13. Compression and Decompression

Algorithms for data compression usually proceed as follows. They encode a text over some finite alphabet into a sequence of bits, hereby exploiting the fact that the letters of this alphabet occur with different frequencies. For instance, an "e" occurs more frequently than a "q" and will therefore be assigned a shorter codeword. The quality of the compression procedure is then measured in terms of the average codeword length.

So the underlying model is probabilistic, namely we consider a finite alphabet and a probability distribution on this alphabet, where the probability distribution reflects the (relative) frequencies of the letters. Such a pair – an alphabet with a probability distribution – is called a source. We shall first introduce some basic facts from Information Theory. Most important is the notion of entropy, since the source entropy characterizes the achievable lower bounds for compressibility.

The source model to be best understood, is the discrete memoryless source. Here the letters occur independently of each other in the text. The use of prefix codes, in which no codeword is the beginning of another one, allows to compress the text down to the entropy of the source. We shall study this in detail. The lower bound is obtained via Kraft's inequality, the achievability is demonstrated by the use of Huffman codes, which can be shown to be optimal.

There are some assumptions on the discrete memoryless source, which are not fulfilled in most practical situations. Firstly, usually this source model is not realistic, since the letters do not occur independently in the text. Secondly, the probability distribution is not known in advance. So the coding algorithms should be universal for a whole class of probability distributions on the alphabet. The analysis of such universal coding techniques is much more involved than the analysis of the discrete memoryless source, such that we shall only present the algorithms and do not prove the quality of their performance. Universal coding techniques mainly fall into two classes.

Statistical coding techniques estimate the probability of the next letters as accurately as possible. This process is called modelling of the source. Having enough information about the probabilities, the text is encoded, where usually arithmetic coding is applied. Here the probability is represented by an interval and this interval will be encoded.

Dictionary–based algorithms store patterns, which occurred before in the text, in a dictionary and at the next occurrence of a pattern this is encoded via its position in the dictionary. The most prominent procedure of this kind is due to Ziv and Lempel.

We shall also present a third universal coding technique which falls in neither of these

two classes. The algorithm due to Burrows and Wheeler has become quite prominent in recent years, since implementations based on it perform very well in practice.

All algorithms mentioned so far are lossless, i. e., there is no information lost after decoding. So the original text will be recovered without any errors. In contrast, there are lossy data compression techniques, where the text obtained after decoding does not completely coincide with the original text. Lossy compression algorithms are used in applications like image, sound, video, or speech compression. The loss should, of course, only marginally effect the quality. For instance, frequencies not realizable by the human eye or ear can be dropped. However, the understanding of such techniques requires a solid background in image, sound or speech processing, which would be far beyond the scope of this paper, such that we shall illustrate only the basic concepts behind image compression algorithms such as JPEG.

We emphasize here the recent developments such as the Burrows–Wheeler transform and the context–tree weighting method. Rigorous proofs will only be presented for the results on the discrete memoryless source which is best understood but not a very realistic source model in practice. However, it is also the basis for more complicated source models, where the calculations involve conditional probabilities. The asymptotic computational complexity of compression algorithms is often linear in the text length, since the algorithms simply parse through the text. However, the running time relevant for practical implementations is mostly determined by the constants as dictionary size in Ziv-Lempel coding or depth of the context tree, when arithmetic coding is applied. Further, an exact analysis or comparison of compression algorithms often heavily depends on the structure of the source or the type of file to be compressed, such that usually the performance of compression algorithms is tested on benchmark files. The most well-known collections of benchmark files are the Calgary Corpus and the Canterbury Corpus.

13.1. Facts from information theory

13.1.1. The discrete memoryless source

The source model discussed throughout this chapter is the ***Discrete Memoryless Source*** (DMS). Such a source is a pair $(\mathcal{X}, P)$, where $\mathcal{X} = \{1, \ldots, a\}$ is a finite alphabet and $P = (P(1), \ldots, P(a))$ is a probability distribution on $\mathcal{X}$. A discrete memoryless source can also be described by a random variable X, where $\text{Prob}(X = x) = P(x)$ for all $x \in \mathcal{X}$. A word $x^n = (x_1 x_2 \ldots x_n) \in \mathcal{X}^n$ is the realization of the random variable $(X_1 \ldots X_n)$, where the X_i-s are identically distributed and independent of each other. So the probability $P^n(x_1 x_2 \ldots x_n) = P(x_1) \cdot P(x_2) \cdot \cdots \cdot P(x_n)$ is the product of the probabilities of the single letters.

Estimations for letter probabilities in natural languages are obtained by statistical methods. If we consider the English language and choose the latin alphabet with an additional symbol for Space and punctuation marks for $\mathcal{X}$, the probability distribution can be derived from the frequency table in 13.1 which is obtained from the copy–fitting tables used by professional printers. So $P(A) = 0.064$, $P(B) = 0.014$, etc.

Observe that this source model is often not realistic. For instance, in English texts e.g. the combination ‘th’ occurs more often than ‘ht’. This could not be the case if an English text was produced by a discrete memoryless source, since then $P(th) = P(t) \cdot P(h) = P(ht)$.

A	64	H	42	N	56	U	31
B	14	I	63	O	56	V	10
C	27	J	3	P	17	W	10
D	35	K	6	Q	4	X	3
E	100	L	35	R	49	Y	18
F	20	M	20	S	56	Z	2
G	14	T	71				

Space/Punctuation mark 166

Figure 13.1. Frequency of letters in 1000 characters of English

In the discussion of the communication model it was pointed out that the encoder wants to compress the original data into a short sequence of binary digits, hereby using a binary code, i. e., a function $c : \mathcal{X} \longrightarrow \{0,1\}^* = \bigcup_{n=0}^{\infty} \{0,1\}^n$. To each element $x \in \mathcal{X}$ a codeword $c(x)$ is assigned. The aim of the encoder is to minimize the average length of the codewords. It turns out that the best possible data compression can be described in terms of the ***entropy*** $H(P)$ of the probability distribution P. The entropy is given by the formula

$$H(P) = -\sum_{x \in \mathcal{X}} P(x) \cdot \lg P(x)$$

where the logarithm is to the base 2. We shall also use the notation $H(x)$ according to the interpretation of the source as a random variable.

13.1.2. Prefix codes

A ***code*** (of variable length) is a function $c : \mathcal{X} \longrightarrow \{0,1\}^*$, $\mathcal{X} = \{1, \ldots, a\}$. Here $\{c(1), c(2), \ldots, c(a)\}$ is the set of ***codewords***, where for $x = 1, \ldots, a$ the codeword is $c(x) = (c_1(x), c_2(x), \ldots, c_{L(x)}(x))$ where $L(x)$ denotes the ***length*** of $c(x)$, i. e., the number of bits used to present $c(x)$.

In the following example some binary codes for the latin alphabet are presented. (SP=Space/punctuation mark)

$\overline{c} : a \longrightarrow 1, b \longrightarrow 10, c \longrightarrow 100, d \longrightarrow 1000, \ldots, z \longrightarrow \underbrace{10\ldots0}_{26}, SP \longrightarrow \underbrace{10\ldots0}_{27}.$

$\hat{c} : a \longrightarrow 00000, b \longrightarrow 00001, c \longrightarrow 00010, \ldots, z \longrightarrow 11001, SP \longrightarrow 11010$. So $\hat{c}(x)$ is the binary representation of the position of letter x in the alphabetic order.

$\check{c} : a \longrightarrow 0, b \longrightarrow 00, c \longrightarrow 1, \ldots$ (the further codewords are not important for the following discussion).

The last code presented has an undesirable property. Observe that the sequence 00 could either be decoded as b or as aa. Hence the messages encoded using this code are not uniquely decipherable.

A code c is ***uniquely decipherable*** (UDC) , if every word in $\{0,1\}^*$ is representable by at most one sequence of codewords.

Code $\overline{c}$ is uniquely decipherable, since the number of 0s between two 1s determines the next letter in a message encoded using $\overline{c}$. Code $\hat{c}$ is uniquely decipherable, since every

letter is encoded with exactly five bits. Hence the first five bits of a sequence of binary digits are decoded as the first letter of the original text, the bits 6 to 10 as the second letter, etc.

A code c is a ***prefix code,*** if for any two codewords $c(x)$ and $c(y)$, $x \neq y$, with $L(x) \leq L(y)$ holds $(c_1(x), c_2(x), \ldots, c_{L(x)}(x)) \neq (c_1(y), c_2(y), \ldots, c_{L(x)}(y))$. So $c(x)$ and $c(y)$ differ in at least one of the first $L(x)$ components.

Messages encoded using a prefix code are uniquely decipherable. The decoder proceeds by reading the next letter until a codeword $c(x)$ is formed. Since $c(x)$ cannot be the beginning of another codeword, it must correspond to letter $x \in \mathcal{X}$. Now the decoder continues until another codeword is formed. The process may be repeated until the end of the message. So after having found codeword $c(x)$ the decoder instantaneously knows that $x \in \mathcal{X}$ is the next letter of the message. Because of this property a prefix code is also denoted as instantaneous code. Observe that code $\overline{c}$ is not instantaneous, since every codeword is the beginning of the following codewords.

The criterion for data compression is to minimize the average length of the codewords. So if we are given a source $(\mathcal{X}, P)$, where $\mathcal{X} = \{1, \ldots, a\}$ and $P = (P(1), P(2), \ldots, P(a))$ is a probability distribution on $\mathcal{X}$, the ***average length*** $\overline{L}(c)$ is defined by

$$\overline{L}(c) = \sum_{x \in \mathcal{X}} P(x) \cdot L(x) \ .$$

If in English texts all letters (incl. Space/punctuation mark) occured with the same frequency, then code $\overline{c}$ would have an average length of $\frac{1}{27}(1 + 2 + \cdots + 27) = \frac{1}{27} \cdot \frac{27 \cdot 28}{2} = 14$. Hence code $\hat{c}$ with an average length of 5 would be more appropriate in this case. From the frequency table of the previous section we know that the occurrence of the letters in English texts cannot be modelled by the uniform distribution. In this case it is possible to find a better code by assigning short codewords to letters with high probability as demonstrated by the following prefix code c with average length $\overline{L}(c) = 3 \cdot 0.266 + 4 \cdot 0.415 + 5 \cdot 0.190 + 6 \cdot 0.101 + 7 \cdot 0.016 + 8 \cdot 0.012 = 4.222$.

$a \longrightarrow 0110,$	$b \longrightarrow 010111,$	$c \longrightarrow 10001,$	$d \longrightarrow 01001\ ,$
$e \longrightarrow 110,$	$f \longrightarrow 11111,$	$g \longrightarrow 111110,$	$h \longrightarrow 00100\ ,$
$i \longrightarrow 0111,$	$j \longrightarrow 11110110,$	$k \longrightarrow 1111010,$	$l \longrightarrow 01010\ ,$
$m \longrightarrow 001010,$	$n \longrightarrow 1010,$	$o \longrightarrow 1001,$	$p \longrightarrow 010011\ ,$
$q \longrightarrow 01011010,$	$r \longrightarrow 1110,$	$s \longrightarrow 1011,$	$t \longrightarrow 0011\ ,$
$u \longrightarrow 10000,$	$v \longrightarrow 0101100,$	$w \longrightarrow 001011,$	$x \longrightarrow 01011011\ ,$
$y \longrightarrow 010010,$	$z \longrightarrow 11110111,$	$SP \longrightarrow 000\ .$	

We can still do better, if we do not encode single letters, but blocks of n letters for some $n \in N$. In this case we replace the source $(\mathcal{X}, P)$ by $(\mathcal{X}^n, P^n)$ for some $n \in N$. Remember that $P^n(x_1 x_2 \ldots x_n) = P(x_1) \cdot P(x_2) \cdot \cdots \cdot P(x_n)$ for a word $(x_1 x_2 \ldots x_n) \in \mathcal{X}^n$, since the source is memoryless. If e.g. we are given an alphabet with two letters, $\mathcal{X} = \{a, b\}$ and $P(a) = 0.9$, $P(b) = 0.1$, then the code c defined by $c(a) = 0$, $c(b) = 1$ has average length $\overline{L}(c) = 0.9 \cdot 1 + 0.1 \cdot 1 = 1$. Obviously we cannot find a better code. The combinations of two letters now have the following probabilities:

$$P^2(aa) = 0.81, \quad P^2(ab) = 0.09, \quad P^2(ba) = 0.09, \quad P^2(bb) = 0.01\ .$$

The prefix code c^2 defined by

$$c^2(aa) = 0, \quad c^2(ab) = 10, \quad c^2(ba) = 110, \quad c^2(bb) = 111$$

has average length $\overline{L}(c^2) = 1 \cdot 0.81 + 2 \cdot 0.09 + 3 \cdot 0.09 + 3 \cdot 0.01 = 1.29$. So $\frac{1}{2}\overline{L}(c^2) = 0.645$ could be interpreted as the average length the code c^2 requires per letter of the alphabet $\mathcal{X}$. When we encode blocks of n letters we are interested in the behaviour of

$$L(n, P) = \min_{cUDC} \frac{1}{n} \sum_{(x_1 \ldots x_n) \in \mathcal{X}^n} P^n(x_1 \ldots x_n) L(x_1 \ldots x_n) = \min_{cUDC} \overline{L}(c) \; .$$

It follows from the noiseless coding theorem, which is stated in the next section, that $\lim_{n \longrightarrow \infty} L(n, P) = H(P)$, the entropy of the source $(\mathcal{X}, P)$.

In our example for the English language we have $H(P) \approx 4.19$. So the code presented above, where only single letters are encoded, is already nearly optimal in respect of $L(n, P)$. Further compression is possible if we consider the dependencies between the letters.

13.1.3. Kraft's inequality and the noiseless coding theorem

We shall now introduce a necessary and sufficient condition for the existence of a prefix code for $\mathcal{X} = \{1, \ldots, a\}$ with prescribed word lengths $L(1), \ldots, L(a)$.

Theorem 13.1 (Kraft's inequality). *Let $\mathcal{X} = \{1, \ldots, a\}$. A prefix code $c : \mathcal{X} \longrightarrow \{0, 1\}^*$ with word lengths $L(1), \ldots, L(a)$ exists, if and only if*

$$\sum_{x \in \mathcal{X}} 2^{-L(x)} \leq 1.$$

Proof. The central idea is to interpret the codewords as nodes of a rooted binary tree with depth $T = \max_{x \in \mathcal{X}}\{L(x)\}$. The tree is required to be complete (every path from the root to a leaf has length T) and regular (every inner node has outdegree 2). The example in Figure 13.2 for $T = 3$ may serve as an illustration.

So the nodes with distance n from the root are labelled with the words $x^n \in \{0, 1\}^n$. The upper successor of $x_1 x_2 \ldots x_n$ is labelled $x_1 x_2 \ldots x_n 0$, its lower successor is labelled $x_1 x_2 \ldots x_n 1$.

The ***shadow*** of a node labelled by $x_1 x_2 \ldots x_n$ is the set of all the leaves which are labelled by a word (of length T) beginning with $x_1 x_2 \ldots x_n$. In other words, the shadow of $x_1 \ldots x_n$ consists of the leaves labelled by a sequence with prefix $x_1 \ldots x_n$. In our example $\{000, 001, 010, 011\}$ is the shadow of the node labelled by 0.

Now assume that we are given a prefix code with word lengths $L(1), \ldots, L(a)$. Every codeword corresponds to a node in the binary tree of depth T. Observe that the shadows of any two codewords are disjoint. If this was not the case, we could find a word $x_1 x_2 \ldots x_T$, which has as prefix two codewords of length s and t, say (w.l.o.g. $s < t$). But these codewords are $x_1 x_2 \ldots x_t$ and $x_1 x_2 \ldots x_s$ which obviously is a prefix of the first one.

The shadow of codeword $c(x)$ has size $2^{T - L(x)}$. There are 2^T words of length T. For the sum of the shadow sizes follows $\sum_{x \in \mathcal{X}} 2^{T - L(x)} \leq 2^T$, since none of these words can be a

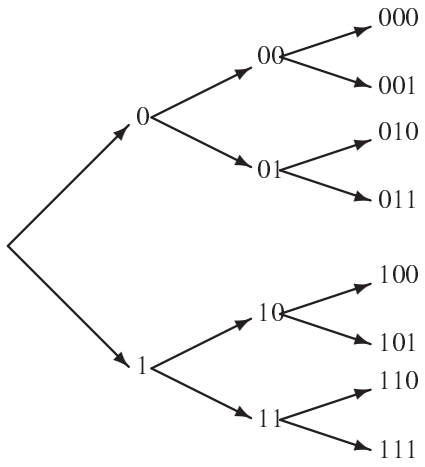

Figure 13.2. Example of a code tree.

member of two shadows. Division by 2^T yields the desired inequality $\sum_{x\in\mathcal{X}} 2^{-L(x)} \le 1$.

Conversely, suppose we are given positive integers $L(1), \dots, L(a)$. We further assume that $L(1) \le L(2) \le \cdots \le L(a)$. As first codeword $c(1) = \underbrace{00\dots0}_{L(1)}$ is chosen. Since $\sum_{x\in\mathcal{X}} 2^{T-L(x)} \le 2^T$, we have $2^{T-L(1)} < 2^T$ (otherwise only one letter has to be encoded). Hence there are some nodes left on the T–th level, which are not in the shadow of $c(1)$. We pick the first of these remaining nodes and go back $T - L(2)$ steps in the direction of the root. Since $L(2) \ge L(1)$, we shall find a node labelled by a sequence of $L(2)$ bits, which is not a prefix of $c(1)$. So we can choose this sequence as $c(2)$. Now again, either $a = 2$, and we are ready, or by the hypothesis $2^{T-L(1)} + 2^{T-L(2)} < 2^T$ and we can find a node on the T–th level, which is not contained in the shadows of $c(1)$ and $c(2)$. We find the next codeword as shown above. The process can be continued until all codewords are assigned. ■

Kraft's inequality gives a necessary and sufficient condition for the existence of a prefix code with codewords of lengths $L(1), \dots, L(a)$. In the following theorem it is shown that this condition is also necessary for the existence of a uniquely decipherable code. This can be interpreted in such a way that it is sufficient to consider only prefix codes, since one cannot expect a better performance by any other uniquely decipherable code.

Theorem 13.2 (Kraft's inequality for uniquely decipherable codes). *A uniquely decipherable code with prescribed word lengths $L(1), \dots, L(a)$ exists, if and only if*

$$\sum_{x\in\mathcal{X}} 2^{-L(x)} \le 1.$$

Proof. Since every prefix code is uniquely decipherable, the sufficiency part of the proof is immediate. Now observe that $\sum_{x\in\mathcal{X}} 2^{-L(x)} = \sum_{j=1}^{T} w_j\, 2^{-j}$, where w_j is the number of codewords with length j in the uniquely decipherable code and T again denotes the maximal word

length. The s–th power of this term can be expanded as

$$\left(\sum_{j=1}^{T} w_j\, 2^{-j}\right)^s = \sum_{k=s}^{T\cdot s} N_k\, 2^{-k}.$$

Here $N_k = \sum_{i_1+\cdots+i_s=k} w_{i_1}\ldots w_{i_s}$ is the total number of messages whose coded representation is of length k. Since the code is uniquely decipherable, to every sequence of k letters corresponds at most one possible message. Hence $N_k \le 2^k$ and $\sum_{k=s}^{T\cdot s} N_k\, 2^{-k} \le \sum_{k=s}^{T\cdot s} 1 = T\cdot s - s + 1 \le T\cdot s$. Taking s–th root this yields $\sum_{j=1}^{T} w_j\, 2^{-j} \le (T\cdot s)^{\frac{1}{s}}$.

Since this inequality holds for any s and $\lim_{s\longrightarrow\infty} (T\cdot s)^{\frac{1}{s}} = 1$, we have the desired result

$$\sum_{j=1}^{T} w_j\, 2^{-j} = \sum_{x\in\mathcal{X}} 2^{-L(x)} \le 1\ .$$

■

Theorem 13.3 (Noiseless coding theorem). *For a source $(\mathcal{X}, P)$, $\mathcal{X} = \{1,\ldots,a\}$, it is always possible to find a uniquely decipherable code $c : \mathcal{X} \longrightarrow \{01,\}^*$ with an average length of*

$$H(P) \le L_{\min}(P) < H(P) + 1.$$

Proof. Let $L(1),\ldots,L(a)$ denote the codeword lengths of an optimal uniquely decipherable code. Now we define a probability distribution Q on $\mathcal{X} = \{1,\ldots,a\}$ by $Q(x) = \frac{2^{-L(x)}}{r}$ for $x \in \mathcal{X}$, where $r = \sum_{x=1}^{a} 2^{-L(x)}$. By Kraft's inequality $r \le 1$.

For two probability distributions P and Q on $\mathcal{X}$ the ***I–divergence*** $D(P\|Q)$ is defined by

$$D(P\|Q) = \sum_{x\in\mathcal{X}} P(x) \lg \frac{P(x)}{Q(x)}$$

I-divergence is a good measure for the distance of two probability distributions. Especially, always the I–divergence $D(P\|Q) \ge 0$. So for any probability distribution P

$$D(P\|Q) = -H(P) - \sum_{x\in\mathcal{X}} P(x)\cdot \lg(2^{-L(x)}\cdot r^{-1}) \ge 0.$$

From this it follows that

$$\begin{aligned} H(P) &\le -\sum_{x\in\mathcal{X}} P(x)\cdot \lg(2^{-L(x)}\cdot r^{-1}) \\ &= \sum_{x\in\mathcal{X}} P(x)\cdot L(x) - \sum_{x\in\mathcal{X}} P(x)\cdot \lg r^{-1} = L_{\min}(P) + \lg r. \end{aligned}$$

Since $r \le 1$, $\lg r \le 0$ and hence $L_{\min}(P) \ge H(P)$.

In order to prove the right-hand side of the noiseless coding theorem for $x = 1,\ldots,a$ we define $L'(x) = \lceil -\lg P(x)\rceil$. Observe that $-\lg P(x) \le L'(x) < -\lg P(x) + 1$ and hence

x	$P(x)$	$Q(x)$	$\overline{Q}(x)$	$\lceil \lg \frac{1}{P(x)} \rceil$	$c_S(x)$	$c_{SFE}(x)$
1	0.25	0	0.125	2	00	001
2	0.2	0.25	0.35	3	010	0101
3	0.11	0.45	0.505	4	0111	10001
4	0.11	0.56	0.615	4	1000	10100
5	0.11	0.67	0.725	4	1010	10111
6	0.11	0.78	0.835	4	1100	11010
7	0.11	0.89	0.945	4	1110	11110
				$\overline{L}$	3.3	4.3

Figure 13.3. Example of Shannon code and Shannon-Fano-Elias code.

$P(x) \geq 2^{-L'(x)}$.

So $1 = \sum_{x \in \mathcal{X}} P(x) \geq \sum_{x \in \mathcal{X}} 2^{-L'(x)}$ and from Kraft's Inequality we know that there exists a uniquely decipherable code with word lengths $L'(1), \ldots, L'(a)$. This code has an average length of

$$\sum_{x \in \mathcal{X}} P(x) \cdot L'(x) < \sum_{x \in \mathcal{X}} P(x)(-\lg P(x) + 1) = H(P) + 1.$$

■

13.1.4. Shannon-Fano-Elias codes and the Shannon-Fano algorithm

In the proof of the noiseless coding theorem it was explicitly shown how to construct a prefix code c to a given probability distribution $P = (P(1), \ldots, P(a))$. The idea was to assign to each $x \in \{1, \ldots, a\}$ a codeword of length $L(x) = \lceil \lg \frac{1}{P(x)} \rceil$ by choosing an appropriate vertex in the tree introduced. However, this procedure does not always yield an optimal code. If e.g. we are given the probability distribution $(\frac{1}{3}, \frac{1}{3}, \frac{1}{3})$, we would encode $1 \longrightarrow 00$, $2 \longrightarrow 01$, $3 \longrightarrow 10$ and thus achieve an average codeword length of 2. But the code with $1 \longrightarrow 00$, $2 \longrightarrow 01, 3 \longrightarrow 1$ has only average length of $\frac{5}{3}$.

Shannon gave an explicit procedure for obtaining codes with codeword lengths $\lceil \lg \frac{1}{P(x)} \rceil$ using the binary representation of cumulative probabilities (Shannon remarked this procedure was originally due to Fano). The elements of the source are ordered according to increasing probabilities $P(1) \geq P(2) \geq \cdots \geq P(a)$. Then codeword $c_S(x)$ consists of the first $\lceil \lg \frac{1}{P(x)} \rceil$ bits of the binary expansion of the sum $Q(x) = \sum_{j<x} P(j)$.

This procedure was further developed by Elias . The elements of the source now may occur in any order. The ***Shannon-Fano-Elias code*** has $c_{SFE}(x)$ the first $\lceil \lg \frac{1}{P(x)} \rceil + 1$ bits of the binary expansion of the sum $\overline{Q}(x) = \sum_{j<x} P(j) + \frac{1}{2}P(x)$ as codewords.

We shall illustrate these procedures by the example in Figure 13.3.

A more efficient procedure is also due to Shannon and Fano. The ***Shannon-Fano algorithm*** will be illustrated by the same example in Figure 13.4:

The messages are first written in order of nonincreasing probabilities. Then the message set is partitioned into two most equiprobable subsets X_0 and X_1. A 0 is assigned to each

x	$P(x)$	$c(x)$	$L(x)$
1	0.25	00	2
2	0.2	01	2
3	0.11	100	3
4	0.11	101	3
5	0.11	110	3
6	0.11	1110	4
7	0.11	1111	4
		$\overline{L}(c)$	2.77

Figure 13.4. Example of the Shannon-Fano algorithm.

message contained in the first subset and a 1 to each of the remaining messages. The same procedure is repeated for subsets of X_0 and X_1; that is, X_0 will be partitioned into two subsets X_{00} and X_{01}. Now the code word corresponding to a message contained in X_{00} will start with 00 and that corresponding to a message in X_{01} will begin with 01. This procedure is continued until each subset contains only one message.

However, this algorithm does not yield an optimal code in general, since the prefix code $1 \longrightarrow 01, 2 \longrightarrow 000, 3 \longrightarrow 001, 4 \longrightarrow 110, 5 \longrightarrow 111, 6 \longrightarrow 100, 7 \longrightarrow 101$ has an average length of 2.75.

13.1.5. The Huffman coding algorithm

The ***Huffman coding algorithm*** is a recursive procedure which we shall illustrate with the same example as for the Shannon-Fano algorithm in Figure 13.5 with $p_x = P(x)$ and $c_x = c(x)$. The source is successively reduced by one element. In each reduction step we add up the two smallest probabilities and insert their sum $P(a) + P(a-1)$ in the increasingly ordered sequence $P(1) \geq \cdots \geq P(a-2)$, thus obtaining a new probability distribution P' with $P'(1) \geq \cdots \geq P'(a-1)$. Finally we arrive at a source with two elements ordered according to their probabilities. The first element is assigned a 0, the second element a 1. Now we again "blow up" the source until the original source is restored. In each step $c(a-1)$ and $c(a)$ are obtained by appending 0 or 1, respectively, to the codeword corresponding to $P(a) + P(a-1)$.

Correctness

The following theorem demonstrates that the Huffman coding algorithm always yields a prefix code optimal with respect to the average codeword length.

Theorem 13.4 *We are given a source $(\mathcal{X}, P)$, where $\mathcal{X} = \{1, \ldots, a\}$ and the probabilities are ordered non–increasingly: $P(1) \geq P(2) \geq \cdots \geq P(a)$. A new probability distribution is defined by*

$$P' = (P(1), \ldots, P(a-2), P(a-1) + P(a)).$$

Let $c' = (c'(1), c'(2), \ldots, c'(a-1))$ be an optimal prefix code for P'. Now we define a code c for the distribution P by

p_1	0.25	p_1	0.25	p_1	0.25	P_{23}	0.31	p_{4567}	0.44	p_{123}	0.56
p_2	0.2	p_{67}	0.22	p_{67}	0.22	p_1	0.25	p_{23}	0.31	p_{4567}	0.44
p_3	0.11	p_2	0.2	p_{45}	0.22	p_{67}	0.22	p_1	0.25		
p_4	0.11	p_3	0.11	p_2	0.2	p_{45}	0.22				
p_5	0.11	p_4	0.11	p_3	0.11						
P_6	0.11	p_5	0.11								
p_7	0.11										

C_{123}	0	c_{4567}	1	c_{23}	00	c_1	01	c_1	01	c_1	01
c_{4567}	1	c_{23}	00	c_1	01	c_{67}	10	c_{67}	10	c_2	000
		c_1	01	c_{67}	10	c_{45}	11	c_2	000	c_3	001
				c_{45}	11	c_2	000	c_3	001	c_4	110
						c_3	001	c_4	110	c_5	111
								c_5	111	c_6	100
										c_7	101

Figure 13.5. Example of a Huffman code.

$$\begin{aligned} c(x) &= c'(x) \text{ for } x = 1, \ldots, a-2, \\ c(a-1) &= c'(a-1)0, \\ c(a) &= c'(a-1)1. \end{aligned}$$

In this case c is an optimal prefix code for P and $L_{\min}(P) - L_{\min}(P') = p(a-1) + p(a)$.

Proof. For a probability distribution P on $\mathcal{X} = \{1, \ldots, a\}$ with $P(1) \geq P(2) \geq \cdots \geq P(a)$ there exists an optimal prefix code c with

i) $L(1) \leq L(2) \leq \cdots \leq L(a)$

ii) $L(a-1) = L(a)$

iii) $c(a-1)$ and $c(a)$ differ exactly in the last position.

This holds, since:

i) Assume that there are $x, y \in \mathcal{X}$ with $P(x) \geq P(y)$ and $L(x) > L(y)$. In this case the code c' obtained by interchanging codewords $c(x)$ and $c(y)$ has average length $\overline{L}(c') \leq \overline{L}(c)$, since

$$\begin{aligned} \overline{L}(c') - \overline{L}(c) &= P(x) \cdot L(y) + P(y) \cdot L(x) - P(x) \cdot L(x) - P(y) \cdot L(y) \\ &= (P(x) - P(y)) \cdot (L(y) - L(x)) \leq 0 \end{aligned}$$

ii) Assume we are given a code c' with $L(1) \leq \cdots \leq L(a-1) < L(a)$. Because of the prefix property we may drop the last $L(a) - L(a-1)$ bits of $c'(a)$ and thus obtain a new code c with $L(a) = L(a-1)$.

iii) If no two codewords of maximal length agree in all places but the last, then we may drop the last digit of all such codewords to obtain a better code.

Now we are ready to prove the statement from the theorem. From the definition of c and c' we have

$$L_{\min}(P) \leq \overline{L}(c) = \overline{L}(c') + p(a-1) + p(a).$$

Now let c'' be an optimal prefix code with the properties ii) and iii) from the preceding lemma. We define a prefix code c''' for

$$P' = (P(1), \ldots, P(a-2), P(a-1) + P(a))$$

by $c'''(x) = c''(x)$ for $x = 1, \ldots, a-2$ and $c'''(a-1)$ is obtained by dropping the last bit of $c''(a-1)$ or $c''(a)$.

Now

$$\begin{aligned} L_{\min}(P) &= \overline{L}(c'') = \overline{L}(c''') + P(a-1) + P(a) \\ &\geq L_{\min}(P') + P(a-1) + P(a) \end{aligned}$$

and hence $L_{\min}(P) - L_{\min}(P') = P(a-1) + P(a)$, since $\overline{L}(c') = L_{\min}(P')$.

■

Analysis

If a denotes the size of the source alphabet, the Huffman coding algorithm needs $a-1$ additions and $a-1$ code modifications (appending 0 or 1). Further we need $a-1$ insertions, such that the total complexity can be roughly estimated to be $O(a \lg a)$. However, observe that with the noiseless coding theorem, the quality of the compression rate can only be improved by jointly encoding blocks of, say, k letters, which would result in a Huffman code of size a^k for the source $\mathcal{X}^k$. So, the price of better compression is a rather drastic increase in complexity. Further, the codewords for all a^k letters have to be stored. Encoding a sequence of n letters can therefore be done in $O(\frac{n}{k} \cdot (a^k \lg a^k))$ steps.

Exercises

13.1-1 Show that the code $c : \{a, b\} \longrightarrow \{0, 1\}^*$ with $c(a) = 0$ and $c(b) = \underbrace{0 \ldots 01}_{n}$ is uniquely decipherable but not instantaneous for any $n > 0$.

13.1-2 Compute the entropy of the source $(\mathcal{X}, P)$, with $\mathcal{X} = \{1, 2\}$ and $P = (0.8, 0.2)$.

13.1-3 Find the Huffman codes and the Shannon-Fano codes for the sources $(\mathcal{X}^n, P^n)$ with $(\mathcal{X}, P)$ as in the previous exercise for $n = 1, 2, 3$ and calculate their average codeword lengths.

13.1-4 Show that $0 \leq H(P) \leq \lg |\mathcal{X}|$ for every source.

13.1-5 Show that the redundancy $\rho(c) = \overline{L}(c) - H(P)$ of a prefix code c for a source with probability distribution P can be expressed as a special I–divergence.

13.1-6 Show that the I-divergence $D(P\|Q) \geq 0$ for all probability distributions P and Q over some alphabet $\mathcal{X}$ with equality exactly if $P = Q$, but that the I–divergence is not a metric.

13.2. Arithmetic coding and modelling

In statistical coding techniques as Shannon-Fano or Huffman coding the probability distribution of the source is modelled as accurately as possible and then the words are encoded such that a higher probability results in a shorter codeword length.

We know that Huffman codes are optimal with respect to the average codeword length. However, the entropy is approached by increasing the block length. On the other hand, for long blocks of source symbols, Huffman coding is a rather complex procedure, since it requires the calculation of the probabilities of all sequences of the given block length and the construction of the corresponding complete code.

For compression techniques based on statistical methods often ***arithmetic coding*** is preferred. Arithmetic coding is a straightforward extension of the Shannon-Fano-Elias code. The idea is to represent a probability by an interval. In order to do so, the probabilities have to be calculated very accurately. This process is denoted as ***modelling*** of the source . So statistical compression techniques consist of two stages: modelling and coding. As just mentioned, coding is usually done by arithmetic coding. The different algorithms like, for instance, DCM (Discrete Markov Coding) and PPM (Prediction by Partial Matching) vary in the way of modelling the source. We are going to present the context–tree weighting method, a transparent algorithm for the estimation of block probabilities due to Willems, Shtarkov, and Tjalkens, which also allows a straightforward analysis of the efficiency.

13.2.1. Arithmetic coding

The idea behind arithmetic coding is to represent a message $x^n = (x_1 \dots x_n)$ by interval $I(x^n) = [Q^n(x^n), Q^n(x^n) + P^n(x^n))$, where $Q^n(x^n) = \sum_{y^n < x^n} P^n(y^n)$ is the sum of the probabilities of those sequences which are smaller than x^n in lexicographic order.

A codeword $c(x^n)$ assigned to message x^n also corresponds to an interval. Namely, we identify codeword $c = c(x^n)$ of length $L = L(x^n)$ with interval $J(c) = [bin(c), bin(c) + 2^{-L})$, where $bin(c)$ is the binary expansion of the nominator in the fraction $\frac{c}{2^L}$. The special choice of codeword $c(x^n)$ will be obtained from $P^n(x^n)$ and $Q^n(x^n)$ as follows:

$$L(x^n) = \lceil \lg \frac{1}{P^n(x^n)} \rceil + 1, \qquad bin(c) = \lceil Q^n(x^n) \cdot 2^{L(x^n)} \rceil.$$

So message x^n is encoded by a codeword $c(x^n)$, whose interval $J(x^n)$ is inside interval $I(x^n)$.

Let us illustrate arithmetic coding by the following example of a discrete memoryless source with $P(1) = 0.1$ and $n = 2$.

x^n	$P^n(x^n)$	$Q^n(x^n)$	$L(x^n)$	$c(x^n)$
00	0.81	0.00	2	00
01	0.09	0.81	5	11010
10	0.09	0.90	5	11101
11	0.01	0.99	8	11111110

At first glance it may seem that this code is much worse than the Huffman code for the same source with codeword lengths $(1, 2, 3, 3)$ we found previously. On the other hand, it can be shown that arithmetic coding always achieves an average codeword length

$\overline{L}(c) < H(P^n) + 2$, which is only two bits apart from the lower bound in the noiseless coding theorem. Huffman coding would usually yield an even better code. However, this "negligible" loss in compression rate is compensated by several advantages. The codeword is directly computed from the source sequence, which means that we do not have to store the code as in the case of Huffman coding. Further, the relevant source models allow to easily compute $P^n(x_1x_2\ldots x_{n-1}x_n)$ and $Q^n(x_1x_2\ldots x_{n-1}x_n)$ from $P^{n-1}(x_1x_2\ldots x_{n-1})$, usually by multiplication by $P(x_n)$. This means that the sequence to be encoded can be parsed sequentially bit by bit, unlike in Huffman coding, where we would have to encode blockwise.

Encoding: The basic algorithm for encoding a sequence $(x_1\ldots x_n)$ by arithmetic coding works as follows. We assume that $P^n(x_1\ldots x_n) = P_1(x_1)\cdot P_2(x_2)\cdots P_n(x_n)$, (in the case $P_i = P$ for all i the discrete memoryless source arises, but in the section on modelling more complicated formulae come into play) and hence $Q_i(x_i) = \sum_{y<x_i} P_i(x_i)$

Starting with $B_0 = 0$ and $A_0 = 1$ the first i letters of the text to be compressed determine the ***current interval*** $[B_i, B_i + A_i)$. These current intervals are successively refined via the recursions

$$B_{i+1} = B_i + A_i \cdot Q_i(x_i), \quad A_{i+1} = A_i \cdot P_i(x_i)$$

$A_i \cdot P_i(x)$ is usually denoted as augend. The final interval $[B_n, B_n + A_n) = [Q^n(x^n), Q^n(x^n) + P^n(x^n))$ will then be encoded by interval $J(x^n)$ as described above. So the algorithm looks as follows.

ARITHMETIC–ENCODER(x)
1 $B \leftarrow 0$
2 $A \leftarrow 1$
3 **for** $i \leftarrow 1$ **to** n
4 $\quad B \leftarrow B + A \cdot Q_i(x[i])$
5 $\quad$ **do** $A \leftarrow A \cdot P_i(x[i])$
6 $L \leftarrow \lceil \lg \frac{1}{A} \rceil + 1$
7 $c \leftarrow \lceil B \cdot 2^L \rceil$
8 **return** c

We shall illustrate the encoding procedure by the following example from the literature. Let the discrete, memoryless source $(\mathcal{X}, P)$ be given with ternary alphabet $\mathcal{X} = \{1, 2, 3\}$ and $P(1) = 0.4$, $P(2) = 0.5$, $P(3) = 0.1$. The sequence $x^4 = (2, 2, 2, 3)$ has to be encoded. Observe that $P_i = P$ and $Q_i = Q$ for all $i = 1, 2, 3, 4$. Further $Q(1) = 0$, $Q(2) = P(1) = 0.4$, and $Q(3) = P(1) + P(2) = 0.9$.

The above algorithm yields

i	B_i	A_i
0	0	1
1	$B_0 + A_0 \cdot Q(2) = 0.4$	$A_0 \cdot P(2) = 0.5$
2	$B_1 + A_1 \cdot Q(2) = 0.6$	$A_1 \cdot P(2) = 0.25$
3	$B_2 + A_2 \cdot Q(2) = 0.7$	$A_2 \cdot P(2) = 0.125$
4	$B_3 + A_3 \cdot Q(3) = 0.8125$	$A_3 \cdot P(3) = 0.0125$

Hence $Q(2, 2, 2, 3) = B_4 = 0.8125$ and $P(2, 2, 2, 3) = A_4 = 0.0125$. From this can be calculated that $L = \lceil \lg \frac{1}{A} \rceil + 1 = 8$ and finally $\lceil B \cdot 2^L \rceil = \lceil 0.8125 \cdot 256 \rceil = 208$ whose binary representation is codeword $c(2, 2, 2, 3) = 11010000$.

Decoding: Decoding is very similar to encoding. The decoder recursively "undoes" the encoder's recursion. We divide the interval $[0,1)$ into subintervals with bounds defined by Q_i. Then we find the interval in which codeword c can be found. This interval determines the next symbol. Then we subtract $Q_i(x_i)$ and rescale by multiplication by $\frac{1}{P_i(x_i)}$.

ARITHMETIC–DECODER(c)
1 **for** $i \leftarrow 1$ **to** n
2 **do** $j \leftarrow 1$
3 **while** $(c < Q_i(j))$ **do** $j \leftarrow j+1$
4 $x[i] \leftarrow j-1$
5 $c \leftarrow (c - Q_i(x[i]))/P_i(x[i])$
6 **return** x

Observe that when the decoder only receives codeword c he does not know when the decoding procedure terminates. For instance $c = 0$ can be the codeword for $x^1 = (1)$, $x^2 = (1,1)$, $x^3 = (1,1,1)$, etc. In the above pseudocode it is implicit that the number n of symbols has also been transmitted to the decoder, in which case it is clear what the last letter to be encoded was. Another possibility would be to provide a special end-of-file (EOF)-symbol with a small probability, which is known to both the encoder and the decoder. When the decoder sees this symbol, he stops decoding. In this case line 1 would be replaced by

1 **while** $(x[i] \neq \text{EOF})$

(and i would have to be increased). In our above example, the decoder would receive the codeword 11010000, the binary expansion of 0.8125 up to $L = 8$ bits. This number falls in the interval $[0.4, 0.9)$ which belongs to the letter 2, hence the first letter $x_1 = 2$. Then he calculates $(0.8075 - Q(2))\frac{1}{P(2)} = (0.815 - 0.4) \cdot 2 = 0.83$. Again this number is in the interval $[0.4, 0.9)$ and the second letter is $x_2 = 2$. In order to determine x_3 the calculation $(0.83 - Q(2))\frac{1}{P(2)} = (0.83 - 0.4) \cdot 2 = 0.86$ must be performed. Again $0.86 \in [0.4, 0.9)$ such that also $x_3 = 2$. Finally $(0.86 - Q(2))\frac{1}{P(2)} = (0.86 - 0.4) \cdot 2 = 0.92$. Since $0.92 \in [0.9, 1)$, the last letter of the sequence must be $x_4 = 3$.

Correctness

Recall that message x^n is encoded by a codeword $c(x^n)$, whose interval $J(x^n)$ is inside interval $I(x^n)$. This follows from $\lceil Q^n(x^n) \cdot 2^{L(x^n)} \rceil 2^{-L(x^n)} + 2^{-L(x^n)} < Q^n(x^n) + 2^{1-L(x^n)} = Q^n(x^n) + 2^{-\lceil \lg \frac{1}{P^n(x^n)} \rceil} \leq Q^n(x^n) + P^n(x^n)$.

Obviously a prefix code is obtained, since a codeword can only be a prefix of another one, if their corresponding intervals overlap – and the intervals $J(x^n) \subset I(x^n)$ are obviously disjoint for different n-s.

Further, we mentioned already that arithmetic coding compresses down to the entropy up to two bits. This is because for every sequence x^n it is $L(x^n) < \lg \frac{1}{P^n(x^n)} + 2$. It can also be shown that the additional transmission of block length n or the introduction of the EOF symbol only results in a negligible loss of compression.

However, the basic algorithms we presented are not useful in order to compress longer files, since with increasing block length n the intervals are getting smaller and smaller, such that rounding errors will be unavoidable. We shall present a technique to overcome this problem in the following.

Analysis

The basic algorithm for arithmetic coding is linear in the length n of the sequence to be

encoded. Usually, arithmetic coding is compared to Huffman coding. In contrast to Huffman coding, we do not have to store the whole code, but can obtain the codeword directly from the corresponding interval. However, for a discrete memoryless source, where the probability distribution $P_i = P$ is the same for all letters, this is not such a big advantage, since the Huffman code will be the same for all letters (or blocks of k letters) and hence has to be computed only once. Huffman coding, on the other hand, does not use any multiplications which slow down arithmetic coding.

For the adaptive case, in which the P_i's may change for different letters x_i to be encoded, a new Huffman code would have to be calculated for each new letter. In this case, usually arithmetic coding is preferred. We shall investigate such situations in the section on modelling.

For implementations in practice floating point arithmetic is avoided. Instead, the subdivision of the interval $[0, 1)$ is represented by a subdivision of the integer range $0, \ldots, M$, say, with proportions according to the source probabilities. Now integer arithmetic can be applied, which is faster and more precise.

Precision problem.

In the basic algorithms for arithmetic encoding and decoding the shrinking of the current interval would require the use of high precision arithmetic for longer sequences. Further, no bit of the codeword is produced until the complete sequence x^n has been read in. This can be overcome by coding ech bit as soon as it is known and then double the length of the current interval $[LO, HI)$, say, so that this expansion represents only the unknown part of the interval. This is the case when the leading bits of the lower and upper bound are the same, i. e. the interval is completely contained either in $[0, \frac{1}{2})$ or in $[\frac{1}{2}, 1)$. The following expansion rules guarantee that the current interval does not become too small.

Case 1 ($[LO, HI) \in [0, \frac{1}{2})$): $LO \leftarrow 2 \cdot Lo$, $HI \leftarrow 2 \cdot HI$.

Case 2 ($[LO, HI) \in [\frac{1}{2}, 1)$): $LO \leftarrow 2 \cdot LO - 1$, $HI \leftarrow 2 \cdot HI - 1$.

Case 3 ($\frac{1}{4} \leq LO < \frac{1}{2} \leq HI < \frac{3}{4}$): $LO \leftarrow 2 \cdot LO - \frac{1}{2}$, $HI \leftarrow 2 \cdot HI - \frac{1}{2}$.

The last case called ***underflow*** (or follow) prevents the interval from shrinking too much when the bounds are close to $\frac{1}{2}$. Observe that if the current interval is contained in $[\frac{1}{4}, \frac{3}{4})$ with $LO < \frac{1}{2} \leq HI$, we do not know the next output bit, but we do know that whatever it is, the following bit will have the opposite value. However, in contrast to the other cases we cannot continue encoding here, but have to wait (remain in the underflow state and adjust a counter *underflowcount* to the number of subsequent underflows, i. e. $underflowcount \leftarrow underflowcount + 1$) until the current interval falls into either $[0, \frac{1}{2})$ or $[\frac{1}{2}, 1)$. In this case we encode the leading bit of this interval – 0 for $[0, \frac{1}{2})$ and 1 for $[\frac{1}{2}, 1)$ – followed by *underflowcount* many inverse bits and then set $underflowcount = 0$. The procedure stops, when all letters are read in and the current interval does not allow any further expansion.

ARITHMETIC-PRECISION-ENCODER(x)
1 $LO \leftarrow 0$
2 $HI \leftarrow 1$
3 $A \leftarrow 1$
4 $underflowcount \leftarrow 0$
5 **for** $i \leftarrow 1$ **to** n
6 **do** $LO \leftarrow LO + Q_i(x[i]) \cdot A$

```
7       A ← P_i(x[i])
8       HI ← LO + A
9       while HI − LO < 1/2 AND NOT (LO < 1/4 AND HI ≥ 1/2)
10          do if HI < 1/2
11                then c ← c||0, underflowcount many 1s
12                     underflowcount ← 0
13                     LO ← 2 · LO
14                     HI ← 2 · HI
15             else if LO ≥ 1/2
16                then c ← c||1, underflowcount many 0s
17                     underflowcount ← 0
18                     LO ← 2 · LO − 1
19                     HI ← 2 · HI − 1
20             else if LO ≥ 1/4 AND HI < 3/4
21                then underflowcount ← underflowcount + 1
22                     LO ← 2 · LO − 1/2
23                     HI ← 2 · HI − 1/2
24 if underflowcount > 0
25    then c ← c||0, underflowcount many 1s)
26 return c
```

We shall illustrate the encoding algorithm in Figure 13.6 by our example – the encoding of the message $(2, 2, 2, 3)$ with alphabet $\mathcal{X} = \{1, 2, 3\}$ and probability distribution $P = (0.4, 0.5, 0.1)$. An underflow occurs in the sixth row: we keep track of the underflow state and later encode the inverse of the next bit, here this inverse bit is the 0 in the ninth row. The encoded string is 1101000.

Precision – decoding involves the consideration of a third variable besides the interval bounds LO and HI.

13.2.2. Modelling

Modelling of memoryless sources with The Krichevsky-Trofimov Estimator

In this section we shall only consider binary sequences $x^n \in \{0, 1\}^n$ to be compressed by an arithmetic coder. Further, we shortly write $P(x^n)$ instead of $P^n(x^n)$ in order to allow further subscripts and superscripts for the description of the special situation. P_e will denote estimated probabilities, P_w weighted probabilities, and P^s probabilities assigned to a special context s.

The application of arithmetic coding is quite appropriate if the probability distribution of the source is such that $P(x_1x_2 \ldots x_{n-1}x_n)$ can easily be calculated from $P(x_1x_2 \ldots x_{n-1})$. Obviously this is the case, when the source is discrete and memoryless, since then $P(x_1x_2 \ldots x_{n-1}x_n) = P(x_1x_2 \ldots x_{n-1}) \cdot P(x_n)$.

Even when the underlying parameter $\theta = P(1)$ of a binary, discrete memoryless source is not known, there is an efficient way due to Krichevsky and Trofimov to estimate the probabilities via

$$P(X_n = 1|x^{n-1}) = \frac{b + \frac{1}{2}}{a + b + 1},$$

Current Interval	Action	Subintervals 1	2	3	Input
$[0.00, 1.00)$	subdivide	$[0.00, 0.40)$	$[0.40, 0.90)$	$[0.90, 1.00)$	2
$[0.40, 0.90)$	subdivide	$[0.40, 0.60)$	$[0.60, 0.85)$	$[0.85, 0.90)$	2
$[0.60, 0.85)$	encode 1				
	expand $[\frac{1}{2}, 1)$				
$[0.20, 0.70)$	subdivide	$[0.20, 0.40)$	$[0.40, 0.65)$	$[0.65, 0.70)$	2
$[0.40, 0.65)$	underflow				
	expand $[\frac{1}{4}, \frac{3}{4})$				
$[0.30, 0.80)$	subdivide	$[0.30, 0.50)$	$[0.50, 0.75)$	$[0.75, 0.80)$	3
$[0.75, 0.80)$	encode 10				
	expand $[\frac{1}{2}, 1)$				
$[0.50, 0.60)$	encode 1				
	expand $[\frac{1}{2}, 1)$				
$[0.00, 0.20)$	encode 0				
	expand $[0, \frac{1}{2})$				
$[0.00, 0.40)$	encode 0				
	expand $[0, \frac{1}{2})$				
$[0.00, 0.80)$	encode 0				

Figure 13.6. Example of Arithmetic encoding with interval expansion.

a	b	0	1	2	3	4	5
0		1	1/2	3/8	5/16	35/128	63/256
1		1/2	1/8	1/16	5/128	7/256	21/1024
2		3/8	1/16	3/128	3/256	7/1024	9/2048
3		5/16	5/128	3/256	5/1024	5/2048	45/32768

Figure 13.7. Table of the first values for the Krichevsky-Trofimov estimator.

where a and b denote the number of 0s and 1s, respectively, in the sequence $x^{n-1} = (x_1 x_2 \dots x_{n-1})$. So given the sequence x^{n-1} with a many 0s and b many 1s, the probability that the next letter x_n will be a 1 is estimated as $\frac{b+\frac{1}{2}}{a+b+1}$. The estimated block probability of a sequence containing a zeros and b ones then is

$$P_e(a,b) = \frac{\frac{1}{2} \cdots (a-\frac{1}{2}) \frac{1}{2} \cdots (b-\frac{1}{2})}{1 \cdot 2 \cdots (a+b)}$$

with initial values $a = 0$ and $b = 0$ as in Figure 13.7, where the values of the ***Krichevsky-Trofimov estimator*** $P_e(a,b)$ for small (a,b) are listed.

Note that the summand $\frac{1}{2}$ in the nominator guarantees that the probability for the next letter to be a 1 is positive even when the symbol 1 did not occur in the sequence so far. In order to avoid infinite codeword length, this phenomenon has to be carefully taken into account when estimating the probability of the next letter in all approaches to estimate the parameters, when arithmetic coding is applied.

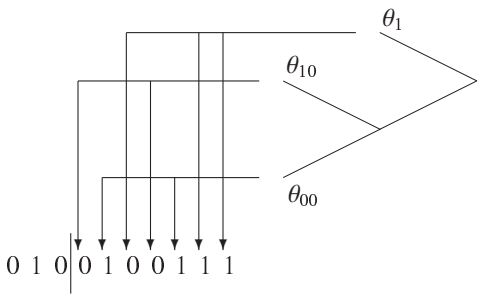

Figure 13.8. An example for a tree source.

Models with known context tree

In most situations the source is not memoryless, i. e., the dependencies between the letters have to be considered. A suitable way to represent such dependencies is the use of a suffix tree, which we denote as ***context tree***. The context of symbol x_t is suffix s preceding x_t. To each context (or leaf in the suffix tree) s there corresponds a parameter $\theta_s = P(X_t = 1|s)$, which is the probability of the occurrence of a 1 when the last sequence of past source symbols is equal to context s (and hence $1 - \theta_s$ is the probability for a 0 in this case). We are distinguishing here between the model (the suffix tree) and the parameters (θ_s).

Example 13.1 Let $\mathcal{S} = \{00, 10, 1\}$ and $\theta_{00} = \frac{1}{2}$, $\theta_{10} = \frac{1}{3}$, and $\theta_1 = \frac{1}{5}$. The corresponding suffix tree jointly with the parsing process for a special sequence can be seen in Figure 13.8.

The actual probability of the sequence '0100111' given the past '...010' is $P^s(0100111|\ldots 010) = (1-\theta_{10})\theta_{00}(1-\theta_1)(1-\theta_{10})\theta_{00}\theta_1\theta_1 = \frac{2}{3}\cdot\frac{1}{2}\cdot\frac{4}{5}\cdot\frac{2}{3}\cdot\frac{1}{2}\cdot\frac{1}{5}\cdot\frac{1}{5} = \frac{4}{1075}$, since the first letter 0 is preceded by suffix 10, the second letter 1 is preceded by suffix 00, etc.

Suppose the model $\mathcal{S}$ is known, but not the parameters θ_s. The problem now is to find a good coding distribution for this case. The tree structure allows to easily determine which context precedes a particular symbol. All symbols having the same context (or suffix) $s \in \mathcal{S}$ form a memoryless source subsequence whose probability is determined by the unknown parameter θ_s. In our example these subsequences are '11' for θ_{00}, '00' for θ_{10} and '011' for θ_1. One uses the Krichevsky-Trofimov-estimator for this case. To each node s in the suffix tree, we count the numbers a_s of zeros and b_s of ones preceded by suffix s. For the children $0s$ and $1s$ of parent node s obviously $a_{0s} + a_{1s} = a_s$ and $b_{0s} + b_{1s} = b_s$ must be satisfied.

In our example $(a_\lambda, b_\lambda) = (3,4)$ for the root λ, $(a_1, b_1) = (1,2)$, $(a_0, b_0) = (2,2)$ and $(a_{10}, b_{10}) = (2,0)$, $(a_{00}, b_{00}) = (0,2)$. Further $(a_{11}, b_{11}) = (0,1)$, $(a_{01}, b_{01}) = (1,1)$, $(a_{111}, b_{111}) = (0,0)$, $(a_{011}, b_{011}) = (0,1)$, $(a_{101}, b_{101}) = (0,0)$,$(a_{001}, b_{001}) = (1,1)$, $(a_{110}, b_{110}) = (0,0)$, $(a_{010}, b_{010}) = (2,0)$, $(a_{100}, b_{100}) = (0,2)$, and $(a_{000}, b_{000}) = (0,0)$. These last numbers are not relevant for our special source $\mathcal{S}$ but will be important later on, when the source model or the corresponding suffix tree, respectively, is not known in advance.

Example 13.2 Let $\mathcal{S} = \{00, 10, 1\}$ as in the previous example. Encoding a subsequence is done by successively updating the corresponding counters for a_s and b_s. For example, when we encode the

sequence '0100111' given the past '...010' using the above suffix tree and Krichevsky–Trofimov–estimator we obtain

$$P_e^s(0100111|\ldots 010) = \frac{1}{2}\cdot\frac{1}{2}\cdot\frac{1}{2}\cdot\frac{3}{4}\cdot\frac{3}{4}\cdot\frac{1}{4}\cdot\frac{1}{2} = \frac{3}{8}\cdot\frac{3}{8}\cdot\frac{1}{16} = \frac{9}{1024},$$

where $\frac{3}{8}$, $\frac{3}{8}$ and $\frac{1}{16}$ are the probabilities of the subsequences '11', '00' and '011' in the context of the leaves. These subsequences are assumed to be memoryless.

The context-tree weighting method

Suppose we have a good coding distribution P_1 for source 1 and another one, P_2, for source 2. We are looking for a good coding distribution for both sources. One possibility is to compute P_1 and P_2 and then 1 bit is needed to identify the best model which then will be used to compress the sequence. This method is called selecting. Another possibility is to employ the weighted distribution, which is

$$P_w(x^n) = \frac{P_1(x^n) + P_2(x^n)}{2}.$$

We shall present now the ***context-tree weighting algorithm*** . Under the assumption that a context tree is a full tree of depth D, only a_s and b_s, i. e. the number of zeros and ones in the subsequence of bits preceded by context s, are stored in each node s of the context tree.

Further, to each node s is assigned a weighted probability P_w^s which is recursively defined as

$$P_w^s = \begin{cases} \frac{P_e(a_s,b_s)+P_w^{0s}P_w^{1s}}{2} & \text{for } 0 \le L(s) < D, \\ P_e(a_s,b_s) & \text{for } L(s) = D, \end{cases}$$

where $L(s)$ describes the length of the (binary) string s and $P_e(a_s, b_s)$ is the estimated probability using the Krichevsky - Trofimov estimator.

Example 13.3 After encoding the sequence '0100111' given the past '...010' we obtain the context tree of depth 3 in Figure 13.9. The weighted probability $P_w^\lambda = \frac{35}{4096}$ of the root node λ finally yields the coding probability corresponding to the parsed sequence.

Recall that for the application in arithmetic coding it is important that probabilities $P(x_1 \ldots x_{n-1}0)$ and $P(x_1 \ldots x_{n-1}1)$ can be efficiently calculated from $P(x_1 \ldots x_n)$. This is possible with the context–tree weighting method, since the weighted probabilities P_w^s only have to be updated, when s is changing. This just occurs for the contexts along the path from the root to the leaf in the context tree preceding the new symbol x_n — namely the $D+1$ contexts $x_{n-1}, \ldots, x_{n-i}$ for $i = 1, \ldots, D-1$ and the root λ. Along this path, $a_s = a_s + 1$ has to be performed, when $x_n = 0$, and $b_s = b_s + 1$ has to be performed, when $x_n = 1$, and the corresponding probabilities $P_e(a_s, b_s)$ and P_w^s have to be updated.

This suggests the following algorithm for updating the context tree $CT(x_1, \ldots, x_{n-1}|x_{-D+1}, \ldots x_0)$ when reading the next letter x_n. Recall that to each node of the tree we store the parameters (a_s, b_s), $P_e(a_s, b_s)$ and P_w^s. These parameters have to be updated in order to obtain $CT(x_1, \ldots, x_n|x_{-D+1}, \ldots x_0)$. We assume the convention that the ordered pair (x_{n-1}, x_n) denotes the root λ.

UPDATE-CONTEXT-TREE$(x_n, CT(x_1 \ldots x_{n-1}|x_{-D+1} \ldots x_0))$

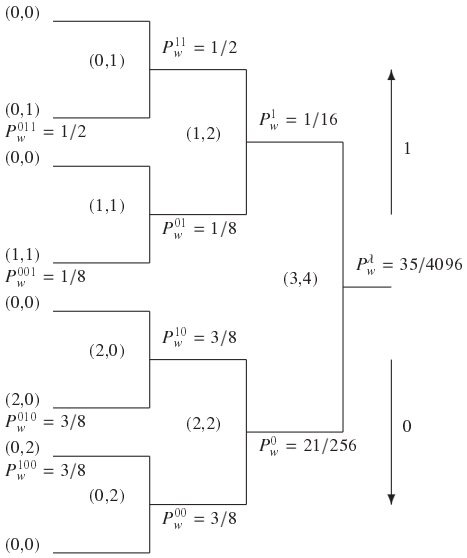

Figure 13.9. Weighted context tree for source sequence '0100111' with past ...010. The pair (a_s, b_s) denotes a_s zeros and b_s ones preceded by the corresponding context s. For the contexts $s = 111, 101, 110, 000$ it is $P_w^s = P_e(0,0) = 1$.

```
1  s ← (x_{n−1} ... x_{n−D})
2  if x_n = 0
3     then P_w^s ← P_w^s · (a_s+1/2)/(a_s+b_s+1)
4          a_s ← a_s + 1
5     else P_w^s ← P_w^s · (b_s+1/2)/(a_s+b_s+1)
6          b_s ← b_s + 1
7  for i ← 1 to D
8      do s ← (x_{n−1}, ..., x_{n−D+i})
9      if x_n = 0
10        then P_e(a_s, b_s) ← P_e(a_s, b_s) · (a_s+1/2)/(a_s+b_s+1)
11             a_s ← a_s + 1
12        else P_e(a_s, b_s) ← P_e(a_s, b_s) · (a_s+1/2)/(a_s+b_s+1)
13             b_s ← b_s + 1
14     P_w^s ← 1/2 · (P_e(a_s, b_s) + P_w^{0s} · P_w^{1s})
15 return P_w^s
```

The probability P_w^λ assigned to the root in the context tree will be used for the successive subdivisions in arithmetic coding. Initially, before reading x_1, the parameters in the context tree are $(a_s, b_s) = (0,0)$, $P_e(a_s, b_s) = 1$, and $P_w^s = 1$ for all contexts s in the tree. In our example the updates given the past $(x_{-2}, x_{-1}, x_0) = (0,1,0)$ would yield the successive probabilities P_w^λ: $\frac{1}{2}$ for $x_1 = 0$, $\frac{9}{32}$ for $(x_1 x_2) = (01)$, $\frac{5}{64}$ for $(x_1 x_2 x_3) = (010)$, $\frac{13}{256}$ for $(x_1 x_2 x_3 x_4) = (0100)$, $\frac{27}{1024}$ for $(x_1 x_2 x_3 x_4) = (01001)$, $\frac{13}{1024}$ for $(x_1 x_2 x_3 x_4 x_5) = (010011)$, $\frac{13}{1024}$ for $(x_1 x_2 x_3 x_4 x_5 x_6) = (010011)$, and finally $\frac{35}{4096}$ for $(x_1 x_2 x_3 x_4 x_5 x_6 x_7) = (0100111)$.

Correctness

Recall that the quality of a code concerning its compression capability is measured with respect to the average codeword length. The average codeword length of the best code comes as close as possible to the entropy of the source. The difference between the average codeword length and the entropy is denoted as the ***redundancy*** $\overline{\rho}(c)$ of code c, hence

$$\overline{\rho}(c) = \overline{L}(c) - H(P),$$

which obviously is the weighted (by $P(x^n)$) sum of the individual redundancies

$$\rho(x^n) = L(x^n) - \lg \frac{1}{P(x^n)}.$$

The individual redundancy $\rho(x^n|\mathcal{S})$ of sequences x^n given the (known) source $\mathcal{S}$ for all $\theta_s \in [0, 1]$ for $s \in \mathcal{S}$, $|\mathcal{S}| \leq n$ is bounded by

$$\rho(x^n|\mathcal{S}) \leq \frac{|\mathcal{S}|}{2} \lg \frac{n}{|\mathcal{S}|} + |\mathcal{S}| + 2.$$

The individual redundancy $\rho(x^n|\mathcal{S})$ of sequences x^n using the context–tree weighting algorithm (and hence a complete tree of all possible contexts as model $\mathcal{S}$) is bounded by

$$\rho(x^n|\mathcal{S}) < 2|\mathcal{S}| - 1 + \frac{|\mathcal{S}|}{2} \lg \frac{n}{|\mathcal{S}|} + |\mathcal{S}| + 2.$$

Comparing these two formulae, we see that the difference of the individual redundancies is $2|\mathcal{S}| - 1$ bits. This can be considered as the cost of not knowing the model, i.e. the model redundancy. So, the redundancy splits into the parameter redundancy, i. e. the cost of not knowing the parameter, and the model redundancy. It can be shown that the expected redundancy behaviour of the context–tree weighting method achieves the asymptotic lower bound due to Rissanen who could demonstrate that about $\frac{1}{2} \lg n$ bits per parameter is the minimum possible expected redundancy for $n \longrightarrow \infty$.

Analysis

The computational complexity is proportional to the number of nodes that are visited when updating the tree, which is about $n(D + 1)$. Therefore, the number of operations necessary for processing n symbols is linear in n. However, these operations are mainly multiplications with factors requiring high precision.

As for most modelling algorithms, the backlog of implementations in practice is the huge amount of memory. A complete tree of depth D has to be stored and updated. Only with increasing D the estimations of the probabilities are becoming more accurate and hence the average codeword length of an arithmetic code based on these estimations would become shorter. The size of the memory, however, depends exponentially on the depth of the tree.

We presented the context–tree weighting method only for binary sequences. Note that in this case the cumulative probability of a binary sequence $(x_1 \ldots x_n)$ can be calculated as

$$Q^n(x_1 x_2 \ldots x_{n-1} x_n) = \sum_{j=1,\ldots,n;x_j=1} P^j(x_1 x_2 \ldots x_{j-1} 0).$$

For compression of sources with larger alphabets, for instance ASCII-files, we refer to

the literature.

Exercises

13.2-1 Compute the arithmetic codes for the sources $(\mathcal{X}^n, P^n)$, $n = 1, 2, 3$ with $\mathcal{X} = \{1, 2\}$ and $P = (0.8, 0.2)$ and compare these codes with the corresponding Huffman codes derived previously.

13.2-2 For the codes derived in the previous exercise compute the individual redundancies of each codeword and the redundancies of the codes.

13.2-3 Compute the estimated probabilities $P_e(a, b)$ for the sequence 0100110 and all its subsequences using the Krichevsky-Trofimov estimator.

13.2-4 Compute all parameters (a_s, b_s) and the estimated probability P_e^s for the sequence 0100110 given the past 110, when the context tree $\mathcal{S} = \{00, 10, 1\}$ is known. What will be the codeword of an arithmetic code in this case?

13.2-5 Compute all parameters (a_s, b_s) and the estimated probability P_λ for the sequence 0100110 given the past 110, when the context tree is not known, using the context-tree weighting algorithm.

13.2-6 Based on the computations from the previous exercise, update the estimated probability for the sequence 01001101 given the past 110.

Show that for the cumulative probability of a binary sequence $(x_1 \dots x_n)$ it is

$$Q^n(x_1 x_2 \dots x_{n-1} x_n) = \sum_{j=1,\dots,n;x_j=1} P^j(x_1 x_2 \dots x_{j-1} 0).$$

13.3. Ziv-Lempel coding

In 1976–1978 Jacob Ziv and Abraham Lempel introduced two universal coding algorithms, which in contrast to statistical coding techniques, considered so far, do not make explicit use of the underlying probability distribution. The basic idea here is to replace a previously seen string with a pointer into a history buffer (LZ77) or with the index of a dictionary (LZ78). LZ algorithms are widely used – "zip" and its variations use the LZ77 algorithm. So, in contrast to the presentation by several authors, Ziv-Lempel coding is not a single algorithm. Originally, Lempel and Ziv introduced a method to measure the complexity of a string – like in Kolmogorov complexity. This led to two different algorithms, LZ77 and LZ78. Many modifications and variations have been developed since. However, we shall present the original algorithms and refer to the literature for further information.

13.3.1. LZ77

The idea of LZ77 is to pass a ***sliding window*** over the text to be compressed. One looks for the longest substring in this window representing the next letters of the text. The window consists of two parts: a history window of length l_h, say, in which the last l_h bits of the text considered so far are stored, and a lookahead window of length l_f containing the next l_f bits of the text. In the simplest case l_h and l_f are fixed. Usually, l_h is much bigger than l_f. Then one encodes the triple (offset, length, letter). Here the ***offset*** is the number of letters

one has to go back in the text to find the matching substring, the length is just the length of this matching substring, and the letter to be stored is the letter following the matching substring. Let us illustrate this procedure with an example. Assume the text to be compressed is *...abaabbaabbaaabbbaaaabbabbbabbb...*, the window is of size 15 with $l_h = 10$ letters history and $l_f = 5$ letters lookahead buffer. Assume, the sliding window now arrived at

$$...aba||abbaabbaaa|bbbaa||,$$

i. e., the history window contains the 10 letters *abbaabbaaa* and the lookahead window contains the five letters *bbbaa*. The longest substring matching the first letters of the lookahead window is *bb* of length 2, which is found nine letters back from the right end of the history window. So we encode $(9, 2, b)$, since b is the next letter (the string *bb* is also found five letters back, in the original LZ77 algorithm one would select the loargest offset). The window then is moved 3 letters forward

$$...abaabb||aabbaaabbb|aaaab||.$$

The next codeword is $(6, 3, a)$, since the longest matching substring is *aaa* of length 3 found 6 letters backwards and a is the letter following this substring in the lookahead window. We proceed with

$$...abaabbaabb||aaabbbaaaa|bbabb||,$$

and encode $(6, 3, b)$. Further

$$...abaabbaabbaaab||bbaaaabbab|babbb||.$$

Here we encode $(3, 4, b)$. Observe that the match can extend into the lookahead window.

There are many subtleties to be taken into account. If a symbol did not appear yet in the text, offset and length are set to 0. If there are two matching strings of the same length, one has to choose between the first and the second offset. Both variations have advantages. Initially one might start with an empty history window and the first letters of the text to be compressed in the lookahead window - there are also further variations.

A common modification of the original scheme is to output only the pair (offset, length) and not the following letter of the text. Using this coding procedure one has to take into consideration the case in which the next letter does not occur in the history window. In this case, usually the letter itself is stored, such that the decoder has to distinguish between pairs of numbers and single letters. Further variations do not necessarily encode the longest matching substring.

13.3.2. LZ78

LZ78 does not use a sliding window but a dictionary which is represented here as a table with an index and an entry. LZ78 parses the text to be compressed into a collection of strings, where each string is the longest matching string α seen so far plus the symbol s following α in the text to be compressed. The new string αs is added into the dictionary. The new entry is coded as (i, s), where i is the index of the existing table entry α and s is the appended symbol.

As an example, consider the string "*abaabbaabbaaabbbaaaabba*". It is divided by LZ78 into strings as shown below. String 0 is here the empty string.

Input	a	b	aa	bb	aab	ba	$aabb$	baa	$aabba$
String Index	1	2	3	4	5	6	7	8	9
Output	$(0, a)$	$(0, b)$	$(1, a)$	$(2, b)$	$(3, b)$	$(2, a)$	$(5, b)$	$(6, a)$	$(7, a)$

Since we are not using a sliding window, there is no limit for how far back strings can reach. However, in practice the dictionary cannot continue to grow infinitely. There are several ways to manage this problem. For instance, after having reached the maximum number of entries in the dictionary, no further entries can be added to the table and coding becomes static. Another variation would be to replace older entries. The decoder knows how many bits must be reserved for the index of the string in the dictionary, and hence decompression is straightforward.

Correctness

Ziv-Lempel coding asymptotically achieves the best possible compression rate which again is the entropy rate of the source. The source model, however, is much more general than the discrete memoryless source. The stochastic process generating the next letter, is assumed to be stationary (the probability of a sequence does not depend on the instant of time, i. e. $P(X_1 = x_1, \ldots, X_n = x_n) = P(X_{t+1} = x_1, \ldots, X_{t+n} = x_n)$ for all t and all sequences $(x_1 \ldots x_n)$). For stationary processes the limit $\lim_{n\to\infty} \frac{1}{n} H(X_1, \ldots X_n)$ exists and is defined to be the entropy rate.

If s(n) denotes the number of strings in the parsing process of LZ78 for a text generated by a stationary source, then the number of bits required to encode all these strings is $s(n) \cdot (\lg s(n) + 1)$. It can be shown that $\frac{s(n)\cdot(\lg s(n)+1)}{n}$ converges to the entropy rate of the source. However, this would require that all strings can be stored in the dictionary.

Analysis

If we fix the size of the sliding window or the dictionary, the running time of encoding a sequence of n letters will be linear in n. However, as usually in data compression, there is a tradeoff between compression rate and speed. A better compression is only possible with larger memory. Increasing the size of the dictionary or the window will, however, result in a slower performance, since the most time consuming task is the search for the matching substring or the position in the dictionary.

Decoding in both LZ77 and LZ78 is straightforward. Observe that with LZ77 decoding is usually much faster than encoding, since the decoder already obtains the information at which position in the history he can read out the next letters of the text to be recovered, whereas the encoder has to find the longest matching substring in the history window. So algorithms based on LZ77 are useful for files which are compressed once and decompressed more frequently.

Further, the encoded text is not necessarily shorter than the original text. Especially in the beginning of the encoding the coded version may expand a lot. This expansion has to be taken into consideration.

For implementation it is not optimal to represent the text as an array. A suitable data structure will be a circular queue for the lookahead window and a binary search tree for the

history window in LZ77, while for LZ78 a dictionary tree should be used.

Exercises

13.3-1 Apply the algorithms LZ77 and LZ78 to the string "abracadabra".

13.3-2 Which type of files will be well compressed with LZ77 and LZ78, respectively? For which type of files are LZ77 and LZ78 not so advantageous?

13.3-3 Discuss the advantages of encoding the first or the last offset, when several matching substrings are found in LZ77.

13.4. The Burrows-Wheeler transform

The ***Burrows-Wheeler transform*** will best be demonstrated by an example. Assume that our original text is $\vec{X}$ = "WHEELER". This text will be mapped to a second text $\vec{L}$ and an index I according to the following rules.

1) We form a matrix M consisting of all cyclic shifts of the original text $\vec{X}$. In our example

$$M = \begin{pmatrix} W & H & E & E & L & E & R \\ H & E & E & L & E & R & W \\ E & E & L & E & R & W & H \\ E & L & E & R & W & H & E \\ L & E & R & W & H & E & E \\ E & R & W & H & E & E & L \\ R & W & H & E & E & L & E \end{pmatrix}.$$

2) From M we obtain a new matrix M' by simply ordering the rows in M lexicograhically. Here this yields the matrix

$$M' = \begin{pmatrix} E & E & L & E & R & W & H \\ E & L & E & R & W & H & E \\ E & R & W & H & E & E & L \\ H & E & E & L & E & R & W \\ L & E & R & W & H & E & E \\ R & W & H & E & E & L & E \\ W & H & E & E & L & E & R \end{pmatrix}.$$

3) The transformed string $\vec{L}$ then is just the last column of the matrix M' and the index I is the number of the row of M', in which the original text is contained. In our example $\vec{L}$ = "HELWEER" and $I = 6$ – we start counting the the rows with row no. 0.

This gives rise to the following pseudocode. We write here X instead of $\vec{X}$ and L instead of $\vec{L}$, since the purpose of the vector notation is only to distinguish the vectors from the letters in the text.

```
BWT-ENCODER(X)
1 for j ← 0 to n − 1
2     do M[0, j] ← X[j]
3 for i ← 0 to n − 1
```

```
4     do for j ← 0 to n − 1
5        do M[i, j] ← M[i − 1, j + 1 mod n]
6  for i ← 0 to n − 1
7     do row i of M′ ← row i of M in lexicographic order
8  for i ← 0 to n − 1
9     do L[i] ← M′[i, n − 1]
10 i = 0
11 while (row i of M′ ≠ row i of M)
12    do i ← i + 1
13 I ← i
14 return L and I
```

It can be shown that this transformation is invertible, i. e., it is possible to reconstruct the original text $\vec{X}$ from its transform $\vec{L}$ and the index I. This is because these two parameters just yield enough information to find out the underlying permutation of the letters. Let us illustrate this reconstruction using the above example again. From the transformed string $\vec{L}$ we obtain a second string $\vec{E}$ by simply ordering the letters in $\vec{L}$ in ascending order. Actually, $\vec{E}$ is the first column of the matrix M' above. So, in our example

$$\vec{L} = \text{“}H \quad E \quad L \quad W \quad E \quad E \quad R''$$

$$\vec{E} = \text{“}E \quad E \quad E \quad H \quad L \quad R \quad W''.$$

Now obviously the first letter $\vec{X}(0)$ of our original text $\vec{X}$ is the letter in position I of the sorted string $\vec{E}$, so here $\vec{X}(0) = \vec{E}(6) = W$. Then we look at the position of the letter just considered in the string $\vec{L}$ – here there is only one W, which is letter no. 3 in $\vec{L}$. This position gives us the location of the next letter of the original text, namely $\vec{X}(1) = \vec{E}(3) = H$. H is found in position no. 0 in $\vec{L}$, hence $\vec{X}(2) = \vec{E}(0) = E$. Now there are three E–s in the string $\vec{L}$ and we take the first one not used so far, here the one in position no. 1, and hence $\vec{X}(3) = \vec{E}(1) = E$. We iterate this procedure and find $\vec{X}(4) = \vec{E}(4) = L$, $\vec{X}(5) = \vec{E}(2) = E$, $\vec{X}(6) = \vec{E}(5) = R$.

This suggests the following pseudocode.

```
BWT-DECODER(L, I)
1 E[0..n − 1] ← sort L[0..n − 1]
2 pi[−1] ← I
3 for i ← 0 to n − 1
4    do j = 0
5       while (L[j]) ≠ E[pi[i − 1]] OR j is a component of pi)
6             do  j ← j + 1
7       pi[i] ← j
8       X[i] ← L[j]
9 return X
```

This algorithm implies a more formal description. Since the decoder only knows $\vec{L}$, he has to sort this string to find out $\vec{E}$. To each letter $\vec{L}(j)$ from the transformed string $\vec{L}$ record the position $\pi(j)$ in $\vec{E}$ from which it was jumped to by the process described above. So the vector pi in our pseudocode yields a permutation π such that for each $j = 0, \ldots, n-1$ row j it is $\vec{L}(j) = \vec{E}(\pi(j))$ in matrix M. In our example $\pi = (3, 0, 1, 4, 2, 5, 6)$. This permutation

can be used to reconstruct the original text $\vec{X}$ of length n via $\vec{X}(n-1-j) = \vec{L}(\pi^j(I))$, where $\pi^0(x) = x$ and $\pi^j(x) = \pi(\pi^{j-1}(x))$ for $j = 1, \ldots, n-1$.

Observe that so far the original data have only been transformed and are not compressed, since string $\vec{L}$ has exactly the same length as the original string $\vec{L}$. So what is the advantage of the Burrows–Wheeler transformation? The idea is that the transformed string can be much more efficiently encoded than the original string. The dependencies among the letters have the effect that in the transformed string $\vec{L}$ there appear long blocks consisting of the same letter.

In order to exploit such frequent blocks of the same letter, Burrows and Wheeler suggested the following ***move–to–front–code***, which we shall illustrate again with our example above.

We write down a list containing the letters used in our text in alphabetic order indexed by their position in this list.

E	H	L	R	W
0	1	2	3	4

Then we parse through the transformed string $\vec{L}$ letter by letter, note the index of the next letter and move this letter to the front of the list. So in the first step we note 1 – the index of the H, move H to the front and obtain the list

H	E	L	R	W
0	1	2	3	4

Then we note 1 and move E to the front,

E	H	L	R	W
0	1	2	3	4

note 2 and move L to the front,

L	E	H	R	W
0	1	2	3	4

note 4 and move W to the front,

W	L	E	H	R
0	1	2	3	4

note 2 and move E to the front,

E	W	L	H	R
0	1	2	3	4

note 0 and leave E at the front,

E	W	L	H	R
0	1	2	3	4

note 4 and move R to the front,

R	E	W	L	H
0	1	2	3	4

So we obtain the sequence $(1, 1, 2, 4, 2, 0, 4)$ as our move-to-front-code. The pseudocode may look as follows, where Q is a list of the letters occuring in the string $\vec{L}$.

```
MOVE-TO-FRONT(L)
1 Q[0..n − 1] ← list of m letters occuring in L ordered alphabetically
2 for i ← 0 to n − 1
3     do j = 0
4        while (j ≠ L[i])
5              do j ← j + 1
6     c[i] ← j
7 for l ← 0 to j
8     do Q[l] ← Q[l − 1 mod j + 1]
9 return c
```

The move–to–front code c will finally be compressed, for instance by Huffman coding.

Correctness

The compression is due to the move-to-front code obtained from the transformed string $\vec{L}$. It can easily be seen that this move-to-front coding procedure is invertible, so one can recover the string $\vec{L}$ from the code obtained as above.

Now it can be observed that in the move-to-front-code small numbers occur more frequently. Unfortunately, this will become obvious only with much longer texts than in our example – in long strings it was observed that even about 70 per cent of the numbers are 0. This irregularity in distribution can be exploited by compressing the sequence obtained after move-to-front coding, for instance by Huffman codes or run–length codes.

The algorithm performed very well in practice regarding the compression rate as well as the speed. The asymptotic optimality of compression has been proven for a wide class of sources.

Analysis

The most complex part of the Burrows–Wheeler transform is the sorting of the block yielding the transformed string $\vec{L}$. Due to fast sorting procedures, especially suited for the type of data to be compressed, compression algorithms based on the Burrows–Wheeler transform are usually very fast. On the other hand, compression is done blockwise. The text to be compressed has to be divided into blocks of appropriate size such that the matrices M and M' still fit into the memory. So the decoder has to wait until the whole next block is transmitted and cannot work sequentially bit by bit as in arithmetic coding or Ziv-Lempel coding.

Exercises

13.4-1 Apply the Burrows-Wheeler transform and the move-to-front code to the text "abracadabra".

13.4-2 Verify that the transformed string $\vec{L}$ and the index i of the position in the sorted text $\vec{E}$ (containing the first letter of the original text to be compressed) indeed yield enough information to reconstruct the original text.

13.4-3 Show how in our example the decoder would obtain the string $\vec{L}$ ="HELWEER" from the move-to-front code $(1, 1, 2, 4, 2, 0, 4)$ and the letters E,H,L,W,R occuring in the text. Describe the general procedure for decoding move-to-front codes.

13.4-4 We followed here the encoding procedure presented by Burrows and Wheeler. Can

the encoder obtain the transformed string $\vec{L}$ even without constructing the two matrices M and M'?

13.5. Image compression

The idea of image compression algorithms is similar to the one behind the Burrows-Wheeler transform. The text to be compressed is transformed to a format which is suitable for application of the techniques presented in the previous sections, such as Huffman coding or arithmetic coding. There are several procedures based on the type of image (for instance, black/white, greyscale or colour image) or compression (lossless or lossy). We shall present the basic steps – representation of data, discrete cosine transform, quantization, coding – of lossy image compression procedures like the standard ***JPEG***.

13.5.1. Representation of data

A greyscale image is represented as a two–dimensional array X, where each entry $X(i, j)$ represents the intensity (or brightness) at position (i, j) of the image. Each $X(i, j)$ is either a signed or an unsigned k-bit integers, i. e., $X(i, j) \in \{0, \ldots, 2^k - 1\}$ or $X(i, j) \in \{-2^{k-1}, \ldots, 2^{k-1} - 1\}$.

A position in a colour image is usually represented by three greyscale values $R(i, j)$, $G(i, j)$, and $B(i, j)$ per position corresponding to the intensity of the primary colours red, green and blue.

In order to compress the image, the three arrays (or channels) R, G, B are first converted to the luminance/chrominance space by the YC_bC_r***-transform*** (performed entry–wise)

$$\begin{pmatrix} Y \\ C_b \\ C_r \end{pmatrix} = \begin{pmatrix} 0.299 & 0.587 & 0.114 \\ -0.169 & -0.331 & 0.5 \\ 0.5 & -0.419 & -0.0813 \end{pmatrix} \cdot \begin{pmatrix} R \\ G \\ B \end{pmatrix}$$

$Y = 0.299R + 0.587G + 0.114B$ is the luminance or intensity channel, where the coefficients weighting the colours have been found empirically and represent the best possible approximation of the intensity as perceived by the human eye. The chrominance channels $C_b = 0.564(B - Y)$ and $C_r = 0.713(R - Y)$ contain the colour information on red and blue as the differences from Y. The information on green is obtained as big part in the luminance Y.

A first compression for colour images commonly is already obtained after application of the YC_bC_r–transform by removing ***irrelevant information***. Since the human eye is less sensitive to rapid colour changes than to changes in intensity, the resolution of the two chrominance channels C_b and C_r is reduced by a factor of 2 in both vertical and horizontal direction, which results after sub-sampling in arrays of $\frac{1}{4}$ of the original size.

The arrays then are subdivided into 8×8 blocks, on which successively the actual (lossy) data compression procedure is applied.

Let us consider the following example based on a real image, on which the steps of compression will be illustrated. Assume that the 8×8 block of 8-bit unsigned integers below is obtained as a part of an image.

$$f = \begin{pmatrix} 139 & 144 & 149 & 153 & 155 & 155 & 155 & 155 \\ 144 & 151 & 153 & 156 & 159 & 156 & 156 & 155 \\ 150 & 155 & 160 & 163 & 158 & 156 & 156 & 156 \\ 159 & 161 & 162 & 160 & 160 & 159 & 159 & 159 \\ 159 & 160 & 161 & 161 & 160 & 155 & 155 & 155 \\ 161 & 161 & 161 & 161 & 160 & 157 & 157 & 157 \\ 162 & 162 & 161 & 163 & 162 & 157 & 157 & 157 \\ 161 & 162 & 161 & 161 & 163 & 158 & 158 & 158 \end{pmatrix}$$

13.5.2. The discrete cosine transform

Each 8×8 block $(f(i,j))_{i,j=0,\ldots,7}$, say, is transformed into a new block $(F(u,v))_{u,v=0,\ldots,7}$. There are several possible transforms, usually the ***discrete cosine transform*** is applied, which here obeys the formula

$$F(u,v) = \frac{1}{4} c_u c_v \left(\sum_{i=0}^{7} \sum_{j=0}^{7} f(i,j) \cdot \cos \frac{(2i+1)u\pi}{16} \cos \frac{(2j+1)v\pi}{16} \right)$$

The cosine transform is applied after shifting the unsigned integers to signed integers by subtraction of 2^{k-1}.

DCT(f)
1 **for** $u \leftarrow 0$ **to** 7
2 **do for** $v \leftarrow 0$ **to** 7
3 **do** $F(u,v) \leftarrow$ DCT - coefficient of matrix f
4 **return** F

The coefficients need not be calculated according to the formula above. They can also be obtained via a related Fourier transform (see Exercises) such that a Fast Fourier Transform may be applied. JPEG also supports wavelet transforms, which may replace the discrete cosine transform here.

The discrete cosine transform can be inverted via

$$f(i,j) = \frac{1}{4} \left(\sum_{u=0}^{7} \sum_{v=0}^{7} c_u c_v F(u,v) \cdot \cos \frac{(2i+1)u\pi}{16} \cos \frac{(2j+1)v\pi}{16} \right),$$

where $c_u = \begin{cases} \frac{1}{\sqrt{2}} & \text{for } u = 0 \\ 1 & \text{for } u \neq 0 \end{cases}$ and $c_v = \begin{cases} \frac{1}{\sqrt{2}} & \text{for } v = 0 \\ 1 & \text{for } v \neq 0 \end{cases}$ are normalization constants.

In our example, the transformed block F is

$$F = \begin{pmatrix} 235.6 & -1.0 & -12.1 & -5.2 & 2.1 & -1.7 & -2.7 & 1.3 \\ -22.6 & -17.5 & -6.2 & -3.2 & -2.9 & -0.1 & 0.4 & -1.2 \\ -10.9 & -9.3 & -1.6 & 1.5 & 0.2 & -0.9 & -0.6 & -0.1 \\ -7.1 & -1.9 & 0.2 & 1.5 & 0.9 & -0.1 & 0.0 & 0.3 \\ -0.6 & -0.8 & 1.5 & 1.6 & -0.1 & -0.7 & 0.6 & 1.3 \\ 1.8 & -0.2 & 1.6 & -0.3 & -0.8 & 1.5 & 1.0 & -1.0 \\ -1.3 & -0.4 & -0.3 & -1.5 & -0.5 & 1.7 & 1.1 & -0.8 \\ -2.6 & 1.6 & -3.8 & -1.8 & 1.9 & 1.2 & -0.6 & -0.4 \end{pmatrix}$$

where the entries are rounded.

The discrete cosine transform is closely related to the discrete Fourier transform and similarly maps signals to frequencies. Removing higher frequencies results in a less sharp image, an effect that is tolerated, such that higher frequencies are stored with less accuracy.

Of special importance is the entry $F(0,0)$, which can be interpreted as a measure for the intensity of the whole block.

13.5.3. Quantization

The discrete cosine transform maps integers to real numbers, which in each case have to be rounded to be representable. Of course, this rounding already results in a loss of information. However, the transformed block F will now be much easier to manipulate. A ***quantization*** takes place, which maps the entries of F to integers by division by the corresponding entry in a luminance quantization matrix Q. In our example we use

$$Q = \begin{pmatrix} 16 & 11 & 10 & 16 & 24 & 40 & 51 & 61 \\ 12 & 12 & 14 & 19 & 26 & 58 & 60 & 55 \\ 14 & 13 & 16 & 24 & 40 & 57 & 69 & 56 \\ 14 & 17 & 22 & 29 & 51 & 87 & 80 & 62 \\ 18 & 22 & 37 & 56 & 68 & 109 & 103 & 77 \\ 24 & 35 & 55 & 64 & 81 & 104 & 113 & 92 \\ 49 & 64 & 78 & 87 & 103 & 121 & 120 & 101 \\ 72 & 92 & 95 & 98 & 112 & 100 & 103 & 99 \end{pmatrix}.$$

The quantization matrix has to be carefully chosen in order to leave the image at highest possible quality. Quantization is the lossy part of the compression procedure. The idea is to remove information which should not be "visually significant". Of course, at this point there is a tradeoff between the compression rate and the quality of the decoded image. So, in JPEG the quantization table is not included into the standard but must be specified (and hence be encoded).

QUANTIZATION(F)
1 **for** $i \leftarrow 0$ **to** 7
2 **do for** $j \leftarrow 0$ **to** 7
3 **do** $T(i,j) \leftarrow \{\frac{F(i,j)}{Q(i,j)}\}$
4 **return** T

This quantization transforms block F to a new block T with $T(i,j) = \{\frac{F(i,j)}{Q(i,j)}\}$, where $\{x\}$ is the closest integer to x. This block will finally be encoded. Observe that in the transformed block F besides the entry $F(0,0)$ all other entries are relatively small numbers, which has

the effect that T mainly consists of 0s .

$$T = \begin{pmatrix} 15 & 0 & -1 & 0 & 0 & 0 & 0 & 0 \\ -2 & -1 & 0 & 0 & 0 & 0 & 0 & 0 \\ -1 & -1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

Coefficient $T(0,0)$, in this case 15, deserves special consideration. It is called DC term (direct current), while the other entries are denoted AC coefficients (alternate current).

13.5.4. Coding

Matrix T will finally be encoded by a Huffman code. We shall only sketch the procedure. First the DC term will be encoded by the difference to the DC term of the previously encoded block. For instance, if the previous DC term was 12, then $T(0,0)$ will be encoded as -3.

After that the AC coefficients are encoded according to the zig–zag order $T(0,1)$, $T(1,0)$, $T(2,0)$, $T(1,1)$, $T(0,2)$, $T(0,3)$, $T(1,2)$, etc.. In our example, this yields the sequence $0,-2,-1,-1,-1,0,0,-1$ followed by 55 zeros. This zig–zag order exploits the fact that there are long runs of successive zeros. These runs will be even more efficiently represented by application of ***run-length coding***, i. e., we encode the number of zeros before the next nonzero element in the sequence followed by this element.

Integers are written in such a way that small numbers have shorter representations. This is achieved by splitting their representation into size (number of bits to be reserved) and amplitude (the actual value). So, 0 has size 0, 1 and -1 have size 1. -3, -2, 2, and 3 have size 2, etc.

In our example this yields the sequence (2)(3) for the DC term followed by $(1,2)(-2)$, $(0,1)(-1)$, $(0,1)(-1)$, $(0,1)(-1)$, $(2,1)(-1)$, and a final $(0,0)$ as an end-of-block symbol indicating that only zeros follow from now on. $(1,2)(-2)$, for instance, means that there is 1 zero followed by an element of size 2 and amplitude -2.

These pairs are then assigned codewords from a Huffman code. There are different Huffman codes for the pairs (run, size) and for the amplitudes. These Huffman codes have to be specified and hence be included into the code of the image.

In the following pseudocode for the encoding of a single 8×8–block T we shall denote the different Huffman codes by encode-1, encode-2, encode-3.

```
RUN-LENGTH-CODE(T)
1 c ← encode-1(size(DC − T[0,0]))
2 c ← c|| encode-3(amplitude(DC − T[00]))
3 DC ← T[0,0]
4 for l ← 1 to 14
5     do for i ← 0 to l
6         do if l = 1 mod 2 then u ← i else u ← l − i
7             if T[u,l-u]=0 then run ← run + 1
8                           else c ← c|| encode-2(run, size(T[u, l − u]))
```

9 $c \leftarrow c \| \text{encode-3}(\text{amplitude}(T[u, l-u])$
10 $run \leftarrow 0$
11 **if** $run > 0$ **then** encode-2$(0,0)$
12 **return** c

At the decoding end matrix T will be reconstructed. Finally, by multiplication of each entry $T(i,j)$ by the corresponding entry $Q(i,j)$ from the quantization matrix Q we obtain an approximation $\overline{F}$ to the block F, here

$$\overline{F} = \begin{pmatrix} 240 & 0 & -10 & 0 & 0 & 0 & 0 & 0 \\ -24 & -12 & 0 & 0 & 0 & 0 & 0 & 0 \\ -14 & -13 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}.$$

To $\overline{F}$ the inverse cosine transform is applied. This allows to decode the original 8×8–block f of the original image – in our example as

$$\overline{f} = \begin{pmatrix} 144 & 146 & 149 & 152 & 154 & 156 & 156 & 156 \\ 148 & 150 & 152 & 154 & 156 & 156 & 156 & 156 \\ 155 & 156 & 157 & 158 & 158 & 157 & 156 & 155 \\ 160 & 161 & 161 & 162 & 161 & 159 & 157 & 155 \\ 163 & 163 & 164 & 163 & 162 & 160 & 158 & 156 \\ 163 & 164 & 164 & 164 & 162 & 160 & 158 & 157 \\ 160 & 161 & 162 & 162 & 162 & 161 & 159 & 158 \\ 158 & 159 & 161 & 161 & 162 & 161 & 159 & 158 \end{pmatrix}.$$

Exercises

13.5-1 Find size and amplitude for the representation of the integers 5, −19, and 32.

13.5-2 Write the entries of the following matrix in zig – zag order.

$$\begin{pmatrix} 5 & 0 & -2 & 0 & 0 & 0 & 0 & 0 \\ 3 & 1 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & -1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 2 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

How would this matrix be encoded if the difference of the DC term to the previous one was −2?

13.5-3 In our example after quantizing the sequence (2)(3), (1, 2)(−2), (0, 1)(−1), (0, 1)(−1), (0, 1)(−1), (2, 1)(−1), (0, 0) has to be encoded. Assume the Huffman codebooks would yield 011 to encode the difference 2 from the preceding block's DC, 0, 01, and

11 for the amplitudes -1, -2, and 3, respectively, and 1010, 00, 11011, and 11100 for the pairs $(0,0)$, $(0,1)$, $(1,2)$, and $(2,1)$, respectively. What would be the bitstream to be encoded for the 8×8 block in our example? How many bits would hence be necessary to compress this block?

13.5-4 What would be matrices T, $\overline{F}$ and $\overline{f}$, if we had used

$$Q = \begin{pmatrix} 8 & 6 & 5 & 8 & 12 & 20 & 26 & 31 \\ 6 & 6 & 7 & 10 & 13 & 29 & 30 & 28 \\ 7 & 7 & 8 & 12 & 20 & 29 & 35 & 28 \\ 7 & 9 & 11 & 15 & 26 & 44 & 40 & 31 \\ 9 & 11 & 19 & 28 & 34 & 55 & 52 & 39 \\ 12 & 18 & 28 & 32 & 41 & 52 & 57 & 46 \\ 25 & 32 & 39 & 44 & 57 & 61 & 60 & 51 \\ 36 & 46 & 48 & 49 & 56 & 50 & 57 & 50 \end{pmatrix}$$

for quantizing after the cosine transform in the block of our example?

13.5-5 What would be the zig–zag code in this case (assuming again that the DC term would have difference -3 from the previous DC term)?

13.5-6 For any sequence $(f(n))_{n=0,\ldots,m-1}$ define a new sequence $(\hat{f}(n))_{n=0,\ldots,2m-1}$ by

$$\hat{f}(n) = \begin{cases} f(n) & \text{for } n = 0, \ldots, m-1 \\ f(2m-1-n) & \text{for } n = m, \ldots, 2m-1 \end{cases}.$$

This sequence can be expanded to a Fourier series via

$$\hat{f}(n) = \frac{1}{\sqrt{2m}} \sum_{n=0}^{2m-1} \hat{g}(u) e^{i\frac{2\pi}{2m}nu} \quad \text{with } \hat{g}(u) = \frac{1}{\sqrt{2m}} \sum_{n=0}^{2m-1} \hat{f}(u) e^{-i\frac{2\pi}{2m}nu}, \quad i = \sqrt{-1}.$$

Show how the coefficients of the discrete cosine transform

$$F(u) = c_u \sum_{n=0}^{m-1} f(n) \cos(\frac{(2n+1)\pi u}{2m}, \quad c_u = \begin{cases} \frac{1}{\sqrt{m}} & \text{for } u = 0 \\ \frac{2}{\sqrt{m}} & \text{for } u \neq 0 \end{cases}$$

arise from this Fourier series.

Chapter notes

The frequency table of the letters in English texts is taken from [24]. The Huffman coding algorithm was introduced by Huffman in [13]. A pseudocode can be found in [8], where the Huffman coding algorithm is presented as a special Greedy algorithm. There are also adaptive or dynamic variants of Huffman coding, which adapt the Huffman code if it is no longer optimal for the actual frequency table, for the case that the probability distribution of the source is not known in advance. The "3/4-conjecture" on Kraft's inequality for fix-free codes is due to Ahlswede, Balkenhol, and Khachatrian [1].

Arithmetic coding has been introduced by Rissanen [19] and Pasco [18]. For a discussion of implementation questions see [16, 16, 27]. In the section on modelling we are

following the presentation of Willems, Shtarkov and Tjalkens in [26]. The exact calculations can be found in their original paper [25] which received the Best Paper Award of the IEEE Information Theory Society in 1996. The Krichevsky-Trofimov estimator had been introduced in [14].

We presented the two original algorithms LZ77 and LZ78 [28, 29] due to Lempel and Ziv. Many variants, modifications and extensions have been developed since that – concerning the handling of the dictionary, the pointers, the behaviour after the dictionary is complete, etc. For a description, see, for instance, [3] or [4]. Most of the prominent tools for data compression are variations of Ziv-Lempel coding. For example "zip" and "gzip" are based on LZ77 and a variant of LZ78 is used by the program "compress".

The Burrows-Wheeler transform was introduced in the technical report [5]. It became popular in the sequel, especially because of the Unix compression tool "bzip" based on the Burrows-Wheeler transform, which outperformed most dictionary – based tools on several benchmark files. Also it avoids arithmetic coding, for which patent rights have to be taken into consideration. Further investigations on the Burrows-Wheeler transform have been carried out, for instance in [2, 10, 15].

We only sketched the basics behind lossy image compression, especially the preparation of the data for application of techniques as Huffman coding. For a detailed discussion we refer to [22], where also the new JPEG2000 standard is described. Our example is taken from [23].

JPEG – short for Joint Photographic Experts Group – is very flexible. For instance, it also supports lossless data compression. All the topics presented in the section on image compression are not unique. There are models involving more basic colours and further transforms besides the YC_bC_r-transform (for which even different scaling factors for the chrominance channels were used, the formula presented here is from [22]). The cosine transform may be replaced by another operation like a wavelet transform. Further, there is freedom to choose the quantization matrix, responsible for the quality of the compressed image, and the Huffman code. On the other hand, this has the effect that these parameters have to be explicitly specified and hence are part of the coded image.

The ideas behind procedures for video and sound compression are rather similar to those for image compression. In principal, they follow the same steps. The amount of data in these cases, however, is much bigger. Again information is lost by removing irrelevant information not realizable by the human eye or ear (for instance by psychoacoustic models) and by quantizing, where the quality should not be reduced significantly. More refined quantizing methods are applied in these cases.

Most information on data compression algorithms can be found in literature on Information Theory, for instance [9, 11], since the analysis of the achievable compression rates requires knowledge of source coding theory. Recently, there have appeared several books on data compression, for instance [4, 12, 17, 20, 21], to which we refer to further reading. The benchmark files of the Calgary Corpus and the Canterbury Corpus are available under [6] or [7].

Problems

13-1. Adaptive Huffman codes

Dynamic and adaptive Huffman coding is based on the following property. A binary code tree has the sibling property if each node has a sibling and if the nodes can be listed in order of nonincreasing probabilities with each node being adjacent to its sibling. Show that a binary prefix code is a Huffman code exactly if the corresponding code tree has the sibling property.

13-2. Generalizations of Kraft's inequality

In the proof of Kraft's inequality it is essential to order the lengths $L(1) \leq \cdots \leq L(a)$. Show that the construction of a prefix code for given lengths $2, 1, 2$ is not possible if we are not allowed to order the lengths. This scenario of unordered lengths occurs with the Shannon-Fano-Elias code and in the theory of alphabetic codes, which are related to special search problems. Show that in this case a prefix code with lengths $L(1) \leq \cdots \leq L(a)$ exists if and only if

$$\sum_{x \in \mathcal{X}} 2^{-L(x)} \leq \frac{1}{2} \, .$$

If we additionally require the prefix codes to be also suffix-free i. e., no codeword is the end of another one, it is an open problem to show that Kraft's inequality holds with the 1 on the right–hand side replaced by 3/4, i. e.,

$$\sum_{x \in \mathcal{X}} 2^{-L(x)} \leq \frac{3}{4} \, .$$

13-3. Redundancy of Krichevsky-Trofimov estimator

Show that using the Krichevsky-Trofimov estimator, when parameter θ of a discrete memoryless source is unknown, the individual redundancy of sequence x^n is at most $\frac{1}{2} \lg n + 3$ for all sequences x^n and all $\theta \in \{0, 1\}$.

13-4. Alternatives to move-to-front codes

Find further procedures which like move-to-front coding prepare the text for compression after application of the Burrows-Wheeler transform.

Bibliography

[1] R. Ahlswede, B. Balkenhol, L. Khachatrian. Some properties of fix-free codes. In *Proceedings of the 1st International Seminarium on Coding Theory and Combinatorics*, 1996, pp. 20–23. 614

[2] B. Balkenhol, S. Kurtz. Universal data compression based on the Burrows-Wheeler transform: theory and practice. *IEEE Transactions on Computers*, 49(10):1043–1053–953, 2000. 615

[3] T. C. Bell, I. H. Witten, J. G. Cleary. Modeling for text compression. *Communications of the ACM*, 21:557–591, 1989. 615

[4] T. C. Bell, I. H. Witten, J. G. Cleary. *Text Compression*. Prentice Hall, 1990. 615

[5] M. Burrows, D. J. Wheeler. A block-sorting lossless data compression algorithm. Research Report 124, http://gatekeeper.dec.com/pub/DEC/SRC/research-reports/abstracts/src-rr-124.html, 1994. 615

[6] Calgary. The Calgary/Canterbury Text Compression. ftp://ftp.cpsc.ucalgary.ca/pub/projects/text.compression.corpus, 2004. 615

[7] Canterbury. The Canterbury Corpus. http://corpus.canterbury.ac.nz, 2004. 615

[8] T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein. *Introduction to Algorithms*. The MIT Press/McGraw-Hill, 2004 (Fifth corrected printing of 2. edition). 614

[9] T. M. Cover, J. A. Thomas. *Elements of Information Theory*. John Wiley & Sons, 1991. 615

[10] M. Effros, K. Viswesvariah, S. Kulkarni, S. Verdú. Universal lossless source coding with the Burrows-Wheeler transform. *IEEE Transactions on Information Theory*, 48(5):1061–1081, 2002. 615

[11] T. S. Han, K. Kobayashi. *Mathematics of Information and Coding*. American Mathematical Society, 2002. 615

[12] D. Hankerson, G. A. Harris, P. D. Johnson. *Introduction to Information Theory and Data Compression*. CRC Press, 2003 (2. edition). 615

[13] D. A. Huffman. A method for the construction of minimum-redundancy codes. *Proceedings of the IRE*, 40(9):1098–1101, 1952. 614

[14] R. Krichevsky, V. Trofimov. The performance of universal encoding. *IEEE Transactions on Information Theory*, 27:199–207, 1981. 615

[15] S. Kurtz, B. Balkenhol. Space efficient linear time computation of the Burrows and Wheeler transformation. in I. Althöfer, N. Cai, G. Dueck, L. Khachatrian, M. Pinsker, A. Sárközy, I. Wegener, Z. Zhang (editors) *Numbers, Information and Complexity*. Kluwer Academic Publishers, 2000, pp. 375–383. 615

[16] G. Langdon. An introduction to arithmetic coding. *IBM Journal of Research and Development*, 28:135–149, 1984. 614

[17] M. Nelson, J. L. Gailly. *The Data Compression Book*. M&T Books, 1996. 615

[18] R. Pasco. *Source Coding Algorithms for Fast Data Compression*. PhD thesis, Stanford University, 1976. 614

[19] J. J. Rissanen. Generalized Kraft inequality and arithmetic coding. *http://www.research.ibm.com/journal/ Journal of Research and Development*, 20:198–203, 1976. 614

[20] D. Salomon. *Data Compression*. Springer-Verlag, 2004 (3. edition). 615

[21] K. Sayood. *Introduction to Data Compression*. Morgan Kaufman Publisher, 2000 (2. edition). 615

[22] D. S. Taubman, M. W. Marcelin. *JPEG 2000 – Image Compression, Fundamentals, Standards and Practice*. Society for Industrial and Applied Mathematics, 1983. 615

[23] G. Wallace. The JPEG still picture compression standard. *Communications of the ACM*, 34:30–44, 1991. 615

[24] D. Welsh. *Codes and Cryptography*. Oxford University Press, 1988. 614

[25] F. M. J. Willems, Y. M. Shtarkov, T. J. Tjalkens. The context-tree weighting method: basic properties. *IEEE Transactions on Information Theory*, 47:653–664, 1995. 615

[26] F. M. J. Willems, Y. M. Shtarkov, T. J. Tjalkens. The context-tree weighting method: basic properties. *IEEE Information Theory Society Newsletter*, 1:1 and 20–27, 1997. 615

[27] I. H. Witten, R. M. Neal, J. G. Cleary. Arithmetic coding for sequential data compression. *Communications of the ACM*, 30:520–540, 1987. 614

[28] J. Ziv, A. Lempel. A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory*, 23:337–343, 1977. 615

[29] J. Ziv, A. Lempel. Compression of individual sequences via variable-rate coding. *IEEE Transactions on Information Theory*, 24:530–536, 1978. 615

Name index

Subject Index

Contents