Datenkompression:

Codes und Codierungen

H. Fernau

email: fernau@uni-trier.de

WiSe 2008/09

Universität Trier

Codierung

Für das Alphabet

$$\Sigma = \{a_1, ..., a_n\}$$

ist ein *Code* eine injektive Funktion

$$K : \Sigma \to \{0, 1\}^+.$$

Ziel: "Gute" Codierung der Folge

$$a_{i_1}a_{i_2}a_{i_3}a_{i_4}a_{i_5}\ldots \quad \to \quad K(a_{i_1})K(a_{i_2})K(a_{i_3})K(a_{i_4})K(a_{i_5})\ldots$$

Maß: Wenn jedes Symbol a_i mit Wahrscheinlichkeit $P(a_i)$ auftritt, definieren wir die *erwartete Länge von K* als

$$\boxed{L_K := \sum_{i=1}^{n} P(a_i) \cdot |K(a_i)|.}$$

Beispiel 1

Σ	P	K_1	K_2	K_3
a	0.5	0	0	0
b	0.25	1	10	01
c	0.125	00	110	011
d	0.125	11	111	0111
erwartete Länge		1.25	1.75	1.875

Eigenschaften

1. Codiere die Folge $baa \rightarrow 100 \rightarrow baa/bc$?
2. Codierung und Decodierung für K_2 – einfach und eindeutig.
3. Decodierung für K_3 – eindeutig, aber *nur mit Verzögerung* (nicht *unmittelbar decodierbar*).

Ein Code $K : \Sigma \rightarrow \{0,1\}^+$, mit $\Sigma = \{a_1, ..., a_n\}$, ist *eindeutig decodierbar*, wenn jeder gegebenen Konkatenation von Codewörtern $w_1 w_2 \dots w_m$ eindeutig eine Sequenz $a_{i_1}, a_{i_2}, \dots, a_{i_m}$ entspricht, wobei

$$K(a_{i_1})K(a_{i_2}) \dots K(a_{i_m}) = w_1 w_2 \dots w_m.$$

Beispiel 2 *Beliebige Verzögerungen sind möglich !*

Σ	K_4
a	0
b	01
c	11

Problem: Decodiere 011111111111111.

$\rightarrow$ 0 11 11 11 11 11 11 11 1**?!**

$\rightarrow$ 01 11 11 11 11 11 11 11

Frage:

Wie kann man entscheiden, ob ein Code eindeutig decodierbar ist?

Σ	K_5
a	0
b	01
c	10

Decodiere 0101010101010101010101010101010.

$\rightarrow$ 0 10 10 10 10 10 10 10 10 10 10 10 10 10 10

$\rightarrow$ 01 01 01 01 01 01 01 01 01 01 01 01 01 01 0

?!

Test für eindeutige Decodierbarkeit (Sardinas-Patterson)

Sei $w = w_1 w_2 \dots w_n \in \Sigma^*$; für $0 < k < n$, $u = w_1 w_2 \dots w_k$ ist ein *Präfix* von w und $\mathtt{suff}_u(w) = w_{k+1} w_{k+2} \dots w_n$.

1. Let C be a set of all codewords and let $S = \emptyset$;

2. For every pair u, w of words in C do
   if u is a prefix of w add $\mathtt{suff}_u(w)$ to S;

3. Repeat until there are no new entries added to S
   - For every $s \in S$ and for every Codewort $c \in C$
     - if s is a prefix of c then add $\mathtt{suff}_s(c)$ to S;
     - if c is a prefix of s then add $\mathtt{suff}_c(s)$ to S;
   - If obtaining a suffix that is a codeword ($S \cap C \neq \emptyset$) then Return(Code not uniquely decodable);

4. Return(Code uniquely decodable).

Frage:

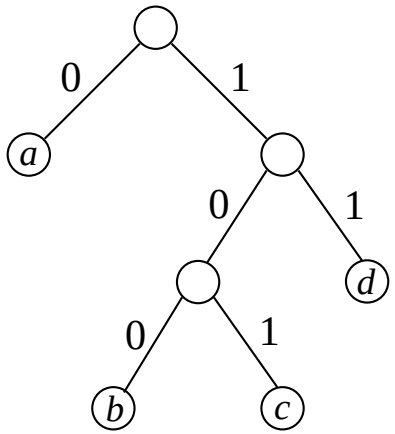
Wie kann man entscheiden, ob ein Code eindeutig decodierbar ist?

Σ	K_5
a	0
b	01
c	10

Eingangs: $C = \{0, 01, 10\}$, $S = \emptyset$.
Nach erster Schleife: $S = \{1\}$.
ein Durchlauf der nächsten Schleife: $S = \{1, 0\}$.
$C \cap S = \{0\} \rightsquigarrow K_5$ ist nicht eindeutig decodierbar.
Konstruktion eines konkreten Gegenbeispiels "von rechts" rückwärts möglich:
Widerspruch ergab sich durch Codewort 0 bzw. Suffix 0, also Codewort 10.
Die erste Decodierungsmöglichkeit muss durch Suffix 1, also Codewort 01, fortgesetzt werden.
$\rightsquigarrow (01)0 = 0(10)$ als kurzes Gegenbeispiel.

Präfixcodes

Die Codierung hat folgende Eigenschaft: *Kein Codewort ist ein Präfix des anderen Codeworts.*



Satz 3 *Jeder Präfixcode ist eindeutig decodierbar.*

Satz 4 *(Kraft-McMillan)*
Seien $\Sigma = \{a_1, \ldots, a_n\}$ ein Alphabet und $\ell_1, \ldots, \ell_n$ die Längen der Codewörter a_1 bis a_n.

1. *Wenn ein eindeutig decodierbarer Code $K : \Sigma \to \{0,1\}^+$ existiert mit $|K(a_1)| = \ell_1, \ldots, |K(a_n)| = \ell_n$, dann ist die Ungleichung*

$$\boxed{\sum_{i=1}^{n} 2^{-\ell_i} \leq 1}$$

erfüllt.

2. *Wenn $\sum_{i=1}^{n} 2^{-\ell_i} \leq 1$ erfüllt ist, dann existiert ein Präfixcode $K : \Sigma \to \{0,1\}^+$, mit $|K(a_1)| = \ell_1, \ldots, |K(a_n)| = \ell_n$.*

Korollar 5 *Zu jedem eindeutig decodierbaren Code gibt es einen Präfixcode mit gleichen Codewortlängen.*

Beweis von 1. Sei $C = \sum_{i=1}^{n} 2^{-\ell_i}$ und betrachten wir C^m, $m \in \mathbb{N}$.

$$\begin{aligned} C^m = [\textstyle\sum_{i=1}^{n} 2^{-\ell_i}]^m &= (\textstyle\sum_{i_1=1}^{n} 2^{-\ell_{i_1}})(\sum_{i_2=1}^{n} 2^{-\ell_{i_2}}) \ldots (\sum_{i_m=1}^{n} 2^{-\ell_{i_m}}) \\ &= \textstyle\sum_{i_1=1}^{n} \sum_{i_2=1}^{n} \cdots \sum_{i_m=1}^{n} 2^{-(\ell_{i_1}+\ell_{i_2}+\ldots+\ell_{i_m})}. \end{aligned}$$

$\ell_{i_1} + \ell_{i_2} + \ldots + \ell_{i_m}$: die Länge einer Konkat. von m Codewörtern.
Sei $\ell = \max\{\ell_1, \ldots, \ell_n\}$. Dann gilt:

$$m \leq \ell_{i_1} + \ell_{i_2} + \ldots + \ell_{i_m} \leq m \cdot \ell$$

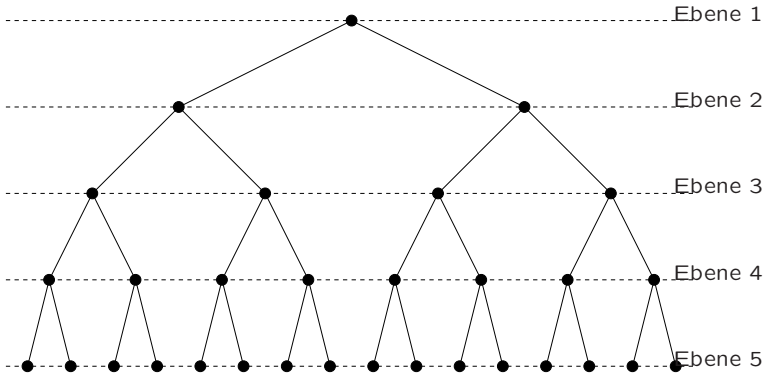
Andererseits: $C^m = \sum_{k=m}^{m\ell} A_k 2^{-k}$ mit
$A_k = \#$ Konkat. der Länge k von m Codewörtern.
Es gilt $\boxed{A_k \leq 2^k,}$ weil K eindeutig decodierbar ist, und das bedeutet, dass

$$C^m = \sum_{k=m}^{m\ell} A_k 2^{-k} \leq \sum_{k=m}^{m\ell} 2^k 2^{-k} = m\ell - m + 1.$$

Ist $C > 1$, dann wächst C^m exponentiell in m. Das heißt

$$\exists m \; C^m > m\ell - m + 1 \text{ ?!}$$

Beweis von 2. Es sei $\sum_{i=1}^{n} 2^{-\ell_i} \leq 1$ und $\ell_1 \leq \ell_2 \leq \ldots \leq \ell_n = m$. Betrachten wir nun einen vollständigen Binärbaum T der Höhe m.



Um den Präfixcode $K = \{c_1, \ldots, c_k\}$ zu erzeugen, benutzen wir:

`for` $k := 1$ `to` n `do`

1. `in Ebene` $\ell_k + 1$ `nimm Binärcodierung von Knoten` v `als` c_k, `sodass` $v \neq c_i$ `für alle` $i < k$ `mit` $\ell_i = \ell_k$
2. `entferne alle Nachfolger von` v `in` T.

Der Shannon-Algorithmus

Es sei $p_1 = P(a_1), \ldots, p_n = P(a_n)$ mit $p_1 \geq p_2 \geq \ldots \geq p_n$.

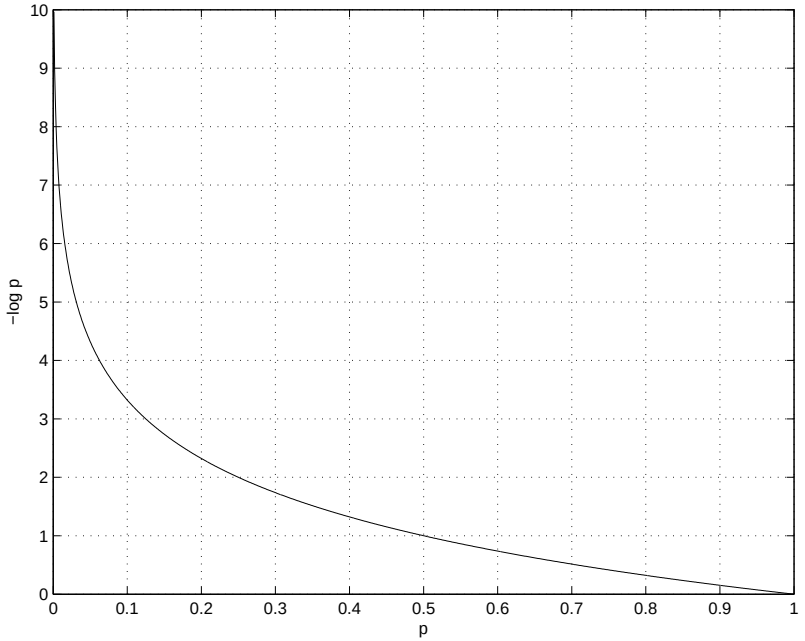
Dann $P_1 := 0$, und für $i > 1$ sei $P_i := p_1 + \ldots + p_{i-1}$.

`for` $i := 1$ `to` n `do`

$\ell_i := \lceil -\log p_i \rceil$;

`Sei` $P_i := 0.b_1 b_2 b_3 \ldots$

$K(a_i) := b_1 b_2 \ldots b_{\ell_i}$;



Beispiel 6 *Sei* $\Sigma = \{a, b, c, d, e\}$ *und*

$P(a) = 0.35$, $P(b) = 0.17$, $P(c) = 0.17$, $P(d) = 0.16$, $P(e) = 0.15$

	p_i	l_i	P_i	P_i (Binär)	Code
a	0.35	2	0.0	0.0000000...	00
b	0.17	3	0.35	0.0101100...	010
c	0.17	3	0.52	0.1000010...	100
d	0.16	3	0.69	0.1011000...	101
e	0.15	3	0.85	0.1101100...	110

$$\rightsquigarrow L_S = 0.35 \cdot 2 + 0.17 \cdot 3 + 0.17 \cdot 3 + 0.16 \cdot 3 + 0.15 \cdot 3 = 2.65$$

Es ist leicht zu sehen, dass die Shannon-Codierung nicht optimal ist.

Wir könnten z.B. b mit 01 codieren.

Satz 7 *Der Shannon-Algorithmus generiert einen Präfixcode.*

Beweis. Es sei $P_i = 0.b_1b_2b_3\ldots = \frac{b_1}{2^1} + \frac{b_2}{2^2} + \cdots$ Nach Konstruktion gilt

$$\log \frac{1}{p_i} \leq \ell_i,$$

womit für jedes $j \geq i+1$ gilt:

$$P_j - P_i \geq P_{i+1} - P_i = p_i \geq \frac{1}{2^{\ell_i}}.$$

Wegen $p_1 \geq p_2 \geq \ldots \geq p_n$ gilt $\ell_1 \leq \ell_2 \leq \ldots \leq \ell_n$.

Angenommen, es gibt ein i und ein j mit $i < j$ und

$$K(a_i) = b_1b_2\ldots b_{\ell_i}, \quad K(a_j) = c_1c_2\ldots c_{\ell_j}$$

wobei $\ell_j \geq \ell_i$. Ist $b_1 = c_1, \ldots, b_{\ell_i} = c_{\ell_i}$, so gilt

$$\begin{aligned} & P_j - P_i \\ = \; & \left(\frac{b_1}{2^1} + \cdots + \frac{b_{\ell_i}}{2^{\ell_i}} + \frac{c_{\ell_i+1}}{2^{\ell_i+1}} + \cdots\right) - \left(\frac{b_1}{2^1} + \cdots + \frac{b_{\ell_i}}{2^{\ell_i}} + \frac{b_{\ell_i+1}}{2^{\ell_i+1}} + \cdots\right) \\ < \; & 2^{-\ell_i}, \end{aligned}$$

also ein Widerspruch. ∎

Shannon-Fano-Codierung

Idee: Baumaufbau / Aufteilung von oben nach unten
Huffman-Codierung (s.u.) arbeitet genau andersherum

make-code(Σ)

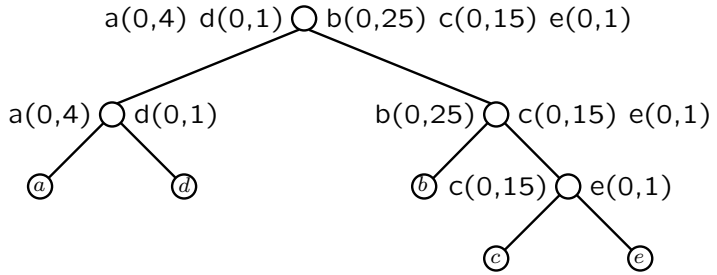
if $|\Sigma| = 1$ then return Binärcodierung von Knoten zu (a)

else

teile Σ auf in Σ_1 und Σ_2 mit $\sum_{a\in\Sigma_1} P(a) \approx \sum_{a\in\Sigma_2} P(a)$;

return node(make-code(Σ_1),make-code(Σ_2)).

Beispiel 8 $P(a) = 0.4,\ P(b) = 0.25,\ P(c) = 0.15$ *und* $P(d) = P(e) = 0.1$.



Der Ausgabecode ist:

	P	Code	Länge
a	0.4	00	2
b	0.25	10	2
c	0.15	110	3
d	0.1	01	2
e	0.1	111	3

Für diese Codierung ist die erwartete Länge

$$L_{S\text{-}F} = 2 \cdot 0.4 + 2 \cdot 0.25 + 3 \cdot 0.15 + 2 \cdot 0.1 + 3 \cdot 0.1 = 2.25$$

Für die Huffman-Codierung gilt hingegen (s.u.):

$$L_H = 1 \cdot 0.4 + 2 \cdot 0.25 + 3 \cdot 0.15 + 2 \cdot 4 \cdot 0.1 = 2.15.$$

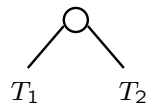
Huffman-Algorithmus

1. Starte mit dem Wald aus Bäumen, in dem jeder Baum ein Symbol darstellt und $w_i = P(a_i)$ das Gewicht des Baumes ist.

2. Repeat until Wald besteht aus nur einem Baum:

   - wähle die zwei Bäume T_1 und T_2 mit den kleinsten Gewichten w_1 und w_2;

   - nimm nun statt T_1 und T_2 den Baum

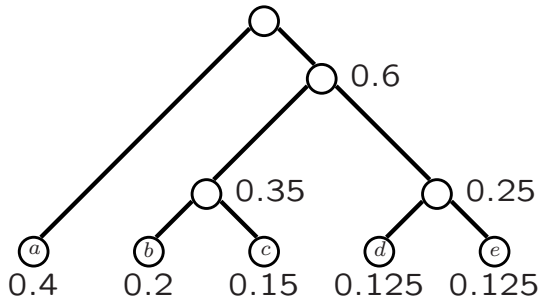


   mit dem Gewicht $w_1 + w_2$.

Beispiel 9 *$\Sigma = \{a, b, c, d, e\}$ und die Wahrscheinlichkeiten:*

$P(a)$	$P(b)$	$P(c)$	$P(d)$	$P(e)$
0.4	0.2	0.15	0.125	0.125

Der Algorithmus konstruiert den Ausgabebaum folgendermaßen:



Der Ausgabecode ist:

a	b	c	d	e
0	100	101	110	111

Satz 10 *Die erwartete Codelänge ist für Huffman-Codierung optimal.*

Betrachten wir jetzt die Qualität der Huffman-Codierung in Vergleich mit der Entropie, oder —anders gesagt— vergleichen wir nun die optimale erwartete Codelänge mit der Entropie.

Satz 11 *Sei Σ ein Quellenalphabet und $K : \Sigma \to \{0,1\}^+$ ein beliebiger Präfixcode. Dann gilt:*

$$L_K \geq H(\Sigma).$$

Satz 12 *Für jedes Quellenalphabet Σ ist die minimale erwartete Codelänge für Präfixcodes höchstens $H(\Sigma) + 1$.*

Dann bekommen wir als Korollar die folgende Abschätzung für die erwartete Länge der Huffman-Codierung:

Korollar 13 *Für jedes Quellenalphabet Σ gilt:*

$$\boxed{H(\Sigma) \leq L_H \leq H(\Sigma) + 1.}$$

Ein mathematischer Exkurs: Jensensche Ungleichung

Eine Funktion f heißt *konvex*, falls

$$f(\lambda x + (1-\lambda)y) \leq \lambda f(x) + (1-\lambda)f(y)$$

Satz 14 *(Jensen)*

Für eine konvexe Funktion f und für positive λ_i mit $\sum_{i=1}^{n} \lambda_i = 1$ gilt:

$$f\left(\sum_{i=1}^{n} \lambda_i x_i\right) \leq \sum_{i=1}^{n} \lambda_i f\left(x_i\right).$$

Korollar 15 *Ist f konvex und X eine Zufallsvariable, dann gilt $f(E(X)) \leq E(f(X))$.*

Daraus folgt unmittelbar, da f konkav, falls $-f$ konvex ist, die "umgekehrte Ungleichung" für konkave Funktionen wie den Logarithmus.

Zusammenhang Codelängen und Entropie

Es sei S eine Quelle mit Alphabet $\{a_1, \ldots, a_n\}$ und Wahrscheinlichkeiten $P(a_i)$.
Sei K ein eindeutig decodierbarer Code mit $\ell_i = |K(a_i)|$.
Sei $\ell = \sum_i P(a_i)\ell_i$ die erwartete Codewortlänge.

$$\begin{aligned} H(S) - \ell &= \sum_i P(a_i)(-\log P(a_i) - \ell_i) \\ &= \sum_i P(a_i) \log \left(\frac{2^{-\ell_i}}{P(a_i)} \right) \\ &\leq \log \sum_i 2^{-\ell_i} \leq 0 \end{aligned}$$

Die erste Ungleichung ergibt sich aus dem Satz von Jensen, die zweite aus dem Satz von Kraft-McMillan.

Huffman-Komprimierung ist moderat asymmetrisch

Codierer:

```
/* -- first pass -- */
initialize_frequencies;
while (ch != eof)
{
   ch = getchar(input);
   update_frequence(ch);
}
construct_tree(T);
puttree(output,T);
/* -- second pass -- */
initialize_input;
while (ch != eof)
{
   ch = getchar(input);
   put(output,encode(ch));
}
put(output,encode(eof));
```

Decodierer:

```
gettree(input,T);
while ((ch=decode(input)) != eof)
   putchar(output,ch);
```

Erweiterte Huffman-Codierung

Beispiel 16 *Sei* $\Sigma = \{a, b\}$ *und* $P(a) = 0.9$; $P(b) = 0.1$.

Dann liefert der Huffman-Algorithmus folgende Codierung:

	P	Code
a	0.9	0
b	0.1	1

mit der erwarteten Länge $1 \cdot 0.9 + 1 \cdot 0.1 = 1$ *Bits/Symbol. Die Entropie ist*

$$H(\Sigma) = 0.9 \cdot \log \frac{10}{9} + 0.1 \cdot \log 10 = 0.47 \ .$$

Die Redundanz – die Differenz zwischen der erwarteten Länge und der Entropie – ist

$$0.53 \text{ Bits/Symbol},$$

also fast 113% der Entropie.

Beispiel 17 *Betrachten wir das neue Alphabet*

$$\Sigma^2 = \{aa, ab, ba, bb\}$$

mit

$$P(aa) = [P(a)]^2,\ P(bb) = [P(b)]^2 \text{ und } P(ab) = P(ba) = P(a) \cdot P(b).$$

Der Huffman-Algorithmus konstruiert folgende Codierung:

	P	Code	Länge
aa	0.81	0	1
ab	0.09	10	2
ba	0.09	110	3
bb	0.01	111	3

Die erwartete Länge ist

$$1 \cdot 0.81 + 2 \cdot 0.09 + 3 \cdot 0.09 + 3 \cdot 0.01 = 1.29 \text{ Bits/Symbol in } \Sigma^2,$$

also

$$0.645 \text{ Bits/Symbol in } \Sigma$$

und jetzt benutzt unsere Codierung nur 37% mehr Bits als eine minimale Codierung.

Beispiel 18 *Für das Alphabet*

$$\Sigma^3 = \{aaa, aab, aba, baa, abb, bab, bba, bbb\}$$

ist der Unterschied noch kleiner:

	P	Code	Länge
aaa	0.729	0	1
aab	0.081	100	3
aba	0.081	101	3
baa	0.081	110	3
abb	0.009	11100	5
bab	0.009	11101	5
bba	0.009	11110	5
bbb	0.001	11111	5

und die erwartete Länge ist

0.52 *Bits/Symbol in* Σ

nur um 11% mehr als die Entropie $H(\Sigma)$.

Sei H^m die erweiterte Huffman-Codierung für die Blöcke der Länge m. Dann bekommen wir die folgende Ungleichung:

Satz 19 *Für jedes Quellenalphabet Σ gilt:*

$$\boxed{H(\Sigma) \leq L_{H^m} \leq H(\Sigma) + \frac{1}{m}.}$$

Beweis Wir zeigen, dass

$$H(\Sigma^m) = mH(\Sigma).$$

Darauf wenden wir Korollar 13 an. ∎

Anmerkungen

Wir haben gesehen, dass die Wahrscheinlichkeit für jedes neue 'Zeichen' $a_{i_1}a_{i_2}\ldots a_{i_m}$ des Alphabets Σ^m das Produkt von Wahrscheinlichkeiten $P(a_1) \times P(a_2) \times \ldots P(a_m)$ ist.
$\rightsquigarrow$ Die Wahrscheinlichkeiten für die Zeichen innerhalb eines Blocks sind unabhängig.

In der Praxis haben wir aber sehr selten mit einer solchen Situation zu tun. Für die englische Sprache z.B. ist die Wahrscheinlichkeit, dass das Zeichen 'a' im U. S. Grundgesetz vorkommt gleich 0.057, aber die Wahrscheinlichkeit für die Reihenfolge von 10 'a's ist nicht 0.057^{10} sondern 0. In der Praxis muss man für jedes m eine neue Statistik für Σ^m konstruieren.

Praktische Probleme mit Huffman

Woher bekommt man die "Wahrscheinlichkeiten" der Quelle ?!
$\leadsto$ Zwei-Pass-Vorgehensweise
Pass 1: Ermittle Häufigkeiten der Zeichen der Quelle
Pass 2: Führe Huffman-Codierung (evtl. erweitert) durch

Praktisch of wirkungsvolle Idee: *Adaption*
Benutze die Statistik der ersten k Zeichen, um das $(k+1)$ste Zeichen zu codieren und erstelle neue Statistik (Update).

Bäume mit Geschwistereigenschaft (s.u.) sind geeignete Datenstruktur für Adaption.

$\rightsquigarrow$ **Adaptive Huffman-Codierung** (sehr symmetrisch)

Codierer:

```
initialize_tree(T);
while (ch != eof)
{
   ch = getchar(input);
   put(output, encode(ch));
   update_tree(ch);
}
```

Decodierer:

```
initialize_tree(T);
while (ch != eof)
{
   ch = decode(input);
   putchar(output,ch);
   update_tree(ch);
}
```

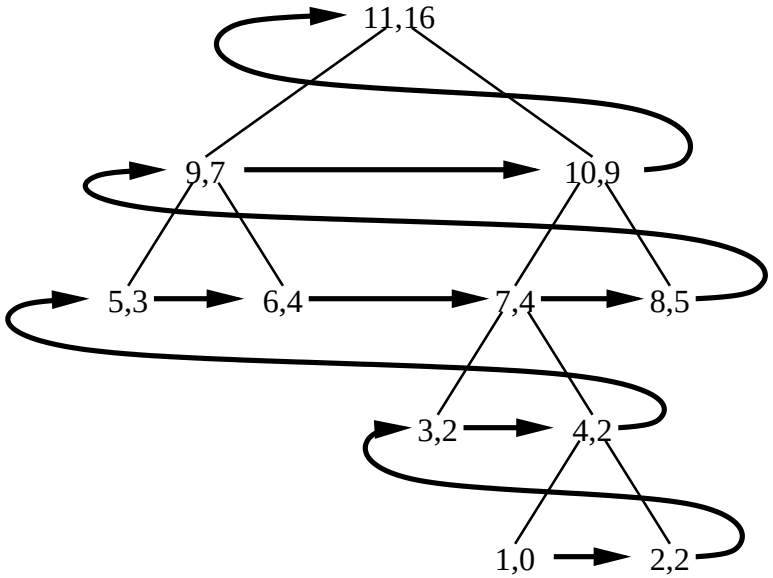
Es sei Σ mit $|\Sigma| = n-1$. Wir betrachten $\Sigma \cup \{\texttt{NYT}\}$ (`NYT` - not yet transmitted).

Jeder Knoten eines Binärbaumes (jeder Knoten *outdeg* $= 2$ oder 0):

Knotennumer $\ell_i \in \{1, 2, \ldots 2n-1\}$ und Knotengewicht $w_i \in \mathbb{N}_0$.

Geschwister-Eigenschaft Für $j = 1, .., n$:

ℓ_{2j-1} und ℓ_{2j} sind Geschwister eines Vaters mit $\ell_k > \ell_{2j-1}, \ell_{2j}$ und $w_k = w_{2j-1} + w_{2j}$ und $w_1 \leq w_2 \leq w_3 \ldots \leq w_{2n-1}$.



Erläuterung:
Linke Zahl: ℓ_i
Rechte Zahl: w_i

Block: alle Knoten mit gleichem Gewicht.

Fakt Jeder Baum mit Geschwister-Eigenschaft ist ein Huffman-Baum.

Initially: `current_number = 2n-1; root.number = 2n-1; root.weight=0;`

```
void update_tree(char ch)
{
   if first_appearance(ch)  {
      p = leaf(NYT); p->weight++;
      q = new node(ch);
      q->father = p; q->weight = 1; q->number = -current_number;
      r = new node(NYT);
      r->father = p; r->weight = 0; r->number = -current_number;
      p = p->father;
   }
   else p = leaf(ch);
   while (!root(p))  {
      if (!(sibling(p, leaf(NYT))) && !max_number_in_block(p))
      { let q highest numbered node in block; switch(p,q); }
      p->weight++;
      p = p->father;
   }
}
```

NAME

pack, pcat, unpack - compress and expand files

DESCRIPTION pack attempts to store the specified files in a compressed form. Wherever possible, each input file name is replaced by a packed file name.z with the same ownership, modes, and access and modification times. The -f option forces packing of name. This is useful for causing an entire directory to be packed even if some of the files do not benefit. If pack is successful, name is removed. Packed files can be restored to their original form using unpack or pcat.

pack uses Huffman (minimum redundancy) codes on a byte-by-byte basis. If the - argument is used, an internal flag is set that causes the number of times each byte is used, its relative frequency, and the code for the byte to be printed on the standard output. Additional occurrences of - in place of name cause the internal flag to be set and reset.

NAME

compact, uncompact, ccat - compact and uncompact files, and cat them

DESCRIPTION compact compresses the named files using an adaptive Huffman code. If no file names are given, standard input is compacted and sent to the standard output. compact operates as an on-line algorithm. Each time a byte is read, it is encoded immediately according to the current prefix code. This code is an optimal Huffman code for the set of frequencies seen so far. It is unnecessary to attach a decoding tree in front of the compressed file because the encoder and the decoder start in the same state and stay synchronized.

	`file: script.ps`	`pack`	`compact`	`gzip`
Size	3135129	1955524	1955681	800374
Compression	0%	37.6%	37.62%	74%
time	–	0.590s	2.854s	1.284s

	`file: f.tar`	`pack`	`compact`	`gzip`
Size	719.3 M	661.87 M	661.87 M	473.1 M
Compression	0%	8.0%	7.99%	34.2%
time	–	1m35s	15m51s	17m16s

Literatur

1. K. Sayood. *Introduction to Data Compression*, Morgan Kaufmann Publishers, Inc., 3. Auflage, 2006.
2. D. Knuth. *Dynamic Huffman Coding*, Journal of Algorithms, 6 (1985), 136-180.
3. J. Vitter. *Design and Analysis of Dynamic Huffman Codes*, Journal of ACM, 34 (1987), 825-845.
4. D. Salomon. *Data Compression; The Complete Reference*, Springer, 1998.