Kapitel 2

Suche nach endlich vielen Wörtern

Übersicht

- Aufgabenstellung
  Gegeben: Text T und eine endliche Menge von Wörtern $\mathcal{P} = \{P_1, \ldots, P_r\}$;
  Gesucht: alle Vorkommen von Wörtern aus $\mathcal{P}$ in T.

  Notation: $|T| = n,\ |\mathcal{P}| = r,\ \sum_{i=1}^{r} |P_i| = M,$
  $\max\{|P_i| : 1 \leq i \leq r\} = m,\ \min\{|P_i| : 1 \leq i \leq r\} = \mu.$

- Anzahl r der Suchwörter häufig sehr groß $\rightarrow$ separate Suche nicht sinnvoll.

- Viele Ideen aus der Suche nach einem Wort sind verallgemeinerbar.

- Verwandtes Problem: Suche in Wörterbüchern

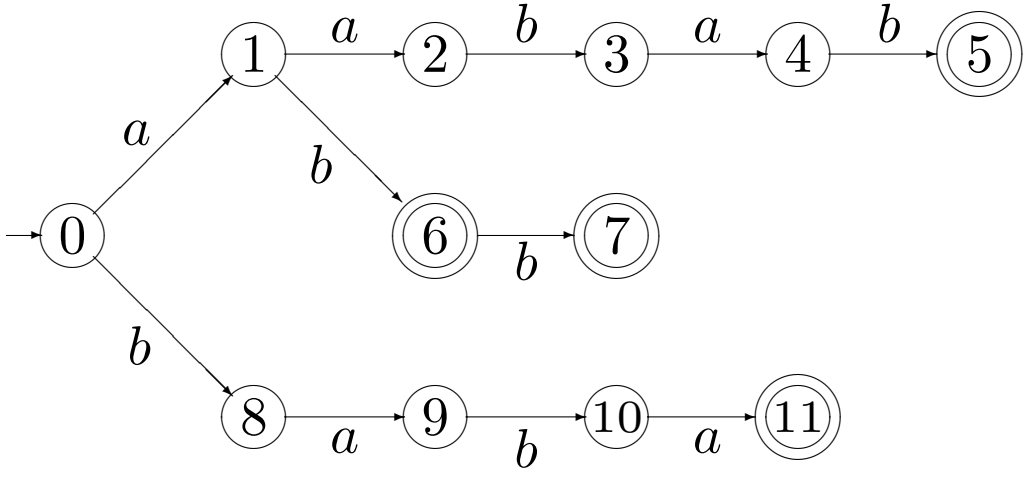
2.1 Suchwort-Bäume (Tries)

Definition. Es sei $\mathcal{P} = \{P_1, \ldots, P_r\}$ eine Menge von Wörtern. Der Suchwort-Baum (Trie) für $\mathcal{P}$ ist der gerichtete Baum $Trie(\mathcal{P})$, der folgende Bedingungen erfüllt:

1. Jede Kante ist mit genau einem Buchstaben beschriftet.
2. Keine zwei Kanten mit dem gleichen Ausgangsknoten haben die gleiche Beschriftung.
3. Jeder Knoten v hat eine Information $info(v) \in \{0, 1, \ldots, r\}$.
   Ist die Beschriftung $\mathcal{L}(v)$ des Weges von der Wurzel nach v ein Wort P_i mit $1 \leq i \leq r$, so gilt $info(v) = i$; anderenfalls ist $info(v) = 0$.
4. Ist v ein Blatt, so gilt $\mathcal{L}(v) \in \mathcal{P}$.
5. Für jedes Wort $P \in \mathcal{P}$ gibt es einen Knoten v mit $\mathcal{L}(v) = P$.

Beispiel – Trie

$\mathcal{P} = \{aabab, ab, abb, baba\}$



Wurzel: 0
$info(5) = 1,\ info(6) = 2,\ info(7) = 3,\ info(11) = 4.$

$Trie(\mathcal{P})$ ist der minimale pDEA, der $\mathcal{P}$ akzeptiert und dessen Graph ein Baum ist.

Algorithmus 2.1 Konstruktion des Suchwort-Baumes

Eingabe: Menge von Wörtern $\mathcal{P} = \{P_1, P_2, \ldots, P_r\}$
Ausgabe: Suchwort-Baum $Trie(\mathcal{P})$

(1) $V \leftarrow \{root\}$; $E \leftarrow \emptyset$;
(2) **for** $k \leftarrow 1$ **to** r
(3) $t \leftarrow 1$; $v \leftarrow root$;
(4) **while** $t \leq |P_k|$ **and** (E enthält die Kante $(v, P_k[t], u)$)
(5) $v \leftarrow u$; $t \leftarrow t + 1$;
(6) **for** $i \leftarrow t$ **to** $|P_k|$
(7) $u \leftarrow$ neuer Knoten; $info(u) \leftarrow 0$;
(8) $V \leftarrow V \cup \{u\}$; $E \leftarrow E \cup \{(v, P[k], u)\}$;
(9) $v \leftarrow u$;
(10) $info(v) \leftarrow k$;
(11) **return** (V, E);

Algorithmus 2.2 Naive Suche mit Tries

Eingabe: Menge von Wörtern $\mathcal{P}$, Text T mit $|T| = n$

Ausgabe: Menge S der Vorkommen von Wörtern aus $\mathcal{P}$ in T

(1) Konstruiere $Trie(\mathcal{P})$ mit der Wurzel $root$.
(2) $S \leftarrow \emptyset$; $k \leftarrow 1$;
(3) **for** $k \leftarrow 1$ **to** n
(4) $\quad j \leftarrow k$; $v \leftarrow root$;
(5) $\quad$ **while** (es gibt in $Trie(\mathcal{P})$ eine Kante $(v, T[j], u)$)
(6) $\quad\quad v \leftarrow u$; $j \leftarrow j + 1$;
(7) $\quad\quad$ **if** $info(v) > 0$ **then** $S \leftarrow S \cup \{(k, info(v))\}$;
(8) **return** S;

Laufzeit

schlechtester Fall: $\Theta(n \cdot m)$, mittlerer Fall: $\Theta(n \cdot \log_\sigma r)$.

2.2 DEA-Suche und Aho-Corasick-Algorithmus

- Verallgemeinerung der DEA-Suche für ein Wort bzw. des MP-Algorithmus
- Verallgemeinerung der $Border$-Tabelle: Fehler-Link (failure link)
- Für korrekte Ausgabe: Ausgabe-Link (output link)
- Laufzeit: $O(M)$ für das Präprozessing, $O(n)$ für die Suche

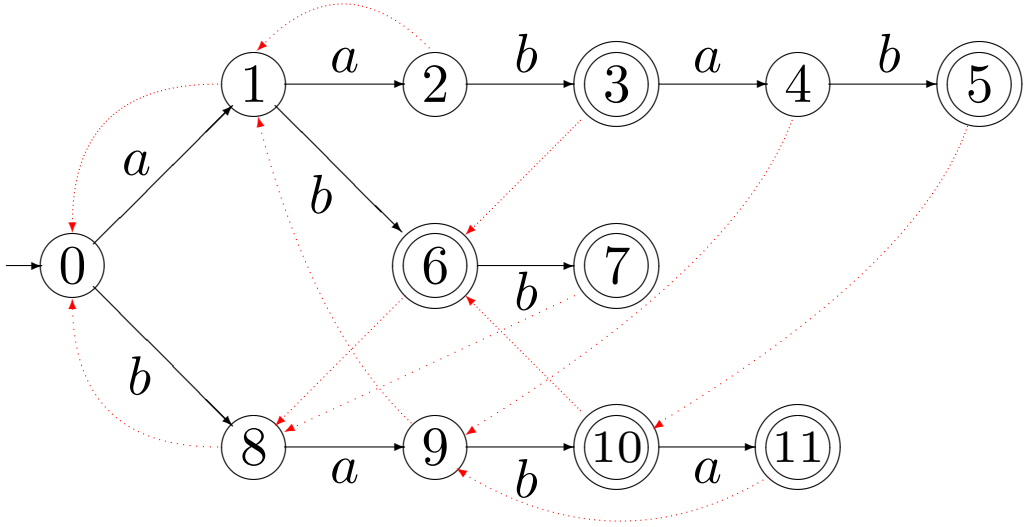
Fehler-Links

Definition. Für einen von der Wurzel verschiedenen Knoten v aus $Trie(\mathcal{P})$ sei $Fail_v$ der Knoten, für den $\mathcal{L}(Fail_v)$ das längste echte Suffix von $\mathcal{L}(v)$ ist, das Präfix eines Wortes aus $\mathcal{P}$ ist. Man bezeichnet $Fail_v$ als Fehler-Link (failure link) von v.

Konstruktion der Fehler-Links in Linearzeit analog zur Bestimmung der $Border$-Tabelle unter Verwendung bereits ermittelter Fehler-Links
Dabei werden die Knoten mit wachsender Tiefe (in BFS-Ordnung) betrachtet

Beispiel

Für $\mathcal{P} = \{aabab, ab, abb, baba\}$ erhalten wir folgende Fehler-Links:



Algorithmus 2.3 Bestimmung der Fehler-Links in einem Trie

Eingabe: $Trie(\mathcal{P})$ für Menge von Wörtern $\mathcal{P}$
Ausgabe: Tabelle $Fail$ der Fehler-Links für $Trie(\mathcal{P})$

```
(1)  foreach Knoten v in Trie(P) der Tiefe 1
(2)    Fail_v ← root;
(3)  foreach Knoten v der Tiefe > 1 in BFS-Ordnung
(4)    x ← Beschriftung der Kante (parent(v), v);
(5)    t ← Fail_{parent(v)};
(6)    while (es gibt keine Kante (t, x, u) and t ≠ root)
(7)      t ← Fail_t;
(8)    if es gibt Kante (t, x, u) then Fail_v ← u;
(9)                               else Fail_v ← root;
(10) return Fail;
```

Satz. Algorithmus 2.3 bestimmt die Fehler-Links mit einem Aufwand von $O(M)$.

Konstruktion des DEA

- Konstruktion analog zum DEA für ein Wort: Grundstruktur ist der Trie
- Bestimmung der Überführungsfunktion in BFS-Ordnung

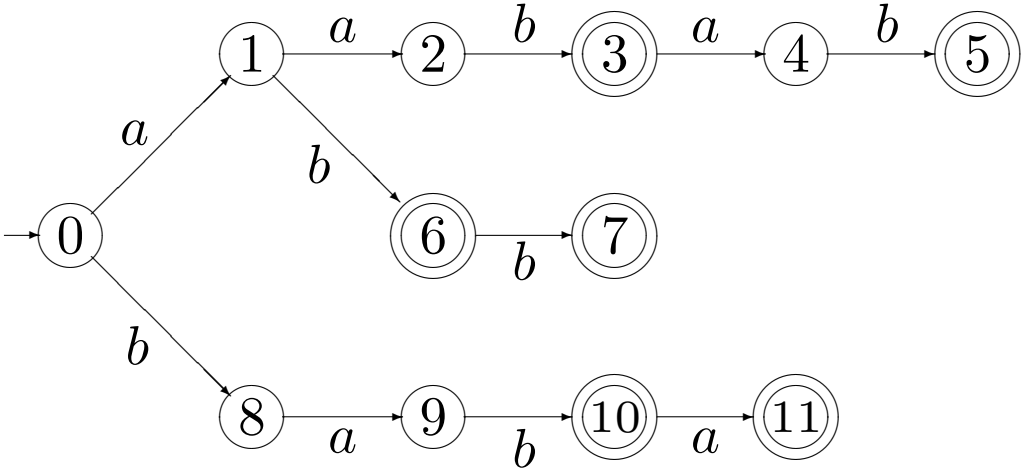
$$\delta(root, x) = \begin{cases} u & \text{falls } \mathit{Trie}(\mathcal{P}) \text{ die Kante } (root, x, u) \text{ enthält,} \\ root & \text{sonst.} \end{cases}$$

$$\delta(v, x) = \begin{cases} u & \text{falls } \mathit{Trie}(\mathcal{P}) \text{ die Kante } (v, x, u) \text{ enthält,} \\ \delta(\mathit{Fail}(v), x) & \text{sonst.} \end{cases}$$

- Endzustände: v ist Endzustand $\iff$ ein Suffix von $\mathcal{L}(v)$ ist in $\mathcal{P}$
  $\iff$ $\mathit{info}(v) > 0$ oder $\mathit{Fail}(v)$ ist Endzustand

Beispiel DEA

$\mathcal{P} = \{aabab, ab, abb, baba\}$; zusätzliche Endzustände: $3, 10$.



	0	1	2	3	4	5	6	7	8	9	10	11
a	1	2	2	4	2	11	9	9	9	2	11	2
b	8	6	3	7	5	7	7	8	8	10	7	10

Ausgabe-Links

Für Endzustand des DEA soll festgestellt werden, welche Suchwörter gefunden wurden.

Beispiel: $3 \to \{P_2\}$, $5 \to \{P_1, P_2\}$, $6 \to \{P_2\}$, $7 \to \{P_3\}$, $10 \to \{P_2\}$, $11 \to \{P_4\}$.

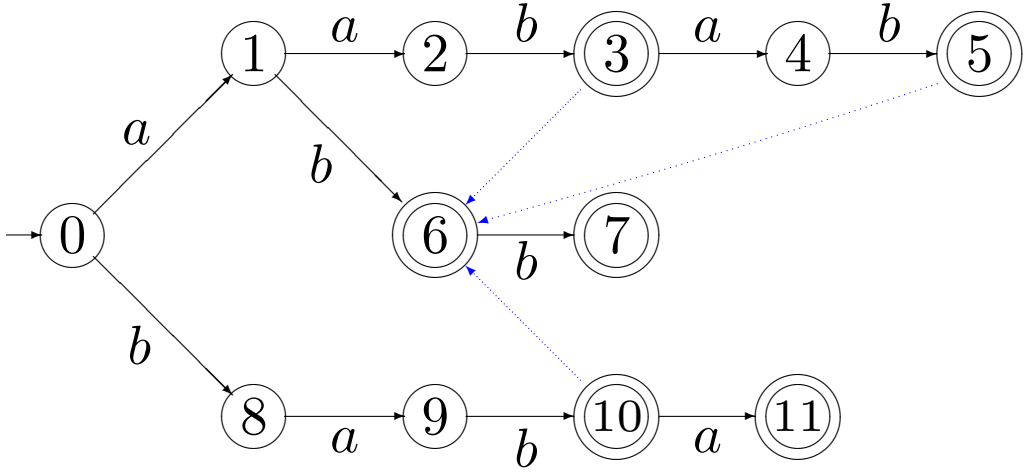
Effiziente Speicherung durch Ausgabe-Links.

Definition. Für einen von der Wurzel verschiedenen Knoten v aus $Trie(\mathcal{P})$ sei Out_v der Knoten, für den $\mathcal{L}(Out_v)$ das längste echte Suffix von v ist, das ein Wort aus $\mathcal{P} \cup \{\varepsilon\}$ ist. Man bezeichnet Out_v als Ausgabe-Link (output link) von v.

Zu Knoten v gehörende Wörter:
0. Keine, falls v die Wurzel ist.
1. P_i, falls $info(v) = i > 0$.
2. Alle zu $Out(v)$ gehörenden Wörter.

Beispiel: Ausgabe-Links

Für $\mathcal{P} = \{aabab, ab, abb, baba\}$ erhalten wir folgende Ausgabe-Links (Links zur Wurzel sind weggelassen):



Algorithmus 2.4 DEA-Suche nach endlich vielen Wörtern

Eingabe: endliche Menge von Wörtern $\mathcal{P}$, Text T

Ausgabe: Menge S der Vorkommen von Wörtern aus $\mathcal{P}$ in T

(1) Präprozessing: Konstruiere den DEA $A_{\mathcal{P}}$ und Ausgabe-Links;
(2) $S \leftarrow \emptyset$; $v \leftarrow root$;
(3) **for** $j \leftarrow 1$ **to** $|T|$
(4) $\quad v \leftarrow \delta(v, T[j])$; $i \leftarrow info(v)$;
(5) $\quad$ **if** $i > 0$ **then**
(6) $\quad\quad S \leftarrow S \cup \{(j - |P_i| + 1, i)\}$;
(7) $\quad t \leftarrow Out_v$; $i \leftarrow info(t)$;
(8) $\quad$ **while** $t \neq root$
(9) $\quad\quad S \leftarrow S \cup \{(j - |P_i| + 1, i)\}$;
(10) $\quad\quad t \leftarrow Out_t$; $i \leftarrow info(t)$;
(11) **return** S;

Algorithmus 2.5 Aho-Corasick-Algorithmus

Eingabe: Menge von Wörtern $\mathcal{P}$, Text T

Ausgabe: Menge S der Vorkommen von Wörtern aus $\mathcal{P}$ in T

(1) Präprozessing: Bestimme $Trie(\mathcal{P})$ mit Fehler- und Ausgabe-Links;
(2) $S \leftarrow \emptyset$; $v \leftarrow root$;$j \leftarrow 1$;
(3) **for** $j \leftarrow 1$ **to** $|T|$
(4) **while** es gibt keine Kante $(v, T[j], u)$ **and** $v \neq root$
(5) $v \leftarrow Fail_v$;
(6) **if** es gibt Kante $(v, T[j], u)$ **then** $v \leftarrow u$;
(7) $i \leftarrow info(v)$;
(8) **if** $i > 0$ **then**
(9) $S \leftarrow S \cup \{(j - |P_i| + 1, i)\}$;
(10) $t \leftarrow Out_v$; $i \leftarrow info(t)$;
(11) **while** $t \neq root$
(12) $S \leftarrow S \cup \{(j - |P_i| + 1, i)\}$;
(13) $t \leftarrow Out_t$; $i \leftarrow info(t)$;
(14) **return** S;

Einfluss der Alphabetgröße bei Algorithmen mit Tries

Alphabetgröße spielt eine Rolle bei der Bestimmung des nächsten Knotens.

Sei v ein Knoten mit d ausgehenden Kanten.
Mögliche Implementierungen der ausgehenden Kanten:

1. Liste; Platzbedarf: $\Theta(d)$; Zeit: $\Theta(d)$

2. geordnetes Array (geordnet nach Beschriftung der Kanten);
   Platzbedarf: $\Theta(d)$; Zeit: $\Theta(\log d)$, da binäre Suche
   ungünstig für dynamische Tries (Einfügen bzw. Löschen von Wörtern)

3. Array über Σ; Platzbedarf: $\Theta(\sigma)$; Zeit: $\Theta(1)$

2.3 Weitere Algorithmen

- Shift-And-Algorithmus für Mengen
  praktikabel für kleine Suchwort-Mengen
- Commentz-Walter-Algorithmus
  Verallgemeinerung des Boyer-Moore-Algorithmus
  historisch erster Algorithmus mit sublinearer mittlerer Laufzeit
- Horspool-Algorithmus für Mengen
  praktikabel für kleine Suchwort-Mengen und große Alphabete
- Wu-Manber-Algorithmus
  Verallgemeinerung des Horspool-Algorithmus mit verallgemeinerter Bad Character Regel;
  zusätzlich Einsatz von Hash-Tabellen
  allgemein sehr effizient
- Rückwärts-Faktor-Algorithmen
  Rückwärts-NEA praktikabel für kleine Suchwort-Mengen
  Orakel-DEA allgemein sehr effizient

Shift-And-Algorithmus für Mengen

- Verallgemeinerung des Shift-And-Algorithmus für ein Wort
- Theoretischer Hintergrund: NEA für Sprache $\Sigma^* \cdot \mathcal{P}$
- Bitvektoren I bzw. F kodieren Start bzw. Ende der einzelnen Wörter
- Länge der Bitvektoren: M. Damit effizient, wenn $M \leq w$.
  w: Länge eines Computer-Wortes

Theorie: NEA für $\Sigma^*\mathcal{P}$

Sei $\mathcal{P} = \{P_1, P_2, \ldots, P_r\}$ über Σ mit $|P_i| = m_i$.

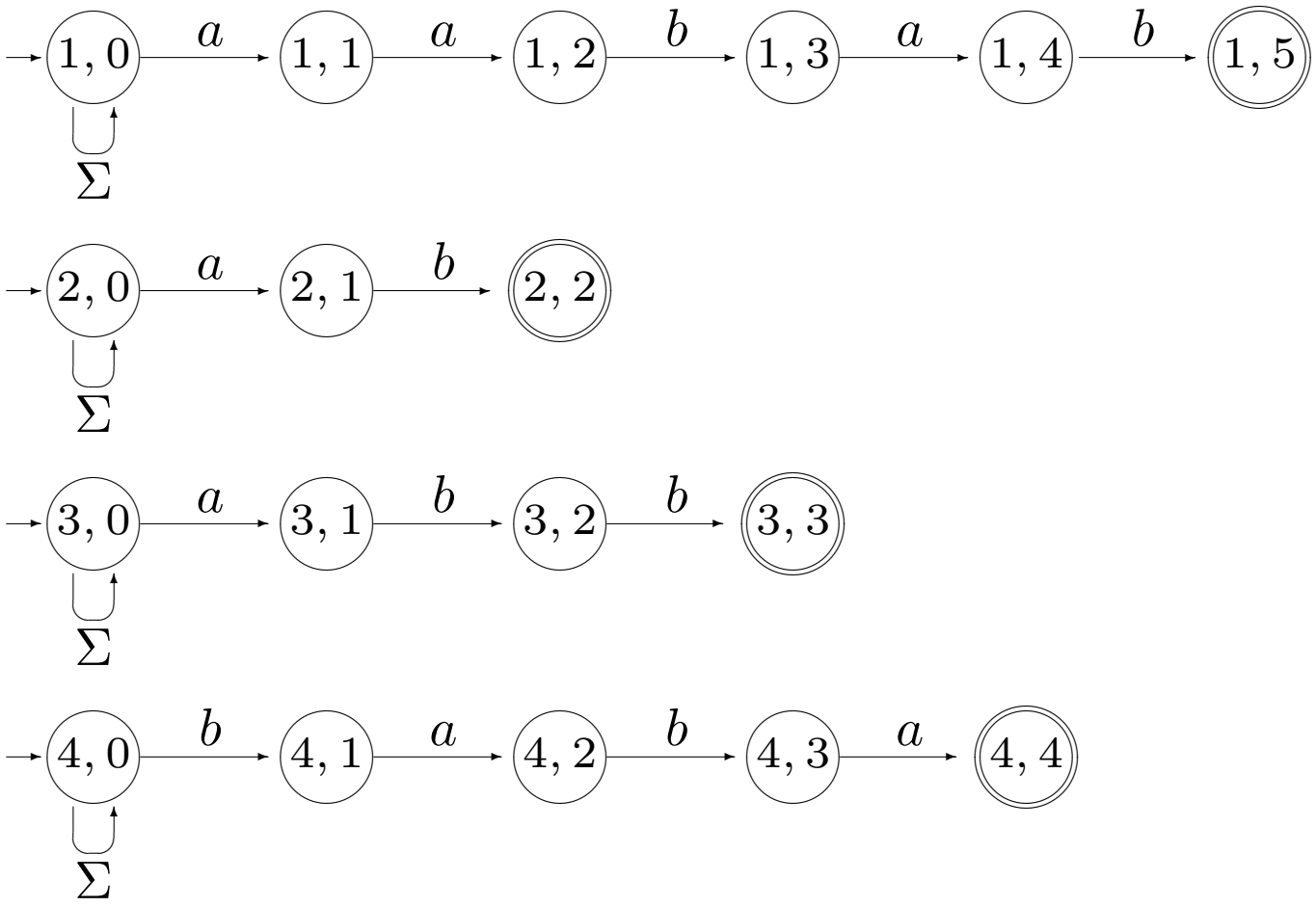
Dann akzeptiert der $NEA_{\mathcal{P}} = (Z, \Sigma, \delta, I, F)$ mit
$Z = \{(i,j) : 1 \leq i \leq r, 0 \leq j \leq m_i\}$,
$I = \{(i,0) : 1 \leq i \leq r\}$,
$F = \{(i,m_i) : 1 \leq i \leq r\}$ und der Menge der Transitionen

$$\begin{aligned} \delta &= \{((i,0), x, (i,0)) : 1 \leq i \leq r, x \in \Sigma\} \cup \\ &\quad \{((i,j-1), P_i[j], (i,j)) : 1 \leq i \leq r, 1 \leq j \leq m_i\} \end{aligned}$$

die Menge $\Sigma^*\mathcal{P}$.

Beispiel: NEA für $\Sigma^*\mathcal{P}$

$\mathcal{P} = \{aabab, ab, abb, baba\}$, $\Sigma = \{a, b, c\}$



Kodierung durch Bitvektoren

Sei $\mathcal{P} = \{P_1, P_2, \ldots, P_r\}$ über Σ mit $|P_i| = m_i$.

- Bitvektoren für $\mathcal{P}$ ergeben sich durch Konkatenation der Bitvektoren für die einzelnen Wörter:
  - $B_x = B_x(\mathcal{P}) = B_x(P_r) \cdots B_x(P_2) B_x(P_1)$ für $x \in \Sigma$;
  - $I = I(\mathcal{P}) = I(P_r) \cdots I(P_2) I(P_1)$ mit $I(P_i) = 0^{m_i - 1}1$;
  - $F = F(\mathcal{P}) = F(P_r) \cdots F(P_2) F(P_1)$ mit $F(P_i) = 10^{m_i - 1}$;
- Initialisierung: $Z \leftarrow 0^M$.
- Schritt für Textzeichen x: $Z \leftarrow ((Z \mathbin{<<} 1) \,|\, I) \,\&\, B_x$.
- Test auf Endzustand: $Z \,\&\, F \neq 0$.

Bitvektoren – Beispiel

Für $\mathcal{P} = \{aabab, ab, abb, baba\}$ und $\Sigma = \{a, b, c\}$ erhalten wir folgende Bitvektoren der Länge $5 + 2 + 3 + 4 = 14$:

$$
\begin{aligned}
B_a &= 1010\ 001\ 01\ 01011, \\
B_b &= 0101\ 110\ 10\ 10100, \\
B_c &= 0000\ 000\ 00\ 00000, \\
I &= 0001\ 001\ 01\ 00001, \\
F &= 1000\ 100\ 10\ 10000.
\end{aligned}
$$

Horspool-Algorithmus für Mengen

- In einer Phase Suche von rechts nach links mit Hilfe von $Trie(P_1^r, P_2^r, \ldots, P_r^r)$.
- Verschiebungsregel für Textzeichen $x \in \Sigma$:
  $Shift_x(\mathcal{P}) = \min\{Shift_x(P_i) : 1 \leq i \leq r\}$.
  Wenn $Shift_x > 0$, Verschiebung um $Shift_x$;
  anderenfalls Suche von rechts nach links mittels $Trie(P_1^r, P_2^r, \ldots, P_r^r)$.
- effizient für große Alphabete und wenige Suchwörter
  (z.B. bis zu 10 Wörter in natürlichsprachigem Text)
- Generell gilt: größtmögliche Verschiebung ist $\mu = \min\{|P_1|, |P_2|, \ldots, |P_r|\}$.

Beispiel

1. Für $\mathcal{P} = \{aabab, ab, abb, baba\}$ und $\Sigma = \{a, b, c\}$ gilt $Shift_a(\mathcal{P}) = Shift_b(\mathcal{P}) = 0$ und $Shift_c(\mathcal{P}) = 2$.
   Der Horspool-Algorithmus ist hier recht ineffizient.

2. Für $\mathcal{P} = \{textalgorithmen, theoretische, informatik\}$ ergibt sich folgende $Shift$-Tabelle, wobei $*$ für alle nicht explizit genannten Symbole steht:

a	c	e	g	h	i	k	l	m	n	o	r	s	t	$*$
3	2	0	8	1	1	0	9	2	0	6	5	3	2	10

   Der Horspool-Algorithmus ist hier sehr effizient.

Wu-Manber-Algorithmus

- Verallgemeinerung des Horspool-Algorithmus mit erweiterter Bad Character Regel
- Optimale Wahl der Blocklänge $q = \lceil \log_\sigma(r \cdot \mu) \rceil$.
  Mittlere Zahl der Vergleiche pro Phase: q, mittlere Verschiebung: μ.
- Aus Platzgründen oft keine Speicherung der $Shift$-Werte aller q-Tupel;
  stattdessen Verwendung von Hash-Werten für die q-Tupel
- Keine Konstruktion eines Tries;
  stattdessen Verwendung von Hash-Werten für die Suchwörter

Erweiterte Bad Character Regel für mehrere Wörter

Für $P \in \Sigma^*$ und $\alpha \in \Sigma^q$ sei $Shift_\alpha(P)$ wie beim Horspool-Algorithmus mit erweiterter Bad-Character-Regel definiert (siehe Folie 69).

Sei $\mathcal{P} = \{P_1, P_2, \ldots, P_r\}$.
Für $\alpha \in \Sigma^q$: $Shift_\alpha(\mathcal{P}) = \min\{Shift_\alpha(P_i) : 1 \leq i \leq r\}$

Erweiterte Bad Character Regel
Die letzten q Zeichen im Suchfenster seien α.
Gilt $Shift_\alpha > 0$, verschiebe um $Shift_\alpha$.
Gilt $Shift_\alpha = 0$, prüfe ob ein Wort aus $\mathcal{P}$ vorliegt, verschiebe anschließend um 1.

Hash-Funktion für die $Shift$-Tabelle

Platzbedarf für $Shift$-Tabelle: σ^q. Häufig zu groß, z.B. $\sigma = 256$, $q \geq 3$.
Platzersparnis durch Hash-Funktion: $Hash_1 : \Sigma^q \to \{1, 2, \ldots, p_1\}$.

Verschiebung ergibt sich aus dem $Hash_1$-Wert. Für sichere Verschiebung wähle kleinste Verschiebung unter den Wörtern mit demselben $Hash_1$-Wert.

Array $SHIFT$ der Größe p_1: $SHIFT_i(\mathcal{P}) = \min\{Shift_\alpha(\mathcal{P}) : Hash_1(\alpha) = i\}$.

Erweiterte Bad Character Regel mit Hash-Funktion
Die letzten q Zeichen im Suchfenster seien α. Bestimme $SHIFT_i$ für $i = Hash_1(\alpha)$.
Gilt $SHIFT_i > 0$, verschiebe um $SHIFT_i$.
Gilt $SHIFT_i = 0$, prüfe ob ein Wort aus $\mathcal{P}$ vorliegt, verschiebe anschließend um 1.

Beispiel

$\Sigma = \{0, 1, 2, 3\}$, $q = 2$, $P_1 = 1200321$, $P_2 = 03123232$, $P_3 = 101232310$.
$p_1 = 11$, $Hash_1(ab) = ((4a + b)\, mod\, 11) + 1$.

Block	00,23	01,30	02,31	03,32	10,33	11	12	13	20	21	22
$Hash_1$	1	2	3	4	5	6	7	8	9	10	11
$SHIFT$	1	6	1	0	0	6	4	6	4	0	6

Einige Situationen

Text-Block	$Hash_1$	$SHIFT$	Aktion
00	1	1	Verschiebe um 1.
03	4	0	Überprüfung notwendig.
12	7	4	Verschiebe um 4.
32	4	0	Überprüfung notwendig.

Hash-Funktion für die Suchwörter

$Hash_2 : \Sigma^q \to \{1, 2, \ldots, p_2\}$.
$Hash_2(P_i) = Hash_2(\alpha_i)$, wobei α_i das Suffix der Länge q von P_i ist.

Array PAT der Größe p_2. $PAT_j = \{i : Hash_2(P_i) = j\}$.

Benutzung: Ist der Textblock α und $SHIFT_{Hash_1(\alpha)} = 0$,
so teste alle P_i mit $i \in PAT_{Hash_2(\alpha)}$ auf ein Vorkommen.

Vorteil: Es muss kein Trie konstruiert werden.
(Platzersparnis, einfache Programmierung)

Nachteil: Schlimmstenfalls muss man alle Suchwörter testen.
(natürlich sehr unwahrscheinlich)

Beispiel

$\Sigma = \{0, 1, 2, 3\}$, $q = 2$, $P_1 = 1200321$, $P_2 = 03123232$, $P_3 = 101232310$.
$p_2 = 3$, $Hash_2(ab) = ((4a + b)\, mod\, 3) + 1$ für $a, b \in \Sigma$.

Suchwort	P_1	P_3	P_2
$Hash_2$	1	2	3
PAT	$\{1\}$	$\{3\}$	$\{2\}$

Einige Situationen

Text-Block	$Hash_1$	$SHIFT$	$Hash_2$	Aktion
00	1	1		Verschiebe um 1.
03	4	0	1	Teste, ob P_1 vorliegt. Dann verschiebe um 1.
12	7	4		Verschiebe um 4.
32	4	0	3	Teste, ob P_2 vorliegt. Dann verschiebe um 1.

2.4 Suche in Wörterbüchern

- **Gegeben:** $\mathcal{P} = \{P_1, P_2, \ldots, P_r\}$ und ein Wort T.
  Frage: Ist T in $\mathcal{P}$ enthalten?

- Nach Vorverarbeitung von $\mathcal{P}$ soll die Frage effizient, d.h. möglichst in Zeit $O(|T|)$ beantwortet werden können.

- **Lösung:** Konstruiere $Trie(\mathcal{P})$. Suche dann in $Trie(\mathcal{P})$ nach dem Pfad T.

- **Anwendung:** Rechtschreibprüfung, Kompression (mit dynamischen Wörterbüchern)

- **Alternative Datenstrukturen**
  1. Minimaler DEA für $\mathcal{P}$ (gerichteter azyklischer Graph)
  2. Sortierung von $\mathcal{P}$ nach lexikographischer Ordnung, binäre Suche.
  Beide Strukturen sparen Platz, sind aber nur für unveränderliche Wörterbücher einfach zu handhaben.

Anwendung von Wörterbüchern: LZW-Kompression

- **Eingabe:** Text $T \in \Sigma^*$ mit $|T| = n$
  Ausgabe: Wort Z über $\mathbb{N}$ (komprimierte Version von T)
  T kann aus Z rekonstruiert werden.

- Benutzung eines dynamischen Wörterbuchs D.
  D wird während der Kompression aufgebaut und enthält Teilwörter von T.
  Länge der Wörter in D unbegrenzt; sehr effizient bei langen Wiederholungen.

- In der Komprimierten Z werden Wörter aus D durch einen Index in D dargestellt.

- Einfacher Kompressionsalgorithmus, angewendet z.B. beim Bildformat `GIF`.

- Kompression und Dekompression in Linearzeit, wenn D als Trie realisiert wird.

LZW-Algorithmus: Kompressionsschritt

Alphabet sei $\Sigma = \{a_1, a_2, \ldots a_\sigma\}$.
Initialisierung: $D = (a_1, a_2, \ldots a_\sigma)$; $Z = \varepsilon$.

Sei $T[1 \ldots i-1]$ bereits komprimiert. D enthalte d Wörter.

Bestimme das längste Präfix α von $T[i \ldots n]$, das in D enthalten ist.
Bestimme die Position p von α in D. Hänge p an Z an.

Gilt $|\alpha| = n - i$, so ist die Kompression beendet.
Anderenfalls füge das Wort $T[i \ldots i + |\alpha|] = \alpha T[i + |\alpha|]$ in D an der Position $d+1$ ein.

Algorithmus 2.6 LZW-Kompression

Eingabe: Text $T \in \Sigma^*$, $|T| = n$, $\Sigma = \{a_1, a_2, \ldots, a_\sigma\}$

Ausgabe: Komprimierter Text (Indexfolge) Z

(1) $D \leftarrow \{1 \rightarrow a_1, 2 \rightarrow a_2, \ldots, \sigma \rightarrow a_\sigma\}$; $d \leftarrow \sigma + 1$;
(2) $Z \leftarrow \varepsilon$; $j \leftarrow 1$;
(3) **while** $j \leq n$
(4) $\quad i \leftarrow j$;
(5) $\quad$ **while** D enthält $T[i \ldots j]$
(6) $\quad\quad j \leftarrow j + 1$;
(7) $\quad Z \leftarrow Z \cdot$ Index von $T[i \ldots j-1]$;
(8) $\quad D \leftarrow D \cup \{d \rightarrow T[i \ldots j]\}$; $d \leftarrow d + 1$;
(9) **return** Z;

LZW-Kompression: Beispiel

Für $\Sigma = \{a, b, c\}$ und $T = bacbacbacba$ ergibt sich folgender Ablauf des Algorithmus. Der Index eines Wortes im Wörterbuch ergibt sich aus seiner Position. Die Position im Text wird durch den senkrechten Strich angezeigt.

Wörterbuch	Text	Ausgabe	Neuer Eintrag
(a, b, c)	$\|bacbacbacba$	2	ba
(a, b, c, ba)	$b\|acbacbacba$	1	ac
(a, b, c, ba, ac)	$ba\|cbacbacba$	3	cb
(a, b, c, ba, ac, cb)	$bac\|bacbacba$	4	bac
$(a, b, c, ba, ac, cb, bac)$	$bacba\|cbacba$	6	cba
$(a, b, c, ba, ac, cb, bac, cba)$	$bacbacb\|acba$	5	acb
$(a, b, c, ba, ac, cb, bac, cba, acb)$	$bacbacbac\|ba$	4	

Die Ausgabe ist somit $(2, 1, 3, 4, 6, 5, 4)$.

LZW-Algorithmus: Dekompressionsschritt

Alphabet sei $\Sigma = \{a_1, a_2, \ldots a_\sigma\}$.
Initialisierung: $D = (a_1, a_2, \ldots a_\sigma)$; $T = \varepsilon$.

Sei $Z[1 \ldots i-1]$ bereits dekomprimiert. D enthalte d Wörter.
Es gilt $Z[i] \leq d$, $Z[i+1] \leq d$ (Beweis: Induktion).

Bestimme das Wort α an der Position $Z[i]$ in D.
Hänge α an T an.

Gilt $|Z| = i$, so ist die Dekompression beendet.
Anderenfalls bestimme das erste Zeichen a des Wortes an der Position $Z[i+1]$ in D und füge αa in D an der Position $d+1$ ein.

Algorithmus 2.7 LZW-Dekompression

Eingabe: LZW-komprimierter Text Z, $|Z| = k$, Alphabet $\Sigma = \{a_1, a_2, \ldots, a_\sigma\}$

Ausgabe: dekomprimierter Text $T \in \Sigma^*$

(1) $D \leftarrow \{1 \rightarrow a_1, 2 \rightarrow a_2, \ldots, \sigma \rightarrow a_\sigma\}$; $d \leftarrow \sigma + 1$;
(2) $T \leftarrow \varepsilon$;
(3) **for** $i \leftarrow 1$ **to** $k - 1$
(4) $\quad w \leftarrow$ Wort mit Index $Z[i]$;
(5) $\quad T \leftarrow T \cdot w$;
(6) $\quad a \leftarrow$ erstes Symbol von Wort mit Index $Z[i+1]$;
(7) $\quad D \leftarrow D \cup \{d \rightarrow wa\}$; $d \leftarrow d + 1$;
(8) $T \leftarrow T \cdot$ Wort mit Index $Z[k]$;
(9) **return** T;

LZW-Dekompression: Beispiel

Für $\Sigma = \{a, b, c\}$ und $Z = (2, 1, 3, 4, 6, 5, 4)$ ergibt sich folgender Ablauf der Dekompression. Der Index eines Wortes im Wörterbuch ergibt sich aus seiner Position. Die Position im komprimierten Text wird durch den senkrechten Strich angezeigt.

Wörterbuch	komprimierter Text	Ausgabe	Neuer Eintrag
(a, b, c)	$(\|2, 1, 3, 4, 6, 5, 4)$	b	ba
(a, b, c, ba)	$(2\|1, 3, 4, 6, 5, 4)$	a	ac
(a, b, c, ba, ac)	$(2, 1\|3, 4, 6, 5, 4)$	c	cb
(a, b, c, ba, ac, cb)	$(2, 1, 3\|4, 6, 5, 4)$	ba	bac
$(a, b, c, ba, ac, cb, bac)$	$(2, 1, 3, 4\|6, 5, 4)$	cb	cba
$(a, b, c, ba, ac, cb, bac, cba)$	$(2, 1, 3, 4, 6\|5, 4)$	ac	acb
$(a, b, c, ba, ac, cb, bac, cba, acb)$	$(2, 1, 3, 4, 6, 5\|4)$	ba	

Die Ausgabe ist somit $bacbacbacba$.