Hauptseminar „Finite State Methoden“bei Prof. Dr. Klaus Schulz, LMU München | CIS
Knuth-Morris-Pratt, Aho-Corasick, Transducers für Rewrite-Dictionaries
Andrea Stubbe, 15.06.2004

1 Knuth-Morris-Pratt

Der Algorithmus von Knuth-Morris-Pratt sucht Vorkommen eines Musters in einem Text. Dabei nutzt er die Tatsache aus, das sich Teile des Musters wiederholen. Bei einem Mismatch wird das Muster so weit im Text verschoben, dass das Suffix des zuletzt passenden Teils Präfix des neuen Match-Versuchs wird (statt – wie beim naiven Algorithmus – nur um ein Zeichen).

Beispiel 1.1 $w = abcaba, t = abcabcaba$

Text	`abcabcaba`
Muster	`abcaba`
	→ `abcaba`

In einem *Preprocessing*-Schritt wird dazu das Muster analysiert und ein Automat aufgebaut.

Definition 1.1 (Maximaler Rand) *Das Subwort von* w*, das gleichzeitig* Präfix(w) *und* Suffix(w) *ist heißt* maximaler Rand *.*

Beispiel 1.2 $w = abcab \Rightarrow$ *maximaler Rand* $(w) = ab$

Definition 1.2 (Label) *Das* Label $l(q_i)$ *ist die Konkatenation aller Zeichen auf dem Weg von* q_0 *bis* q_i*.*

1.1 Fehlerfunktion

Zu jedem Zustand q_i muss zunächst seine *Fehlerfunktion* f berechnet werden. Sie ist wie folgt definiert:

$$f(i) = \begin{cases} \max\{s < i \mid a_1 \dots a_s = a_{i-s+1} \dots a_i\} & \text{falls es solche s gibt} \\ 0 & \text{sonst} \end{cases}$$

Das Label von $q_{f(i)}$ ist also der maximale Rand des Labels von q_i

Beispiel 1.3 $w = abcaba \Rightarrow$

$$\begin{array}{lcl@{\qquad}lcl} f(0) & = & 0 & f(4) & = & 1 \\ f(1) & = & 0 & f(5) & = & 2 \\ f(2) & = & 0 & f(6) & = & 1 \\ f(3) & = & 0 & & & \end{array}$$

1.2 Algorithmus

1. Erzeuge Trie für Suchwort $w = a_1 \dots a_k$
2. Berechne Fehlerfunktion f für alle q_i $(0 < i < k)$
3. Ergänze Trie um neue Übergänge $\delta(q_i, \sigma) := \begin{cases} \delta(q_{f(i)}, \sigma) & (\text{für } 0 < i < k \text{ und } \sigma \neq a_{i+1}) \\ q_0 & (\text{für } i = 0 \text{ und } \sigma \neq a_1) \end{cases}$

Der Automat lässt sich in Zeit $O(|w|)$ erzeugen, die Suche in einem Text t hat die Komplexität $O(|t|)$. Insgesamt ergibt sich (da $|w| \leq |t|$) also $O(|t|)$.

1.3 Automat

Für ein Suchwort w ergibt sich also der Automat $A(w) = \langle Q, \Sigma, q_0, F, \delta \rangle$ mit der Zustandsmenge Q, dem Alphabet Σ, dem Startzustand $q_0 \in Q$, der Menge der Finalzustände $F \subseteq Q$ und der Übergangsfunktion δ, wobei

- $Q = \{q_i \mid 0 \leq i \leq k\}$
- $F = \{q_k\}$
- $\delta(q_i, \sigma) = \begin{cases} q_{i+1} & \text{falls } \sigma = a_{i+1} \\ q_0 & \text{falls } i = 0 \text{ und } \sigma \neq a_i \text{ (also: bleibe in } q_0 \text{ bis } a_i \text{ gelesen)} \\ \delta(q_{f(i)}, \sigma) & \text{sonst} \end{cases}$

Beispiel 1.4 *Für $w = abcaba$ und $\Sigma = \{a, b, c, d\}$ ergibt sich:*

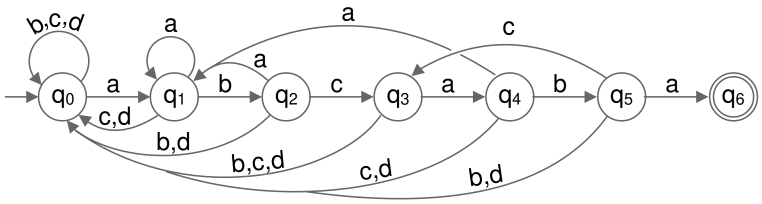

Abbildung 1: Automat für Suchwort *abcaba*

2 Aho-Corasick

Der Aho-Corasick-Algorithmus ist eine Erweiterung der Idee von Knuth-Morris-Pratt auf die Suche nach mehreren Suchwörtern $w_1 \dots w_m$.
Mit Hilfe von *Failure-Links* werden die Muster bei einem Mismatch (von allen Patterns) so weit verschoben, dass das Suffix des bisher verglichenen Texts Präfix eines der Suchwörter ist.

Definition 2.1 (Failure Link) *Übergang $fl(q_i)$ für alle $\sigma \in \Sigma$, für die bisher kein Übergang definiert war (also $(q_i, \sigma) \notin Dom(\delta)$).*

2.1 Berechnung Failure-Links

Mit den *Failure-Links* fl springt der Automat in den Zustand, dessen Label l zugleich Suffix von P und Präfix eines der Suchwörter ist.

Berechnet werden die Failure-Links folgendermaßen:

$$fl(q_i) = \begin{cases} fail & \text{für } i = 0 \\ q_j \mid l(q_j) = \text{längstes echtes Suffix von } l(q_i) & \text{falls es eins gibt und } i > 0 \\ q_0 & \text{sonst} \end{cases}$$

2.2 Algorithmus

1. Erzeuge Keyword-Trie aus Liste der Suchwörter $w_1 \dots w_m$
2. berechne für alle Zustände q_i den Failure-Link $fl(q_i)$

Die Komplexität des Algorithmus ist $O(\sum_{i=1}^{m} |w_i|)$ für die Erzeugung des Automaten und $O(|t|)$ für die Suche nach Patterns im Text.

2.3 Automat und Suche

Für eine Menge von Suchwörtern $W = \{w_1 \dots w_k\}$ ergibt sich also der Automat $A(W) = \langle Q, \Sigma, q_0, F, \delta \rangle$, wobei

- Q = Knoten des Tries
- $F = \{q_i \mid l(q_i) \in W\}$
- $\delta(q_i, \sigma)$ = Übergänge des Tries
- zusätzlich erweitert um Failure-Links $fl(q_i)$

Beispiel 2.1 *Für $W = \{abcaba, abac, bcabac\}$ ergibt sich:*

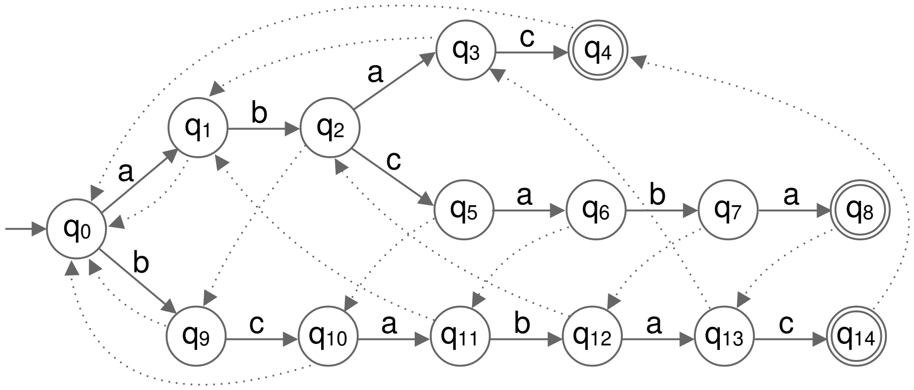

Abbildung 2: Automat für Suchwörter abcaba, abac, bcabac

Bei der Suche durchläuft man den Automaten entlang der Trie-Übergänge. Bei einem Mismatch mit dem Zeichen σ verfolgt man so lange die Failure-Links, bis man einen Zustand erreicht, der eine σ- Kante besitzt und folgt dann dieser Kante. Erreicht man vorher den Zustand $fail$, liest man das nächste Zeichen des Texts und springt wieder zurück zu q_0.
Diesen letzten Fall kann man auch durch Übergänge $\{(q_0, \sigma, q_0) \mid (q_0, \sigma) \notin Dom(\delta)\}$ simulieren – es werden im Startzustand einfach alle nicht-passenden Zeichen im Suchtext übersprungen.

3 Transducers für Rewrite-Dictionaries

Dieser Algorithmus von Klaus Schulz/Stoyan Mihov erzeugt aus einem *Rewrite Dictionary* einen sequentiellen Transducer, der in einem Text alle Vorkommen von Lexikon-Einträgen durch ihre Übersetzung ersetzt. Dabei arbeitet der Transducer nach dem *leftmost largest match*-Prinzip.

3.1 Lexikon-basiertes Übersetzen von Texten

Definition 3.1 (Rewriting Dictionary) *Ein* Rewriting Dictionary *ist eine partielle Funktion $D : \Sigma^+ \rightarrow \Sigma^*$, die* Einträge *auf ihre* Ersetzungen *abbildet.*

Definition 3.2 (leftmost largest) *Ein Vorkommen $\langle u, w, v\rangle \in \Sigma^*$ von w heißt genau dann* leftmost*, wenn sein Prätext u der kleinste aller möglichen Prätexte für das Vorkommen ist.*
Es heißt leftmost largest*, wenn es leftmost ist und w der längste vorkommende Lexikoneintrag beginnend an dieser Stelle ist. Wenn es ein Vorkommen eines Lexikoneintrags im Text t gibt, so existiert immer ein leftmost-largest Vorkommen $lml_D(t)$.*

Beispiel 3.1 *Sei D_1 ein Rewrite-Dictionary mit folgenden Einträgen:*

$$\begin{array}{lcl} a & \mapsto & 1 \\ ab & \mapsto & 2 \\ abcc & \mapsto & 3 \\ babc & \mapsto & 4 \\ c & \mapsto & 5 \end{array}$$

und der Text $t = abcbbbabccb$. Die beiden leftmost-Vorkommen (mit $u = \epsilon$) sind a und ab. Da ab das längste dieser Vorkommen ist, ergibt sich: $lml_D(t) = \langle \epsilon, ab, cbbbabccb\rangle$

Definition 3.3 (sequentieller Transducer) *Ein* sequentieller Transducer *ist ein Tupel $T = \langle \Sigma, \Delta, Q, q_0, F, \delta, \lambda, \Psi\rangle$ mit Eingabealphabet Σ, Ausgabealphabet Δ, Zustandsmenge Q, Startzustand q_0, Menge von Endzuständen $F \subseteq Q$, Übergangsfunktion $\delta : Q \times \Sigma \to \Sigma$, Übergangs-Ausgabefunktion $\lambda : Q \times \Sigma \to \Delta^*$ (mit $Dom(\lambda) = Dom(\delta)$) und Zustands-Ausgabefunktion $\Psi : F \to \Delta^*$.*

Der Rewriting-Transducer erhält als Input den Text $t \subseteq \Sigma^*$ und liefert als Output die durch das Dictionary bestimmte Übersetzung $t' \subseteq \Delta^*$.

3.2 Algorithmus

1. erzeuge Transducer-Trie T_D aus Lexikoneinträgen mit
   - $\Psi_D(q_f) = D(l(q_f)) \quad \forall q_f \in F_D$
   - $\lambda_D(q_i, a) = \epsilon \quad \forall (q_i, a) \in Dom(\delta_D)$
2. durchlaufe T_D in *Breadth-First-Search*, für jedes q':

   (a) $q' = q_0 \Rightarrow$ ergänze F_D, δ_D, λ_D und Ψ_D um
   - $F' := q_0$
   - $\delta' := \{\langle q_0, \sigma, q_0\rangle \mid \sigma \in \Sigma, \langle q_0, \sigma\rangle \notin Dom(\delta_D)\}$
   - $\lambda' := \{\langle q_0, \sigma, \sigma\rangle \mid \sigma \in \Sigma, \langle q_0, \sigma\rangle \notin Dom(\delta_D)\}$
   - $\Psi' := \langle q_0, \epsilon\rangle$

   (b) sonst: sei $l(q') = a_1 \ldots a_k$, also $q' = \delta_D^*(a_1 \ldots a_k)$ und T_i der bisher konstruierte Transducer.

   i. Fall 1: $F_D \cap \{q_0 \ldots q_k\} = \emptyset$
   Sei $p' := \begin{cases} q_0 & \text{falls } |l(q')| = 1 \\ \delta_i^*(q_0, a_2 \ldots a_k) & \text{sonst} \end{cases}$
   Sei $\beta' := \begin{cases} \epsilon & \text{falls } |l(q')| = 1 \\ \lambda_i^*(q_0, a_2 \ldots a_k) & \text{sonst} \end{cases}$

   ergänze F_i, δ_i, λ_i und Ψ_i um
   - $F' := q'$
   - $\delta' := \{\langle q', \sigma, \delta_i(p', \sigma)\rangle \mid \sigma \in \Sigma, \langle q', \sigma\rangle \notin Dom(\delta_D)\}$
   - $\lambda' := \{\langle q', \sigma, a_1 \cdot \beta' \cdot \lambda_i(p', \sigma)\rangle \mid \sigma \in \Sigma, \langle q', \sigma\rangle \notin Dom(\delta_D)\}$
   - $\Psi' := \langle q', a_1 \cdot \beta' \cdot \Psi_i(p')\rangle$

   ii. Fall 2: $F_D \cap \{q_0 \ldots q_k\} \neq \emptyset$
   Sei q_f der letzte Finalzustand auf Pfad $\pi(q')$.

$$\text{Sei } p' := \begin{cases} q_0 & \text{falls } q_f = q' \\ \delta_i^*(q_0, a_{f+1} \dots a_k) & \text{sonst} \end{cases}$$

$$\text{Sei } \beta' := \begin{cases} \epsilon & \text{falls } q_f = q' \\ \lambda_i^*(q_0, a_{f+1} \dots a_k) & \text{sonst} \end{cases}$$

ergänze F_i, δ_i, λ_i und Ψ_i um

- $F' := q'$
- $\delta' := \{\langle q', \sigma, \delta_i(p', \sigma)\rangle \mid \sigma \in \Sigma, \langle q', \sigma\rangle \notin Dom(\delta_D)\}$
- $\lambda' := \{\langle q', \sigma, \Psi_D(q_f) \cdot \beta' \cdot \lambda_i(p', \sigma)\rangle \mid \sigma \in \Sigma, \langle q', \sigma\rangle \notin Dom(\delta_D)\}$
- $\Psi' := \langle q', \Psi_D(q_f) \cdot \beta' \cdot \Psi_i(p')\rangle$

Der Transducer lässt sich in $O(|\Sigma| \cdot m)$ erzeugen (wobei m = Anzahl der Zustände), das Übersetzen des Textes t erfolgt in $O(|t|)$.

3.3 Transducer

Für ein Rewrite-Dictonary D ergibt sich also der Transducer $T = \langle \Sigma, \Delta, Q, q_0, F, \delta, \lambda, \Psi\rangle$ mit

- Q = Knoten des Tries
- $F = Q$
- $\delta(q_i, \sigma)$, $\lambda(q_i, \sigma)$ wie in Algorithmus erzeugt

Beispiel 3.2 *Für das Dictionary D_1 aus Beispiel 3.1 und $\Sigma = \{a, b, c\}$ ergibt sich:*

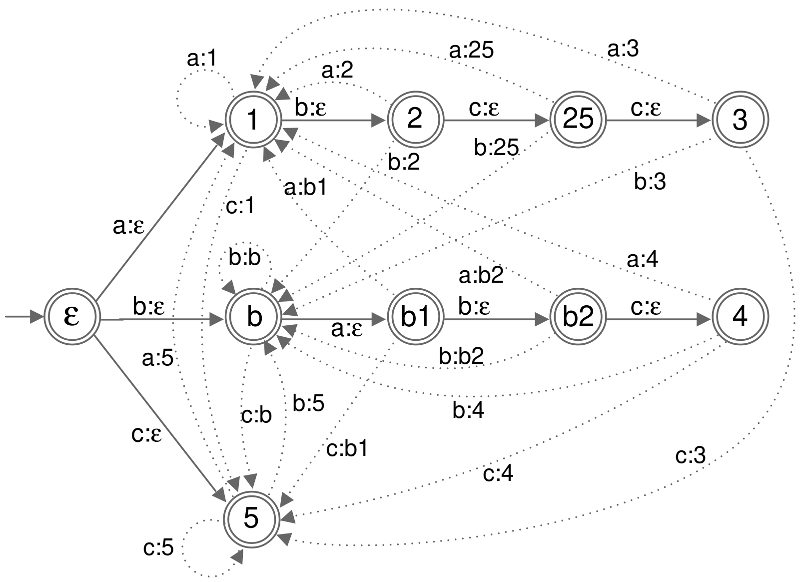

Abbildung 3: Transducer für D_1 (durchgezogene Linien sind Kanten des Tries)

Literatur

[1] H. W. Lang. 2001. *String-Matching-Algorithmen.* http://www.iti.fh-flensburg.de/lang/algorithmen/pattern/kmp.htm

[2] Ulf Leser. *Skript zur Vorlesung Bioinformatik*. Humboldt Universität Berlin. http://www.informatik.hu-berlin.de/wbi/teaching/ws0304/bioinfo/05_keyword_trees_regexp.pdf

[3] Dieter Buehler. *Animation of the Aho/Corasick Pattern Matching Automaton*. http://www-sr.informatik.uni-tuebingen.de/~buehler/AC/AC.html

[4] Stoyan Mihov und Klaus U. Schulz. 2004. *Efficient Dictionary-Based Text Rewriting using Sequential Transducers*. Technical Report, CIS (to appear).