Skript zum Kurs Finite-State-Technologien beim Indizieren und Suchen

Estelle Perez

2. Mai 2013

Inhaltsverzeichnis

1 Einleitung

Das folgende Skript stellt Begleitmaterial zum Kurs *Finite-State-Technologien* im Wintersemester 2012/2013 dar. Es baut maßgeblich auf gesammelten Beiträgen von Ulrich Reffle aus diversen Kursen zur Automatenimplementierung in vorhergehenden Semestern auf (z.B. [Ref10]), ebenso wie auf Kursmaterialen zum Thema *Formale Sprachen und Automaten* von Prof. Dr. Schulz ([Sch08b]). Einige der Erklärungen wurden in ähnlicher Weise auch in meiner Magisterarbeit [Per12] verwendet, in der Automaten ebenso einen zentralen Bestandteil einnehmen. Insbesondere Abbildungen wurden übernommen (dies ist entsprechend gekennzeichnet).
Ziel dieses Skripts ist die Beschreibung hocheffizienter Wege zum Aufbauen und Speichern azyklischer Lexikonautomaten. Abschnitt 2 widmet sich kurz der Wiederholung zentraler Begrifflichkeiten zum Thema endliche Automaten. Aufbauend auf diesem theoretischen Hintergrund erläutert Abschnitt 3 Grundfragen der tatsächlichen Implementierung solcher Automaten und insbesondere von Tries, einer häufig zur Speicherung von Lexika verwendeter Unterart azyklischer endlicher Automaten. Abschnitt 4 erläutert eine geschickte Form der Kompression solcher Tries. Eine andere Form der effizienten Speichernutzung, nämlich die Automatenminimierung, wird in Abschnitt 5 vorgestellt, wobei das hier vorgestellte Verfahren *online* arbeitet, also darauf verzichtet, zuerst den nicht minimierten Trie aufzubauen, bevor der Automat minimiert wird. Abschnitt 6 zeigt, wie trotz Minimierung zu jedem Wort geeignete Zusatzinformationen gespeichert werden können. Abschnitt 7 zeigt schließlich eine geschickte Möglichkeit des approximativen Matchings.

2 Grundlagen

2.1 Alphabete und Wörter

Im Folgenden werden kurz einige Grundlagen zu formalen Sprachen und Automaten wiederholt[1].

Definition 1 (Alphabet)**.** [Sch08b, 5] Ein *Alphabet* ist eine endliche Menge Σ von Zeichen (Symbolen).

Definition 2 (Wort)**.** Ein *Wort* über einem Alphabet Σ ist eine geordnete Folge von Symbolen aus Σ.

Definition 3 (leeres Wort)**.** Mit ϵ wird im Allgemeinen das *leere Wort* bezeichnet. Es gilt $|\epsilon| = 0$.

Definition 4. (siehe auch [Sch08b, 6]) Es bezeichnen

- Σ^n: die Menge aller Wörter der Länge n über Σ.
- $\Sigma^+ = \cup_{i>0}\Sigma^i$: die Menge aller nicht-leeren Wörter über dem Alphabet Σ.
- $\Sigma^* = \cup_{i\in\mathbb{N}}\Sigma^i$: die Menge aller Wörter über Σ (inkl. dem leeren Wort ϵ).

[1] Inhalte und Beschreibungen orientieren sich an den Mitschriften aus dem Kurs *Implementierung von endlichen Automaten* im Sommersemester 2010, Definitionen wurden meist mit dem Skript [Sch08b] abgeglichen. Dort können auch genauere Beschreibungen und Beispiele nachgelesen werden.

Definition 5 (Konkatenation)**.** [Sch08b, 8] Die *Konkatenation* $\circ : \Sigma^* \times \Sigma^* \to \Sigma^*$ ist für zwei Wörter $u \in \Sigma^k$ und $v \in \Sigma^l$ wie folgt definiert:

$$u \circ v(i) := \begin{cases} u(i) & \text{wenn } 1 \leq i \leq k \\ v(i-k) & \text{wenn } k < i \leq k+1 \end{cases}$$

2.2 Endliche Automaten und Sprachen

Definition 6 (Endlicher Automat)**.** (siehe auch [Sch08b, 23]) Ein *endlicher Automat* (EA) ist ein Quintupel $A = (Q, \Sigma, I, F, \Delta)$ mit:

- Q: einer endlichen und nicht-leeren Zustandsmenge
- Σ: einem endlichen Eingabealphabet
- $I \subseteq Q$: einer Menge von Initialzuständen
- $F \subseteq Q$: einer Menge von Finalzuständen
- $\Delta \subseteq Q \times \Sigma^* \to Q$ einer Übergangsrelation (statt Σ kann auch $\Sigma \cup \{\epsilon\}$ gewählt werden)

Anmerkung: ein solchermaßen definierter Automat wird auch *nicht-deterministischer endlicher Automat* (NDEA) genannt.

Definition 7 (Deterministischer Endlicher Automat)**.** (siehe auch [Sch08b, 18]) Ein *deterministischer endlicher Automat* (DEA) ist ein Quintupel $A = (Q, \Sigma, I, F, \delta)$. Σ, Q, F sind wie oben definiert. Statt einer Übergangs*relation* Δ ist nun eine Übergangs*funktion* $\delta : Q \times \Sigma \to Q$ definiert. Außerdem besitzt ein DEA nur einen ausgewiesenen Startzustand s.
Die Funktion δ muss nicht total sein, d.h. es muss nicht an jedem Zustand $q \in Q$ für jedes Symbol $\sigma \in \Sigma$ ein expliziter Übergang definiert sein. Man schreibt hierfür auch oft $\delta(q, \sigma) = \perp$, wobei $\perp$ auch *Fallenzustand* genannt wird.

Definition 8 (Pfad)**.** Ein *Pfad* im Automaten bezeichnet eine Folge von Tripeln aus Δ der Form:

$$(q_1, w_1, q_2), (q_2, w_2, q_3) \dots (q_{n-1}, w_{n-1}, q_n)$$

(Pfad von q_1 nach q_n mit den Labels w_1, w_2, w_{n-1})

Definition 9 (Von A akzeptierte Sprache)**.** Die von A *akzeptierte Sprache* ist:

$$L(A) = \{w \in \Sigma^* | w \text{ ist Label eines Pfades von einem Start- zu einem Finalzustand}\}$$

Für $q \in Q$ ist:

$$L(q, A) = \{w \in \Sigma^* | w \text{ ist Label eines Pfades von } q \text{ zu einem Finalzustand } q'\}$$

$L(q, A)$ wird oft auch *rechte Sprache* oder *Rechtssprache* eines Zustandes $q \in Q$ genannt.

Definition 10 (verallgemeinerte Übergangsrelation Δ^*)**.** (siehe auch [Sch08b, 23]) Sei $A = (Q, \Sigma, I, F, \Delta)$ ein NDEA. Die *verallgemeinerte Übergangsrelation* $\Delta^* \subseteq Q \times \Sigma^* \times Q$ wird induktiv wie folgt definiert: $\forall q \in Q, w \in \Sigma^*, \sigma \in \Sigma$:

- $(q, \epsilon, q) \in \Delta^*$

- ist $(q_1, w, q_2) \in \Delta^*$ und $(q_2, \sigma, q_3) \in \Delta$, so auch $(q_1, w\sigma, q3) \in \Delta^*$

Damit gilt für $p, q \in Q, w \in \Sigma^*$:

$$(p, w, q) \in \Delta^* \Leftrightarrow \text{es existiert ein Pfad mit Label } w \text{ von } p \text{ nach } q$$

Analog lässt sich die *verallgemeinerte Übergangsfunktion* für einen DEA mit $q \in Q, w \in \Sigma^*, \sigma \in \Sigma$ wie folgt definieren (siehe auch [Sch08b, 19]):

$$\delta^*(q, \epsilon) = q,$$
$$\delta^*(q, w\sigma) = \delta(\delta^*(q, w), \sigma)$$

Statt δ^* (bzw. Δ^*) wird oft auch $\hat{\delta}$ (bzw. $\hat{\Delta}$) geschrieben.

Definition 11 (Äquivalente Automaten)**.** (siehe auch [Sch08b, 28]) Zwei Automaten A und A' sind *äquivalent*, wenn sie dieselbe Sprache erkennen, d.h. wenn gilt: $L(A) = L(A')$

Definition 12 (Kleene-Stern einer Sprache)**.** (siehe auch [Sch08b, 9]) Der *Konkatenationsabschluss* oder *Kleene-Stern* einer Sprache ist definiert durch $L^* := \cup_{n \in \mathbb{N}} L^n$

Definition 13 (Abschlusseigenschaften – Auswahl)**.** Seien $A_1 = (Q_1, \Sigma, I_1, F_1, \Delta_1)$, $A_2 = (Q_2, \Sigma, I_2, F_2, \Delta_2)$ und $Q_1 \cap Q_2 = \emptyset$.

- *Vereinigung*: $A = (Q_1 \cup Q_2, \Sigma, I_1 \cup I_2, F_1 \cup F_2, \Delta_1 \cup \Delta_2)$
  erkennt $L(A_1) \cup L(A_2)$
- *Konkatenation*: $A = (Q_1 \circ Q_2, \Sigma, I_1, F_2, \Delta_1 \circ \Delta_2 \circ \{(f, \epsilon, q) | f \in F_1, q \in I_2\})$
  erkennt $L(A_1) \circ L(A_2)$
- *Kleene-Stern*: Sei q_0 ein neuer Zustand.
  $A = (Q_1 \cup \{q_0\}, \Sigma, \{q_0\}, F_1 \cup \{q_0\}, \Delta)$ mit
  $\Delta = \Delta_1 \cup \{(q_0, \epsilon, q | q \in I_1\} \cup \{(f, \epsilon, q_0) | f \in F_1\}$
  erkennt $L(A_1)^*$

2.3 Reguläre Ausdrücke und reguläre Sprachen

Definition 14 (Reguläre Ausdrücke)**.** (siehe auch [Sch08b, 13-14]) *Reguläre Ausdrücke* werden wie folgt induktiv definiert:

- $\emptyset$ oder 0 (mit $L(0)$ für die leere Sprache) ist ein regulärer Ausdruck
- 1 (mit $L(1)$ die Sprache, die das leere Wort ϵ erkennt) ist ein regulärer Ausdruck
- Für jedes $\sigma \in \Sigma$ ist σ ein regulärer Ausdruck
- falls α und β reguläre Ausdrücke sind, so auch:
  - α^*
  - $\alpha \cup \beta$ (auch als $\alpha + \beta$ notiert)
  - $\alpha \cdot \beta$ (auch als $\alpha\beta$ notiert)

Definition 15 (Reguläre Sprachen)**.** (siehe auch [Sch08b, 14]) Induktiv lassen sich *reguläre Sprachen* wie folgt definieren:

- $L(\emptyset) = \emptyset (= L(0))$ (die leere Sprache) ist eine reguläre Sprache
- $L(1) = \{\epsilon\}$ ist eine reguläre Sprache
- für $\alpha \in \Sigma$ ist $L(\alpha) = \{\alpha\}$ eine reguläre Sprache
- sind L_1 und L_2 reguläre Sprachen, so auch $L_1 \cup L_2, L_1 \cdot L_2, L_1^*$.
- es gibt keine weiteren regulären Sprachen

Theorem 1 (Kleenes Theorem)**.** [Sch08b, 43] Die Klasse der regulären Sprachen ist genau die Klasse von Sprachen, die durch einen EA akzeptiert werden kann.

Ein Beweis findet sich unter anderem in [Sch08b, 43].

2.4 Determinisierung von endlichen Automaten

Theorem 2 (Satz von Rabin/Scott)**.** (aus [Sch08a, 24]) Jede von einem NDEA akzeptierbare Sprache ist auch durch einen DEA akzeptierbar.

Sei $A = (Q, \Sigma, I, F, \Delta)$ ein NDEA[2].
Nun entsteht ein DEA $A' = (Q', \Sigma, s, F', \delta)$, der dieselbe Sprache akzeptiert folgendermaßen:

- für jede mögliche Teilmenge von Zuständen aus A wird jeweils ein einzelner Zustand in A' vorgesehen wird. D.h., $Q' = \mathcal{P}(Q)$.
- Übergänge von A' von einem solchen *Mengenzustand* umfassen jeweils *alle* A-Übergänge der im Mengenzustand enthaltenen A-Zustände.

  D.h. $\delta(q', \sigma) = \{q \in Q | \exists q_1 \in q', (q_1, \sigma, q) \in \Delta^*\}$ für $q' \subseteq Q, \sigma \in \Sigma$.
- Als Startzustand s wird derjenige Mengenzustand gewählt, der als "Bestandteile" alle A-Zustände enthält, die auch in I enthalten sind, sowie alle Zustände, die von Zuständen aus I mit ϵ-Übergängen erreicht werden können.
- Als Finalzustände werden all jene Mengenzustände $q' \subset Q$ gewählt, bei denen gilt $q' \cap F \neq \emptyset$

In der Praxis kann man sich auf solche Zustände beschränken, die vom Startzustand aus erreichbar sind (die unter *Spontannachfolger* abgeschlossen sind)[3] Für genauere (teilweise leicht abweichende) Ausführungen siehe [Sch08a, 24-25] und [Sch08b, 30-31].
Für einen NDEA mit n Zuständen kann der durch Potenzmengenkonstruktion erstellte äquivalente DEA bis zu 2^n Zustände haben. Selbst wenn man sich bei der Konstruktion nur auf solche Zustände beschränkt, die vom Startzustand aus erreichbar sind, ergibt sich dadurch bereits für vergleichsweise kleine NDEA ein Speicherplatzproblem (siehe auch [Sch08b, 33]). Andererseits ist die Berechnung des *Erkennungsproblems* (also, ob ein gegebener Automat A ein Wort w akzeptiert) mit einem NDEA komplexer als mit einem DEA (dort ist sie linear zur Länge des Eingabewortes). Daher wird man in der Praxis lieber auf deterministische als auf nicht-deterministische Automaten zurückgreifen wollen (zur Zeitkomplexität des Erkennungsproblems von NDEA siehe auch [Sch08b, 24-27]).

[2] Hier wird angenommen, dass jedes Übergangslabel stets die Form ϵ oder $\sigma \in \Sigma$ hat. Sollte dies nicht der Fall sein, es also Labels der Länge $k \geq 2$ geben, so müssen diese Labels zunächst durch Einfügen geeigneter Zwischenzustände gesplittet werden.

[3] Als *Spontannachfolger* eines Zustands $p \in Q$ bezeichnet man: $SNF_A(p) = \{q \in Q | (p, \epsilon, q) \in \Delta^*\}$ bzw. $SNF_A(p) = \{q \in Q | (p, \epsilon) \Longrightarrow_A^* (q, \epsilon\})$, d.h. q ist von p in endlich vielen ϵ-Schritten erreichbar (siehe auch [Sch08b, 29]).

2.5 Minimale deterministische Automaten

2.5.1 Definitionen

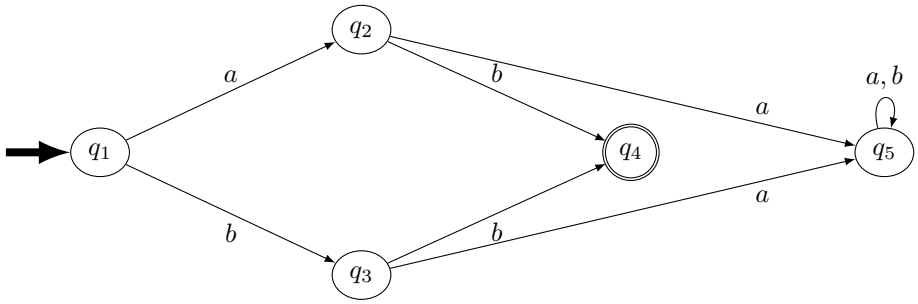
Zu jeder regulären Sprache L existiert ein bis auf Namen der Zustände eindeutig bestimmter kleinster deterministischer endlicher Automat, der genau L erkennt.

Definition 16. Reduzierter DEA (siehe auch [Sch08b, 46]) Ein DEA $A = (Q, \Sigma, s, F, \delta)$ heißt *reduziert* bzw. *minimiert*, falls:

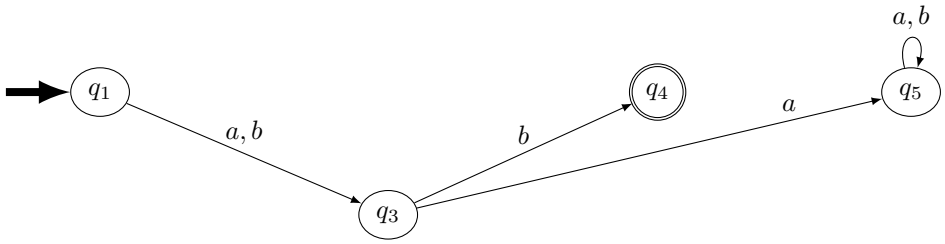
- jeder Zustand q vom Startzustand s aus erreichbar ist (d.h. es gibt ein Eingabewort, mit dem man von s zu q gelangen kann)
- A keine zwei voneinander verschiedenen Zustände p und q hat, so dass p und q äquivalent sind.

Definition 17 (ununterscheidbar)**.** (siehe auch [Sch08b, 45]) Zwei Zustände $p, q \in Q$ werden als *k-ununterscheidbar* bezeichnet (für $k \geq 0$), genau dann wenn gilt:
$\forall w \in \Sigma^*, |w| \leq k : \delta^*(p, w) \in F \Leftrightarrow \delta^*(q, w) \in F$
Zwei Zustände p und q heißen *äquivalent* oder *ununterscheidbar*, wenn sie für jedes $k \in \mathbb{N}$ ununterscheidbar sind, also falls gilt:
$\forall w \in \Sigma^* : \delta^*(p, w) \in F \Leftrightarrow \delta^*(q, w) \in F$

Als einfaches Beispiel sei der folgende NDEA A gegeben:



Die Zustände q_1, q_2, q_3 und q_5 sind paarweise 0-ununterscheidbar (sie sind alle nicht-final). q_1 und q_5 sind 1-ununterscheidbar (beide sind nicht final, von beiden kommt man mit a und b jeweils zu einem nicht-finalen Zustand), aber nicht 2-ununterscheidbar (mit ab kommt man von q_1 zu einem Finalzustand, von q_5 jedoch nicht). q_2 und q_3 sind ununterscheidbar.
Daher lässt sich A in einen – eindeutig bestimmten – kleinsten DEA A', der dieselbe Sprache $L(A) = L(A') = \{ab, bb\}$ erkennt, umwandeln.



Im Wesentlichen lässt sich das Auffinden des Minimalautomaten für eine Sprache L wie folgt beschreiben:

- starte mit einem beliebigen DEA A, der L erkennt
- eliminiere Zustände, die vom Startzustand aus nicht erreichbar sind
- suche nach äquivalenten Zuständen und identifiziere diese

2.5.2 Algebraische Konstruktion des minimierten DEA

Sei $L \in \Sigma^*$ eine beliebige Sprache.

Definition 18 (Nerode-Äquivalenzrelation)**.** (siehe auch [Sch08b, 49]) Für eine beliebige (auch nicht-reguläre) Sprache L über dem Alphabet Σ wird die *Nerode-Äquivalenzrelation* $\sim_L$ wie folgt definiert:
$\forall u, v \in \Sigma^* : u \sim_L v :\Leftrightarrow (\forall w \in \Sigma : u \cdot w \in L \Leftrightarrow v \cdot w \in L)$

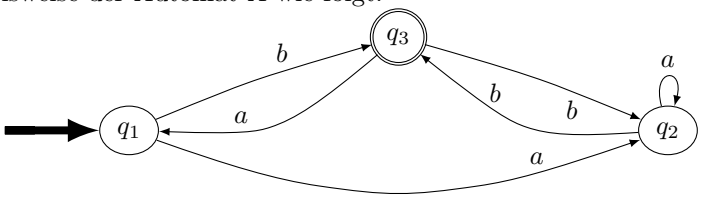
Theorem 3 (Satz von Nerode)**.** [Sch08b, 49] Sei $L \subseteq \Sigma^*$ und $\sim_L$ wie oben definiert. Dann ist L regulär gdw. die Zahl der $\sim_L$-Äquivalenzrelationen von Σ^* endlich ist.

Beweis 1 ($\Rightarrow$)**.** (siehe auch [Sch08b, 49]) Sei L regulär. Dann existiert (nach Kleene-Theorem) ein DEA $A = (Q, \Sigma, s, F, \delta)$ mit $L = L(A)$. Es sei $L_{A,s}(q) := \{w \in \Sigma^* | \delta^*(s, w) = q\}$ die Sprache bestehend aus der Menge aller Wörter, die vom Startzustand zum Zustand q führen. Für $u, v \in L_{A,s}(q)$ gilt offenkundig $u \sim_L v$ (da beide zum Zustand q führen). Aus diesem Grund kann die Zahl der $\sim_L$-Äquivalenzklassen die Zahl der Zustände nicht übersteigen (und ist also endlich).

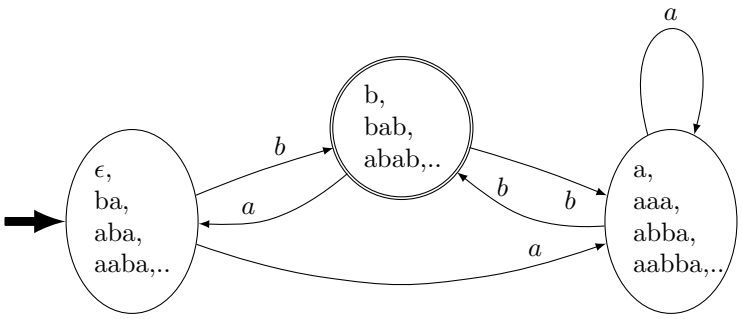
Für $\Leftarrow$ siehe [Sch08b, 49].
Für eine reguläre Sprache L ist $A = (\Sigma^*/\sim_L, \Sigma, [\epsilon]_L, \{[w]_{\sim_L} | w \in L\}, \delta)$, mit:

- $\Sigma^*/\sim_L$ Menge der Äquivalenzklassen von $\sim_L$
- $[\epsilon]_{\sim_L}$ Äquivalenzklasse von ϵ
- $\{[w]_{\sim_L} | w \in L\}$ Äquivalenzklassen der Wörter aus L
- $\delta([u]_{\sim_L}, a) = [ua]_{\sim_L}$

der eindeutig bestimmte kleinste deterministische Automat für L.
Sei beispielsweise der Automat A wie folgt:



In Anlehnung an die obengenannte Definition könnte man die Zustände auch mit Repräsentanten ihrer Äquivalenzklassen labeln, also z.B. so:



3 Automaten – Grundfragen zur Implementierung

Der folgende Abschnitt baut auf den obigen – sehr theoretischen – Definitionen auf und untersucht sie insbesondere in ihrer Bedeutung für die tatsächliche Implementierung[4].

3.1 Alphabet

Eine wichtige Implementierungsentscheidung für die Programmierung eines Lexikonautomaten ist die Festlegung des Alphabets, das der Automat verarbeiten können soll. Sollen nur die 128 ASCII-Zeichen unterstützt werden (z.B. für englischsprachige Texte)? Reichen die 256 Symbole, die von ISO 8859-1 (Latin-1) zu Verfügung gestellt werden? Benötigt man gar alle Unicode-Zeichen (mindestens 2^{16})? Oder soll – angesichts dieser erschreckend großen Zahl – lieber ein eigenes Alphabet konstruiert werden, welches dann beliebige Sonderzeichen, davon aber nur relativ wenige enthalten soll?
Eine weitere Frage ist jene nach der Kodierung, d.h. die Form des Mappings von Buchstaben zu Zahlenwert. Gerade bei einem benutzerdefinierten Alphabet kommt dieser Frage besondere Bedeutung zu.
Schließlich steht außerdem zu Debatte, ob Groß- und Kleinbuchstaben berücksichtigt werden sollen (und wenn nicht, wie etwaige Großbuchstaben in der Eingabe korrekt umgewandelt werden können).

3.2 Zustände und Übergangsfunktionen

An Zustände und Übergangsfunktionen werden im Wesentlichen zwei Anforderungen gestellt:

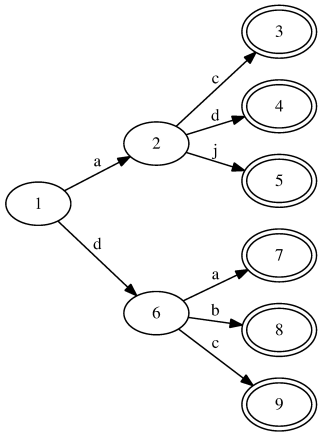
- der *lookup*, also die Frage, ob von einem gegebenen Zustand $q \in Q$ ein Übergang mit Label $\sigma \in \Sigma$ existiert und wohin dieser ggf. führt.
- der *traversal*, d.h. die Möglichkeit, die Labels aller Übergänge (bzw. deren Zielzustände) ausgehend von einem Zustand $q \in Q$ festzustellen.

Aus den obigen Anforderungen ergeben sich verschiedene Möglichkeiten zum Aufbau der Datenstruktur.

[4]Inhalte stark orientiert an [Ref10], Abbildungen größtenteils entnommen aus [Per12]

3.2.1 Listenbasierte Speicherung

Eine Variante basiert auf Listen bzw. Hashmaps. Abbildung 1 zeigt einen beispielhaften *Trie* (einen deterministischen endlichen Automaten) mit passender Datenstruktur: jeder Zustand besitzt dabei eine Liste von Übergängen (kodiert durch Label und Zielzustand). Die hier dargestellte Übergangsfunktion ist partiell – nicht jeder Zustand hat für jedes Symbol aus dem Eingabealphabet einen Übergang.



(a) Trie für *ac, ad, aj, da, db, dc*

1: $< a, 2 >, < d, 6 >$
2: $< c, 3 >, < d, 4 >, < j, 5 >$
3:
4:
5:
6: $< a, 7 >, < b, 8 >, < d, 9 >$
7:
8:
9:

(b) Listenbasierte Speicherung

Abbildung 1: Trie und passende listenbasierte Speicherung (Abbildungen auch verwendet in [Per12])

Während ein *traversal* in diesem Fall einfach durchzuführen ist (eine einfache Iteration über alle Elemente der assoziierten Liste eines Zustands), ist der *lookup* zeitaufwendig. Bei einer sortierten Liste benötigt er $O(log(n))$.

3.2.2 Übergangstabelle

Ein anderer Ansatz ist daher, den Automaten in einer sogenannten *transition table*, einer Matrix der Größe $|Q| \times |\Sigma|$, abzulegen (siehe Abbildung 1(a)). Hier kann der *lookup* in konstanter Zeit durchgeführt werden, da es sich um einen Arrayzugriff handelt. Diese Lösung bewährt sich jedoch nur für sehr kleine Automaten – für größere Automaten (oder auch nur gößere Alphabete) wächst die Tabelle viel zu schnell.
Dennoch ist scheint Großteil des verbrauchten Speichers vergeudet: in unserem Fall sind nur 8 der 90 Felder der Tabelle tatsächlich belegt. Für Lexikonautomaten (und für viele andere DEA) ist es charakteristisch, dass die Anzahl der Übergänge der meisten Zustände weitaus kleiner ist als $|\Sigma|$. Aus diesem Grund wird der Speicherplatzbedarf einer $|Q| \times |\Sigma|$-Tabelle für größere Automaten natürlich unnötig wachsen.

	1	2	3	4	5	6	7	8	9
a	2					7			
b						8			
c		3				9			
d	6	4							
e									
f									
g									
h									
i									
j		5							

Abbildung 2: Tabelle für Trie aus Abbildung 1(a) (Abbildungen auch verwendet in [Per12])

3.3 Einfacher Algorithmus zur Konstruktion eines Tries

Im Folgenden wird ein einfacher Algorithmus zur Konstruktion eines Tries dargestellt.

```
1  void compileTrie( char const* filename ) {
2      std::wstring word;
3      std::wifstream in (filename);
4
5      while( std::getline(in, word).good() ) {
6
7          wchar_t const* c = word.cstr();
8          State_t* splitState = getRoot();
9
10         while( splitState->delta(*c) ) {
11             splitState = splitState->delta(*c);
12             ++c;
13         }
14
15         State *last = splitState;
16         while( *c != 0 ) {
17             State_t *next = newState();
18             last->setTransition(*c, next);
19             last = next;
20             ++c;
21         }
22         last->setFinal(true);
23     }
24     std::wcerr
25     << "csl::SimpleFSA::Trie::compileTrie::automaton has"
26         << Automaton::getNumberOfStates()
27         << "states." << std::endl;
28 }
```

Abbildung 3: Methode `compileTrie` – Codebeispiel zum Erzeugen eines Tries (aus [Ref10])

4 Speicheroptimierung durch Tarjan Tables[5]

Wie bereits in Abschnitt 3 dargestellt, sind "naive" Speicherstrategien zu langsam bzw. zu speicherintensiv. Die Ziele einer effizienten Speicherstruktur lassen sich wie folgt formulieren:

- schneller und wahlfreier Zugriff auf alle Übergänge eines Zustandes (notwendig für einen effizienten *lookup*)
- kompakte Darstellung im Speicher
- Darstellung in einer zusammenhängenden Region im Speicher: einmal aufgebaut kann der Automat auf die Festplatte geschrieben und beim nächsten Einsatz einfach zurückgelesen werden (ohne explizit neu aufgebaut zu werden).

Weitere wichtige Ziele können sein:

- Möglichkeit zur schnellen Iteration über alle Übergänge eines Zustandes (notwendig, um z.B. das Lexikon komplett auszulesen)
- Möglichkeit, gespeicherte Wörter mit Zusatzinformationen zu annotieren

Die hier vorgestellte Datenstruktur genügt allen genannten Zielen. Sie hat dabei allerdings auch den Nachteil, dass Übergänge nicht *in place* hinzugefügt werden können. Von jedem Zustand müssen erst alle Übergänge bekannt sein, bevor dieser in die Datenstruktur eingefügt werden kann. Daher wird auf temporäre Zustände (im Folgenden auch *tempstates*) zurückgegriffen.
In ihrem Artikel *Storing a sparse table*([TY79]) präsentieren Robert Tarjan und Andrew Yao eine Methode, um sogenannte *sparse tables*, also dürftig besetzte Tabellen, effizient zu speichern. Hier wird eine vereinfachte Version ihrer Idee vorgestellt.
Abbildung 4 verdeutlich das angewandte Prinzip. Auf der linken Seite sieht man jene drei Spalten aus Tabelle 2, die überhaupt Einträge aufweisen (also Zustände mit Übergängen). In der Mitte der Abbildung sind diese drei Spalten nun dergestalt verschoben, dass sie kollisionsfrei ineinander verzahnt werden können (rechte Seite der Abbildung). In diesem Sinn kann das allgemeine Prinzip so beschrieben werden: alle Spalten, die Übergänge einzelner Zustände repräsentieren, werden in eine eindimensionale Struktur überführt, was die Anzahl leerer Felder drastisch reduziert. Die dergestalt aufgebaute Tabelle besteht aus *Zellen* oder *cells*. Für jeden Zustand wird eine *Zustandszelle* oder *state cell* belegt, die den Nullpunkt relativ zum Zustand darstellt. Für jeden seiner Übergänge werden nun *Übergangszellen* oder *transitions cells* belegt. Zwei zentrale Fragen stellen sich nun:

- Wie wird ein Zustand in der *tarjan table* referenziert und wie werden seine jeweiligen Übergänge gefunden?
- Wie kann eine passende Position gefunden werden, um einen neuen Zustand mit all seinen Übergängen kollisionsfrei einzufügen, ohne dabei unnötig Platz zu verschenken?

4.1 Informationsspeicherung im komprimierten Datenfeld

Hier wird die Frage beantwortet, wie ein Zustand im komprimierten Datenfeld referenziert werden kann. In unkomprimierten Übergangstabellen gibt es eine eindeutige Entsprechung von (fortlaufenden) Zustandsnummern und deren entsprechenden Spalten in der Tabelle.

[5] Abschnitt stark angelehnt an[Ref10], Abbildungen entnommen aus [Per12]

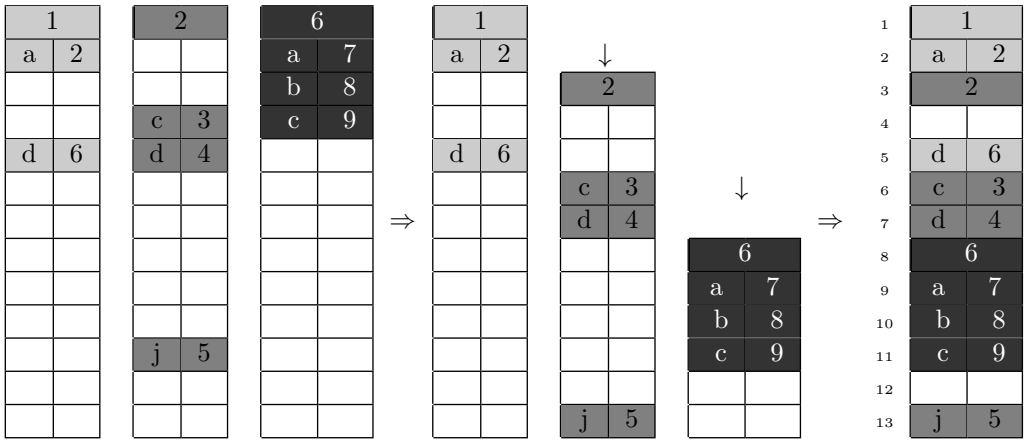


Abbildung 4: Komprimierungsmöglichkeit für die *sparse table* aus Abbildung 2 (Idee angelehnt an [Ref10], Abbildung auch verwendet [Per12])

Dies ist hier nicht mehr gegeben. Statt einer fortlaufenden Id wird nun der Index, also die Stelle im Datenfeld, in dem die *state cell* des Zustands gespeichert ist genutzt. Damit ist einerseits eine eindeutige Zuordnung garantiert (an jedem Index im Datenfeld kann sich maximal eine Zustandszelle befinden). Außerdem kann in konstanter Zeit auf den jeweiligen Zustand zugegriffen werden (ein Datenfeld oder Array ermöglicht wahlfreien Zugriff). Der Zustand mit Id 6 wird also derjenige sein, der sich im Datenfeld an Index 6 befindet. Wir schreiben s_n um den Zustand am n-ten Index zu bezeichnen. Ein Nachteil dieser Vorgehensweise ist, dass Zustände nur dann eindeutig referenziert werden können, wenn sie bereits in der Tabelle eingetragen sind. Daher wird der (azyklische) Lexikonautomat von rechts nach links in die Tabelle eingetragen: so ist garantiert, dass für jeden neu einzufügenden Zustand bereits alle seine Nachfolgerzustände in der Tabelle eingetragen (und daher eindeutig referenzierbar) sind.

4.2 Zuordnung der Übergänge zum passenden Zustand

Nachdem alle Spalten der ursprünglichen Übergangstabelle zu einer eindimensionalen Struktur verzahnt wurden, stellt sich die Frage, wie Zustände und Übergänge – z.B. zum Zwecke des *lookup* – wieder "entwirrt" werden können. Um dies zu ermöglichen, werden in Übergangszellen stets zwei Werte gespeichert. Der zweite Wert enthält die Id des Zielzustandes, der erste hingegen das Label des Übergangs. Mithilfe dieses Labels kann ein Zusammenhang zwischen Zustand und Übergang hergestellt werden. Befindet sich z.B. das d an der vierten Stelle im Alphabet[6], so wird der passende d-Übergang vier Positionen unterhalb des zugehörigen Zustandes zu finden sein.
Abbildung 5 zeigt ein Beispiel eines Tries mit seiner komprimierten Entsprechung[7]. Betrachtet man Zustand s_4, also den Eintrag an Index 4 des Datenfeldes, und geht von dort aus 4 Schritte für ein Symbol d weiter, so landet man an Index 8(= 4+4) in einer Übergangszelle

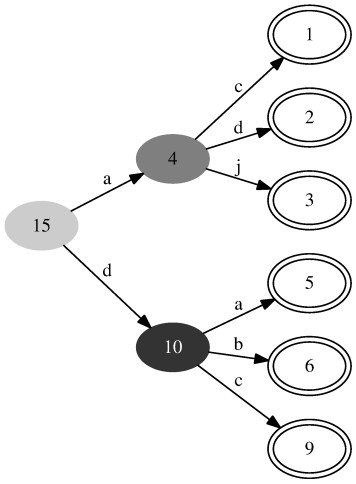
[6] Um diese Methode zu verwenden, wird eine totale Ordnung über dem Alphabet Σ erwartet. Diese ist in unserem Beispiel durch die Auftretensreihenfolge im Alphabet gegeben, in der Praxis aber einfach durch das vom jeweiligen Encoding bereitgestellte Mapping von Zeichen zu Zahlenwerten.

[7] Die gespeicherten Wörter sind die gleichen wie im Trie aus Abbildung 1(a), aufgrund der oben beschrieben Speicherreihenfolge von rechts nach links ergibt sich jedoch eine andere Nummerierung der Zustände. Außerdem wird der Zustand an Stelle 0 als Fallenzustand markiert.

mit Label d. Zustand s_4 hat also einen d-Übergang. Sucht man hingegen einen f-Übergang und geht dafür sechs Schritte von s_4 aus weiter, so erhält man an Index 10(= 4+6) eine Zustands- und keine Übergangszelle. Zustand s_4 hat also keinen f-Übergang. Sucht man ein i, erreicht man Index 13(= 4 + 9) und findet eine mit c gelabelte Übergansgzelle: Zustand s_4 hat also auch keinen i-Übergang. Erwartungsgemäß werden nur für die tatsächlich vorhandenen Übergänge mit Label c, d und j korrekt gelabelte Übergangszellen an den erwarteten Positionen gefunden.

0	FAIL	
1	S	
2	S	
3	S	
4	S	
5	S	
6	S	
7	c	1
8	d	2
9	S	
10	S	
11	a	5
12	b	6
13	c	9
14	j	3
15	S	
16	a	4
17		
18		
19	d	10

(a) Übergangstabelle



(b) Trie für *ac, ad, ai, da, db, dc*

Abbildung 5: Übergangstabelle und Trie in tatsächlicher Speicherreihenfolge (Abbildung auch verwendet in [Per12])

4.3 Finden eines passenden Tabellenindex zum Einfügen eines neuen Zustands

Wie der Titel dieses Abschnitts bereits suggeriert, geschieht das Einfügen der Zustände schrittweise. Tatsächlich waren wir ursprünglich davon ausgegangen, dass der Trie in seiner unkomprimierten Form gar nicht im Speicher gehalten werden kann. Aus diesem Grund macht es auch keinen Sinn, nach einem Algorithmus zu suchen, der eine optimale Speicherung der Tabelle garantiert (dafür müsste man ja zunächst den kompletten Trie kennen). Daher wird hier ein ganz einfacher Ansatz verfolgt: für jeden neu einzufügenden Zustand wird

das Datenfeld schrittweise durchsucht, bis eine passende Einfügestelle gefunden wird. Das Codefragment in Abbildung 6 verdeutlicht dies.

```
1  int TransitionTable::findSlot( State s ) {
2    int slot = compArray.firstFreeCell;
3    while( 1 ) {
4      success = true;
5      for( all outgoing transitions (s, c, nextState) of s ) {
6        if( compArray[slot + c].cellType != empty ) {
7          success = false;
8          break;
9        }
10     } // for all transitions
11
12     if( success ) {
13       return slot;
14     }
15     else {
16       // find the next empty slot and start over
17       ++slot;
18       while( compArray[slot].cellType != empty ) ++slot;
19     }
20   } // while( 1 )
21 } // method findSlot
```

Abbildung 6: Methode `findSlot` – Finden einer passenden Zelle in der `TransitionTable` (aus [Ref10])

Neue Zustandszellen können dann wie in Abbildung 7 gespeichert werden.

```
1  int TransitionTable::storeState( State s ) {
2    int slot = findSlot(s);
3    compArray[slot].cellType = state;
4    if( s.isFinal ) compArray[slot].isFinal = true;
5    for( all outgoing transitions (s, c, s') of s ) {
6      compArray[slot + c].cellType = transition;
7      compArray[slot + c].label = c;
8      compArray[slot + c].target = s';
9    }
10
11   // update firstFreeCell
12   while( compArray[compArray.firstFreeCell] != empty ) {
13     ++compArray.firstFreeCell;
14   }
15   return slot;
16 }
```

Abbildung 7: Methode `storeState` der Klasse `TransitionTable` (aus [Ref10])

Um zu vermeiden, dass wiederholt an Stellen gesucht wird, in denen nur wenige – kleine – Lücken vorhanden sind, kann außerdem noch ein Suchfenster der Größe m definiert werden.

Die Suche nach der nächsten freien Zelle wird dann m Schritte hinter dem zurückgegeben Einfügeindex gestartet, sofern $slot - m > firstFreeCell$.

4.4 Effizienter *lookup*, schneller *traversal*

Die folgenden zwei Codefragmente zeigen grob den Aufbau einer Zelle sowie die Methode `delta` zum Finden eines mit $\sigma \in \Sigma$ gelabelten Übergangs für einen beliebigen Zustand $q \in Q$.

```
1 class Cell {
2   CellType CellType; // state, transition or empty
3   char label;
4   int target;
5   bool isFinal;
6 }
```

Abbildung 8: Beispielimplementierung der Klasse `Cell` (aus [Ref10])

```
1 int delta( int stateId, char c ) {
2   if( compArray[stateId + c].label == c )
3     return compArray[ stateId + c ].target;
4   else
5     return 0; // FAIL state
6 }
```

Abbildung 9: Methode `delta` - Codebeispiel (aus [Ref10])

An der Implementierung der `delta`-Methode kann man sehen, dass, gegeben ein Zustand s und ein Label c, in konstanter Zeit entschieden werden kann, ob s einen mit c gelabelten Übergang hat, und, wenn ja, wohin dieser führt. Um den *lookup* für ein Wort w durchzuführen, muss also höchstens $|w|$ Übergängen gefolgt werden, so dass der *lookup* für ein Wort w die Zeitkomplexität $O(|w|)$ besitzt.
Neben dem *lookup* ist eine weitere häufig notwendige Funktion der *traversal*. Eine einfache Verwendung einer solchen Funktion wäre z.B. die Methode *countEntries*, die einen *depth-first traversal* auf dem Lexikon durchführt, um die Anzahl der gespeicherten Wörter zu zählen. Um eine solche Methode bereitstellen zu können, muss die oben skizzierte Tabelle also erweitert werden: an jeder Zustandszelle müssen die Labels all seiner Übergänge abgefragt werden können. Hierfür lassen sich verschiedene Strategien verfolgen. Zum einen könnte man an jedem Label (und an der Zustandszelle selber) den Index des nächsten zugehörigen Übergangs speichern. Damit könnte man durch einfachen wahlfreien Zugriff über alle Übergänge eines Zustandes iterieren. Eine andere Möglichkeit ist es, an jeder Zustandszelle einen Zeiger auf einen String – bestehend aus den Labels aller Übergänge des Zustandes – zu speichern. Die Übergänge eines Zustands können dann iteriert werden, indem für jedes Zeichen des Übergangsstrings der jeweilige Zahlenwert zum Index des Zustandes hinzuaddiert wird (siehe auch den Pseudocode für *countEntries*). Wie ein solcher *transitionString* gespeichert wird, kann unterschiedlich erfolgen. Für große Lexika kann man sich vorstellen, beim Aufbau etwas mehr Zeit zu benötigen und alle *transitionStrings* in einer Hashmap[8] zu speichern: bevor ein Übergangsstring gespeichert wird, würde zunächst nachgesehen, ob

[8] Zu Hashes siehe Punkt 6.1

dieser nicht bereits existiert. Wenn ja, so würde der Zeiger auf den bereits existierenden String zeigen, was auf Dauer Speicher spart. Die Hashmap kann natürlich am Ende des Trie-Aufbaus wieder gelöscht werden, um neuen Speicher freizugeben.

```
1  int countEntries( int stateId ) {
2    int count = 0;
3    if( compArray[stateId].isFinal )
4      count = 1;
5    for( int i = 0; i < alphabetSize; ++i ) {
6      nextState = delta( stateId, transitionString[i] );
7      if( nextState != 0 ) {
8        count += count( nextState );
9      }
10   } // for all transitions
11   return count;
12 } // method countEntries
```

Abbildung 10: Methode `countEntries` – Codebeispiel zum Zählen der Einträge eines Tries (aus [Ref10])

```
1  int countEntries( int stateId ) {
2    int count = 0;
3    if( compArray[stateId].isFinal )
4      count = 1;
5    char* transitionString = getTransitionString( stateId );
6    for( int i = 0; transitionString[i] != 0; ++i ) {
7      nextState = delta( stateId, transitionString[i] );
8      count += count( nextState );
9    } // for all transitions
10   return count;
11 } // method countEntries
```

Abbildung 11: Alternative Methode `countEntries`(aus [Ref10])

5 Konstruktion eines minimierten Automaten anhand einer sortierten Wortliste

Der folgende Abschnitt stellt ein in [DMWW00] entwickeltes Verfahren vor, welches es ermöglicht, einen Automaten bereits zum Konstruktionszeitpunkt stets minimal zu halten[9]. Für wirklich große Lexika reicht die durch den Einsatz einer Tarjan Table erzielte Speicherplatzersparnis oft nicht aus. Die Automatenminimierung bietet sich als weitere Kompressionsmöglichkeit an. Der Vorteil des hier vorgestellten Algorithmus ist der, dass nicht – wie bei üblichen Verfahren – erst der komplette Trie aufgebaut und damit im Speicher gehalten werden muss (was unter Umständen gar nicht möglich ist), sondern stets nur der kleinstmögliche Automat im Speicher liegt.

[9] Die Ausführungen sind eng an denen in [Per12] orientiert, sämtliche Abbildungen sind von dort entnommen.

5.1 Minimierung eines vorhandenen Tries

Wie bereits in Punkt 2.5 gesehen, können DEAs minimiert werden, indem Äquivalenzklassen über ihre Zustände gebildet werden. Behält man von jeder Äquivalenzklasse eines DEA A nur einen Repräsentanten, so erhält man den minimalen DEA A' für A. Zur Verdeutlichung zeigt Abbildung 12 noch einmal einen nicht minimierten Trie, während in Abbildung 13 der äquivalente reduzierte DEA zu sehen ist.

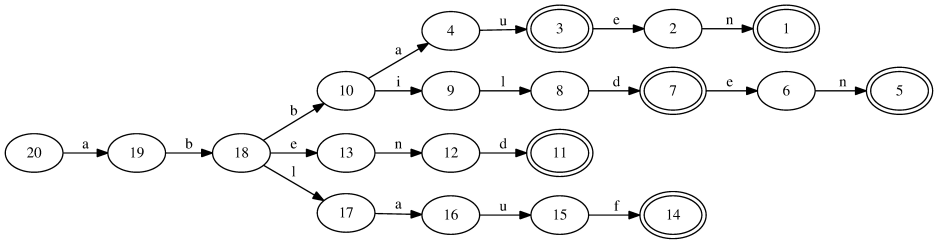


Abbildung 12: Nicht-minimierter Trie für *abbau, abbauen, abbild, abbilden, abend, ablauf* – Zustände in *postorder*-Nummerierung (Abbildung auch verwendet in [Per12])

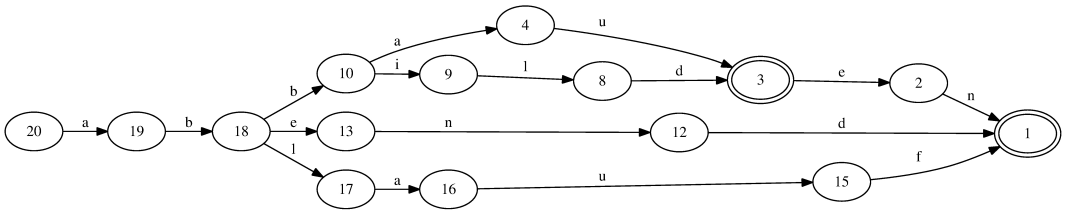


Abbildung 13: Minimierte Form des Tries aus Abbildung 12 unter Beibehaltung der gleichen Nummerierung (Abbildung auch verwendet in [Per12])

Es stellt sich nun die Frage, wie die Äquivalenz zweier Zustände festgestellt werden kann. Angelehnt an Definition 17 müssen wir für je zwei Zustände prüfen:

1. ob beide final oder beide nicht-final sind
2. ob beide über die gleiche Anzahl ausgehender Übergänge verfügen
3. ob diese Übergänge die gleichen Labels tragen
4. ob gleich gelabelte Übergänge zu jeweils äquivalenten Zuständen führen (für die wiederum die obigen Bedingungen gelten)

Durchläuft man den nicht-minimierten Trie in *postorder*-Reihenfolge (analog zur Nummerierung der Zustände in Abbildung 12), so kann man garantieren, dass zu keinem Zeitpunkt zwei äquivalente Zustände endgültig (also in der minimierten Struktur) gespeichert sind. Daher reicht es, die – aufwendige – Prüfung unter 4. durch folgende zu ersetzen:

4'. ob gleich gelabelte Übergänge zu gleichen Zielzuständen führen.

5.2 Online-Minimierung

Die Idee ist nun, das obige Vorgehen sozusagen zu imitieren, konkret, einen postorder-Durchlauf durchzuführen, bei dem nur solche Zustände gespeichert werden, von denen man sicher sein kann, dass sie nicht mehr verändert werden können (also insbesondere keine neuen Übergänge mehr erhalten können). Um dies zu garantieren, wird eine sortierte Wortliste eingelesen. Zustände, von denen man erwarten kann, dass sie sich zu einem späteren Zeitpunkt noch ändern werden, werden *temporär* gehalten, also noch nicht endgültig abgespeichert. Das Beispiel in Abbildung 14 verdeutlicht dies: während die ellipsenförmigen Zustände bereits endgültig abgespeichert sind, sind die rechteckigen Zustände noch veränderbar. Veränderbar sind stets nur Zustände, die den Pfad für das gerade gelesene Wort repräsentieren. Für alle anderen Zustände garantiert die lexikographische Vorab-Sortierung der Wortliste, dass sie keine weiteren Übergänge mehr erhalten können.

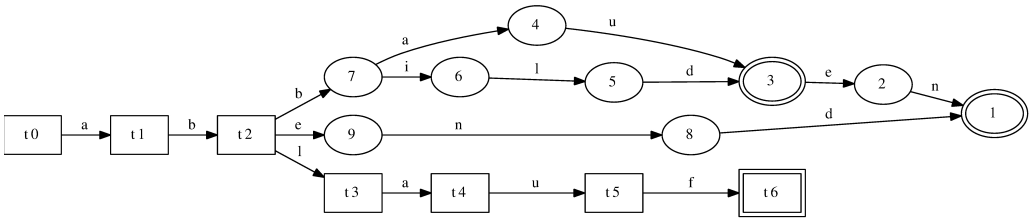


Abbildung 14: Trie für *abbau, abbauen, abbild, abbilden, abend, ablauf* während der Minimierung (Abbildung auch verwendet in [Per12])

Wird ein neues Wort hinzugefügt:

- beginnt es entweder mit einem neuen (lexikografisch größeren) Buchstaben
- oder es teilt sich ein gemeinsames Präfix mit dem letzten hinzugefügten Wort
- etwaige Änderungen der Zustände durch Hinzufügen eines neuen Wortes betreffen also höchstens die Zustände des zuletzt eingelesenen Wortes (und im zuvor gelesenen Wort nur das etwaige gemeinsame Präfix mit dem aktuellen Wort).
- andere bestehende Zustände erhalten höchstens neue eingehende Übergänge, die deren rechte Sprache nicht ändern
- Wörter, die ‘älter‘ als das vorletzte eingefügte Wort sind, bleiben unveränderbar

Daraus lässt sich der Pseudocode aus Abbildung 15 ableiten.

```
1  Register:= {}
2  while there is another word {
3     Word:= next word in lexicographic order;
4     CommonPrefix:= common_prefix(Word);
5     SplitState:=delta_string(q0,CommonPrefix);
6     CurrentSuffix:=
7        Word[length(CommonPrefix)...length(Word)-1];
8     if( has_children(SplitState) ){
9        replace_or_register(SplitState);
10    }
11    add_suffix(SplitState,CurrentSuffix);
12 }
13 replace_or_register(q0);
14
15 replace_or_register(State) {
16    Child:=last_child(State);
17    if( has_children(Child) ) {
18       replace_or_register(Child);
19    }
20    if(exists q in Register and q equals Child ) {
21       last_child(State):=q: (q in Register and q equals Child);
22       delete(Child);
23    } else {
24       Register := Register + {Child};
25    }
26 }
```

Abbildung 15: Pseudocode für *replace_or_register* zur Online-Minimierung

Die Methode `common_prefix` ermittelt das größte gemeinsame Präfix w zwischen aktuellem und zuletzt gelesenen Wort (so dass `delta_string(`q_0, w`) != NULL`). Die Methode `last_child` gibt für eine Zustand q denjenigen Zustand zurück, welcher mit seinem (bisher) letzten Übergang erreicht wird.
Sei der bisher aufgebaute Automat wie in Abbildung 14 und das danach gelesene Wort *abrufen*. Dann gibt die Methode `common_prefix` in Zeile 4 das Präfix ab zurück. Mit `delta_string(`q_0, ab`)` aus der nächsten Zeile wird der Zustand t_2 als `SplitState` gesetzt. In Zeile 8 wird nun geprüft, ob $t2$ *Kinder*, also Übergänge und damit Nachfolgerzustände hat. Da dies der Fall ist, wird `replace_or_register` aufgerufen. Dort wird in Zeile 16 der letzte Nachfolgerzustand von t_2, nämlich t_3 zurückgeben (also immer ein Zustand, der bis jetzt noch temporär ist). Nun wird rekursiv die Methode `replace_or_register` aufgerufen, bis man bei Zustand t_6 ankommt, der selber keine Nachfolgerzustände besitzt. In Zeile 20 wird nun geprüft, ob t_6 bereits im `Register` vorhanden ist, ob also bereits ein äquivalenter Zustand vorhanden ist. Wir suchen einen Finalzustand ohne Übergänge: ein solcher ist in Zustand 1 bereits vorhanden. Daher wird t_6 durch 1 ersetzt und t_6 gelöscht. Aufgrund der Rekursion wird nun nacheinander auch für t_5, t_4 und t_3 nach etwaigen bereits eingetragenen äquivalenten Zuständen gesucht (und – da solche nicht vorhanden sind – die drei Zustände in das Register eingetragen).
Das `Register` (in der Praxis z.B. ein eigens entwickeltes Hash[10]) muss nur solange existieren,

[10] Zu Hashes siehe Punkt 6.1

wie der Aufbau des Automaten noch nicht abgeschlossen ist. Danach kann es gelöscht werden.

6 Speicherung von Zusatzinformationen für einzelne Lexikonwörter

In der Regel wird ein Lexikon nicht nur dazu benötigt werden, nachzusehen, ob ein Wort in ihm enthalten ist oder nicht, sondern auch dazu, Zusatzinformationen zum gefundenen Wort auszugeben. So kann es z.B. interessant sein, die Frequenz eines Wortes zu kennen, sein Lemma oder Ähnliches. In einem Trie, für den jedes Wort an einem eigenen Zustand endet, können am Finalzustand einfach Zeiger auf die gewünschte Zusatzinformation gespeichert werden. Bei einem minimierten Automaten, bei dem alle Wörter an einem einzigen Finalzustand enden, ist dies nicht mehr möglich.
Dennoch gibt es einen einfachen Trick, mit dem jedes Wort einen eindeutigen Schlüssel erhalten kann (mit dem dann wiederum auf die gespeicherte Zusatzinformation zugegriffen werden kann). Dieses Verfahren wird als *static perfect hashing* bezeichnet (vgl. [DMS05]).

6.1 Speicherung von Schlüssel-/Wert-Paaren

Um Schlüssel-/Wert-Paare zu speichern, bieten sich im Allgemeinen mehrere Möglichkeiten an. Hat man einen numerischen Schlüssel und ist insgesamt die Anzahl Schlüssel beschränkt, so ist die Nutzung eines Datenfeldes (Arrays) sicher die einfachste Möglichkeit: für ein Schlüssel-/Wert-Paar (k, v) speichert man einfach den Wert v an Index k. Oft ist die grundsätzlich mögliche Anzahl Schlüssel jedoch weit größer als der Speicherplatz, den man zu Verfügung stellen will, vorallem, wenn es nicht für jeden Schlüssel einen Eintrag geben wird (siehe auch [CLRS07, 221]). Außerdem sind oft keine numerischen Schlüssel vorhanden. Nun bieten sich zweierlei Speicherstrategien an. Einerseits kann man auf sogenannte *Baumstrukturen* zurückgreifen (bzw. *verkettete Listen*, sofern Elemente sortiert eingefügt werden). Ein Beispiel dafür ist der STL-Container[11] `map`. Der Vorteil einer solchen Struktur ist, dass leicht über alle Elemente, oder aber auch nur über einen definierten Ausschnitt innerhalb von Grenzwerten iteriert werden kann. Ein konkretes Element über seinen Schlüssel zu erreichen erfordert aber $O(log\ n)$ (Zeitkomplexität für Binärsuche).
Eine andere Möglichkeit stellen sogenannte *Hashtabellen* dar. Dabei wird für jeden Eintrag mittels einer *Hashfunktion* ein Schlüssel berechnet, der wiederum angibt, an welcher Stelle der Eintrag gespeichert werden soll. In (C++) bieten die STL-Container `unordered_map` eine solche Implementierung. Dabei werden die gespeicherten Elemente nicht sortiert, sondern anhand ihres Schlüssels in sogenannten *buckets* gespeichert: nur wenn zufälligerweise mehrere Elemente den gleichen Schlüssel haben sollten (z.B. aufgrund einer ungeschickt gewählten Hashfunktion), muss ein einzelnes *buckets* nach dem tatsächlichen Vorhandensein eines Elements durchsucht werden. Ist die Hashfunktion gut gewählt, ist im Durchschnitt der Zugriff auf ein einzelnes Element konstant. Damit ist ein solcher Container gut geeignet, wenn man häufig nach einzelnen Elementen sucht (Näheres hierzu ist in in der C++-Referenz – z.B. unter *http://www.cplusplus.com* – oder auch in [Had10] nachzulesen).

6.2 Perfektes Hashing mittels Automaten

Für unseren Anwendungsfall ist eine `unordered_map` jedoch nicht geeignet: wenn man explizit um Speicherplatz kämpft, macht es wenig Sinn, Strings als Schlüssel zum Speichern von

[11] STL für *Standard Template Library*

Zusatzinformation zu verwenden (was z.B. mit einer `unordered_map<wstring,wstring>` möglich wäre). Es liegt daher nahe, nach einem Verfahren zu suchen, welches für jedes Lexikonwort einen eindeutigen numerischen und monoton wachsenden Schlüssel berechnet, um dann Zusatzinformation in einem linearen Datenfeld zu speichern. Das hier skizzierte Verfahren erfüllt genau dies: für jedes Lexikonwort ist der Schlüssel genau der Wert, der angibt, an welcher Stelle in der sortierten Wortliste es sich befand (er entspricht also der lexikographischen Sortierung, die ja wie oben dargestellt auch die Reihenfolge bestimmt, in der das Wort in den Automaten eingefügt wird).
Ein mögliches Verfahren hierzu wurde in [DMS05] vorgestellt (siehe dort, Erklärungen und Abbildungen in ähnlicher Weise auch verwendet in [Per12]).

Definition 19 (Größe der rechten Sprache)**.** ([DMS05, 171]):
$|\overrightarrow{L}(q)| = (\sum_{a:\delta(q,a)\neq\perp} |\overrightarrow{L}(\delta(q,a))|) + \begin{cases} 0 & q \notin F \\ 1 & q \in F \end{cases}$

Die Größe der rechten Sprache eines Zustandes q berechnet sich also aus der Summe der Größen der rechten Sprachen aller seiner Nachfolgerzustände. Hierzu wird noch der Wert eins addiert, falls q final ist (um anzuzeigen, dass am Finalzustand ein Wort gelesen werden kann).
Abbildung 16 stellt den bereits mehrmals als Beispiel verwendeten Trie dar, wobei die Größe der rechten Sprache eines jeden Zustandes in Klammern unterhalb der Zustandsnummer steht. Man kann die Korrektheit der Werte leicht nachprüfen. An Zustand 3 können z.B. genau zwei weitere Teilworte gelesen werden: das leere Wort ϵ, welches an Zustand 3 selbst endet, und das Teilwort *en*. Ebenso können an Zustand 7 vier Teilworte gelesen werden, nämlich *au, auen, ild* und *ilden*.

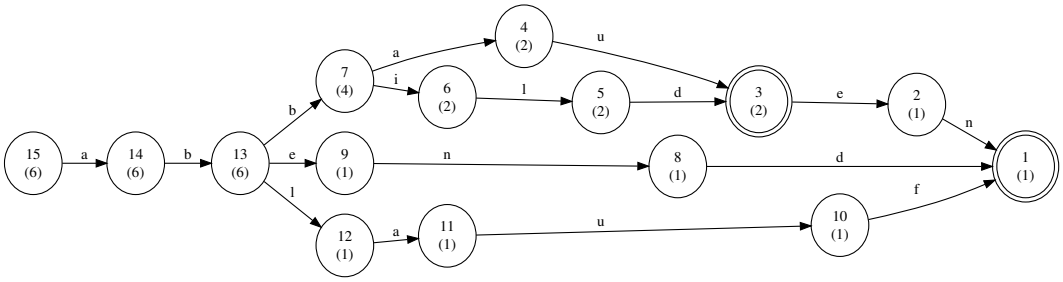


Abbildung 16: Minimierter Trie aus Abbildung 13 mit Größen der rechten Sprachen an den Zuständen

Eine andere Darstellungsmöglichkeit ergibt sich, wenn die berechneten *Rechtssprachengrößen* (oder auch *Gewichte*) nicht an den Zuständen selber, sondern an den jeweiligen Labels gespeichert werden. Abbildung 17 stellt diese Möglichkeit dar. Dabei erhält jedes Label σ eines Zustandes q als zugeordneten Wert die Anzahl Wörter, die von q aus mit lexikographisch kleineren Labels $\tau < \sigma$ gelesen werden konnten. Der erste Übergang eines nicht-finalen Zustandes erhält also stets den Wert 0. Der erste Übergang eines Finalzustandes erhält den Wert 1 (mit einem leeren Übergang ϵ, bzw. ohne vom Finalzustand q aus weiterzugehen, konnte ja bereits ein Wort gelesen werden).
Der Hashwert eines Wortes setzt sich nun aus der Summe der gespeicherten Werte für seine einzelnen Labels zusammen. So hat in unserem Fall das Wort *abend* den Hashwert $0 + 0 + 4 + 0 + 0 = 4$.

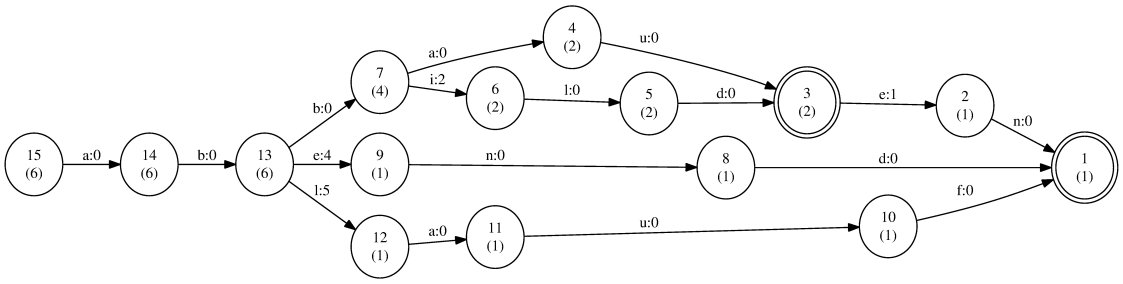


Abbildung 17: Minimierter Trie aus Abbildung 13 mit Größen der rechten Sprachen und entsprechenden Einträgen für die Labels (Abbildung auch verwendet in [Per12])

In der Praxis lässt sich dieses Verfahren mit den oben erwähnten Vorgehensweisen zur Online-Minimierung unter Einsatz einer komprimierten Übergangstabelle kombinieren. Die Codefragmente in Abbildung 18 verdeutlichen dies.

```
1  /**
2   * return the current perfect hash value of this TempState
3   */
4  size_t getPhValue() const {
5    return phSum_ + ( isFinal() ? 1 : 0 );
6  }
7
8  /**
9   * add an outgoing edge to this TempState
10  * @param label
11  * @param target
12  */
13 void addTransition( wchar_t label,
14      size_t target, size_t targetPhNumber ) {
15   // check that alphabetic order is not violated - code omitted
16   transitions_.push_back(Transition(label, target, getPhValue()))
17   // store current label in the string of labels - code omitted
18   phSum_ += targetPhNumber;
19 }
20
21 /**
22  * reset the state for re-use
23  */
24 inline void reset() {
25   transitions_.clear();
26   // clear string of labels - code omitted
27   annotation_ = 0;
28   isFinal_ = false;
29   phSum_ = 0;
30 }
```

Abbildung 18: Auszüge aus Klasse TempState zur Erläuterung des *perfect hashing*

Wie bereits in Punkt 5.2 erwähnt, wird für die Online-Minimierung des Tries ein Array aus temporären Zuständen, bzw. *TempState*s benötigt. Der in Abbildung 18 dargestellte Code-Auszug zeigt einige der Klassenmethoden. Zeile 4 zeigt die Methode `getPhValue`, welche den aktuellen Hashwert des *TempState* zurückgibt. Sofern der Zustand final ist, wird der gespeicherte Hashwert um eins inkrementiert (analog zur vorgestellten Formel). Zeile 14 zeigt Auszüge der Methode `addTransition`, die einem temporären Zustand einen neuen Übergang hinzufügt. Dabei wird ersichtlich, dass zunächst (Zeile 16) für den gerade zu speichernden Übergang der aktuelle Hashwert des Zustandes gespeichert wird (also die Größe der rechten Sprache, die bisher vom Zustand aus gelesen werden konnte). Danach wird der Hashwert des Zustandes um die Größe der rechten Sprache, die mit dem gerade hinzufügten Übergang erreicht werden kann (hier `targetPhNumber` genannt) inkrementiert. Dies entspricht der obigen Beobachtung, dass der erste Übergang eines Zustands stets den Hashwert 0 (bzw. 1 für Finalzustände) hat. Schließlich wird in `reset` in Zeile 24 gezeigt, wie ein temporärer Zustand zur erneuten Verwendung zurückgesetzt werden kann.

```
1 //in MinDic::addToken:
2 for( size_t i = wcslen(lastKey_); i > commonPrefix; --i ) {
3   storedState = replaceOrRegister(tempStates_[i]);
4   tempStates_[i-1].addTransition( lastKey_[i-1],
5                   storedState,
6                   tempStates_[i].getPhValue());
7   tempStates_[i].reset();
8 }
```

Abbildung 19: Auszüge aus Klasse `MinDic`

In der Klasse `MinDic`, die das eigentliche minimierte Lexikon (unter Nutzung von `TempStates`) aufbauen soll, erfolgt nun die Berechnung der Hashwerte. Abbildung 19 zeigt den relevanten Programmausschnitt. In Zeile 2 wird rückwärts über alle Symbole des zuletzt gelesenen Wortes iteriert und der jeweilige temporäre Zustand i für dieses Symbol mit `replaceOrRegister` gespeichert (als Pseudocode dargestellt in Abbildung 15). Da als zugrundeliegende Klasse eine `TransitionTable` verwendet wird, erhält man den Speicherindex `storedState` des gespeicherten Zustandes zurück. Nun kann man diesen neu ermittelten Index als jenen Nachfolgerzustand für den temporären Zustand $i-1$ speichern, der mit Lesen des Label an Stelle $i-1$ erreicht wird. Gleichzeitig wird auch der Hashwert des gerade gespeicherten temporären Zustandes i übergeben. Wie bereits vorher in *addTransition* (Zeile 14 von Abbildung 18) gesehen, wird dieser Wert dem Hashwert des temporären Zustandes $i-1$ hinzugefügt (und zum Tragen kommen, falls später ein neuer Übergang vom Zustand $i-1$ aus gespeichert werden soll).

```
1  //in TransitionTable:
2  int TransitionTable::storeState( State s ) {
3    int slot = findSlot(s);
4    compArray[slot].cellType = state;
5    if( s.isFinal ) compArray[slot].isFinal = true;
6    for( all outgoing transitions (trans) of s ) {
7      compArray[slot + c].cellType = transition;
8      compArray[slot + c].label = trans.getLabel();
9      compArray[slot + c].target = trans.getTarget();
10     compArray[slot + c].phValue = trans.getPhNumber();
11   }
12
13   // update firstFreeCell
14   while( compArray[compArray.firstFreeCell] != empty ) {
15     ++compArray.firstFreeCell;
16   }
17   return slot;
18 }
19 //usage:
20   size_t perfHashValue;
21   dictFW_->walkPerfHash( dicPos2, *c, &perfHashValue );
22   dictFW_->getAnnotation( perfHashValue )
```

Abbildung 20: Methode `storeState` der Klasse `TransitionTable` ergänzt um Hashwert

Abbildung 20 zeigt den Code, mit dem ein Zustand tatsächlich in der *TransitionTable* gespeichert wird (nachdem `replaceOrRegister` ermittelt hat, dass der Zustand bisher noch kein Äquivalent hat und daher neu gespeichert werden muss). Dieser Code wurde bereits ähnlich in Abbildung 7 gezeigt, nur dass Übergänge hier als Objekte (`trans`) dargestellt sind). Außerdem wurde Zeile 10 hinzugefügt. Vereinfacht gesagt, wird in einer zusätzlichen Zelle (neben den bisher bekannten für Label und Nachfolgerzustand) auch noch der Hashwert für dieses Label gespeichert. Dieser setzt sich (wenn man den Codefragmenten oben folgt), also jeweils aus den Summen der Hashwerte aller lexikografisch kleineren Labels, die vom zugehörigen Zustand ausgehen, zusammen. Damit ist die gewünschte Implementierung erreicht.
Anmerkung: obige Codefragmente lassen natürlich einiges aus und können so nicht kompiliert werden!

7 Approximatives Matching

Lexikonautomaten, wie in den vorherigen Abschnitten beschrieben, ermöglichen es, für ein gegebenen Wort zu ermitteln, ob dieses in exakt der vorhandenen Schreibweise auch im Lexikon existiert (*lookup*). In der Praxis stellt sich jedoch häufig eine andere Anforderung: für ein vorhandenes Wort, welches als solches nicht im Lexikon existiert, sollen möglichst passende Korrekturkandidaten gefunden werden. “Passende” Korrekturkandidaten sind dabei zunächst einmal solche, die einen möglichst geringen Editierabstand zum gesuchten Wort aufweisen. Als Abstandsmaß wird hier der Levenshtein-Abstand gewählt (siehe Punkt 7.1). Mögliche Anwendungen hierfür sind z.B. Rechtschreibkorrektur bei Texteditoren, Korrektur von Suchanfragen bzw. Vorschlagssuche, oder die Ermittlung von Korrekturkandidaten für

fehlerhaft digitalisierte Wörter. Der folgende Abschnitt stellt ein in [MS04] beschriebenes Tool vor, welches – gemeinsam mit einem Lexikonautomaten – genau solche Korrekturkandidaten findet. Genauer gesagt, wird ein Lösungsvorschlag für folgendes algorithmisches Problem gegeben:
Gegeben ein Pattern P, ein Lexikon D und ein kleiner Grenzwert k, soll auf effiziente Weise die Menge aller Einträge W in D ermittelt werden, so dass der Levenshtein-Abstand zwischen P und W nicht größer als k ist[MS04, 2].

7.1 Der Levenshtein-Abstand

Definition 20 (Abstandsmaß)**.** [Sch06, 53] Eine *Metrik* oder ein *Abstandsmaß* auf M ist eine reellwertige Funktion d auf M für die gilt:

$$\begin{aligned} d(x,y) \geq 0 \quad \forall x,y \in M & \quad \text{Nicht-Negativität} \\ d(x,y) = 0 \Leftrightarrow x = y \quad \forall x,y \in M & \quad \text{Null-Eigenschaft} \\ d(x,y) = d(y,x) \quad \forall x,y \in M & \quad \text{Symmetrie} \\ d(x,z) \leq d(x,y) + d(y,z) \quad \forall x,y,z \in M & \quad \text{Dreiecks-Ungleichung} \end{aligned}$$

Definition 21 (Levenshtein-Abstand)**.** [MS04, 5] Seien P und W Wörter über dem Alphabet Σ. Der (Standard-)Levenshtein-Abstand zwischen P und W, geschreiben $d_L(P,W)$, ist die minimale Anzahl primitiver Editieroperationen (Ersetzungen, Löschungen, Einfügungen) die notwendig sind, um P in W umzuwandeln.

Der Levenshtein-Abstand zweier Wörter P und W kann dabei mittels dynamischer Programmierung wie folgt ermittelt werden (siehe [WF74] aus [MS04, 6]):

$$d_L(\epsilon, W) = |W| \tag{1}$$

$$d_L(P, \epsilon) = |P| \tag{2}$$

$$d_L(Pa, Wb) = \begin{cases} d_L(P,W) & \text{falls } a = b \\ 1 + min(d_L(P,W), d_L(Pa,W), d_L(P,Wb)) & \text{falls } a \neq b \end{cases} \tag{3}$$

Die beiden ersten Zeilen der Formel entsprechen der Intuition, dass aus dem leeren Wort durch $|W|$ Einfügungen (korrekter Symbole) das Wort W entstehen kann, bzw., dass aus P durch $|P|$ Löschungen das leere Wort entsteht. Die darauffolgende Formel definiert die sonstige (rekursive) Berechnung des Levenshtein-Abstandes. Im ersten Fall kann der Editierabstand zweier Wörter Pa und Wb auf den Editierabstand zwischen P und W zurückgeführt werden (falls a und b gleich sind). Im anderen Fall ist zumindest ein Fehler aufgetreten. Je nachdem, ob es sich um eine Ersetzung, eine Einfügung oder eine Löschung handelt, wird zudem der Editierabstand $d_L(P,W), d_L(Pa,W), d_L(P,Wb)$ hinzuaddiert.
Zur Verdeutlichung obiger Formel sei ein Rekursionsschritt gezeigt. Angenommen, man müsste den Editierabstand $d_L(hau, haus)$ berechnen. Laut obiger Formel wäre die dritte Zeile anzuwenden, und hier wieder der Fall $a \neq b$, da $u \neq s$. Nun ergibt sich:

$$\begin{aligned} d_L(hau, haus) &= 1 + min(d_L(ha, hau), d_L(hau, hau), d_L(ha, haus)) \\ &= 1 + min(1, \mathbf{0}, 2) \text{ (zur Berechnung wird hier rekursiv weitergearbeitet)} \\ &= 1 \text{ (wie erwartet muss die Formel für eine Einfügung benutzt werden)} \end{aligned}$$

Für ein Pattern $P = p_1 \dots p_m$ und ein Wort $W = w_1 \dots w_n$ kann man zur Anwendung dieser Methode ein Matrix $T_L(P,W)$ der Größe $(m+1) \times (n+1)$ aufbauen. Von oben

		h	c	h	o	l	d
	0	1	2	3	4	5	6
c	1	1	1	2	3	4	5
h	2	1	2	1	2	3	4
o	3	2	2	2	1	2	3
l	4	3	3	3	2	1	2
d	5	4	4	4	3	2	1

Abbildung 21: Berechnung der Levenshtein-Distanz zwischen *chold* und *hchold* mittels Matrix und dynamischer Programmierung (Abbildung aus [MS04, 6], eigene Hervorhebungen hinzugefügt)

nach unten und von links nach rechts vorgehend, befüllt man nun die Zelle (i, j) jeweils mit $d_L(p_1 \dots p_i, w_1 \dots w_j)$ (mit $0 \leq i \leq m, 0 \leq j \leq m$) ([WF74] aus [MS04, 6]).
Abbildung 21 zeigt die Berechnung des Editierabstandes zwischen *chold* und *hchold*.
Wichtig ist, dass für das oben genannte Verfahren stets gewährleistet ist, dass ein einmal ersetzter, gelöschter oder hinzugefügter Buchstabe nicht erneut von einem Fehler betroffen sein kann.
Die Zeitkomplexität von $O(m \cdot n)$, die zur Berechnung der gesamten Tabelle notwendig wäre, kann reduziert werden, wenn maximal k Fehler zugelassen werden. Hier reicht es, Diagonalen in einem Band von $2k + 1$ zu berechnen (in der Tabelle hellgrau hinterlegt – Erläuterung siehe [Ukk85]).

7.2 Erweiterungen

Die oben aufgeführten Beispiele gehen von gleichen Kosten für jede der primitiven Editieroperationen, unabhängig von den beteiligten Symbolen aus. Es sind jedoch auch Kostenfunktionen denkbar, bei denen symbolabhängige Kosten definiert werden. So scheint es z.B. in der OCR naheliegender, dass z.B. ein i mit einem l verwechselt wird, als das ein k zu einem g wird. Das oben erwähnte Verfahren kann auch an solche Varianten angepasst werden. Dabei muss jedoch weiterhin die Dreiecksungleichung $d(x, z) \leq d(x, y) + d(y, z)$ gewährleistet sein (die Symmetrie ist u.U. nicht mehr gegeben, da man sich z.B. vorstellen kann, dass die Umwandlung eines i in ein l weniger kostet als die Umwandlung von l in i).
Neben den primitiven Editieroperationen Einfügung, Löschung und Ersetzung können außerdem noch Transpositionen, Verschmelzungen (zweier Buchstaben zu einem) oder Splits (eines Buchstabens in zwei) berücksichtigt werden. Insbesondere die letzten beiden Operationen kommen in der OCR häufig vor – man denke nur an das Umformen von u in ii.
Näheres zu verallgemeinerten Kostenfunktionen und Verfahren ist z.B. in [Sch06, 58f] zu finden. Im Folgenden werden jedoch weiterhin nur die primitiven Editieroperationen mit Kostenfunktion 1 betrachtet.

7.3 Approximative Suche eines Patterns in einem Text

Mit einem ähnlichen Verfahren lässt sich auch für einen Text T die Menge der Substrings T' ermitteln, die eine bestimmte Ähnlichkeit zu einem Pattern P aufweisen (siehe [MS04, 8f]). Für einen Text $T = t_1 \dots t_n$ und ein Pattern $P = p_1 \dots p_m$ wird dabei wie bisher eine $(m + 1) \times (n + 1)$ Matrix $T_{AST}(P, T)$ aufgebaut. Das Schema wird nun insoweit verändert, als das die erste Zeile der Matrix nun mit Nullen befüllt wird. In der Folge entspricht jeder Eintrag h in Zelle (i, j) dem minimalen Levenshtein-Abstand zwischen $p_1 \dots p_i$ und einem

Substring aus dem Text, der an Position t_j endet. Um nun alle passenden Substrings zu finden, die nicht mehr als k Editieroperationen von P abweichen, werden alle Positionen j ausgegeben, so dass der Eintrag an Zelle (m, j) nicht größer als k ist.

		t	h	i	s	_	c	h	i	l	d
	0	0	0	0	0	0	0	0	0	0	0
c	1	1	1	1	1	1	0	1	1	1	1
h	2	2	1	2	2	2	1	0	1	2	2
o	3	3	2	2	3	3	2	1	1	2	3
l	4	4	3	3	3	4	3	2	2	1	2
d	5	5	4	4	4	4	4	3	3	2	1

Abbildung 22: Approximative Suche des Patterns *chold* in einem Text mit dynamischer Programmierung (Abbildung aus [MS04, 7])

Das hier skizzierte Vorgehen ist jedoch aufwendig. Weniger aufwendige Verfahren nutzen einen nicht-deterministischen Automaten $A_{AST}(P)$, der als Sprache alle Wörter mit Levenshtein-Abstand $\leq k$ zum Pattern P hat (weitere Ausführungen dieses Unterabschnitts entnommen aus [MS04, 7f]). Abbildung 23 stellt einen solchen *Stockwerksautomaten* dar. Dabei bezeichnet die Nummer eines Zustandes stets die Position im Pattern, während der Exponent die Anzahl bisher gemachter Fehler repräsentiert. In Zustand 3^1 befindet man sich also an Position 3 im Pattern und hat bis jetzt (höchstens) einen Fehler gemacht. Im Automaten repräsentieren horizontale Übergänge solche Übergänge, bei denen das im Text gelesene Symbol dem nächsten Patternsymbol entspricht. Vertikale Übergänge zeigen eine Einfügung an. Diagonale Übergänge mit einem Symbol $\sigma \in \Sigma$ deuten Ersetzungen an, wohingegen diagonale Übergänge mit ϵ für Löschungen stehen. Finalzustände sind all jene Zustände an der Endposition des Patterns, in unserem Fall also 5^0, 5^1 und 5^2. Da der Automat nicht-deterministisch ist, kann ein String also an mehreren Finalzuständen im Automat akzeptiert werden (z.B. könnte man in unserem Fall durchaus den Text *chold* lesen, jedoch zwischenzeitlich statt eines horizontalen Übergangs auch einen diagonalen nehmen, und so den Text *chold* an Zustand 5^1 oder 5^2 akzeptieren). Um also die Anzahl tatsächlich gemachter Fehler am Automaten abzulesen, muss jeweils der Finalzustand mit dem niedrigsten Exponenten ermittelt werden, der noch erreicht werden kann.
Die Matrix $T_{AST}(P, T)$ und der NDEA $A_{AST}(P, k)$ stehen in Beziehung zueinander. In Abbildung 22 sind jene Zellen markiert, die nach Lesen von *th* aktiv sein können. Diese weisen respektive 0, 1, 1 und 2 Fehler auf (je nachdem, wie weit man im Pattern voranschreitet). Analog dazu sind jene Zustände in Abbildung 23 grau hinterlegt, die nach Lesen von *th* aktiv sein können. Die Exponenten der jeweils untersten dieser Zustände in den Spalten i zeigen dabei die Anzahl Fehler an, die gemacht wurden, um mit dem gelesenen Text bis zur Position i im Pattern vorzudringen. Auch hier entspricht die Folge dieser Exponenten aus den markierten Zuständen 0, 1, 1, 2.

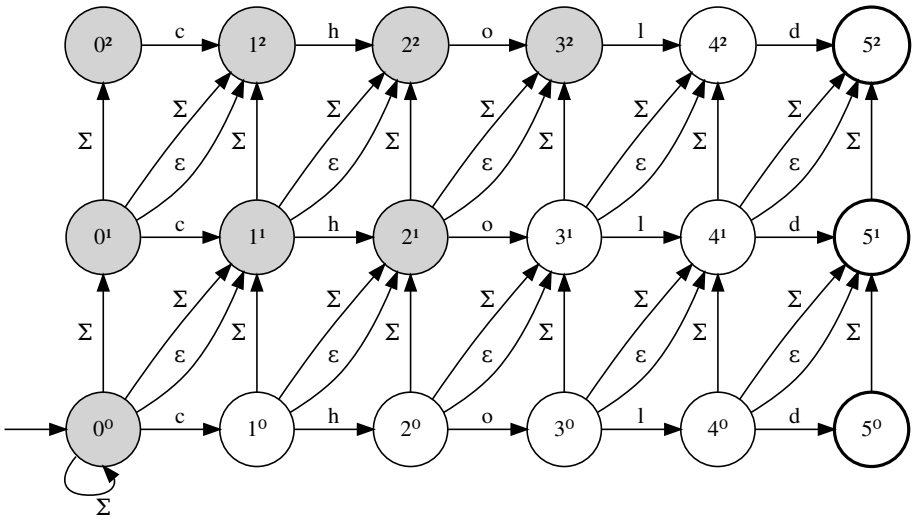


Abbildung 23: Stockwerksautomat $A_{AST}(chold, 2)$ für *chold* und maximalen Editierabstand 2 (aus [MS04, 7])

7.4 Universelle Levenshtein-Automaten

Indem man die Σ-Schleife am Startzustand entfernt, kann der oben gezeigte Automat $A_{AST}(P, k)$ zu einem Automaten $A(P, k)$ angepasst werden, der für ein gegebenes Pattern P und ein Wort W feststellt, ob der Editierabstand zwischen P und W höchstens k beträgt. Wie in Abbildung 24 ersichtlich, sind Finalzustände (fett umrandet) nun all jene Zustände, von denen aus man mit beliebig vielen ϵ-Übergängen noch zu einem Zustand in Spalte m gelangen kann.

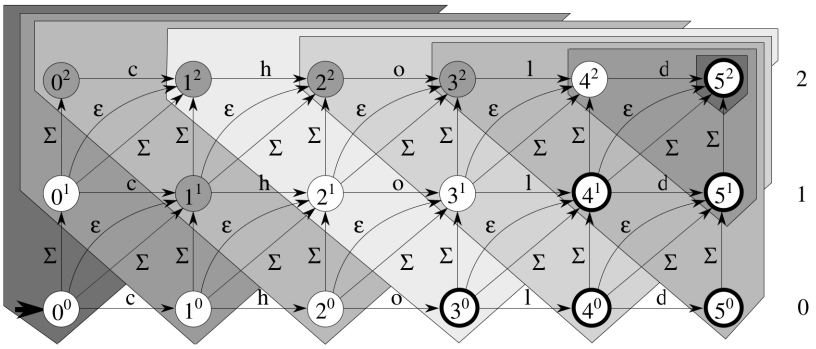


Abbildung 24: NDEA $A(chold, 2)$ für *chold* und maximalen Editierabstand 2 (aus [MS04, 7])

Mihov und Schulz stellen nun einen Ansatz vor, bei dem für eine ausreichend kleine Fehlerschranke k die explizite Berechnung des obigen Stockwerksautomaten vermieden werden kann ([MS04]). Dabei wird für den Vergleich zwischen einem Wort W und einem Pattern P zunächst eine Sequenz sogenannter *charakteristischer* Vektoren berechnet (siehe unten),

welche dann verwendet wird, um den universellen Levenshtein-Automaten $A^{\forall}(k)$ zu traversieren (die Ausführungen dieses Unterabschnitts lassen sich genauer nachlesen in [MS04, 9f]).

Definition 22 (charakteristischer Vektor)**.** ([MS04, 9]) Der *charakteristische Vektor* $\overrightarrow{\chi}(w, V)$ für ein Symbol $w \in \Sigma$ in einem Wort $V = v_1 \dots v_n \in \Sigma^*$ ist der Bitvektor der Länge n, bei dem das i-te Bit auf 1 gesetzt ist, genau dann wenn $w = v_i$.

Definition 23 (Dreiecksbereich)**.** ([MS04, 9]) Sei P ein Pattern der Länge m. Der *Dreiecksbereich eines Zustandes* p aus $A(P, k)$ besteht aus allen Zuständen q von $A(P, k)$, die von p aus erreicht werden können, indem eine (ggf. leere) Folge von u Übergängen nach oben genutzt wird, und außerdem $h \leq u$ horizontale oder umgekehrt horizontale (d.h. nach links führenden) Übergänge. Sei $0 \leq i \leq m$. Mit *Dreiecksbereich* i ist der Dreiecksbereich des Zustandes i^0 gemeint. Für $j = 1, \dots, k$ ist mit *Dreiecksbereich* $m+j$ der Dreiecksbereich des Zustandes m^j gemeint.

Für ein Wort $W = w_1 \dots w_n$ wird die Menge der Zustände von $A(P, k)$, welche nach dem Lesen des i-ten Symbols w_i von W noch aktiv ist, stets eine Teilmenge des Dreiecksbereich i darstellen (für $0 \leq i \leq min\{n, m + k\}$) bzw. leer sein, falls $i > m + k$ ([MS04, 10]). Welche Zustände innerhalb des Dreiecksbereich aktiv sind, hängt dabei maßgeblich von zwei Faktoren ab ([MS04, 10]):

- welche Zustände zuletzt (nach Lesen von $w_1 \dots w_{i-1}$) aktiv waren
- welcher charakteristische Vektor $\overrightarrow{\chi}(w_i, p_l \dots p_r)$ gerade gelesen wurde (mit $l = max\{1, i-k\}$, $r = min\{m, i + k\}$).

Analog zur Beziehung zwischen $A_{AST}(P, k)$ und $T_{AST}(P)$ existiert ein ähnlicher Zusammenang zwischen dem NDEA $A(P, k)$ und der Matrix $T_L(P, W)$ ([MS04, 9]). In Abbildung 24 sieht man die Zustände (dunkel hinterlegt), die nach Lesen der zwei Symbole *ch* aktiv sind. Erwartungsgemäß befinden sie sich im Dreiecksbereich 2. Wieder entsprechen die Exponenten 2, 1, 2, 2 der jeweils untersten aktiven Zustände einer jeden Spalte den Einträgen in der entsprechenden Spalte aus Abbildung 21 (ebenfalls dunkler hinterlegt). Der Dreiecksbereich i entspricht dabei dem Teilbereich im Diagonalband der Breite $2k + 1$ in der i-ten Spalte der Matrix (statt einer Rechts-Links-Ausrichtung wie im NDEA weist die Matrix eine Ausrichtung von oben nach unten auf).
Es bleibt zu klären, wie sich einerseits die *charakteristischen Vektoren* berechnen lassen, und andererseits, wie *Zustände* im universellen Levenshtein-Automaten $A^{\forall}(k)$ ermittelt werden.

7.4.1 Charakteristische Vektoren

(aus [MS04, 10]). Um den universellen Levenshtein-Automaten zu benutzen, sollen die berechneten Vektoren stets eine Länge $\leq 2k + 1$ haben. Um die Länge der Vektoren zu standardisieren, wird an den Beginn des Patterns P eine Folge von speziellen, nicht im Alphabet vorhandenen Symbolen angefügt. Genauer wird $p_0 = p_{-1} = \dots = p_{k-1} := \$$ gesetzt. Eine zweite Änderung ergibt sich aus folgendem Grund: solange man sich in einem Dreiecksbereich i befindet, für den $i \leq m - k - 1$ gilt, ist mit Lesen von w_i noch kein Dreiecksbereich erreicht worden, der Finalzustände enthält (sonst wäre ja ein Zustand final, für den mindesten $k + 1$ Fehler gelten). Um diese Information zu speichern, wird an jeden Vektor eine weitere Position $i + k + 1$ gehängt, solange $i + k + 1 \leq m$. Für $0 \leq i \leq m - k - 1$ haben charakteristische Vektoren also die Länge $2k + 2$ und nehmen für alle späteren Positionen kontinuierlich ab.

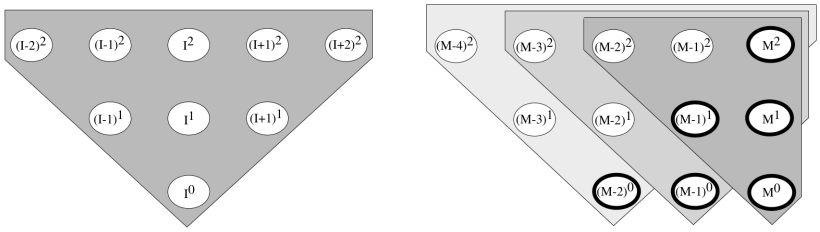


Abbildung 25: Symbolische Dreiecksbereiche und symbolische Finalpositionen für $k = 2$ (aus [MS04, 11])

7.4.2 Zustände

(aus [MS04, 11f]). Zur Bezeichnung der Zustände im universellen Levenshtein-Automaten werden sogenannte *symbolische* Dreiecksbereiche eingeführt. Statt wie bisher explizite Zustandsnummern zu nennen, wird ein sogenannter *I-Bereich* eingeführt für all jene Dreiecksbereiche, die keine Finalzustände enthalten, und $k + 1$ *M-Bereiche* für die Dreiecksbereiche aus $A(P, k)$, die Finalzustände enthalten. I (für *Integer*) abstrahiert dabei von konkreten Positionen i im Pattern, während M für die variable Patternlänge m steht. Man kann nun *symbolische Zustandsnummern* der Form $I, I + 1, I - 1 \ldots$ bzw. $M, M - 1, \ldots$ nutzen. Weiterhin zeigen Exponenten die Anzahl gemachter Fehler an. Ein Beispiel symbolischer Dreiecksbereiche findet sich in Abbildung 25.
Zustände aus $A^\forall(k)$ werden nun als Teilmengen symbolischer Dreiecksbereiche konstruiert. Jede solche Teilmenge, die eine Finalposition enthält, wird zum Finalzustand. $\{I^0\}$ stellt den Startzustand dar.
Nicht jede mögliche Teilmenge eines Dreiecksbereichs wird jedoch zum eigenen Zustand. Dreiecksbereiche in $A(P, k)$ enthalten oft Positionen $p = g^e$ und $q = h^f$, bei denen q von p *subsumiert* wird. Dies bedeutet, dass, sofern man mit einem fixen Restinput U von q zu einer Finalposition kommt, dies auch von p mit Lesen von U gelingt. Ebenso lassen sich Subsumtionen für symbolische Dreiecksbereiche definieren. Zustände in $A^\forall(k)$ werden dann nur solche Teilmengen symbolischer Dreiecksbereiche, die subsumtionsfrei sind. So wird z.B. für $A^\forall(1)$ nur der Zustand $\{I^0\}$ den symbolischen Dreiecksbereich I^0 enthalten, nachdem jede der symbolischen Positionen $(I - 1)^1$, I^1 und $(I + 1)^1$ von I^0 subsumiert wird.

7.4.3 Übergangsfunktion

(aus [MS04, 12]). Im Folgenden wird die Grundidee zur Berechnung der Übergangsfunktion $\delta^\forall$ beschrieben. Sei $A(P, k)$ ein Stockwerksautomat für ein Pattern P der Länge m. Sei $W = w_1 \ldots w_n$ das Eingabewort. Sei S_i^P die Menge aktiver Positionen aus $A(P, k)$ die erreicht werden, nachdem das i-te Symbol w_i $(1 \leq i \leq n)$ gelesen wird. In $A^\forall(k)$ gibt es eine parallele Akzeptanzprozedur bei der z.B. $S_i^\forall$ nach Lesen von $\overrightarrow{\chi}(w_i, p_{i-k} \ldots p_i \ldots p_r)$ (mit $r = min\{m, i + k + 1\}$) für $1 \leq i \leq n$ erreicht wird. Nun sind Übergänge folgendermaßen definiert:

$C1$ für alle parallelen Mengen S_i^P und $S_i^\forall$ zweier Sequenzen

$$S_0^P \; S_1^P \; \ldots \; S_i^P \; \ldots \; S_n^P$$
$$S_0^\forall \; S_1^\forall \; \ldots \; S_i^\forall \; \ldots \; S_n^\forall$$

wird die Menge S_i^P von $S_i^\forall$ abgeleitet, indem I mit i instanziert wird, sofern $S_i^\forall$ die Variable I nutzt, und andernfalls M mit m instanziert.

$C2$ immer wenn S_i^P eine Finalposition enthält, ist $S_i^\forall$ final.

7.4.4 Beispiele

(aus [MS04, 12]) Der universelle Levenshtein-Automat für Editierabstand 1 ist in Abbildung 26 dargestellt. Dabei symbolisiert ein "_" ein beliebiges Bit (0 oder 1). Geklammerte Bits sind optional. Als Beispiel sei das Pattern *chold* und das Eingabewort *child* gewählt. Daraus lassen sich nun folgende charakteristische Vektoren berechnen:

- $\overrightarrow{\chi}_1 = \overrightarrow{\chi}(c, \$cho) = 0100$
- $\overrightarrow{\chi}_2 = \overrightarrow{\chi}(h, chol) = 0100$
- $\overrightarrow{\chi}_3 = \overrightarrow{\chi}(i, hold) = 0000$
- $\overrightarrow{\chi}_4 = \overrightarrow{\chi}(l, old) = 010$
- $\overrightarrow{\chi}_5 = \overrightarrow{\chi}(d, ld) = 01$

Mit diesen Vektoren erreicht man, ausgehend vom Startzustand I^0, nacheinander die Zustände $\{I^0\}, \{I^0\}, \{(I-1)^1, I^1\}, \{I^1\}, \{M^1\}$. Daher wird *child* akzeptiert. Ähnlich erhielte man für das Eingabewort *cold* die charakteristischen Vektoren 0100-0010-0010-001 und erreicht damit vom Startzustand aus folgende Zustände: $\{I^0\}, \{(I-1)^1, I^1, (I+1)^1\}, \{(I+1)^1\}$ und $\{M^1\}$. Auch *cold* wird daher akzeptiert. Schließlich wird *hchold* in die Sequenz 0010-1000-1000-100-10-1 übersetzt, was folgende Zustandsfolge ergibt: $\{(I-1)^1, I^1, (I+1)^1\}, \{(I-1)^1\}, \{(I-1)^1\}, \{(I-1)^1\}, \{(I-1)^1\}, \{M^1\}$. Auch *hchold* wird also akzeptiert.

7.5 Ermittlung von Korrekturkandidaten

(aus [MS04, 14]). Im Folgenden wird ein Algorithmus vorgestellt, mit dem anhand eines Lexikons D (z.B. implementiert als minimierter Automat $A_D = \langle \Sigma, Q^D, q_0^D, F^D, \delta^D \rangle$ wie weiter oben beschrieben) und des universellen Levenshtein-Automaten $A^\forall(k) = \langle \Gamma, Q^\forall, q_0^\forall, F^\forall, \delta_\forall \rangle$ für einen Fehlerschranke k alle Korrekturkandidaten für ein Pattern P ermittelt werden können. Dabei sei angenommen, dass für jedes $\sigma \in \Sigma$ und jeden Index $1 \leq i \leq m + k$ der charakteristische Vektor $\overrightarrow{\chi}(\sigma, p_{i-k} \dots p_i \dots p_r)$ (mit $r = min\{m, i+k+1\}$) in konstanter Zeit ermittelt werden kann (in Abbildung 27 wird gezeigt, wie eine solche Berechnung implementiert werden könnte). Nun werden die beiden Automaten A_D und $A^\forall$ parallel durchlaufen, indem ein Standard *Backtracking*-Verfahren verwendet wird. Bei jedem Schritt wird das aktuell in A_D gelesene Symbol σ (das i-te Symbol auf dem aktuellen Lexikonpfad) in einen Bitvektor der Form $\overrightarrow{\chi}(\sigma, p_{i-k} \dots p_i \dots p_r)$ (mit $r = min\{m, i+k+1\}$) übersetzt, der als Eingabe für $A^\forall(k)$ dient.

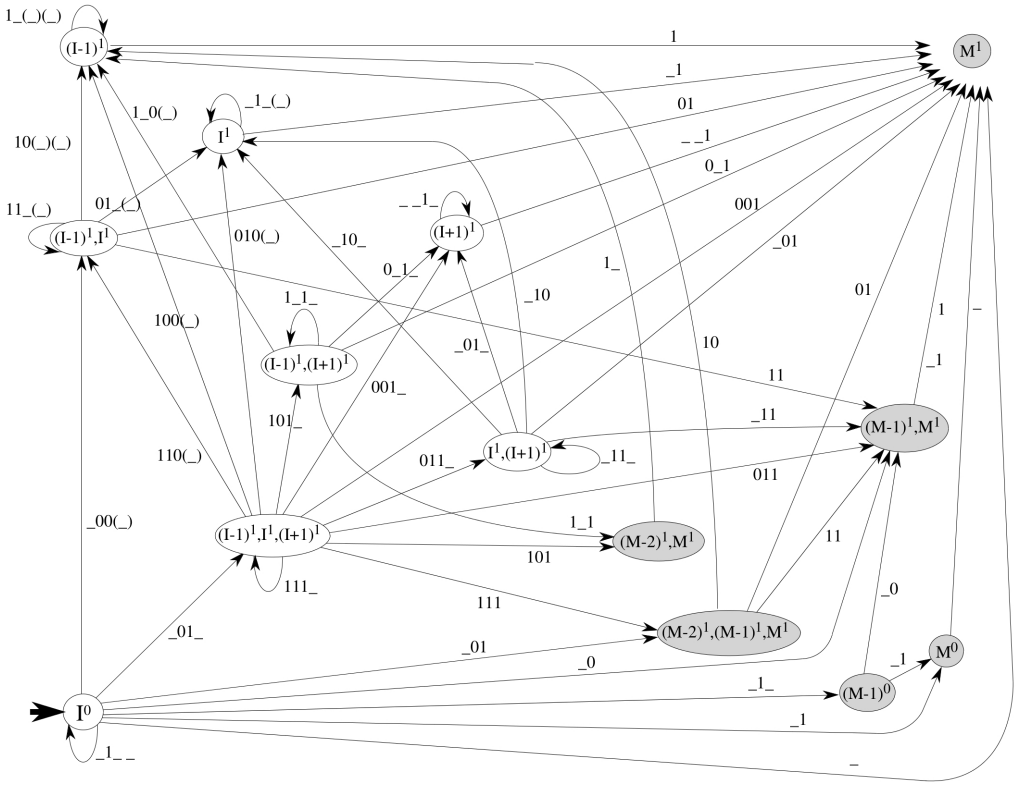


Abbildung 26: Universeller Levenshtein-Automat für Abstand 1 (aus [MS04])

```
1: push(< 0, ϵ, q_0^D, q_0^∀ >);
2: while not empty(stack) do
3:     pop(< i, W, q^D, q^∀ >);
4:     for σ in Σ do
5:         χ⃗ := χ⃗(σ, p_{i−k} ... p_i ... p_r);
6:         q_1^D := δ^D(q^D, σ);
7:         q_1^∀ := δ_∀(q^∀, χ⃗)
8:         if (q_1^D ≠ NIL) and (q_1^∀ ≠ NIL) then
9:             W_1 :=concat(W, σ);
10:            push(< i + 1, W_1, q_1^D, q_1^∀ >);
11:            if (q_1^D ∈ F^D) and (q_1^∀ ∈ F^∀) then output (W_1);
12:            end if
13:        end if
14:    end for
15: end while
```

Man beginnt also mit dem Zustandspaar $\langle q_0^D, q_0^\forall \rangle$ der Startzustände aus Lexikon- und Levenshtein-Automat, der Position $i = 0$ im Pattern und dem leeren Wort ϵ. Nun wird bei jedem Traversierungsschritt im Lexikon ein neues Symbol $\sigma \in \Sigma$ zum aktuell gelesenen Wort W hinzugefügt. Außerdem wird das Zustandspaar $\langle q^D, q^\forall \rangle$ im nächsten Schritt durch $\langle \delta^D(q^D, \sigma), \delta_\forall(q^\forall, \overrightarrow{\chi}) \rangle$ ersetzt, sofern beide Zustände im neu berechneten Zustandspaar nicht

dem (jeweiligen) Fallenzustand `NIL` entsprechen. Sobald man sowohl im Levenshtein- als auch im Lexikonautomaten einen Finalzustand erreicht, wird das aktuell gelesene Wort W als passender Korrekturkandidat für das Pattern P ausgegeben. Jeder dieser Korrekturkandidaten hat einen maximalen Editierabstand $d_L(P, W) \leq k$ zum Pattern.
Die Zeitkomplexität dieses Algorithmus ist im Wesentlichen durch die Größe des Lexikonautomaten A_D sowie die Fehlerschranke k limitiert. Sollte k z.B. die Länge des längsten Lexikonwortes erreichen, muss das gesamte Lexikon traversiert werden. In der Regel wird die Fehlerschranke k jedoch klein gewählt sein[12]. Daher werden auch nur kleine Teile des Lexikons tatsächlich traversiert werden.

7.5.1 Beispielhafte Berechnung charakteristischer Vektoren

Im Folgenden wird beispielhaft gezeigt, wie charakteristische Vektoren im konkreten Programm berechnet werden können. Dazu sei vorab gesagt, das Bitvektoren intern als Ganzzahlwerte gespeichert werden, da dies die einfachste Repräsentation solcher Vektoren ist und C++ es erlaubt, alle bekannten Bit-Operationen (wie bitweises UND, bitweises ODER, Links- und Rechstshifts) auf Ganzzahlwerten auszuführen.
Daraus ergibt sich jedoch ein Nachteil: es ist nun nicht mehr einfach möglich, "kürzere" von "längeren" Vektoren zu unterscheiden. Möchte man z.B. den Vektor 001110 darstellen, so verwendet man die Zahl 14. Diese repräsentiert jedoch auch den kürzeren Vektor 1110. Um solche Längenunterschiede zu kennzeichnen, behilft man sich mit einem einfachen Trick: "kürzere" Vektoren erhalten sogenannte *leading Bits* (werden als Ganzahl also größer) an Stellen, die normalerweise nicht gesetzt sein könnten. Wenn ein Vektor, welcher nicht das Wortende kennzeichnet, also die Länge $2k + 2$ haben muss, so muss ein "kürzerer" Vektor nun die Länge $2k + 3$ bekommen.
In Abbildung 27 werden relevante Programmauszüge zur Vektorenberechnung gezeigt (auch hier kompiliert der Programmcode so nicht). Zunächst werden mittels `typedef` eigene Datentypen definiert. Dies dient vorallem der besseren Lesbarkeit. Zeile 2 definiert eine 64-Bit Zahl, wohingegen Zeile 3 eine 32-Bit Zahl (die übliche Speichergröße für Integer) definiert. Die 64-Bit Zahl wird dazu benötigt, um für ein neues Pattern die charakteristischen Vektoren zu berechnen, welche noch nicht auf eine Länge $2k + 1$ eingeschränkt sind (ein Pattern darf also nicht mehr als 64 Buchstaben haben, was jedoch für unsere Zwecke vollauf genügt). Einige wiederholt benötigte Zahlen werden statisch definiert. Dies sind `zff` (Zeile 6), eine Zahl, deren Binärrepräsentation aus 64 Einsern besteht und `z10` (Zeile 8), eine Zahl, deren Binärrepräsentation mit Eins beginnt und mit 63 Nullen endet. `zff` kann dazu verwendet werden, eine korrekte Anzahl *leading bits* zu definieren, während `z10` dazu dient, die Bits im jeweiligen Vektor zu setzen (siehe unten).
Im Konstruktor werden nun weitere Zahlen definiert (um diese nicht stets neu berechnen zu müssen), nämlich `z2k1` und `z2k3` (Zeilen 12 und 16), jeweils eine Sequenz aus $2k + 1$ bzw. $2k + 3$ Einsen. Diese Zahlen ermöglichen es, mit bitweisem AND andere Zahlen auf die gewünschte Länge ($2k + 1$ oder $2k + 3$) zu kürzen.
Die ungekürzten Vektoren werden nun in einem `std::vector` von 64-Bit-Zahlen namens `charvecs_` gespeichert, der zunächst mit 65536 Nullen (für die möglichen Unicode Zeichen) initalisiert wird (Zeile 20). Wenn ein Pattern geladen wird, wird zunächst die Methode `calcCharvec` (Zeile 22) aufgerufen. Dort wird ein Bitvektor `c` zunächst auf `z10`, also eine Eins mit 63 Nullen gesetzt. Man iteriert nun über das gesamte Pattern (Zeile 25) und verschiebt für jeden Schritt im Pattern den Bitvektor `c` um eins nach rechts. Für jedes Symbol `*pat` an Stelle i im Pattern fügt man im Vektor `charvecs_` am entsprechenden

[12] In der Praxis werden meistens Schranken $k \leq 3$ betrachtet

Index (dem Unicode-Wert für `*pat`) per bitweisem ODER den aktuellen Wert von `c` ein. Taucht ein Symbol σ nun an mehreren Stellen auf, so wird an den entsprechenden Stellen auch jeweils ein Bit gesetzt sein. Bevor man ein neues Pattern lädt, muss man die belegten Indizes im Vektor auf Null setzen (wieder, indem man das Pattern symbolweise durchläuft). Die Methode `calc_k_charvec` in Zeile 30 zeigt nun, wie man für ein konkretes gelesenes Symbol `c` an einer Position `i` den entsprechend Vektorausschnitt berechnet.
Als Beispiel sei angenommen, wir würden das Pattern *highlight* gelesen haben, und genau *highlight* auch als Eingabewort bekommen. Die Vektoren nach der ursprünglichen Berechnung wären dann folgende:

- $\overrightarrow{\chi}_0 = \overrightarrow{\chi}(h, \$\$high) = 001001$
- $\overrightarrow{\chi}_1 = \overrightarrow{\chi}(i, \$highl) = 001000$
- $\overrightarrow{\chi}_2 = \overrightarrow{\chi}(g, highli) = 001000$
- $\overrightarrow{\chi}_3 = \overrightarrow{\chi}(h, ighlig) = 001000$
- $\overrightarrow{\chi}_4 = \overrightarrow{\chi}(l, ghligh) = 001000$
- $\overrightarrow{\chi}_5 = \overrightarrow{\chi}(i, hlight) = 001000$
- $\overrightarrow{\chi}_6 = \overrightarrow{\chi}(g, light) = 00100$
- $\overrightarrow{\chi}_7 = \overrightarrow{\chi}(h, ight) = 0010$
- $\overrightarrow{\chi}_8 = \overrightarrow{\chi}(t, ght) = 001$

Zeile 35 zeigt, wie der Vektor aus der Datenstruktur `charvecs_` gewonnen wird. Natürlich greift man auf den entsprechenden Index für das Symbol `c` zu. Um den korrekten Ausschnitt des Vektors zu bekommen, verschiebt man ihn nach rechts. Für die Position 0 und eine Fehlerschranke $k = 2$ würde man ihn um $64 - (2 + 1 + 0) = 61$ verschieben. Sucht man z.B. den Vektor für h an Position 0 im Pattern *highlight*, erhielte man aus $1001000 \ldots 0$ durch den Rechtsshift den Vektor 100. Da man sich nicht am Wortende befindet, wird dieser Vektor in Zeile 44 nochmal um eins nach links verschoben, man erhält also 1000. Da die letzten Bits im universellen Levenshtein-Automaten jedoch für nicht-finale Vektoren irrelevant sind, entspricht $001000 \simeq 001001 \simeq 00100\_$, was korrekt ist.
Für Vektoren, die das Patternende signalisieren, ist die Sache komplizierter. Betroffene Vektoren sind all jene, bei denen für die Patternlänge m gilt: $m - i \leq k + 1$. Für unser Beispiel ist dies ab Position 6 der Fall (Positionen starten bei 0). Zunächst wird nun ein Wert `highBits` berechnet, indem man die Zahl `zff` (64 Einser) um $(m - i + k + 1)$ nach links verschiebt und dann mit `z2k3` (hier 7 Einser) entsprechend kürzt. Für Position 6 erhielte man also 1000000, für Position 7 1100000, für Position 8 1110000 bis hin zu 1111100 für Position 10. Diese kann nur erreicht werden, wenn das Pattern bis dahin korrekt gelesen wurde, und dann weitere zwei Einfügungen erfolgt sind. Mehr Fehler darf man bei Fehlerschranke $k = 2$ nicht machen. Dennoch kann es natürlich vorkommen, dass man versucht, ein bis dato korrekt gelesenes Wort um 3 Buchstaben zu verlängern (z.B. nach *highlighting* für das Pattern *highlight* sucht). Da ein solcher Vektor nicht definiert ist, wird in diesem Fall der Wert von `highBits` wieder auf den kürzesten möglichen Nullvektor korrigiert (Zeile 40).
Zeile 41 zeigt, wie der konkrete verkürzte Vektor berechnet wird. Zunächst wird ein Rechtshift mit $k\_ + 1 - (m - i)$ durchgeführt. Damit wird der Tatsache Rechnung getragen, dass Bitvektoren am Wortende kürzer werden müssen (im obigen Beispiel sieht man, dass, obwohl stets der richtige Buchstabe gelesen wird, das entsprechend gesetzte Bit immer weiter

nach rechts wandert). Dann wird die oben berechnete Zahl `highBits` mit bitweisem ODER verknüpft.

```
1  //in header
2  typedef unsigned long long bits64;
3  typedef int bits32;
4
5  static const bits64 zff = 0xffffffffffffffffll;
6      // that's 64 '1's
7  static const bits64 z10 = 1ll << 63;
8      // that's a '1' followed by 63 '0's
9  std::vector< unsigned long long > charvecs_;
10
11 //in constructor
12 z2k1 = 1ll;
13 z2k1 <<=  2 * k_ + 1;
14 z2k1--; // a sequence of 2k+1 1-values
15
16 z2k3 = 1ll;
17 z2k3 <<= 2 * k_ + 3;
18 z2k3--; // a sequence of 2k+3 1-values
19
20 charvecs_( maxNrOfChars, 0 );
21
22 void calcCharvec() {
23   bits64 c;
24   const wchar_t* pat;
25   for ( c = z10, pat= pattern_; *pat; ++pat, c >>= 1 ) {
26       charvecs_[*pat] |= c;
27   }
28 }
29
30 bits32 calc_k_charvec( wchar_t c, int i ) const {
31   bits64 r;
32   // after the next line,
33   // the bits i, i+1, i+2 of chv are the lowest bits of r.
34   // All other bits of r are 0
35   r = ( charvecs_[c] >> ( 64 - ( k_ + 1 + i ) ) ) & z2k1;
36   if ( patLength_ - i <= k_ + 1 ) {
37         // the last few chars of the word
38     int highBits = ( zff << ( (patLength_ - i + k_ + 1 ) & z2k3 );
39     if (highBits >= z2k3 - 1)
40       highBits -=2;
41       r = ( ( r >> ( k_ + 1 - ( patLength_ - i ) ) ) | highBits );
42     }
43   else {
44     r = r << 1;
45   }
46   return ( (bits32) r );
47 }
```

Abbildung 27: Codebeispiele zur Berechnung charakteristischer Vektoren

Die weitere Benutzung des universellen Levenshtein-Automaten ist denkbar einfach. Es genügt, die wie oben berechneten charakteristischen Vektoren zum Traversieren zu verwenden: als Rückgabewerte einer solchen $\delta_\forall$-Methode erhält man entweder einen neuen Zustand im Levenshtein-Automaten, oder aber einen Default-Wert für den Fallenzustand (in diesem Fall akzeptiert der Automat nicht und die Suchprozedur für das gerade gelesene Eingabewort kann beendet werden). Etwas gänzlich anderes wäre es natürlich, den Automaten selber zu berechnen!

Abbildungsverzeichnis

Literatur

[CLRS07] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, und Clifford Stein. *Algorithmen - Eine Einführung*. Oldenbourg Verlag, München, 2007.

[DMS05] Jan Daciuk, Denis Maurel, und Agata Savary. *Dynamic Perfect Hashing with Finite-State Automata*. In: MieczysławA. Kłopotek, SławomirT. Wierzchoń, und Krzysztof Trojanowski, (Hrsg.), Intelligent Information Processing and Web Mining, Bd. 31 von *Advances in Soft Computing*, Seiten 169–178. Springer Berlin Heidelberg, 2005.

[DMWW00] Jan Daciuk, Stoyan Mihov, Bruce W. Watson, und Richard E. Watson. *Incremental Construction of Minimal Acyclic Finite State Automata*. Computational Linguistics, 26(1):3–16, March 2000.

[Had10] Max Hadersbeck. *Programmierung mit C++ für Computerlinguisten*. Skript, März 2010. Skript zum gleichnamigen Seminar.

[MS04] Stoyan Mihov und Klaus U. Schulz. *Fast Approximate Search in Large Dictionaries*. Computational Linguistics, 30(4):451–477, December 2004. Version aus `www.cis.uni-muenchen.de/download/publikationen/fastapproxsearch.pdf`.

[Per12] Estelle Perez. *Effiziente Berechnung von Fehler- und Sprachprofilen*. Magisterarbeit, 2012.

[Ref10] Ulrich Reffle. *Implementation of large dictionary automata*. Skript-Entwurf, Juli 2010. Skript-Entwurf zum Seminar *Implementieren von endlichen Automaten*.

[Sch06] Klaus U. Schulz. *Nachkorrektur von Ergebnissen eine Optischen Charaktererkennung*. Skript, October 2006. Skript zum Kurs OCR-Nachkorrektur.

[Sch08a] Uwe Schöning. *Theoretische Informatik - kurz gefasst*. Spektrum Verlag, Heidelberg, 2008.

[Sch08b] Klaus U. Schulz. *Formale Sprachen und Automaten*. Skript, October 2008. Skript zum gleichnamigen Seminar.

[TY79] Robert E. Tarjan und Andrew Chi-Chih Yao. *Storing a sparse table*. Commun. ACM, 22(11):606–611, November 1979.

[Ukk85] Esko Ukkonen. *Algorithms for approximate string matching*. Inf. Control, 64(1-3):100–118, March 1985.

[WF74] Robert A. Wagner und Michael J. Fischer. *The String-to-String Correction Problem*. J. ACM, 21(1):168–173, January 1974.