismatch in Runde r' vorkommen, bevor das linke Ende von T_r erreicht ist. Angenommen, das Mismatch komme an Position k'' in t vor. Nach diesem Mismatch wird s nach rechts verschoben. Nach Lemma 2.20 kann $d_{r'}$ nicht das rechte Ende von s zu dem rechten Ende von β' verschieben, und wir werden zeigen, dass die Verschiebung das rechte Ende von s auch nicht jenseits dem rechten Ende von β' verschieben kann.

Wie erwähnt, verschiebt die Suffix-Regel s (wenn möglich) um den kleinstmöglichen Wert, so dass alle Zeichen von t, die in Runde r' übereingestimmt haben, mit dem verschobenen s übereinstimmen und die zwei Zeichen von s, die gleichauf mit $t_{k''}$ vor und nach der Verschiebung sind, verschieden sind. Wir behaupten, dass diese Bedingungen gelten, wenn das rechte Ende von s gleichauf mit dem rechten Ende von β' ist.

Betrachte diese Ausrichtung. Da S semiperiodisch ist mit Periode β ist, würden die Aursichtung von s und t zumindest bis zum linken Ende von T_r übereinstimmen, und so auch bei $t_{k''}$. Daher können die Zeichen in s, die gleichauf mit $t_{k''}$ vor und nach der Verschiebung sind, nicht übereinstimmen. Wenn das Ende von s also mit dem Ende von β' ausgerichtet ist, dann würden alle Zeichen in t, die in Runde r' übereingestimmt haben, wieder übereinstimmen, und die Zeichen in s gleichauf mit $t_{k''}$ vor und nach der Verschiebung wären verschieden. Daher würde die Suffix-Regel das rechte Ende von s nicht jenseits von β' verschieben.

Daraus folgt, dass wenn das rechte Ende von s gleichauf mit dem Inneren von β' in Runde r' ist, dann muss es auch gleichauf mit dem Inneren von β' in Runde $r' + 1$ sein. Da aber r' beliebig ist, würde die Verschiebung in Runde $r' + 1$ auch nicht das rechte Ende von s jenseits von β' verschieben. Wenn also das rechte Ende von s im Inneren von β' ist, bleibt es dort für immer. Das ist unmöglich, da in Runde $r > r'$ das rechte Ende von s mit dem rechten Ende von T_r übereinstimmt, was rechts von β' ist. Daher ist das rechte Ende von s nicht im Inneren von β', was das Lemma beweist. □

Lemma 2.23 *Falls s nicht in t vorkommt, dann ist $d_r \geq v_r/3$ für alle Runden r.*

Beweis. Das ist trivialerweise wahr, wenn $d_r \geq (|T_r|+1)/3$. Wir nehmen also an, dass $|T_r|+1 > 3d_r$ ist. Nach Lemma 2.22 ist in jeder Runde $r' < r$ das rechte Ende von s gleichauf mit entweder einem der $|\beta|-1$ Zeichen am linken Ende von T_r oder einem der $|\beta|$ Zeichen am rechten Ende von T_r. Nach Lemma 2.21 gilt, dass höchstens $|\beta|$ Vergleiche in Runde $r' < r$ gemacht werden. Daher sind die einzigen Zeichen in Runde r, die eventuell schon vor Runde r betrachtet worden sind, die $|\beta|-1$ Zeichen am linken Ende von T_r oder die $2|\beta|$ Zeichen am rechten Ende von T_r, oder das Zeichen zur Linken von T_r. Also gilt $v_r \leq 3|\beta| \leq 3d_r$ wenn $|T_r|+1 > 3d_r$. In beiden Fällen gilt also $d_r \geq v_r/3$. □

Lemma 2.23 ergibt den Satz. □

Der Boyer-Moore Algorithmus ist also im worst case asymptotisch genauso schnell wie der Knuth-Morris-Pratt Algorithmus, um das erste Vorkommen von s in t zu finden. In der Praxis kann er das allerdings wesentlich schneller, in der Größenordnung $O(n/m)$.

Falls alle Vorkommen eines Wortes gewünscht sind und s k-mal in t vorkommt, so verschlechtert sich die Laufzeit des Boyer-Moore Algorithmus auf $O(n + k \cdot m)$. In diesem Fall ist also der Knuth-Morris-Pratt Algorithmus zu empfehlen.

2.6 Aho-Corasick Algorithmus

Bis jetzt haben wir nur den Fall betrachtet, dass wir ein einziges Suchwort s haben. Was aber, wenn wir den Text t nach allen Vorkommen eines Suchworts in $S = \{s_1, \ldots, s_k\}$ durchsuchen möchten? Wir werden zeigen, wie wir den KMP-Algorithmus verallgemeinern können, um das effizient durchzuführen.

Im folgenden sei $m_i = |s_i|$ und $m = \sum_{i=1}^{k} m_i$. Die einfachste Möglichkeit zur Lösung unseres Textsuchproblems wäre, den KMP-Algorithmus parallel für alle Suchwörter laufen zu lassen. Das kann aber im worst case eine Laufzeit von $O(m + k \cdot n)$ haben. Um eine bessere Laufzeit hinzubekommen, nehmen wir zunächst einmal an, dass $k = 1$ ist und setzen $s = s_1$.

Anstatt einer Tabelle von d_j-Werten, können wir uns die Suche nach s in t im KMP-Algorithmus auch als das Durchlaufen eines endlichen Automaten mit Fehlerübergängen vorstellen. Betrachten wir dazu einmal das Suchwort $s = abaaba$. Dann ist die Tabelle der d_i-Werte wie folgt definiert:

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

Alternativ lässt sich das auch als Automat in Abb. 9 darstellen. Diesen Automaten nennt man auch Aho-Corasick oder AC-Automat.

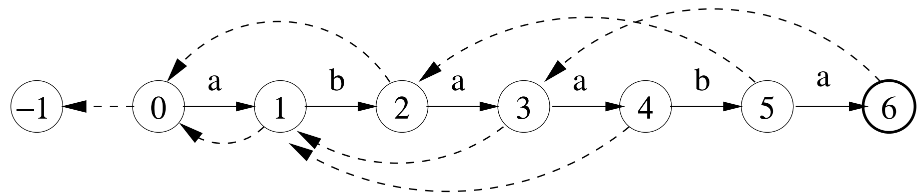


Figure 9: Der AC-Automat zu $s = abaaba$.

AC-Automat für ein Suchwort

Definition 2.24 *Der AC-Automat besteht aus den folgenden Komponenten:*

- *Einer endlichen Menge Q von Zuständen,*
- *einem endlichen Eingabealphabet $\Gamma = \Sigma \cup \{fail\}$,*
- *einer* Transitionsfunktion $\delta : Q \times \Gamma \to Q$
- *einem* initialen Zustand $q_0 \in Q$ *und*
- *einer Menge* akzeptierender Endzustände $F \subseteq Q$.

Der AC-Automat für ein Suchwort s ist nun wie folgt definiert mit $m = |s|$:

- $Q = \{-1, \ldots, m\}$, $q_0 = 0$ und $F = \{m\}$.
- Für alle $i \in \{0, \ldots, m-1\}$ gilt $\delta(i, s_i) = i+1$ und $\delta(i, fail) = i-d_i+1$. (D.h. $\delta(0, fail) = -1$, was einen Fehlschlag anzeigt.)

Der AC-Automat unterscheidet sich von einem gewöhnlichen endlichen Automaten dadurch, dass es extra $fail$-Übergänge gibt, bei denen kein Zeichen im Eingabetext vorgerückt wird, sondern der Automat solange bei dem aktuellen Zeichen stehenbleibt, bis ein gültiger Übergang mit einem Zeichen $c \in \Sigma$ gefunden worden ist.

Ist der AC-Automat in Zustand i, dann bedeutet das, dass bisher eine Übereinstimmung von $(s_1 \ldots s_{i-1})$ mit einem Teilwort in t erzielt werden konnte. Daraus ergibt sich sofort die Korrektheit der Regel $\delta(i, s_i) = i + 1$. Auch die Regel für $\delta(i, fail)$ ist korrekt, da wir im Falle eines Mismatches das kleinstmögliche j suchen, do dass $(s_j \ldots s_i) = (s_1 \ldots s_{i-j+1})$ ist, um die Suche nach s in t fortzusetzen, und dieses j gleich d_i ist. Der Algorithmus zur Berechnung eines AC-Automaten für ein Suchwort s arbeitet also wie in Abb. 10.

Satz 2.25 *Das AC-Preprocessing hat eine Laufzeit von $O(m)$.*

Beweis. Folgt sofort aus der Laufzeit des KMP-Preprocessing. □

Das Aho-Corasick Algorithmus arbeitet nun wie in Abb. 11. Da er eine äquivalente Darstellung des KMP-Algorithmus ist, gilt der folgende Satz.

Satz 2.26 *Der Aho-Corasick Algorithmus ist korrekt und läuft in $O(n)$ Zeit.*

```
Algorithmus AC-Preprocessing:
    d[0] := 2; d[1] := 2
    j := 2    // für Berechnung der d_i-Werte
    FOR i := 2 TO m DO
        WHILE j ≤ i AND s[i] ≠ s[i − j + 1] DO
            j := j + d[i − j] − 1
        d[i] := j
    // f[0..m]: für fehlerhafte Übergänge
    FOR i := 0 TO m DO f[i] := i − d[i] + 1
```

Figure 10: Berechnung des AC-Automaten für ein Suchwort.

```
Aho-Corasick Algorithmus:
    führe AC-Preprocessing durch
    j := 0    // aktuelle Position im Automaten
    FOR i := 1 TO n DO
        WHILE (j ≠ −1 AND t[i] ≠ s[j + 1]) DO
            j := f[j]
        IF j = −1 THEN j := 0 ELSE j := j + 1
        IF j = m THEN gib j aus
```

Figure 11: Aho-Corasick Algorithmus für ein Suchwort.

AC-Automat für mehrere Suchworte

Jetzt sind wir soweit, den AC-Automaten auf mehrere Suchworte zu verallgemeinern. Dieser ist wie folgt definiert.

- $Q = \{w \in \Sigma^* \mid w$ ist ein Präfix von einem $s \in S\} \cup \{fail\}$ und $q_0 = \epsilon$.
- $F = F_1 \cup F_2$, wobei
  - $F_1 = S$, d.h. die Menge aller Suchworte.
  - $F_2 = \{w \in Q \mid \exists s \in S: \ s$ ist ein echtes Suffix von $w\}$.
- Für alle $w \in Q$ und $a \in \Sigma$ gilt
  - $\delta(w, a) = wa$ genau dann, wenn $wa \in Q$ und
  - $\delta(w, fail) = w'$ für dasjenige $w' \in Q$, das den längsten echten Suffix von w darstellt. Für $w = \epsilon$ gilt $\delta(w, fail) = fail$.

Das Preprocessing erfolgt dann wie in Abb. 12 angegeben. Falls $|\Sigma|$ konstant ist, so gilt:

Satz 2.27 *Das allgemeine AC-Preprocessing berechnet einen AC-Automaten in $O(m)$ Zeit.*

Beweis. Folgt direkt aus dem Algorithmus. □

Das allgemeine AC-Algorithmus arbeitet nun wie in Abb. 13.

Satz 2.28 *Der allgemeine Aho-Corasick Algorithmus hat eine Laufzeit von* $O(n+m)$.

2.7 Matching regulärer Ausdrücke

Statt einer endlichen Anzahl an Mustern möchte man auch eventuell ein Musterschema vorgeben. Zur Spezifikation von Musterschemata eignen sich reguläre Ausdrücke. Zur Erinnerung, ein regulärer Ausdruck über einem Alphabet Σ ist wie folgt rekursiv definiert:

- ϵ und jedes $c \in \Sigma$ ist ein regulärer Ausdruck.
- Für jedes Paar r_1, r_2 regulärer Ausdrücke ist auch $r_1 + r_2$ ein regulärer Ausdruck. ("+" bedeutet "r_1 oder r_2".)
- Für jedes Paar r_1, r_2 regulärer Ausdrücke ist auch $r_1 r_2$ ein regulärer Ausdruck. ($r_1 r_2$ bedeutet die Konkatenation von r_1 und r_2.)
- Für jeden regulären Ausdruck r ist auch r^* ein regulärer Ausdruck, wobei $r^* = \{r^i \mid i \geq 0\}$ ist.

So sind zum Beispiel $(a+b)^*c(a+b)^*$ oder $(a(b+ac)^*)^*d(bb+a)^*$ reguläre Ausdrücke. Ein regulärer Ausdruck ist eigentlich eine potentiell unendliche Wortmenge. So gilt:

$$(a+b)^*c(a+b)^* = \{w \in \Sigma^* \mid \exists y, z \in \{a,b\}^* : \; w = xcy\}$$

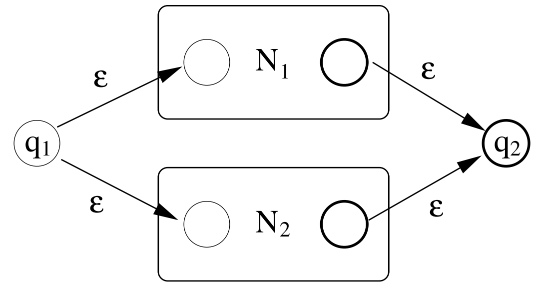
Aus der Automatentheorie ist bekannt, dass es zu jedem regulären Ausdruck einen endlichen Automaten gibt, der diesen akzeptiert. D.h. zu jedem regulären Ausdruck r gibt es einen endlichen Automaten M_r, so dass es für jedes Wort $w \in r$, und nur für diese Worte, eine Menge von Zustandsübergängen in M_r vom Anfangszustand in einen akzeptierenden Endzustand gibt. Wir wollen nun das folgende Problem lösen:

Gegeben ein Text t und ein regulärer Ausdruck r, gib aus, ob es ein Teilwort w in t gibt mit $w \in r$.

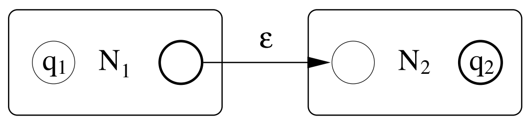
Um dieses Problem zu lösen, werden wir einen nichtdeterministischen endlichen Automaten (NEA) für r konstruieren und diesen dann simulieren während wir t durchlaufen. Die Konstruktion des NEA basiert auf den folgenden 4 Regeln. (q_1 und q_2 zeigen jeweils den neuen Start- und Endzustand an.)

- Für die elementaren regulären Ausdrücke ϵ und $c \in \Sigma$ werden die folgenden Automaten konstruiert:

- Für zwei Ausdrücke r_1 und r_2 mit Automaten N_1 und N_2 wird der Automat für $r_1 + r_2$ wie folgt konstruiert:

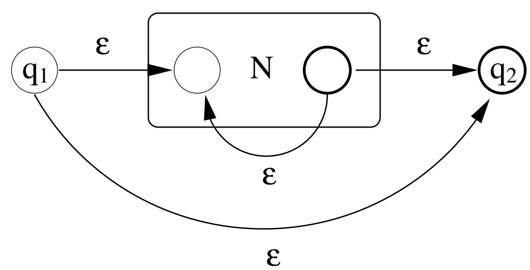


- Für zwei Ausdrücke r_1 und r_2 mit Automaten N_1 und N_2 wird der Automat für $r_1 r_2$ wie folgt konstruiert:



  Anstatt eines ϵ-Übergangs in der Mitte kann auch der Endzustand von N_1 mit dem Anfangszustand von N_2 identifiziert werden.

- Für einen Ausdruck r mit Automat N wird der Automat für r^* wie folgt konstruiert:



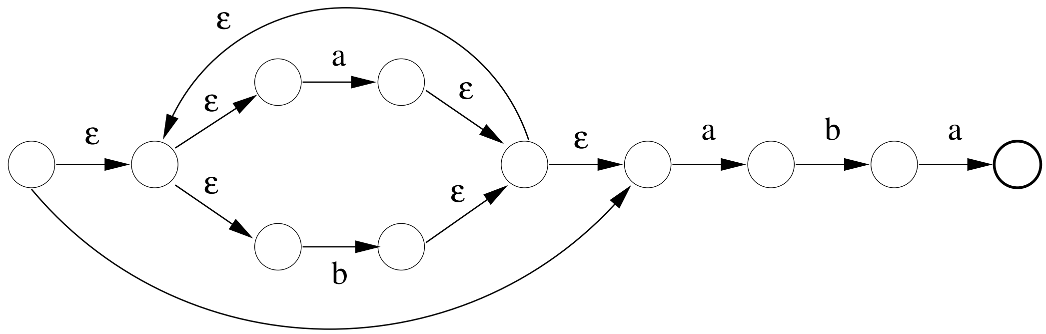
Der so erzeugte Automat N_r hat die folgenden Eigenschaften:

- N_r hat höchstens $2m$ Zustände, wobei $m = |r|$.
- Die Regeln (1) bis (4) benötigen jeweils nur konstante Zeit.
- N_r hat genau einen Start- und genau einen Endzustand.
- Der Startzustand hat keine eingehende Kante, und der Endzustand hat keine ausgehende Kante.
- Jeder Zustand hat entweder
  - genau eine ausgehende Kante, die mit dem Symbol $c \in \Sigma$ markiert ist, oder
  - höchstens zwei mit ϵ markierte ausgehende Kanten.

Als Konsequenz daraus ergibt sich das folgende Lemma.

Lemma 2.29 *Zu einem regulären Ausdruck r kann in $O(m)$ Zeit ein äquivalenter NEA mit ϵ-Übergängen erzeugt werden.*

Betrachten wir zum Beispiel den regulären Ausdruck $(a+b)^*aba$. Dessen NEA sieht wie folgt aus:



Zum Verständnis des Algorithmus zur Simulation von N_r geben wir hier einige Bemerkungen:

- q_0 bezeichnet den Startzustand und q_F den Endzustand des NEA.
- Q bezeichnet die aktuelle Zustandsmenge.
- $\delta(Q, c)$ bezeichnet die Menge aller Zustände, die von einem Zustand in Q aus über eine mit c markierte Kante erreichbar sind.
- $eps(Q)$ bezeichnet die Menge aller Zustände, die von einem Zustand in Q aus über einen ϵ-Übergang erreichbar sind.
- Da das Muster an einer beliebigen Stelle im Text auftreten kann, wird in jeder Iteration der Startzustand q_0 zur aktuellen Zustandsmenge hinzugenommen.

Der Algorithmus zur Simulation eines NEA ist in Abb. 14 gegeben.

Satz 2.30 *Der NEA-Algorithmus hat eine Laufzeit von $O(n \cdot m)$.*

Beweis. Die Konstruktion des NEAs benötigt $O(m)$ Schritte. Weiterhin wird die FOR-Schleife n-mal durchlaufen, und das Update von Q in jedem Durchlauf kostet $O(m)$ Schritte. $\square$

Eine schnellere Laufzeit kann erzielt werden, wenn der NEA zunächst in einen deterministischen endlichen Automaten umgeformt wird. Das kann aber sehr teuer sein, da die Anzahl der Zustände eines deterministischen endlichen Automaten exponentiell in der Anzahl der Zustände der nichtdeterministischen Automaten sein kann.

```
Algorithmus AC-Preprocessing:
    // d[q][c]: für Übergäng von q bei Zeichen c ≥ 1
    // d[q][0] ∈ {0,...,|S|}: gibt Suchwort an, wenn q ∈ F
    // f[q]: Fehlerfunktion für Zustand q ∈ {0,...,m}
    // s[k][i]: i-tes Zeichen im k-ten Suchwort
    // Berechnung des Tries und F_1
    FOR q := 0 TO m DO
        FOR c := 0 TO |Σ| DO d[q][c] := 0
    qnew := 1; q := 0;
    FOR k := 1 TO |S| DO
        FOR i := 1 TO |m_i| DO
            IF d[q][s[k][i]] = 0 THEN
                d[q][s[k][i]] = qnew; qnew := qnew + 1
            q := d[q][s[k][i]]
        d[q][0] := k    // füge q in F_1 ein
    // Berechnung der Fehlerfunktion und F_2
    f[0] := −1;    // Fehlschlag falls Fehler bei 0
    Q := new Queue    // Q für BFS-Durchlauf
    FOR c := 1 to |Σ| DO
        IF d[0][c] ≠ 0 THEN
            f[d[0][c]] := 0    // bei Fehlschlag nach 0
            ENQUEUE(Q, f[d[0][c]])
    WHILE not EMPTY(Q) DO
        q := DEQUEUE(Q)
        FOR c := 1 TO |Σ| DO
            IF d[q][c] ≠ 0 THEN
                q' := d[q][c]
                r := f[q]
                WHILE (r ≠ −1 AND d[r][c] = 0) DO r := f[r]
                IF r=-1 THEN f[q']=0 ELSE f[q'] := d[r][c]
                IF d[f[q']][0] > 0 AND d[q'][0] = 0 THEN    // q' in F_2?
                  d[q'][0] = d[f[q']][0]
                ENQUEUE(Q, q')
```

Figure 12: Berechnung des AC-Automaten für mehrere Suchworte.

```
Aho-Corasick Algorithmus:
    führe AC-Preprocessing durch
    q := 0    // aktuelle Position im Automaten
    FOR i := 1 TO n DO
        WHILE (q ≠ −1 AND d[q][t[i]] = 0) DO
            q := f[q]
        IF q = −1 THEN q := 0 ELSE q := d[q][t[i]]
        IF d[q][0] > 0 THEN
            gib i − m_{d[q][0]} + 1 und d[q][0] aus
```

Figure 13: Aho-Corasick Algorithmus für mehrere Suchworte.

```
Simulationsalgorithmus:
    führe NEA-Preprocessing durch
    Q := eps({q_0})
    IF q_F ∈ Q THEN return true
    FOR i := 1 TO n DO
        Q := eps(δ(Q, t[i]) ∪ {q_0})
        IF q_F ∈ Q THEN return true
    return false
```

Figure 14: Algorithmus zur Simulation eines NEA.