ADS: Algorithmen und Datenstrukturen 2
Teil 5

Prof. Peter F. Stadler & Sebastian Will

Bioinformatik/IZBI
Institut für Informatik
& Interdisziplinäres Zentrum für Bioinformatik
Universität Leipzig

07. Mai 2014

Datenkompression

Bisheriges Hauptziel bei der Formulierung von Algorithmen

- schnelle Verarbeitung von Daten
- Effizienz bezüglich *Zeit*-Komplexität

Jetzt:

- *Speicher*-effiziente *Kodierung* (Darstellung)
- Datenkompression

Datenkompression

Zwei grundsätzlich unterschiedliche Typen der Kompression

- **verlustfrei** (lossless)
  Codierung eindeutig umkehrbar.
- **verlustbehaftet** (lossy)
  Originaldaten können i.A. nicht eindeutig aus der komprimierten Codierung zurückgewonnen worden.
  Beispiele: JPEG, MPEG (mp3)

Kompressionsverfahren: Übersicht

- **Lauflängenkodierung:** Aufeinanderfolgende identische Zeichen werden zusammengefasst.
- **Huffman-Kodierung:** Häufiger auftretende Zeichen werden mit weniger Bits codiert.
- **LZW, LZ77, gzip:** Mehrfach auftretende Zeichen*folgen* werden indiziert (*Wörterbuch*/*Codetabelle*), um dann durch ein einzelnes Zeichen dargestellt zu werden.

Allgemeines Prinzip:
Ausnutzen von ***Redundanz***
(Abweichungen von zufälliger Zeichenfolge, wiederkehrende Muster)

Lauflängencodierung

- Einfachster Typ von Redundanz:
  Läufe (runs) = lange Folgen sich wiederholender Zeichen.
- Beispiel:
  AAAABBBAABBBBBCCCCCCCCDABCBAAABBBBCCCD.
- Ersetze jeden Run durch seine Länge und das wiederholte Zeichen:
  4A3B2A5B8CDABCB3A4B3CD
- Lohnt sich nur für Läufe mit Länge > 2.

Wie unterscheidet man Text-Zeichen von Längenangaben? Insbesondere, wenn alle Zeichen (auch Ziffern) im zu codierenden Text vorkommen dürfen.

Lauflängencodierung: Codierung mit Escape-Zeichen

- Wähle als ***Escape-Zeichen*** ein Zeichen Q, das in der Eingabe wahrscheinlich nur selten auftritt.
- Jedes Auftreten dieses Zeichens besagt, dass die folgenden beiden Buchstaben ein Paar (Zähler, Zeichen) bilden.
- Ein solches Tripel wird als *Escape-Sequenz* bezeichnet.
- Zählerwert i wird durch i-tes Zeichen des Alphabets dargestellt.
- Codierung erst für Läufe ab Länge 4 sinnvoll.
- Auftreten des Escape-Zeichens Q selbst muss speziell codiert werden, z.B. als Run der Länge 1 oder Q$_\sqcup$ ($\sqcup$=Leerzeichen), etc.
- Beispiel:

Text	AAAABBBAABBBBBCCCCCCCCDABCQC.
Codierung	**Q**DABBBAA**Q**EB**Q**HCDABC**Q**$_\sqcup$C

Lauflängencodierung: Spezialfall "Binäre Daten"

Für Sequenzen von Bitwerten $\in \{0, 1\}$:

- Beobachtung: Läufe von 0 und 1 wechseln sich ab.
- Nutze das aus: gebe nur noch Lauflängen an.
- Beispiel:

Sequenz	000111011111011000111111
Kodierung	3 3 1 4 1 2 3 5

Was ist, wenn die Sequenz mit 1 beginnt?

Codierung mit variabler Länge

- Ansatzpunkt für Kompression: In Sprache kommen Zeichen mit unterschiedlicher Häufigkeit vor, z.B. **E** häufiger als **Y**.
- Beispiel: ABRACADABRA
- Standardcodierung (unkomprimiert) mit 5 Bits pro Zeichen:
  00001 00010 10010 00001 00011 00001 00100 00001 00010 10010 00001
- Häufigkeit der Buchstaben:

A	B	C	D	R
5	2	1	1	2

- **Idee:** verwende für häufige Zeichen kurze Bitsequenzen, für seltene dafür längere. Minimiere so die Gesamtzahl der benötigten Bits.

Wie kann man, trotz unterschiedlich langer Bitfolgen, erkennen welche Bits welches Zeichen codieren?

Codierung mit variabler Länge: Präfixcodes

Präfixcode (Präfix-freier Code)

- Kein Codewort ist Präfix eines anderen Codeworts.
- Durch Präfix-freiheit: Codierung eindeutig umkehrbar.
- Im Beispiel (ABRACADABRA):

Buchstabe	A	B	C	D	R
Häufigkeit	5	2	1	1	2
Code	0	100	1010	1011	11

Codierung:

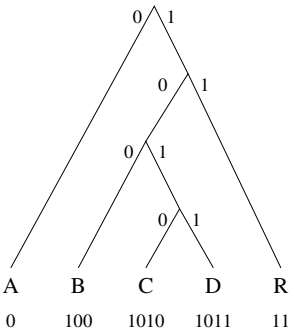
01001101010010110100110

AB R AC AD AB R A

Codierung mit variabler Länge: Tries

Binärer Präfixcode lässt sich durch Binärbaum (Trie) darstellen:

- **Blätter** = zu codierende Zeichen
- Zur **Codierung eines Zeichens**: laufe von Wurzel zum Blatt des Zeichens und gebe Kantenlabel aus (0 bei Verzweigung nach links; 1, nach rechts)
- Übliche Bezeichnung: **Trie** (dazu später mehr)

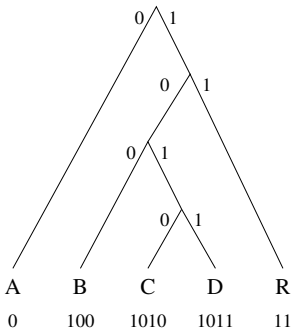


Warum wird der Binärbaum sinnvollerweise strikt gewählt?

Dekodierung

Gegeben: Bitsequenz und Präfixcode als Trie

1. Beginne bei Wurzel im Baum
2. Wiederhole bis zum Ende der Bitsequenz
   1. Lies nächstes Bit. Bei 0 verzweige nach links im Baum, sonst nach rechts.
   2. Falls erreichter Knoten ein Blatt ist, gib das Zeichen des Blatts aus und springe zurück zur Wurzel.



Der Huffman-Code

Der *Huffman-Code* ist ein präfix-freier Code mit variabler Länge, der die unterschiedlichen Zeichen-Häufigkeiten ausnutzt und systematisch erzeugt werden kann.

Erzeugung des Huffman-Codes

Schritt 1: Zähle Häufigkeit der Zeichen in der zu codierenden Zeichenfolge

Das folgende Programm ermittelt die Buchstaben-Häufigkeiten einer Zeichenfolge a und trägt diese in ein Feld count ein.

```
for ( i = 0 ; i <= 26 ; i++ ) count[i] = 0;
for ( i = 0 ; i < a.length ; i++ ) count[index(a[i])]++;
```

Dabei liefert `index(c)` den Index eines Zeichens c:
`index(A)`=1, `index(B)`=2, ...; `index(␣)`:=0

Huffman-Code: Beispiel

"A SIMPLE STRING TO BE ENCODED USING
A MINIMAL NUMBER OF BITS"

Dazugehörige Häufigkeits-Tabelle

	_	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
k	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26
count[k]	11	3	3	1	2	5	1	2	0	6	0	0	2	4	5	3	1	0	2	4	3	2	0	0	0	0	0

Huffman-Code: Erzeugung des Tries

Schritt 2: Aufbau des Tries (entsprechend den Häufigkeiten)

Für jedes Zeichen mit Häufigkeit$\neq 0$, erzeuge einen Baum jeweils mit dem Zeichen als einzigem Knoten.

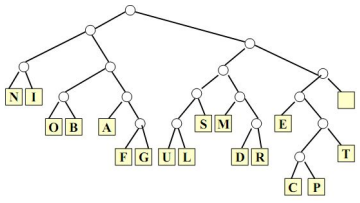
Schritt 3: Kombiniere iterativ die Bäume mit kleinsten Häufigkeiten

- Wähle zwei Bäume mit minimalen Häufigkeiten (Die Häufigkeit eines Baums ist die Summe der Häufigkeiten aller seiner Zeichen. Während der Erzeugung speichern wir Häufigkeit jedes Baums ab.)
- Erzeuge neuen Knoten, der diese beiden Bäume als Nachfolger hat
- Iteriere Schritt 3 bis alle Knoten miteinander zu einem einzigen Baum verbunden sind.

Am Ende sind Zeichen mit geringen Häufigkeiten weit unten im Baum; Zeichen mit großen Häufigkeiten nah an der Wurzel.

Trie für die Huffman-Codierung von "A SIMPLE STRING ..."

	_	A	B	C	D	E	F	G	I	L	M	N	O	P	R	S	T	U
k	0	1	2	3	4	5	6	7	9	12	13	14	15	16	18	19	20	21
count[k]	11	3	3	1	2	5	1	2	6	2	4	5	3	1	2	4	3	2

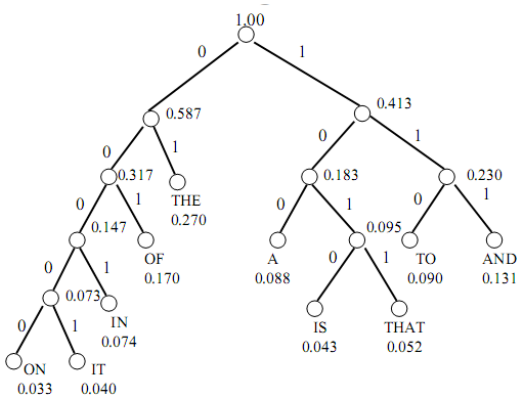


Ableitung des eigentlichen Codes aus dem Trie: Kantenlabel 0 nach links und 1 nach rechts, lese Code eines Zeichens auf Pfad von Wurzel zu Zeichen ab.

Huffman-Codierung für Wörter der Englischen Sprache

Codierungs-Einheit	Wahrscheinlichkeit des Auftretens	Code-Wert	Code-Länge
the	0.270	01	2
of	0.170	001	3
and	0.137	111	3
to	0.099	110	3
a	0.088	100	3
in	0.074	0001	4
that	0.052	1011	4
is	0.043	1010	4
it	0.040	00001	5
on	0.033	00000	5

Huffman-Codierungs-Baum für englische Wörter



Fragen zur Kompression

- Welche Arten von Redundanz können Lauf- längencodierung und Huffman-codierung ausnutzen bzw. nicht ausnutzen?
- Welches Verfahren sollte wann benutzt werden?
- Welches Verfahren ist gut für Text?
- Sind Kombinationen, d.h. die Hintereinanderausführung von Kompressions-Verfahren sinnvoll?

- Verschiedene Kompressionsverfahren arbeiten unterschiedlich gut bei Daten unterschiedlichen Typs (Bilder, numerische Daten, Text).
- Schlussfolgerung: Wir interessieren uns für weitere Verfahren.

Die LZ-Familie von Kompressionsalgorithmen

LZ77, LZ78, LZW, ...

Nutze unterschiedliche Häufigkeit von *Zeichenfolgen* aus.

Grundlegende Idee: verwende ein "Wörterbuch" von Zeichenfolgen, so dass häufige Zeichenfolgen durch Verweise auf das Wörterbuch platzsparend codiert werden können. (LZ78,LZW: explizites Wörterbuch; LZ77: implizit, "sliding window")

Text-Beispiel: RABARBABARBARABARBARBARENBART[1]

Grundsätzlicher Konflikt: Je grösser das Wörterbuch, desto

+ mehr und längere Zeichenfolgen können durch Verweise codiert werden
- mehr Platzbedarf pro Verweis
- mehr Platzbedarf für Wörterbuch
- höhere Laufzeit und Platzbedarf für Kompression und Dekompression

[1] in kompressionsfreundlich-reformierter Rechtschreibung

Historischer Exkurs (LZ-Familie)

1977	Abraham Lempel und Jacob Ziv erfinden den Kompressions-algorithmus LZ77
1983	Terry A. Welch (Sperry Corporation - später Unisys) paten-tiert LZW (Variante von LZ78)
1987	CompuServe veröffentlicht GIF als freie und offene Spezifika-tion (Version 87a)
1989	Vorstellung von GIF 89a
1993	Unisys informiert CompuServe über die Verwendung ihres pa-tentierten LZW-Algorithmus in GIF
1994	Unisys gibt öffentlich bekannt, Gebühren für die Verwendung des LZW-Algorithmus einzufordern
1995	die PNG Gruppe wird gegründet, erste PNG Bilder werden ins Netz gestellt, die PNG-Spezifikation 0.92 steht im W3C
1997	PNG-Unterstützung in Netscape 4.04 und IE 4.0

Lempel-Ziv-Welch (LZW)

Algorithmus:

- Initialiere das Wörterbuch mit den erlaubten Zeichen.
- Kodierungsschleife:
  - Bestimme das längste Wort aus dem Wörterbuch, das mit dem Anfang des noch nicht kodierten Teils der Eingabefolge übereinstimmt.
  - Schreibe die Nummer dieses Worts wird in die Ausgabe.
  - Erstelle einen neue Wörterbucheintrag: die Verlängerung der eben gefundenen Buchstabenfolge um den nächsten Buchstaben.
- Das Wörterbuch wird so immer größer. Deshalb: setze Maximalgrösse; bei Überschreiten initialisere das Wörterbuch wieder neu.

LZW Algorithmus – Kodieren

```
Initialisiere Codetabelle (jedes Zeichen erhält einen Code/eine Nummer);
präfix := " ";
while Ende der Eingabe noch nicht erreicht do
    suffix := nächstes Zeichen der Eingabe;
    wort := präfix + suffix;
    if wort in Codetabelle then präfix := wort;
    else
        gib Code von präfix aus;
        trage wort in Codetabelle ein;
        präfix := suffix;
    end
end
if präfix ≠ ∅ then
    gib Code von präfix aus;
end
```


LZW Algorithmus – Beispiel "Kodieren"

Kodieren von ABCABCABCD

Initiale Codetabelle: 0:A 1:B 2:C 3:D

	präfix	wort	suffix	Eingabe	Code	Ausgabe
0		A	A	ABCABCABCD		
1	A	AB	B	BCABCABCD	4:AB	0 (A)
2	B	BC	C	CABCABCD	5:BC	1 (B)
3	C	CA	A	ABCABCD	6:CA	2 (C)
4	A	AB	B	BCABCD		
5	AB	ABC	C	CABCD	7:ABC	4 (AB)
6	C	CA	A	ABCD		
7	CA	CAB	B	BCD	8:CAB	6 (CA)
8	B	BC	C	CD		
9	BC	BCD	D	D	9:BCD	5 (BC)
10	D					3 (D)

LZW Algorithmus – Dekodieren

```
Initialisiere Codetabelle (jedes Zeichen erhält einen Code);
lies Code;
präfix := dekodiere Code (laut Codetabelle);
gib präfix aus;
while Ende der Eingabe noch nicht erreicht do
    lies Code;
    if Code in Codetabelle then
        wort := dekodiere Code (laut Codetabelle);
        neues_wort := präfix + erstes Zeichen von wort;
    else
        wort := präfix + erstes Zeichen von präfix;
        neues_wort := wort;
    end
    gib wort aus;
    trage neues_wort in Codetabelle ein;
    präfix := wort;
end
```

LZW Algorithmus – Beispiel "Dekodieren"

Dekodieren von 0 1 2 4 6 5 3

Initiale Codetabelle: 0:A 1:B 2:C 3:D

Code	präfix	wort	Code	Ausgabe
0		A		A
1	A	B	4:AB	B
2	B	C	5:BC	C
4	C	AB	6:CA	AB
6	AB	CA	7:ABC	CA
5	CA	BC	8:CAB	BC
3	BC	D	9:BCD	D

Vorteile von LZW

- Wörterbuch und Codierung werden häufig gewechselt: LZW kann sich einem Kontextwechsel im Eingabestrom gut anpassen.
- Um den damit verbundenen "Gedächtnisverlust" zu beschränken, kann man die Zeichen aus dem Wörterbuch (nach Anzahl der Benutzung und Länge) bewerten und die besten n Zeichen behalten.

Lempel-Ziv Algorithmen

LZ77 (Sliding Window)

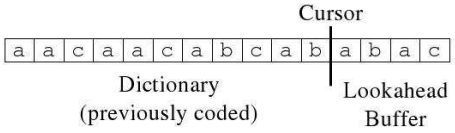
- Varianten: LZSS (Lempel-Ziv-Storer-Szymanski)
- Anwendungen: `gzip`, Squeeze, LHA, PKZIP, ZOO

LZ78 (explizites Dictionary)

- Varianten: LZW (Lempel-Ziv-Welch), LZC (Lempel-Ziv-Compress)
- Anwendungen: `compress`, GIF, CCITT (modems), ARC, PAK

LZ77 galt ursprünglich im Vergleich zu LZ78 als besser komprimierend, aber zu langsam; auf heutigen, leistungsfähigeren Rechnern ist LZ77 ausreichend schnell.

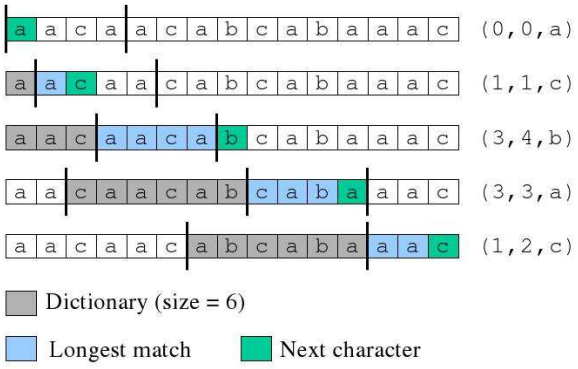
LZ77: Sliding Window Wörterbuch



Dictionary- und Buffer-Windows haben feste Länge und verschieben sich zusammen mit dem Cursor.
An jeder Cursor-Position passiert folgendes:

- Ausgabe des Tripels `(p,l,c)`
  - `p` = relative Position des longest match im Dictionary
  - `l` = Länge des longest match
  - `c` = nächstes Zeichen rechts vom longest match
- Verschiebe das Window um `l`

LZ77: Example



LZ77 Dekodierung

Der Dekodierer arbeitet mit den selben Windowlängen wie der Kodierer

- Im Falle des Tripels `(p,l,c)` geht er `p` Schritte zurück, liest die nächsten `l` Zeichen und kopiert diese nach hinten. Dann wird noch `c` angefügt.

Was ist im Falle `l>p`? (d.h. nur ein Teil der zu kopierenden Nachricht ist im Dictionary)

- Beispiel `dict = abcd, codeword = (2,9,e)`
- Lösung: Kopiere einfach zeichenweise:

```
for (i = 0; i < length; i++)
out[cursor+i] = out[cursor-offset+i]
```

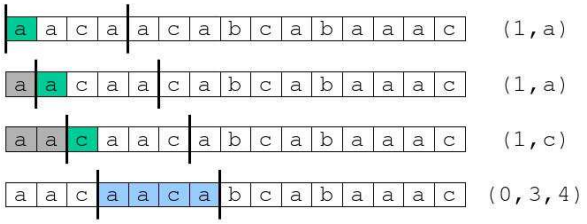
- `Out = abcdcdcdcdcdce`

Optimierungen in `gzip`: Verwendung von LZSS

LZSS verbessert LZ77: Im kodierten Output sind zwei verschiedene Formate erlaubt:

`(0, position, length)` oder `(1,char)`

Benutze das zweite Format, falls `length < 3`.



Weitere Optimierungen in `gzip`: `Deflate`-Algorithmus

- Nachträgliche Huffman-Codierung der Ausgabe
- Clevere Strategie bei der Codierung: Möglicherweise erlaubt ein kürzerer Match in aktuellen Schritt einen viel längeren Match im nächsten Schritt
- Benutze eine Hash-Tabelle für das Wörterbuch.
  - Hash-Funktion für Strings der Länge drei.
  - Suche für längere Strings im entsprechenden Überlaufbereich die längste Übereinstimmung.

LZ77 ist *asymptotisch optimal* [Wyner-Ziv,94], d.h.
LZ77 komprimiert hinreichend lange Strings entsprechend seiner Entropie, falls die Fenstergröße gegen unendlich geht.

$$H_n = \sum_{X \in A^n} p(X) \log \frac{1}{p(X)}$$

$$H = \lim_{n \to \infty} H_n$$

Achtung: um nah an dieses Optimum zu kommen, braucht man sehr grosse Fenster. Als Default in gzip wird z.B. 32KB verwendet.