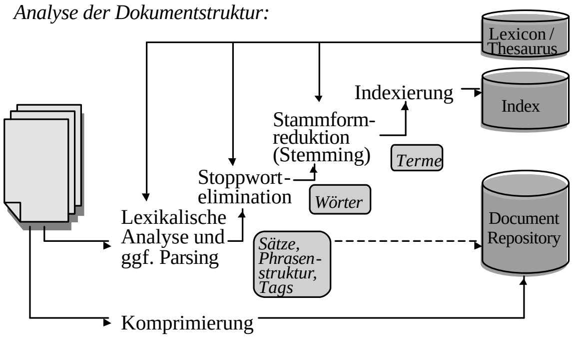
4 Textverarbeitung

Analyse der Dokumentstruktur:



Stoppwortelimination

Nachschlagen in Stoppwortliste
(ggf. unter Berücksichtigung korpusspezifischen Jargons
$\rightarrow$Lexikon/Thesaurus,
z.B. „Definition“ oder „Theorem“ bei Mathematik-Korpus)

Typische Stoppwörter im Englischen
(Artikel, Präpositionen, Konjunktionen, Pronomen,
„überladene“ Verben usw. – u.U. einige Hundert Stoppwörter):

a, also, an, and, as, at, be, but, by,
can, could, do, for, from, go,
have, he, her, here, his, how,
I, if, in, into, it, its,
my, of, on, or, our, say, she,
that, the, their, there, therefore, they,
this, these, those, through, to, until,
we, what, when, where, which, while, who, with, would,
you, your

Morphologische Reduktion (Lemmatisierung)

Reduktion auf grammatikalische **Grundform**:
bei Substantiven der Nominativ, bei Verben der Infinitiv,
Plural auf Singular, Passiv auf Aktiv, usw.
Beispiele:

- „Winden“ auf „Wind“, „Winde“ oder „winden“ je Phrasenstruktur und ggf. Kontext
- „finden“ und „gefundenes“ beides auf „finden“,
- „Gefundenes“ auf „Fund“

Reduktion auf linguistische **Stammform**:
Zurückverfolgen von Derivationen wie
Flexion (z.B. Deklination), Komposition, Substantivierung, usw.
Beispiele:

- „Flüssen“, „einflößen“ auf „Fluß“,
- „finden“ und „Gefundenes“ auf „finden“
- „Du brachtest ... mit“ auf „mitbringen“,
- „Schweinkram“, „Schweinshaxe“ und „Schweinebraten“ auf „Schwein“ usw.
- „Feinschmecker“ und „geschmacklos“ auf „schmecken“

Stammformreduktion (Stemming)

Ansätze:

- Nachschlagen in umfassendem Wörterbuch (Lexikon)
- Erkennung durch Analyse der linguistischen Struktur
- Affix-Entfernung:
  Entfernung von Präfixen und/oder Suffixen
  aufgrund von (heuristischen) Regeln
  Beispiel:
  stresses $\rightarrow$ stress, stressing $\rightarrow$ stress, symbols $\rightarrow$ symbol
  aufgrund von Regeln sses $\rightarrow$ ss, ing $\rightarrow \varepsilon$, s $\rightarrow \varepsilon$, usw.

Bemerkung:
Der Nutzen von Stemming im IR ist nicht unumstritten.
Beispiel:
Bill is operating a company.
On his computer he runs the ... operating system.

Stemming durch Suffixentfernung nach Porter

Notation: C – Konsonant, V – Vokal, L – beliebiger Buchstabe
sowie Wildcard # und Kleene-Star * bzw. +

1) select rule with longest suffix among *Plural*
{sses $\rightarrow$ ss; ies $\rightarrow$ i; s $\rightarrow \varepsilon$}

2) select rule with longest suffix among *Partizipien*
{if $C^*(V^+C^+)^+V^*$eed then eed $\rightarrow$ ee;
if #V#ed or #V#ing then
{select rule with longest suffix {ing $\rightarrow \varepsilon$; ed $\rightarrow \varepsilon$};
select rule with longest suffix among
{at $\rightarrow$ ate; bl $\rightarrow$ ble; iz $\rightarrow$ ize;
if #C_1C_2 and C_1=C_2 and $C_1 \notin$ {l, s, z} then $C_1C_2 \rightarrow C_1$;
if $C^*V^+C^+C_1V_1C_2$ and $C_2 \notin$ {w, x, y}
then $C_1V_1C_2 \rightarrow C_1V_1C_2$e }
}
}

Stemming durch Suffixentfernung nach Porter (2)

3) if #V#y then y $\rightarrow$ i;
if $C^*(V^+C^+)^+V^*$suffix then select rule with longest suffix among
{ational $\rightarrow$ ate; tional $\rightarrow$ tion; enci $\rightarrow$ ence; anci $\rightarrow$ ance; izer $\rightarrow$ ize;
abli $\rightarrow$ able; alli $\rightarrow$ al; entli $\rightarrow$ ent; eli $\rightarrow$ e; ousli $\rightarrow$ ous; ization $\rightarrow$ ize;
ation $\rightarrow$ ate; ator $\rightarrow$ ate; alism $\rightarrow$ al; iveness $\rightarrow$ ive; fulness $\rightarrow$ ful;
ousness $\rightarrow$ ous; aliti $\rightarrow$ al; iviti $\rightarrow$ ive; biliti $\rightarrow$ ble}

4) if $C^*(V^+C^+)^+V^*$suffix then select rule with longest suffix among
{icate $\rightarrow$ ic; ative $\rightarrow \varepsilon$; alize $\rightarrow$ al; iciti $\rightarrow$ ic; ical $\rightarrow$ ic; ful $\rightarrow \varepsilon$; ness $\rightarrow \varepsilon$}

5) if $C^*V^+C^+(V^+C^+)^+V^*$suffix then select rule with longest suffix among
{al $\rightarrow \varepsilon$; ance $\rightarrow \varepsilon$; ence $\rightarrow \varepsilon$; er $\rightarrow \varepsilon$; ic $\rightarrow \varepsilon$; able $\rightarrow \varepsilon$; ible $\rightarrow \varepsilon$;
ant $\rightarrow \varepsilon$; ement $\rightarrow \varepsilon$; ment $\rightarrow \varepsilon$; ent $\rightarrow \varepsilon$; ou $\rightarrow \varepsilon$; ism $\rightarrow \varepsilon$; ate $\rightarrow \varepsilon$;
iti $\rightarrow \varepsilon$; ous $\rightarrow \varepsilon$; ive $\rightarrow \varepsilon$; ize $\rightarrow \varepsilon$; if #sion or #tion then ion $\rightarrow \varepsilon$}

6) select rule with longest suffix among
{if $C^*V^+C^+(V^+C^+)^+V^*$e then e $\rightarrow \varepsilon$;
if $C^*V^+C^+V^*$e and not (#$C_1V_1C_2$e and $C_2 \notin$ {w, x, y}) then e $\rightarrow \varepsilon$}

7) if $C^*V^+C^+(V^+C^+)^+V^*$ll then ll $\rightarrow$ l;

Thesaurusaufbau und -verwaltung

Zu jedem **Konzept (word sense)** wird verwaltet:
- eine Menge von **Synonymen** bzw. Ausprägungen (words)
- eine Menge von **Oberbegriffen** und **Unterbegriffen** im Sinne von
  - Generalisierung und Spezialisierung (Hypernyme, Hyponyme) (z.B. Nager und Ratte)
  - Teile-Ganzes-Beziehungen (Meronyme, Holonyme) (z.B. Computer und Platine)
  - Begriff-Beispiel-Beziehungen (z.B. Märchen und Aschenputtel)
- eine Menge von **Antonymen** (Gegensätzen)

Zu jedem **Wort (word)** – im Sinne einer Konzeptausprägung – wird verwaltet:
- eine Menge von Konzepten, ggf. mit statistischen Zusatzangaben (zur Desambiguierung von Polysemen bzw. Homonymen)

Beispiel eines umfangreichen Thesaurus:
WordNet, http://www.cogsci.princeton.edu/~wn

Komprimierung

- Text als Folge von Symbolen (unterschiedlicher Häufigkeit) eines Alphabets.
- Symbole können
  - Buchstaben bzw. Zeichen (Bytes),
  - Strings fester Länge (z.B. 3 aufeinanderfolgende Zeichen)
  - oder Wörter sein

  (oder u.U. auch Bits → Run-Length Encoding oder auch Silben, Phrasen etc.)

Grenzen der Komprimierung:
Sei pi die Wahrscheinlichkeit (bzw. relative Häufigkeit) des i-ten Symbols im Text d.
Dann ist die **Entropie** des Texts: $H(d) = \sum_i p_i \log_2 \frac{1}{p_i}$
eine untere Schranke
für die Bits pro Symbol bei bestmöglicher Komprimierung.

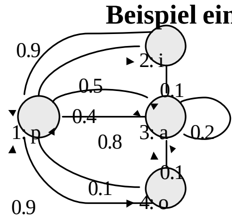
Entropie von Markovquellen

Eine (zeitdiskrete) endliche **Markovkette** ist ein Paar (Σ, p) bestehend aus einer Zustandsmenge Σ={s1, ..., sn} und einer Transitionswahrscheinlichkeitsfunktion p: Σ×Σ → [0,1] mit der Eigenschaft $\sum_j p_{ij} = 1$ für alle i, wobei p_{ij} := p(si, sj).

Eine Markovkette heißt **ergodisch (stationär)**, wenn für jeden Zustand sj der Grenzwert $p_j := \lim_{t\to\infty} p_{ij}^{(t)}$ existiert und von si unabhängig ist.
Dabei ist $p_{ij}^{(t)} := \sum_k p_{ik}^{(t-1)} p_{kj}$ für t>1 und gleich p_{ij} für t=1.

Die **Entropie** (eines Texts aus) einer stationären Markovquelle (Σ, p) ist
$H(\Sigma, p) = \sum_i p_i H(s_i)$ mit $H(s_i) := -\sum_j p_{ij} \log_2 p_{ij}$

Beispiel einer Markovquelle



p1 = 0.9 p2 + 0.8 p3 + 0.9 p4
p2 = 0.5 p1
p3 = 0.4 p1 + 0.2 p3 + 0.1 p2 + 0.1 p4
p4 = 0.1 p1
p1 + p2 + p3 + p4 = 1

⇒ p1 ≈ 0.459, p2= ≈ 0.229, p3 ≈ 0.264, p4 ≈ 0.045

H = 0.459*(0.5 log 2 + 0.4 log 2.5 + 0.1 log 10) +
0.229*(0.9 log 10/9 + 0.1 log 10) +
0.264*(0.8 log 10/8 + 0.2 log 5) +
0.045*(0.9 log 10/9 + 0.1 log 10) ≈ 0.9

Bei Gleichverteilung, d.h. pi= 0.25 und pij = 0.25 für alle i, j, wäre H = 4* 0.25 * (4 * 0.25 * log 4) = 2
die minimale Anzahl der Bits pro Zeichen

Komprimierung mit Huffman-Code

Codiere jedes Symbol s∈Σ durch variabel langen Bitstring c(s), so daß für keine zwei Symbole s1, s2 gilt: c(s1) ist ein Präfix von c(s2) (Präfix-Code).

Die Länge von c(s), soll umso kürzer sein, je höher die Wahrscheinlichkeit p(s) bzw. Häufigkeit f(s) von s ist.

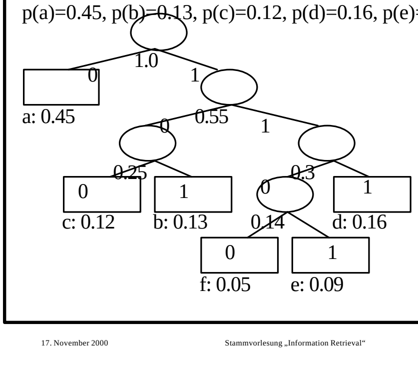
Damit ist die erwartete Anzahl der Bits pro Zeichen: $\sum_{s\in\Sigma} p(s)\,|c(s)|$

und die erwartete Länge eines komprimierten Textes d = a1...al ∈Σ* der unkomprimierten Länge l ist: $\sum_{i=1}^{l} f(a_i)\,|c(a_i)|$

Repräsentation des Codes als vollständiger Binärbaum mit Blättern der Form (s, p(s))
und inneren Knoten der Form $p_i = \sum_{\text{Blätter } s \text{ von } i} p(s)$

Beispiel eines Huffman-Codes

Beispiel:
|Σ|=6, Σ={a,b,c,d,e,f},
p(a)=0.45, p(b)=0.13, p(c)=0.12, p(d)=0.16, p(e)=0.09, p(f)=0.05



Algorithmus zur Berechnung eines Huffman-Codes

n := |Σ|
priority queue Q := Σ (aufsteigend sortiert nach p(s) für s∈Σ
for i:=1 to n-1 do
  z := MakeTreeNode( )
  z.left := ExtractMin(Q)
  z.right := ExtractMin(Q)
  p(z) := p(z.left) + p(z.right)
  Insert (Q, z)
od
return ExtractMin(Q)

Satz:
Der konstruierte Huffman-Code ist ein optimaler Präfix-Code.

Bemerkung:
Huffmann-Codes benötigen zur Komprimierung eines Textes zweimaliges Text-Scanning (oder textunabhängige Statistiken)

Komprimierung mit Ziv-Lempel-Algorithmus

LZ77-Algorithmus (Adaptives Dictionary):
Beim Text-Scanning wird mit einem *Lookahead-Fensters* der längste wiederholte, in einem *Rückwärtssuchfenster* liegende String ermittelt und durch einen „Zeiger“ ersetzt.
Codierung des Textes durch eine Liste von Tripeln der Form
  <*back, count, new*>,
wobei
• *back* die Rückwärtsdistanz zu einem früheren Vorkommen des an der aktuellen Position beginnenden Strings ist,
• *count* die Länge dieses wiederholten Strings ist und
• *new* das nächste Zeichen ist, das auf den wiederholten String folgt
Die Tripel können selbst (variabel lang) codiert werden.

Bemerkungen:
• Diese Art der Codierung wird in gzip und winzip verwendet.
• Die Komprimierung ist etwas aufwendig, kommt aber mit einem Text-Scan aus; die Dekomprimierung ist sehr schnell.
• Auf Texten werden häufig Kompressionsfaktoren von 2 bis 5 erreicht.
• Die Variante LZ78 verwendet ein explizites Dictionary statt des Rückwärtssuchfensters.

Beispiel für Ziv-Lempel-Algorithmus

peter_piper_picked_a_peck_of_pickled_peppers

<0, 0, p> für Zeichen 1:	p
<0, 0, e> für Zeichen 2:	e
<0, 0, t> für Zeichen 3:	t
<-2, 1, r> für Zeichen 4 und 5:	er
<-0, 0, _> für Zeichen 6:	_
<-6, 1, i> für Zeichen 7 und 8:	pi
<-8, 2, r> für Zeichen 9, 10 und 11:	per
<-6, 3, c> für Zeichen 12 bis 15:	_pic
usw.	

Substring-Suche (String-Matching)

Gegeben:
  Text d: array[1..n] of char
  Pattern p: array[1..m] of char
Gesucht:
  alle „Shifts“ s1, ..., sk ($0 \le si \le n-m$),
  so daß d[si+1..si+m] = p[1..m]

Naiver Algorithmus:
  for s:=0 to n-m do
    if p[1..m] = d[s+1..s+m] then
    print „pattern occurs with shift“ s

Worst-Case-Komplexität (# Zeichenvergleiche): $\Theta((n-m+1)m)$

Korrektheitsbeweise und Laufzeitanalysen siehe z.B.:
T. Cormen, C. Leiserson, R. Rivest: Introduction to Algorithms, MIT Press, 1990, Chapter 34

String-Matching-DEA

Ein **deterministischer endlicher Automat (DEA)** ist ein 5-Tupel
$(\Sigma, S, s, E, \delta)$ mit $s \in S$, $E \subseteq S$, $\delta: S \times \Sigma \to S$
Der DEA erkennt die Sprache $L := \{w \in \Sigma^*: \delta^*(s, w) \in E\}$,
wobei $\delta^*(s, wx) = \delta(\delta^*(s,w),x)$ für alle $w \in \Sigma^*$, $x \in \Sigma$.

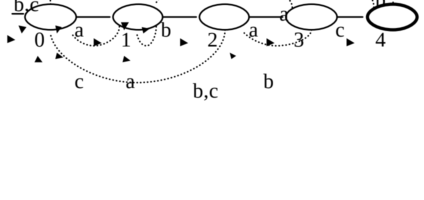
Ansatz:
Konstruiere DEA für Pattern p[1..m] mit Zuständen {0, 1, ..., m}, s=0, E={m} und Transition δ(i-1, p[i]) = i sowie geeigneten „Rückführungen“

Suffix-Funktion $p: \Sigma^* \to N_0$ mit
$\pi(w)$:= Länge des max. echten Präfixes von w, der auch Suffix von w ist

→ bei Mismatch in Zustand i mit Eingabezeichen x
  gehe in Zustand π(p[1..i]x)
→ vollständiger DEA: $\delta(i, x) = \pi(p[1..i]x)$

Beispiel für String-Matching-DEA

d[1..11] = a b a b a c a b a b a $\in$ {a, b, c}*
p[1..4] = a b a c



Knuth-Morris-Pratt-Algorithmus (1)

Grundidee:
Wenn $d[s+1..s+q] = p[1..q]$ mit $q<m$, aber $d[s+q+1] \neq p[q+1]$, kann anschließend der getestete Shift - abhängig von p - häufig um mehr als 1 erhöht werden.

Beispiel:
d[1..15] = b a c b a b a b a a b c b a b
p[1..7] = a b a b a c a
s = 4   q = 5

Bestimmung des nächsten zu testenden Shifts s‘:
Bestimme das kleinste s‘ > s, so daß
$p[1..k] = d[s'+1..s'+k]$ mit $s'+k=s+q$
Berechne dazu für gegebenes p die Funktion
$\pi: \{1, ..., m\} \rightarrow \{0, ..., m-1\}$ mit
$\pi(q) := \max \{k: k<q$ und $p[1..k]$ ist ein Suffix von $p[1..q]\}$

Knuth-Morris-Pratt-Algorithmus (2)

```
ComputePrefixFunction (p):
  π[1] := 0; k := 0;
  for q:=2 to m do
    while k>0 and p[k+1] ≠ p[q] do k:= π[k] od;
    if p[k+1] = p[q] then k:= k+1 fi;
    π[q]:=k;
  od; return π

KMPsearch (d, p):
  π := ComputePrefixFunction (p); q:= 0;
  for i:=1 to n do
    while q>0 and p[q+1] ≠ d[i] do q:= π[q] od;
    if p[q+1] = d[i] then q:= q+1 fi;
    if q = m then print „pattern occurs with shift“ i-m; q:=π[q] fi;
  od;
```

Worst-Case-Komplexität: $\Theta(n+m)$

Beispiel für KMP-Algorithmus

ComputePrefixFunction (p[1..7] = a b a b a c a):
$\pi[1] := 0$
$\pi[2] := 0$
$\pi[3] := 1$
$\pi[4] := 2$
$\pi[5] := 3$
$\pi[6] := 0$
$\pi[7] := 1$

KMPsearch (d[1..15] = b a c b a b a b a a b c b a b ,
p[1..7] = a b a b a c a):

Boyer-Moore-Algorithmus (1)

Grundidee:
vergleicht Muster mit aktuellem Textausschnitt von rechts nach links, erhöht den getesteten Shift häufig um mehr als 1 aufgrund
a) des nicht übereinstimmenden Zeichens in d
(z.B. wenn dieses in p gar nicht vorkommt) → *Bad Character Rule*
b) des bisher übereinstimmenden Suffix von p → *Good Suffix Rule*

Beispiel:

bad char *good suffix*

d = w r i t t e n _ n o t i c e _ t h a t ...
p = r e m i n i s c e n c e
s = 2
r e m i n i s c e n c e
s = 5 — aufgrund Bad Char
r e m i n i s c e n c e
s = 4 — aufgrund Good Suffix

Boyer-Moore-Algorithmus (2)

```
BMsearch (d, p):
  λ := ComputeLastOccurrenceFunction (p, m, Σ);
  σ := ComputeGoodSuffixFunction (p, m); s:= 0;
  while s ≤ n-m do
    j := m;
    while j>0 and p[j] = d[s+j] do j := j-1 od;
    if j = 0 then print „pattern occurs at shift“ s; s := s + σ[0]
         else s := s + max (σ[j], j - λ[d[s+j]] ) fi;
  od;
```

Worst-Case-Komplexität: $O((n-m+1)m+|\Sigma|)$,
Average-Case-Komplexität sehr viel besser

ComputeLastOccurrenceFunction (p, m, Σ): $\Sigma \rightarrow \{0, ..., m\}$
$\lambda[a]$:= Index des rechtesten Vorkommens von Zeichen a in p bzw. 0, wenn a in p nicht vorkommt

ComputeGoodSuffixFunction (p, m): $\{1, ..., m\} \rightarrow \{0, ..., m-1\}$
$\sigma[j] := m - \max \{k: 0 \leq k < m$ und $p[j+1..m]$ ist ein Suffix von $p[1..k]$ oder umgekehrt$\}$

Boyer-Moore-Horspool-Algorithmus

Simplifizierte, in der Praxis sehr schnelle BM-Variante:
Verwendet nur die LastOccurrenceFunction,
wendet diese aber bei d[s+m]=p[m] (auch) auf p[m] an

```
unoptimizedBMHsearch (d, p):
  λ := ComputeLastOccurrenceFunction (p, m, Σ);
  while s ≤ n-m do
    j := m;
    while j>0 and p[j] = d[s+j] do j := j-1 od;
    if j = 0 then print „pattern occurs at shift“ s fi;
    t := s + m - λ[p[m]] );
    if j>0 then s := max(t, s + j - λ[d[s+j]] else s: = t fi;
  od;
```

Optimierte Implementierungen verwenden diverse Codierungstricks, um u.a. den mehrfachen Zugriff auf λ in der Schleife zu vermeiden.

Ähnlichkeitsberechnung auf Strings (1)

Hamming-Distanz von Strings s1, s2 $\in \Sigma^*$ mit |s1|=|s2|:
Anzahl verschiedener Zeichen (Kardinalität von {i: $s1_i \neq s2_i$})

Levenshtein-Distanz (Editierdistanz) von Strings s1, s2 $\in \Sigma^*$:
minimale Anzahl der Editieroperationen auf s1
(Ersetzen, Löschen oder Einfügen eines Zeichens),
um s1 in s2 zu ändern

Für edit (i, j): Levenshtein-Distanz von s1[1..i] und s2[1..j]
gilt: edit (0, 0) = 0, edit (i, 0) = i, edit (0, j) = j
edit (i, j) = min { edit (i-1, j) + 1,
edit (i, j-1) + 1,
edit (i-1, j-1) + diff (i, j) }
mit diff (i, j) = 1 falls $s1_i \neq s2_j$, 0 sonst
→ Berechnung durch dynamische Programmierung

Ähnlichkeitsberechnung auf Strings (2)

Damerau-Levenshtein-Distanz von Strings s1, s2 $\in \Sigma^*$:
minimale Anzahl an Ersetzungs-, Einfüge-, Lösch- oder
Transpositionsoperationen (Vertauschen benachbarter Zeichen),
um s1 in s2 zu ändern

Für edit (i, j): Damerau-Levenshtein-Distanz von s1[1..i] und s2[1..j]
gilt: edit (0, 0) = 0, edit (i, 0) = i, edit (0, j) = j
edit (i, j) = min { edit (i-1, j) + 1,
edit (i, j-1) + 1,
edit (i-1, j-1) + diff (i, j),
edit (i-2, j-2) + diff(i-1, j) + diff(i, j-1) +1 }
mit diff (i, j) = 1 falls $s1_i \neq s2_j$, 0 sonst

Ähnlichkeit auf der Basis von N-Grammen

Bestimme für String s die Menge seiner N-Gramme:
G(s) = {Substrings von s mit der Länge N}
(häufig werden Trigramme betrachtet mit N=3)

Ähnlichkeit von Strings s1 und s2:
$|G(s1)| + |G(s2)| - 2|G(s1) \cap G(s2)|$

Beispiel:
G(rodney) = {rod, odn, dne, ney}
G(rhodnee) = {rho, hod, odn, dne, nee}
Ähnlichkeit (rodney, rhodnee) = 4 + 5 – 2*2 = 5

Phonetische Ähnlichkeit (1)

Soundex-Code:
Abbildung von Wörtern (speziell: Nachnamen) auf 4-Zeichen-Codes,
so daß ähnlich ausgesprochene Wörter denselben Code haben
• Erstes Codezeichen = erstes Zeichen des Wortes
• Codezeichen zwei bis vier (a, e, i, o, u, y, h, w werden ignoriert):

b, p, f, v	→ 1	c, s, g, j, k, q, x, z	→ 2
d, t	→ 3	l	→ 4
m, n	→ 5	r	→ 6

• Aufeinanderfolgende identische Codes werden zusammengefasst
(es sei denn, sie sind durch h getrennt)

Beispiele:
Powers → P620 , Perez → P620
Penny → P500, Penee → P500
Tymczak → T522, Tanshik → T522

Phonetische Ähnlichkeit (2)

Editex-Ähnlichkeit:
Editierdistanz mit Berücksichtigung phonetischer Codes
Für editex (i, j): Editex-Distanz von s1[1..i] und s2[1..j]
gilt: editex (0, 0) = 0,
editex (i, 0) = editex (i-1, 0) + d(s1[i-1], s1[i]),
editex (0, j) = editex (0, j-1) + d(s2[j-1], s2[j]),
editex (i, j) = min { editex (i-1, j) + d(s1[i-1], s1[i]),
editex (i, j-1) + d(s2[j-1], s2[j]),
edit (i-1, j-1) + diffcode (i, j) }
mit diffcode (i, j) = 0 falls $s1_i = s2_j$,
1 falls group($s1_i$)= group($s2_j$), 2 sonst
und d(X, Y) = 1 falls X ≠ Y und X ist h oder w,
diffcode (X, Y) sonst

mit group:
{a e i o u y}, {b p}, {c k q}, {d t}, {l r},
{m n}, {g j}, {f p v}, {s x z}, {c s z}

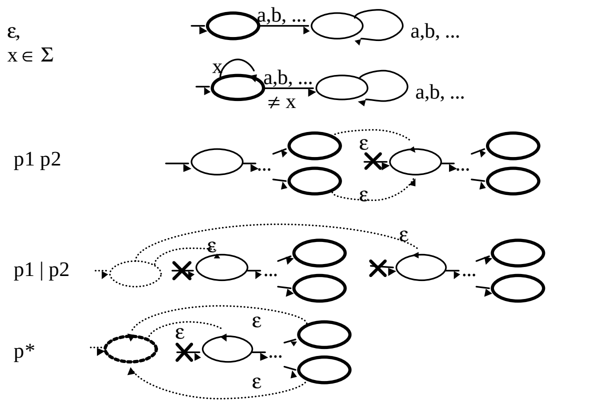
Pattern-Matching für reguläre Ausdrücke

Ein deterministischer endlicher Automat (DEA) ist ein 5-Tupel
(Σ, S, s, E, δ) mit $s \in S$, $E \subseteq S$, δ: $S \times \Sigma \rightarrow S$
Der DEA erkennt die Sprache L := {$w \in \Sigma^*$: $\delta^*(s, w) \in E$},
wobei $\delta^*(s, wx) = \delta(\delta^*(s,w),x)$ für alle $w \in \Sigma^*$, $x \in \Sigma$.

Gegeben sei ein regulärer Ausdruck p über dem Alphabet Σ
(gebildet aus Stringkonstanten, Konkatentation, Vereinigung |,
Kleene-Stern * und Klammern), der eine Sprache L(p) $\subseteq \Sigma^*$ beschreibt.

Problem:
Teste String $d \in \Sigma^*$, ob er einen Teilstring t enthält, der in L(p) ist.
(Anwendung: Unix grep)
Lösung:
Simuliere NEA mit δ: $S \times (\Sigma \cup \{\varepsilon\}) \rightarrow 2^S$, der Σ^*L(p) akzeptiert
→ Algorithmus mit Worst-Case-Laufzeit O(|p|*|d|) und Platz O(|p|)

Vom regulären Ausdruck zum NEA



NEA-Simulation für regulären Ausdruck

closure ($\sigma \in S$) returns $\{\sigma' \in \delta^*(\sigma, \varepsilon)\}$

nextstates ($S' \subseteq S$, $x \in \Sigma$) returns $\{\sigma' \in \delta^*(\sigma, x)$: $\sigma \in S'\}$

RegExpTester (p, $d \in \Sigma^*$) returns Boolean:
  construct NEA for p;
  states := closure (s) for initial state s;
  for i:=1 to |d| do
    states := closure (nextstates (states, d[i])) od;
  return $E \cap$ states $\neq \emptyset$

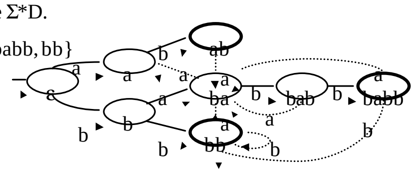
RegExpMatcher (p, $d \in \Sigma^*$) returns Boolean:
  construct NEA for p;
  states := closure (s) for initial state s; if $E \cap$ states $\neq \emptyset$ return true fi;
  for i:=1 to |d| do
    states := closure (nextstates (states, d[i]) $\cup$ {s});
    if $E \cap$ states $\neq \emptyset$ then return true
  od;

Dictionary-Matching

Testen eines Textes auf mehrere konstante Suchpatterns - ein endliches „Dictionary“ $D \subseteq \Sigma^*$

Der DEA (Σ, S = prefix(D), s=ε, E = prefix(D) $\cap$ Σ^*D, δ) mit δ(w, x) = längster Suffix von wx, der in prefix(D) liegt, erkennt die Sprache Σ^*D.

Beispiel: D = {ab, babb, bb}



Optimierungen führen auf Algorithmen mit Worst-Case-Laufzeit O(|D| * log |Σ| + |d| * log |Σ|)

Limited-Error-Matching

Matching eines konstanten Strings p in Text d
mit Maximalzahl erlaubter Fehler (z.B. ≤ k Substitutionen)

Ein möglicher Ansatz:
- BM-Matching
  mit Rückwärtsvergleich von p[1..m] mit d[s..s+m+k]
  bis k+1 unpassende Zeichen gefunden
- Die so gefundenen Kandidaten werden dann genauer untersucht

Beispiel 1 für KMP-Algorithmus

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
d:	b	a	c	b	a	b	a	b	a	a	b	c	b	a	b

	1	2	3	4	5	6	7
p:	a	b	a	b	a	c	a

Beispiel 2 für KMP-Algorithmus

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
d:	a	a	a	b	a	b	a	b	a	b	a	b	a	a	a

	1	2	3
p:	a	b	a

Vom regulären Ausdruck zum NEA

ε,
$x \in \Sigma$

p1 p2

p1 | p2

p*