4

Data compression: efficient coding of a random message

In this chapter we will consider a new type of coding. So far we have concentrated on codes that can help detect or even correct errors; we now would like to use codes to represent some information more *efficiently*, i.e. we try to represent the same information using fewer digits on average. Hence, instead of protecting data from errors, we try to *compress* it such as to use less storage space.

To achieve such a compression, we will assume that we know the probability distribution of the messages being sent. If some symbols are more probable than others, we can then take advantage of this by assigning shorter codewords to the more frequent symbols and longer codewords to the rare symbols. Hence, we see that such a code has codewords that are not of fixed length.

Unfortunately, variable-length codes bring with them a fundamental problem: at the receiving end, how do you recognize the end of one codeword and the beginning of the next? To attain a better understanding of this question and to learn more about how to design a good code with a short average codeword length, we start with a motivating example.

4.1 A motivating example

You would like to set up your own telephone system that connects you to your three best friends. The question is how to design efficient binary phone numbers. In Table 4.1 you find six different ways of how you could choose them.

Note that in this example the phone number is a *codeword* for the person we want to talk to. The set of all phone numbers is called *code*. We also assume that you have different probabilities when calling your friends: Bob is your best friend whom you will call in 50% of the times. Alice and Carol are contacted with a frequency of 25% each.

Table 4.1 *Binary phone numbers for a telephone system with three friends*

Friend	Probability	Phone number					
Alice	1/4	0011	001101	0	00	0	10
Bob	1/2	0011	001110	1	11	11	0
Carol	1/4	1100	110000	10	10	10	11
		(i)	(ii)	(iii)	(iv)	(v)	(vi)

Let us discuss the different designs in Table 4.1.

(i) In this design, Alice and Bob have the same phone number. The system obviously will not be able to connect properly.

(ii) This is much better, i.e. the code will actually work. However, the phone numbers are quite long and therefore the design is rather inefficient.

(iii) Now we have a code that is much shorter and, at the same time, we have made sure that we do not use the same codeword twice. However, a closer look reveals that the system will not work. The problem here is that this code is not *uniquely decodable*: if you dial 10 this could mean "Carol" or also "Bob, Alice." Or, in other words, the telephone system will never connect you to Carol, because once you dial 1, it will immediately connect you to Bob.

(iv) This is the first quite efficient code that is functional. But we note something: when calling Alice, why do we have to dial two zeros? After the first zero it is already clear to whom we would like to be connected! Let us fix that in design (v).

(v) This is still uniquely decodable and obviously more efficient than (iv). Is it the most efficient code? No! Since Bob is called most often, he should be assigned the shortest codeword!

(vi) This is the optimal code. Note one interesting property: even though the numbers do not all have the same length, once you finish dialing any of the three numbers, the system immediately knows that you have finished dialing. This is because no codeword is the *prefix*[1] of any other codeword, i.e. it never happens that the first few digits of one codeword are identical to another codeword. Such a code is called *prefix-free* (see Section 4.2). Note that (iii) was not prefix-free: 1 is a prefix of 10.

[1] According to the *Oxford English Dictionary*, a *prefix* is a word, letter, or number placed before another.

From this example we learn the following requirements that we impose on our code design.

- A code needs to be *uniquely decodable*.
- A code should be short; i.e., we want to *minimize the average codeword length* L_{av}, which is defined as follows:

$$\mathrm{L}_{\mathrm{av}} \triangleq \sum_{i=1}^{r} p_i l_i. \tag{4.1}$$

  Here p_i denotes the probability that the source emits the ith symbol, i.e. the probability that the ith codeword $\mathbf{c}_i$ is selected; l_i is the length of the ith codeword $\mathbf{c}_i$; and r is the number of codewords.
- We additionally require the code to be *prefix-free*. Note that this requirement is not necessary, but only convenient. However, we will later see that we lose nothing by asking for it.

Note that any prefix-free code is implicitly uniquely decodable, but not vice versa. We will discuss this issue in more detail in Section 4.7.

4.2 Prefix-free or instantaneous codes

Consider the following code with four codewords:

$$\begin{aligned} \mathbf{c}_1 &= 0 \\ \mathbf{c}_2 &= 10 \\ \mathbf{c}_3 &= 110 \\ \mathbf{c}_4 &= 111 \end{aligned} \tag{4.2}$$

Note that the zero serves as a kind of "comma": whenever we receive a zero (or the code has reached length 3), we know that the codeword has finished. However, this comma still contains useful information about the message as there is still one codeword without it! This is another example of a prefix-free code. We recall the following definition.

Definition 4.1 A code is called *prefix-free* (sometimes also called *instantaneous*) if no codeword is the prefix of another codeword.

The name *instantaneous* is motivated by the fact that for a prefix-free code we can decode instantaneously once we have received a codeword and do not need to wait for later until the decoding becomes unique. Unfortunately, in the literature one also finds that people call a prefix-free code a *prefix code*. This

name is confusing because rather than having prefixes it is the point of the code to have *no* prefix! We will stick to the name of *prefix-free codes*.

Consider next the following example:

$$\begin{aligned} \mathbf{c}_1 &= 0 \\ \mathbf{c}_2 &= 01 \\ \mathbf{c}_3 &= 011 \\ \mathbf{c}_4 &= 111 \end{aligned} \tag{4.3}$$

This code is not prefix-free (0 is a prefix of 01 and 011; 01 is a prefix of 011), but it is still uniquely decodable.

Exercise 4.2 *Given the code in (4.3), split the sequence* 0011011110 *into codewords.* ◊

Note the drawback of the code design in (4.3): the receiver needs to wait and see how the sequence continues before it can make a unique decision about the decoding. The code is *not instantaneously* decodable.

Apart from the fact that they can be decoded instantaneously, another nice property of prefix-free codes is that they can very easily be represented by leaves of *decision trees*. To understand this we will next make a small detour and talk about trees and their relation to codes.

4.3 Trees and codes

The following definition is quite straightforward.

Definition 4.3 (Trees) A *rooted tree* consists of a root with some branches, nodes, and leaves, as shown in Figure 4.1. A *binary tree* is a rooted tree in which each node (hence also the root) has exactly two children,[2] i.e. two branches stemming forward.

The clue to this section is to note that *any* binary code can be represented as a binary tree. Simply take any codeword and regard each bit as a decision

[2] The alert reader might wonder why we put so much emphasis on having *exactly* two children. It is quite obvious that if a parent node only had one child, then this node would be useless and the child could be moved back and replace its parent. The reason for our definition, however, has nothing to do with efficiency, but is related to the generalization to D-ary trees where every node has exactly D children. We do not cover such trees in this book; the interested reader is referred to, e.g., [Mas96].

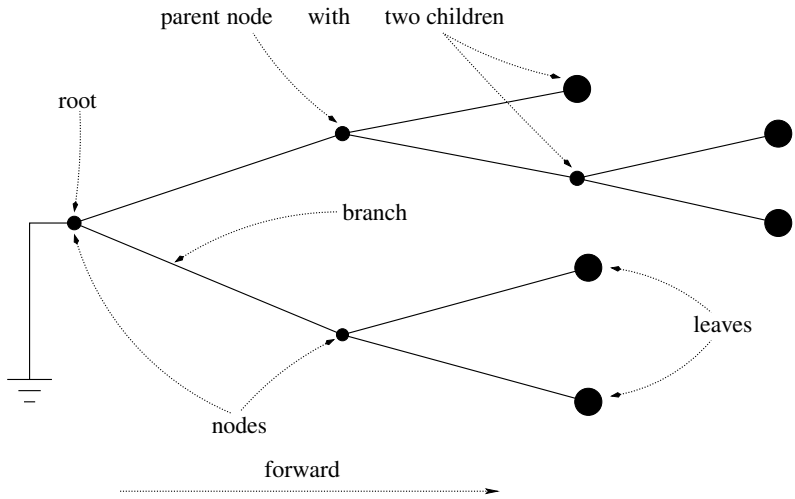

Figure 4.1 A rooted tree (in this case a binary tree) with a *root* (the node that is grounded), four *nodes* (including the root), and five *leaves*. Note that in this book we will always clearly distinguish between *nodes* and *leaves*: a node always has children, while a leaf always is an "end-point" in the tree.

whether to go up ("0") or down[3] ("1"). Hence, every codeword can be represented by a particular path traversing through the tree. As an example, Figure 4.2 shows the binary tree of a binary code with five codewords. Note that on purpose we also keep branches that are not used in order to make sure that the tree is binary.

In Figure 4.3, we show the tree describing the prefix-free code given in (4.2). Note that here every codeword is a leaf. This is no accident.

Lemma 4.4 *A binary code* $\{\mathbf{c}_1, \ldots, \mathbf{c}_r\}$ *is prefix-free if, and only if, in its binary tree every codeword is a leaf. (But not every leaf necessarily is a codeword; see, e.g., code (iv) in Figure 4.4.)*

Exercise 4.5 *Prove Lemma 4.4.*

Hint: Think carefully about the definition of prefix-free codes (see Definition 4.1). ◊

As mentioned, the binary tree of a prefix-free code might contain leaves that are not codewords. Such leaves are called *unused leaves*.

Some more examples of trees of prefix-free and non-prefix-free codes are shown in Figure 4.4.

[3] It actually does not matter whether 1 means up and 0 down, or vice versa.

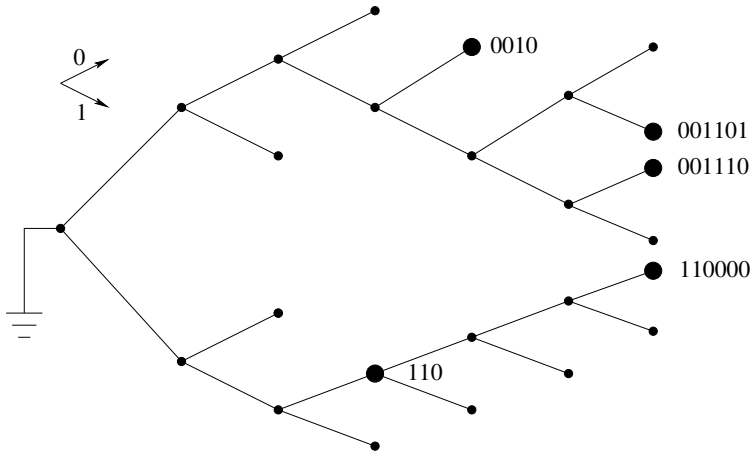

Figure 4.2 An example of a binary tree with five codewords: 110, 0010, 001101, 001110, and 110000. At every node, going upwards corresponds to a 0, and going downwards corresponds to a 1. The node with the ground symbol is the *root* of the tree indicating the starting point.

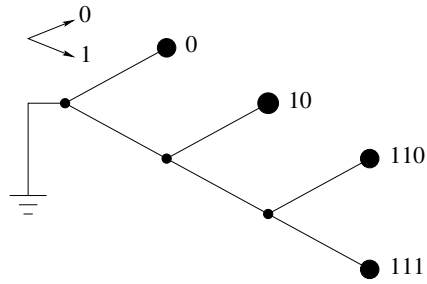

Figure 4.3 Decision tree corresponding to the prefix-free code given in (4.2).

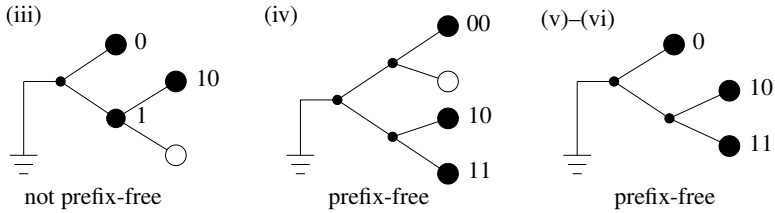

Figure 4.4 Examples of codes and their corresponding trees. The examples are taken from Table 4.1. The prefix-free code (iv) has one unused leaf.

An important concept of trees is the *depths* of their leaves.

Definition 4.6 The *depth of a leaf* in a binary tree is the number of steps it takes when walking from the root forward to the leaf.

As an example, consider again Figure 4.4. Tree (iv) has four leaves, all of them at depth 2. Both tree (iii) and tree (v)–(vi) have three leaves, one at depth 1 and two at depth 2.

We now will derive some interesting properties of trees. Since codes can be represented as trees, we will then be able to apply these properties directly to codes.

Lemma 4.7 (Leaf-Counting and Leaf-Depth Lemma) *The number of leaves n and their depths $l_1, l_2, \ldots, l_n$ in a binary tree satisfy:*

$$n = 1 + \mathrm{N}, \tag{4.4}$$

$$\sum_{i=1}^{n} 2^{-l_i} = 1, \tag{4.5}$$

where N *is the number of nodes (including the root).*

Proof By *extending a leaf* we mean changing a leaf into a node by adding two branches that stem forward. In that process

- we reduce the number of leaves by 1,
- we increase the number of nodes by 1, and
- we increase the number of leaves by 2,

i.e. in total we gain one node and one leaf. This process is depicted graphically in Figure 4.5.

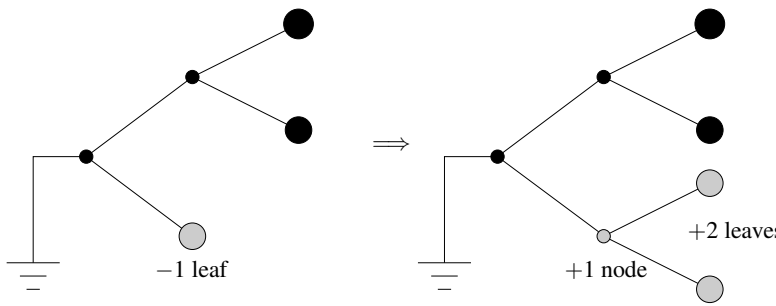

Figure 4.5 Extending a leaf: both the total number of nodes and the total number of leaves is increased by 1.

To prove the first statement (4.4), we start with the *extended root*; i.e., at the beginning we have the root and $n = 2$ leaves. In this case we have $\mathrm{N} = 1$ and (4.4) is satisfied. Now we can grow any tree by continuously extending some leaf, every time increasing the number of leaves and nodes by one each. We see that (4.4) remains valid. By induction this proves the first statement.

We will prove the second statement (4.5) also by induction. We again start with the extended root.

(1) An extended root has two leaves, all at depth 1: $l_i = 1$. Hence,

$$\sum_{i=1}^{n} 2^{-l_i} = \sum_{i=1}^{2} 2^{-1} = 2 \cdot 2^{-1} = 1; \tag{4.6}$$

i.e., for the extended root, (4.5) is satisfied.

(2) Suppose $\sum_{i=1}^{n} 2^{-l_i} = 1$ holds for an arbitrary binary tree with n leaves. Now we extend one leaf, say the nth leaf.[4] We get a new tree with $n' = n+1$ leaves, where

$$\sum_{i=1}^{n'} 2^{-l_i} = \underbrace{\sum_{i=1}^{n-1} 2^{-l_i}}_{\substack{\text{unchanged}\\ \text{leaves}}} + 2 \cdot \underbrace{2^{-(l_n+1)}}_{\substack{\text{new leaves}\\ \text{at depth}\\ l_n+1}} \tag{4.7}$$

$$= \sum_{i=1}^{n-1} 2^{-l_i} + 2^{-l_n} \tag{4.8}$$

$$= \sum_{i=1}^{n} 2^{-l_i} = 1. \tag{4.9}$$

Here the last equality follows from our assumption that $\sum_{i=1}^{n} 2^{-l_i} = 1$. Hence, by extending one leaf, the second statement continues to hold.

(3) Since any tree can be grown by continuously extending some leaves, the proof follows by induction. □

We are now ready to apply our first insights about trees to codes.

4.4 The Kraft Inequality

The following theorem is very useful because it gives us a way of finding out whether a prefix-free code exists or not.

[4] Since the tree is arbitrary, it does not matter how we number the leaves!

Theorem 4.8 (Kraft Inequality) *There exists a binary prefix-free code with r codewords of lengths $l_1, l_2, \ldots, l_r$ if, and only if,*

$$\sum_{i=1}^{r} 2^{-l_i} \leq 1. \tag{4.10}$$

If (4.10) is satisfied with equality, then there are no unused leaves in the tree.

Example 4.9 Let $l_1 = 3$, $l_2 = 4$, $l_3 = 4$, $l_4 = 4$, $l_5 = 4$. Then

$$2^{-3} + 4 \cdot 2^{-4} = \frac{1}{8} + \frac{4}{16} = \frac{3}{8} \leq 1; \tag{4.11}$$

i.e., there exists a binary prefix-free code consisting of five codewords with the given codeword lengths.

On the other hand, we cannot find any prefix-free code with five codewords of lengths $l_1 = 1$, $l_2 = 2$, $l_3 = 3$, $l_4 = 3$, and $l_5 = 4$ because

$$2^{-1} + 2^{-2} + 2 \cdot 2^{-3} + 2^{-4} = \frac{17}{16} > 1. \tag{4.12}$$

These two examples are shown graphically in Figure 4.6. ◊

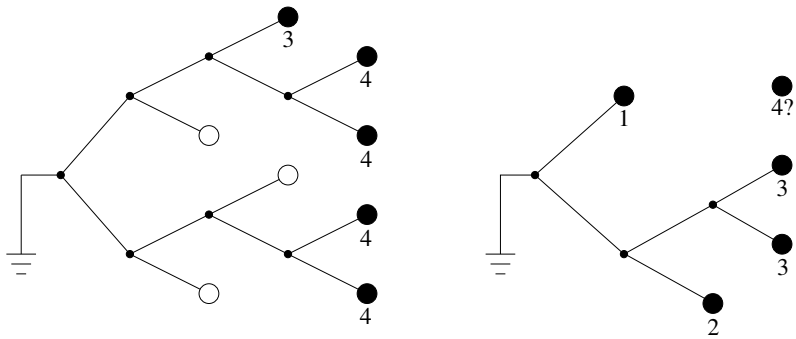

Figure 4.6 Examples of the Kraft Inequality.

Proof of the Kraft Inequality We prove the two directions separately.

$\Longrightarrow$: Suppose that there exists a binary prefix-free code with the given codeword lengths. From Lemma 4.4 we know that all r codewords of a binary prefix-free code are leaves in a binary tree. The total number n of (used and unused) leaves in this tree can therefore not be smaller than r, i.e.

$$r \leq n. \tag{4.13}$$

Hence,

$$\sum_{i=1}^{r} 2^{-l_i} \le \sum_{i=1}^{n} 2^{-l_i} = 1, \tag{4.14}$$

where the last equality follows from the Leaf-Depth Lemma (Lemma 4.7).

$\Longleftarrow$: Suppose that $\sum_{i=1}^{r} 2^{-l_i} \le 1$. We now can construct a prefix-free code as follows:

Step 1 Start with the extended root, i.e. a tree with two leaves, set $i = 1$, and assume, without loss of generality, that $l_1 \le l_2 \le \cdots \le l_r$.

Step 2 If there is an unused leaf at depth l_i, put the ith codeword there. Note that there could be none because l_i can be strictly larger than the current depth of the tree. In this case, extend any unused leaf to depth l_i, and put the ith codeword to one of the new leaves.

Step 3 If $i = r$, stop. Otherwise $i \to i+1$ and go to Step 2.

We only need to check that Step 2 is always possible, i.e. that there is always some unused leaf available. To that goal, note that if we get to Step 2, we have already put $i-1$ codewords into the tree. From the Leaf-Depth Lemma (Lemma 4.7) we know that

$$1 = \sum_{j=1}^{n} 2^{-\tilde{l}_j} = \underbrace{\sum_{j=1}^{i-1} 2^{-l_j}}_{\text{used leaves}} + \underbrace{\sum_{j=i}^{n} 2^{-\tilde{l}_j}}_{\text{unused leaves}}, \tag{4.15}$$

where $\tilde{l}_j$ are the depths of the leaves in the tree at that moment; i.e., $(\tilde{l}_1, \ldots, \tilde{l}_{i-1}) = (l_1, \ldots, l_{i-1})$ and $\tilde{l}_i, \ldots, \tilde{l}_n$ are the depths of the (so far) unused leaves. Now note that in our algorithm $i \le r$, i.e.

$$\sum_{j=1}^{i-1} 2^{-l_j} < \sum_{j=1}^{r} 2^{-l_j} \le 1, \tag{4.16}$$

where the last inequality follows by assumption. Hence,

$$\underbrace{\sum_{j=1}^{i-1} 2^{-l_j}}_{<1} + \sum_{j=i}^{n} 2^{-\tilde{l}_j} = 1 \quad \Longrightarrow \quad \sum_{j=i}^{n} 2^{-\tilde{l}_j} > 0 \tag{4.17}$$

and there still must be some unused leaves available! □

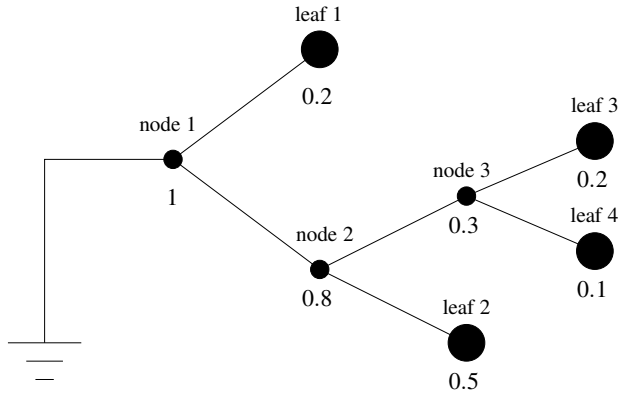

Figure 4.7 Rooted tree with probabilities.

4.5 Trees with probabilities

We have seen already in Section 4.1 that for codes it is important to consider the probabilities of the codewords. We therefore now introduce probabilities in our trees.

Definition 4.10 A *rooted tree with probabilities* is a finite rooted tree with probabilities assigned to each node and leaf such that

- the probability of a node is the sum of the probabilities of its children, and
- the root has probability 1.

An example of a rooted tree with probabilities is given in Figure 4.7. Note that the probabilities can be seen as the overall probability of passing through a particular node (or reaching a particular leaf) when making a random walk from the root to a leaf. Since we start at the root, the probability that our path goes through the root is always 1. Then, in the example of Figure 4.7, we have an 80% chance that our path will go through node 2 and a 10% chance to end up in leaf 4.

Since in a prefix-free code all codewords are leaves and we are particularly interested in the average codeword length, we are very much interested in the average depth of the leaves in a tree (where for the averaging operation we use the probabilities in the tree). Luckily, there is an elegant way to compute this average depth, as shown in the following lemma.

Lemma 4.11 (Path Length Lemma) *In a rooted tree with probabilities, the*

average depth L_{av} *of the leaves is equal to the sum of the probabilities of all nodes (including the root).*

To clarify our notation we refer to leaf probabilities by small p_i while node probabilities are denoted by capital P_ℓ.

Example 4.12 Consider the tree of Figure 4.7. We have four leaves: one at depth $l_1 = 1$ with a probability $p_1 = 0.2$, one at depth $l_2 = 2$ with a probability $p_2 = 0.5$, and two at depth $l_3 = l_4 = 3$ with probabilities $p_3 = 0.2$ and $p_4 = 0.1$, respectively. Hence, the average depth of the leaves is given by

$$\mathrm{L}_{\mathrm{av}} = \sum_{i=1}^{4} p_i l_i = 0.2 \cdot 1 + 0.5 \cdot 2 + 0.2 \cdot 3 + 0.1 \cdot 3 = 2.1. \tag{4.18}$$

According to Lemma 4.11, this must be equal to the sum of the node probabilities:

$$\mathrm{L}_{\mathrm{av}} = P_1 + P_2 + P_3 = 1 + 0.8 + 0.3 = 2.1. \tag{4.19}$$

◊

Proof of Lemma 4.11 The lemma is easiest understood when looking at a particular example. Let us again consider the tree of Figure 4.7: the probability $p_1 = 0.2$ of leaf 1 needs to be counted once only, which is the case as it is only part of the probability of the root $P_1 = 1$. The probability $p_2 = 0.5$ must be counted twice. This is also the case because it is contained in the root probability $P_1 = 1$ and also in the probability of the second node $P_2 = 0.8$. Finally, the probabilities of leaf 3 and leaf 4, $p_3 = 0.2$ and $p_4 = 0.1$, are counted three times: they are part of P_1, P_2, and P_3:

$$\begin{aligned}
\mathrm{L}_{\mathrm{av}} &= 2.1 && (4.20)\\
&= 1 \cdot 0.2 + 2 \cdot 0.5 + 3 \cdot 0.2 + 3 \cdot 0.1 && (4.21)\\
&= 1 \cdot (0.2 + 0.5 + 0.2 + 0.1) + 1 \cdot (0.5 + 0.2 + 0.1) \\
&\quad + 1 \cdot (0.2 + 0.1) && (4.22)\\
&= 1 \cdot P_1 + 1 \cdot P_2 + 1 \cdot P_3 && (4.23)\\
&= P_1 + P_2 + P_3. && (4.24)
\end{aligned}$$

□

4.6 Optimal codes: Huffman code

Let us now connect the probabilities in the tree with the probabilities of the code, or actually, more precisely, the probabilities of the random messages that

shall be represented by the code. We assume that we have in total r different message symbols. Let the probability of the ith message symbol be p_i and let the length of the corresponding codeword representing this symbol be l_i. Then the average length of the code is given by

$$\mathrm{L_{av}} = \sum_{i=1}^{r} p_i l_i. \tag{4.25}$$

With no loss in generality, the p_i may be taken in nonincreasing order. If the lengths l_i are then not in the opposite order, i.e. we do not have both

$$p_1 \geq p_2 \geq p_3 \geq \cdots \geq p_r \tag{4.26}$$

and

$$l_1 \leq l_2 \leq l_3 \leq \cdots \leq l_r, \tag{4.27}$$

then the code is not *optimal* in the sense that we could have a shorter average length by reassigning the codewords to different symbols. To prove this claim, suppose that for some i and j with $i < j$ we have both

$$p_i > p_j \qquad \text{and} \qquad l_i > l_j. \tag{4.28}$$

In computing the average length, originally, the sum in (4.25) contains, among others, the two terms

$$\text{old:} \qquad p_i l_i + p_j l_j. \tag{4.29}$$

By interchanging the codewords for the ith and jth symbols, we get the terms

$$\text{new:} \qquad p_i l_j + p_j l_i, \tag{4.30}$$

while the remaining terms are unchanged. Subtracting the old from the new we see that

$$\begin{aligned} \text{new} - \text{old:} \qquad (p_i l_j + p_j l_i) - (p_i l_i + p_j l_j) &= p_i(l_j - l_i) + p_j(l_i - l_j) && (4.31)\\ &= (p_i - p_j)(l_j - l_i) && (4.32)\\ &< 0. && (4.33) \end{aligned}$$

From (4.28) this is a negative number, i.e. we can decrease the average codeword length by interchanging the codewords for the ith and jth symbols. Hence the new code with exchanged codewords for the ith and jth symbols is better than the original code – which therefore cannot have been optimal.

We will now examine the optimal binary code which is called the *Huffman code* due to its discoverer. The trick of the derivation of the optimal code is the insight that the corresponding code tree has to be grown backwards, starting

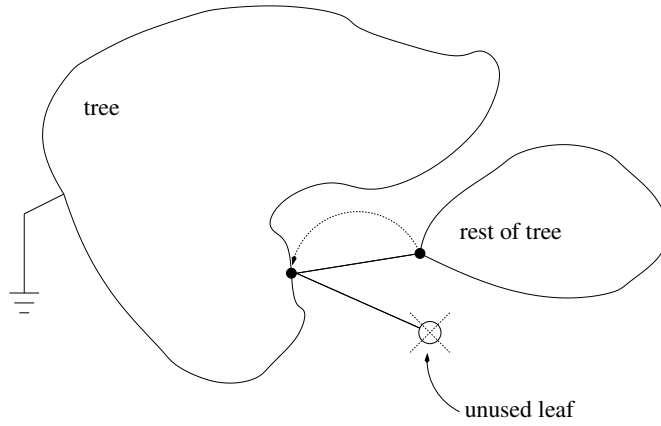

Figure 4.8 Code performance and unused leaves: by deleting the unused leaf and moving its sibling to the parent, we can improve on the code's performance.

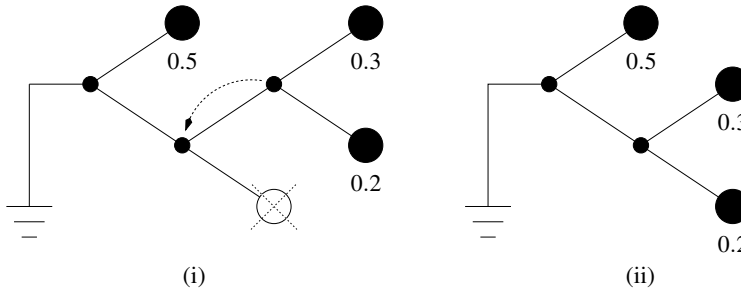

Figure 4.9 Improving a code by removing an unused leaf.

from the leaves (and not, as might be intuitive at a first glance, starting from the root).

The clue of binary Huffman coding lies in two basic observations. The first observation is as follows.

Lemma 4.13 *In a binary tree of an optimal binary prefix-free code, there is no unused leaf.*

Proof Suppose that the tree of an optimal code has an unused leaf. Then we can delete this leaf and move its sibling to the parent node; see Figure 4.8. By doing so we reduce L_{av}, which contradicts our assumption that the original code was optimal. □

Example 4.14 As an example consider the two codes given in Figure 4.9, both of which have three codewords. Code (i) has an average length[5] of $\mathrm{L}_{\mathrm{av}} = 2$, and code (ii) has an average length of $\mathrm{L}_{\mathrm{av}} = 1.5$. Obviously, code (ii) performs better. ◊

The second observation basically says that the two most unlikely symbols must have the longest codewords.

Lemma 4.15 *There exists an optimal binary prefix-free code such that the two least likely codewords only differ in the last digit, i.e. the two most unlikely codewords are siblings.*

Proof Since we consider an optimal code, the codewords that correspond to the two least likely symbols must be the longest codewords (see our discussion after (4.27)). If they have the same parent node, we are done. If they do not have the same parent node, this means that there exist other codewords of the same length (because we know from Lemma 4.13 that there are no unused leaves). In this case, we can simply swap two codewords of equal maximum length in such a way that the two least likely codewords have the same parent, and we are done. □

Because of Lemma 4.13 and the Path Length Lemma (Lemma 4.11), we see that the construction of an optimal binary prefix-free code for an r-ary random message U is equivalent to constructing a binary tree with r leaves such that the sum of the probabilities of the nodes is minimum when the leaves are assigned the probabilities p_i for $i = 1, 2, \ldots, r$:

$$\mathrm{L}_{\mathrm{av}} = \underbrace{\sum_{\ell=1}^{\mathrm{N}} P_\ell}_{\longrightarrow \text{minimize!}} . \tag{4.34}$$

But Lemma 4.15 tells us how we may choose one node in an optimal code tree, namely as the parent of the two least likely leaves p_{r-1} and p_r:

$$P_{\mathrm{N}} = p_{r-1} + p_r. \tag{4.35}$$

So we have fixed one P_ℓ in (4.34) already. But, if we now pruned our binary tree at this node to make it a leaf with probability $p = p_{r-1} + p_r$, it would become one of $(r-1)$ leaves in a new tree. Completing the construction of the optimal code would then be equivalent to constructing a binary tree with these

[5] Remember Lemma 4.11 to compute the average codeword length: summing the node probabilities. In code (i) we have $P_1 = 1$ and $P_2 = P_3 = 0.5$ (note that the unused leaf has, by definition, zero probability), and in code (ii) $P_1 = 1$ and $P_2 = 0.5$.

$(r-1)$ leaves such that the sum of the probabilities of the nodes is minimum:

$$\mathrm{L_{av}} = \underbrace{\sum_{\ell=1}^{\mathrm{N}-1} P_\ell}_{\longrightarrow \text{minimize!}} + \underbrace{P_{\mathrm{N}}}_{\substack{\text{optimally} \\ \text{chosen}}}. \tag{4.36}$$

Again Lemma 4.15 tells us how to choose one node in this new tree, and so on. We have thus proven the validity of the following algorithm.

Huffman's Algorithm for Optimal Binary Codes

Step 1 Create r leaves corresponding to the r possible symbols and assign their probabilities $p_1, \ldots, p_r$. Mark these leaves as active.

Step 2 Create a new node that has the two *least likely* active leaves or nodes as children. Activate this new node and deactivate its children.

Step 3 If there is only one active node left, root it. Otherwise, go to Step 2.

Example 4.16 In Figure 4.10 we show the procedure of producing a Huffman code for the example of a random message with four possible symbols with probabilities $p_1 = 0.4$, $p_2 = 0.3$, $p_3 = 0.2$, $p_4 = 0.1$. We see that the average codeword length of this Huffman code is

$$\mathrm{L_{av}} = 0.4 \cdot 1 + 0.3 \cdot 2 + 0.2 \cdot 3 + 0.1 \cdot 3 = 1.9. \tag{4.37}$$

Using Lemma 4.11 this can be computed much easier as follows:

$$\mathrm{L_{av}} = P_1 + P_2 + P_3 = 1 + 0.6 + 0.3 = 1.9. \tag{4.38}$$

◊

Note that the code design process is not unique in several respects. Firstly, the assignment of the 0 or 1 digits to the codewords at each forking stage is arbitrary, but this produces only trivial differences. Usually, we will stick to the convention that going upwards corresponds to 0 and downwards to 1. Secondly, when there are more than two least likely (active) nodes/leaves, it does not matter which we choose to combine. The resulting codes can have codewords of different lengths; however, the average codeword length will always be the same.

Example 4.17 As an example of different Huffman encodings of the same random message, let $p_1 = 0.4$, $p_2 = 0.2$, $p_3 = 0.2$, $p_4 = 0.1$, $p_5 = 0.1$. Figure 4.11 shows three different Huffman codes for this message: the list of

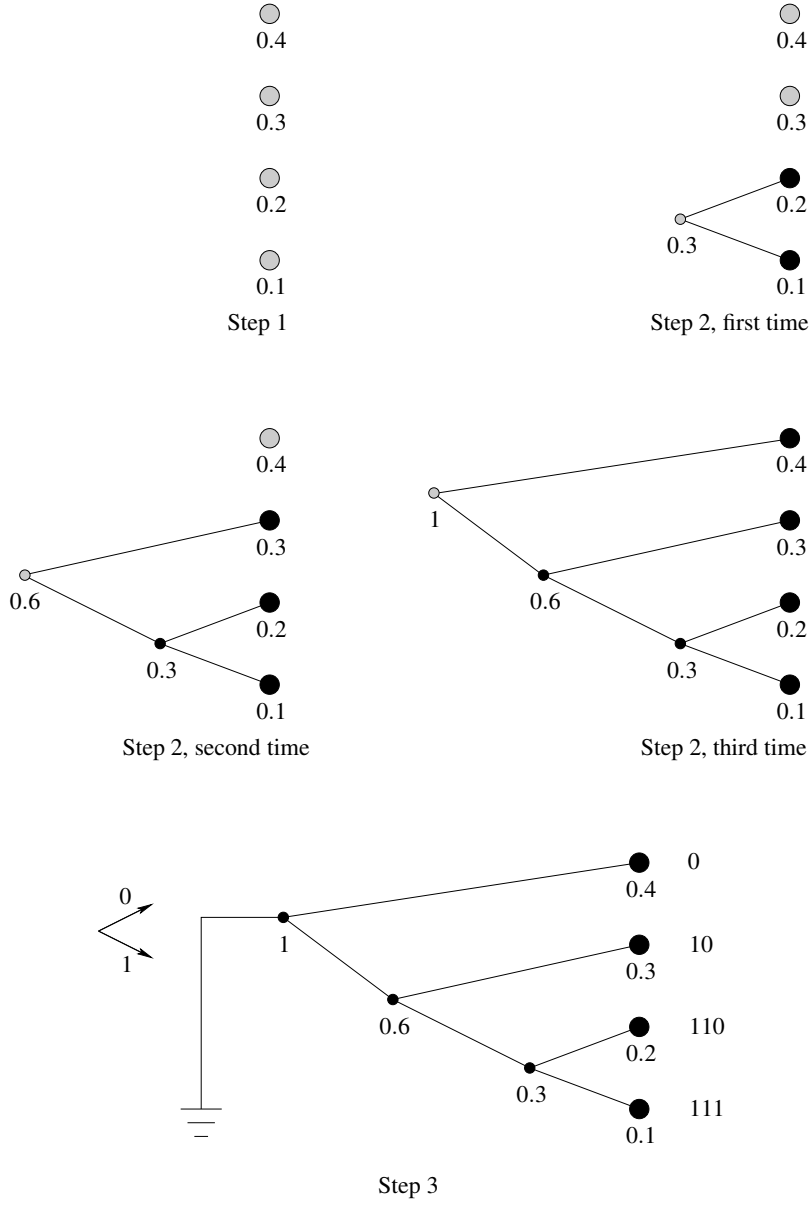

Figure 4.10 Creation of a binary Huffman code. Active nodes and leaves are shaded.

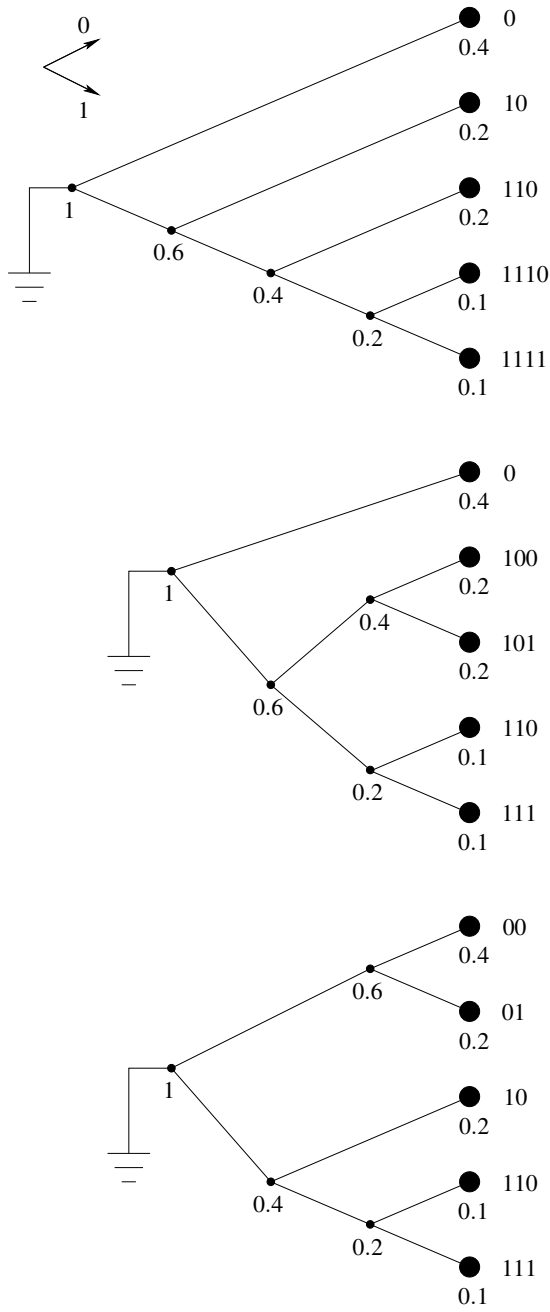

Figure 4.11 Different binary Huffman codes for the same random message.

codeword lengths are $(1,2,3,4,4)$, $(1,3,3,3,3)$, and $(2,2,2,3,3)$, respectively. But all of these codes have the same performance, $L_{av} = 2.2$. ◊

Exercise 4.18 *Try to generate all three codes of Example 4.17 (see Figure 4.11) yourself.* ◊

4.7 Types of codes

Note that in Section 4.1 we have restricted ourselves to prefix-free codes. So, up to now we have only proven that Huffman codes are the optimal codes *under the assumption that we restrict ourselves to prefix-free codes*. We would now like to show that Huffman codes are actually optimal among all useful codes.

To reach that goal, we need to come back to a more precise definition of "useful codes," i.e. we continue the discussion that we started in Section 4.1. Let us consider an example with a random message U with four different symbols and let us design various codes for this message as shown in Table 4.2.

Table 4.2 *Various codes for a random message with four possible values*

U	Code (i)	Code (ii)	Code (iii)	Code (iv)
a	0	0	10	0
b	0	010	00	10
c	1	01	11	110
d	1	10	110	111

We discuss these different codes.

Code (i) is useless because some codewords are used for more than one symbol. Such a code is called *singular*.

Code (ii) is nonsingular. But we have another problem: if we receive 010 we have three different possibilities how to decode it: it could be (010) giving us b, or it could be $(0)(10)$ leading to ad, or it could be $(01)(0)$ corresponding to ca. Even though nonsingular, this code is not *uniquely decodable* and therefore in practice is as useless as code (i).[6]

[6] Note that adding a comma between the codewords is not allowed because in this case we change the code to be *ternary*, i.e. the codewords contain three different letters "0", "1", and "," instead of only two "0" and "1". By the way, it is not very difficult to generalize all results given in this chapter to D-ary codes. See, for example, [Mas96]. In this book, we will stick to binary codes.

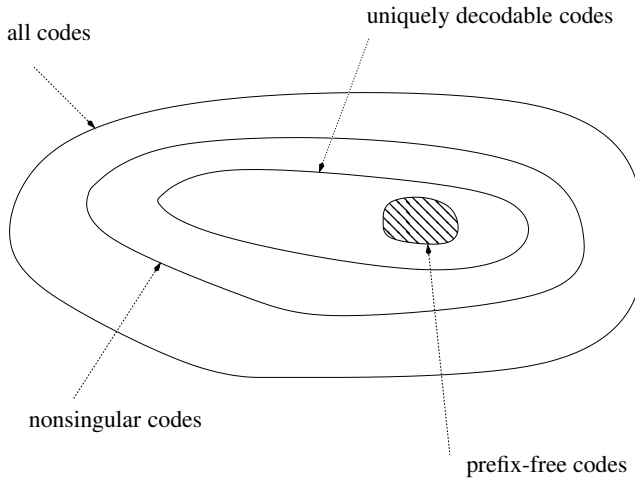

Figure 4.12 Set of all codes.

Code (iii) is uniquely decodable, even though it is not prefix-free! To see this, note that in order to distinguish between c and d we only need to wait for the next 1 to show up: if the number of 0s in between is even, we decode 11, otherwise we decode 110. Example:

$$11000010 = (11)(00)(00)(10) \quad \Longrightarrow \quad cbba, \tag{4.39}$$
$$110000010 = (110)(00)(00)(10) \quad \Longrightarrow \quad dbba. \tag{4.40}$$

So in a uniquely decodable but not prefix-free code we may have to delay the decoding until later.

Code (iv) is prefix-free and therefore trivially uniquely decodable.

We see that the set of all possible codes can be grouped as shown in Figure 4.12. We are only interested in the uniquely decodable codes. But so far we have restricted ourselves to prefix-free codes. So the following question arises: is there a uniquely decodable code that is not prefix-free, but that has a better performance than the best prefix-free code (i.e. the corresponding Huffman code)?

Luckily the answer to this question is No, i.e. the Huffman codes are the best uniquely decodable codes. This can be seen from the following theorem.

Theorem 4.19 (McMillan's Theorem) *The codeword lengths l_i of any*

uniquely decodable code must satisfy the Kraft Inequality

$$\sum_{i=1}^{r} 2^{-l_i} \le 1. \tag{4.41}$$

Why does this help to answer our question about the most efficient uniquely decodable code? Well, note that we know from Theorem 4.8 that every prefix-free code also satisfies (4.41). So, for any uniquely decodable, but non-prefix-free code with given codeword lengths, one can find another code with the same codeword lengths that is prefix-free. But if the codeword lengths are the same, the performance is identical! Hence, there is no gain in designing a non-prefix-free code.

Proof of Theorem 4.19 Suppose we are given a random message U that takes on r possible values $u \in \mathcal{U}$ (here the set $\mathcal{U}$ denotes the message alphabet). Suppose further that we have a *uniquely decodable* code that assigns to every possible symbol $u \in \mathcal{U}$ a certain codeword of length $l(u)$.

Now choose an arbitrary positive integer ν and design a new code for a vector of ν symbols $\mathbf{u} = (u_1, u_2, \ldots, u_\nu) \in \mathcal{U}^\nu = \mathcal{U} \times \cdots \times \mathcal{U}$ by simply *concatenating* the original codewords.

Example 4.20 Consider a ternary message with the possible values $u = a$, $u = b$, or $u = c$, i.e. $\mathcal{U} = \{a, b, c\}$. If the probabilities of these possible values are

$$\Pr[U = a] = \frac{1}{2}, \quad \Pr[U = b] = \Pr[U = c] = \frac{1}{4}, \tag{4.42}$$

a binary (single-letter) Huffman code would map

$$a \mapsto 0, \qquad b \mapsto 10, \qquad c \mapsto 11. \tag{4.43}$$

If we now choose $\nu = 3$, we get a new source with $3^3 = 27$ possible symbols, namely

$$\begin{aligned}\mathcal{U}^3 = \{&aaa, aab, aac, aba, abb, abc, aca, acb, acc,\\ &baa, bab, bac, bba, bbb, bbc, bca, bcb, bcc,\\ &caa, cab, cac, cba, cbb, cbc, cca, ccb, ccc\}.\end{aligned} \tag{4.44}$$

The corresponding 27 codewords are then as follows (given in the same order):

$$\begin{aligned}\{&000, 0010, 0011, 0100, 01010, 01011, 0110, 01110, 01111,\\ &1000, 10010, 10011, 10100, 101010, 101011, 10110, 101110, 101111,\\ &1100, 11010, 11011, 11100, 111010, 111011, 11110, 111110, 111111\}.\end{aligned} \tag{4.45}$$

♢

The clue observation now is that because the original code was uniquely decodable, it immediately follows that this new concatenated code also must be uniquely decodable.

Exercise 4.21 *Explain this clue observation, i.e. explain why the new concatenated code is also uniquely decodable.*

Hint: Note that the codewords of the new code consist of a sequence of uniquely decodable codewords. ◊

The lengths of the new codewords are given by

$$\tilde{l}(\mathbf{u}) = \sum_{j=1}^{\nu} l(u_j). \tag{4.46}$$

Let $l_{\max}$ be the maximal codeword length of the original code. Then the new code has a maximal codeword length $\tilde{l}_{\max}$ satisfying

$$\tilde{l}_{\max} = \nu l_{\max}. \tag{4.47}$$

We now compute the following:

$$\left(\sum_{u\in\mathcal{U}} 2^{-l(u)}\right)^{\nu} = \left(\sum_{u_1\in\mathcal{U}} 2^{-l(u_1)}\right)\left(\sum_{u_2\in\mathcal{U}} 2^{-l(u_2)}\right)\cdots\left(\sum_{u_\nu\in\mathcal{U}} 2^{-l(u_\nu)}\right) \tag{4.48}$$

$$= \sum_{u_1\in\mathcal{U}}\sum_{u_2\in\mathcal{U}}\cdots\sum_{u_\nu\in\mathcal{U}} 2^{-l(u_1)}2^{-l(u_2)}\cdots 2^{-l(u_\nu)} \tag{4.49}$$

$$= \sum_{\mathbf{u}\in\mathcal{U}^\nu} 2^{-l(u_1)-l(u_2)-\cdots-l(u_\nu)} \tag{4.50}$$

$$= \sum_{\mathbf{u}\in\mathcal{U}^\nu} 2^{-\sum_{j=1}^{\nu} l(u_j)} \tag{4.51}$$

$$= \sum_{\mathbf{u}\in\mathcal{U}^\nu} 2^{-\tilde{l}(\mathbf{u})}. \tag{4.52}$$

Here (4.48) follows by writing the exponentiated sum as a product of ν sums; in (4.50) we combine the ν sums over $u_1,\ldots,u_\nu$ into one huge sum over the ν-vector $\mathbf{u}$; and (4.52) follows from (4.46).

Next we will rearrange the order of the terms by collecting all terms with the same exponent together:

$$\sum_{\mathbf{u}\in\mathcal{U}^\nu} 2^{-\tilde{l}(\mathbf{u})} = \sum_{m=1}^{\tilde{l}_{\max}} w(m)2^{-m}, \tag{4.53}$$

where $w(m)$ counts the number of such terms with equal exponent, i.e. $w(m)$ denotes the number of codewords of length m in the new code.

Example 4.22 (Continuation from Example 4.20) We see from (4.45) that the new concatenated code has one codeword of length 3, six codewords of length 4, twelve codewords of length 5, and eight codewords of length 6. Hence,

$$\sum_{\mathbf{u}\in\mathcal{U}^\nu} 2^{-\tilde{l}(\mathbf{u})} = 1\cdot 2^{-3} + 6\cdot 2^{-4} + 12\cdot 2^{-5} + 8\cdot 2^{-6}, \tag{4.54}$$

i.e.

$$w(m) = \begin{cases} 1 & \text{for } m = 3, \\ 6 & \text{for } m = 4, \\ 12 & \text{for } m = 5, \\ 8 & \text{for } m = 6, \\ 0 & \text{otherwise.} \end{cases} \tag{4.55}$$

Also note that $\tilde{l}_{\max} = 6 = \nu \cdot l_{\max} = 3\cdot 2$ in this case. ◊

We combine (4.53) and (4.52) and use (4.47) to write

$$\left(\sum_{u\in\mathcal{U}} 2^{-l(u)}\right)^\nu = \sum_{m=1}^{\nu l_{\max}} w(m) 2^{-m}. \tag{4.56}$$

Note that since the new concatenated code is uniquely decodable, every codeword of length m is used at most once. But in total there are only 2^m different sequences of length m, i.e. we know that

$$w(m) \le 2^m. \tag{4.57}$$

Thus,

$$\left(\sum_{u\in\mathcal{U}} 2^{-l(u)}\right)^\nu = \sum_{m=1}^{\nu l_{\max}} w(m) 2^{-m} \le \sum_{m=1}^{\nu l_{\max}} 2^m 2^{-m} = \nu l_{\max} \tag{4.58}$$

or

$$\sum_{u\in\mathcal{U}} 2^{-l(u)} \le (\nu l_{\max})^{1/\nu}. \tag{4.59}$$

At this stage we are back to an expression that depends only on the original uniquely decodable code. So forget about the trick with the new concatenated code, but simply note that we have shown that for any uniquely decodable code and any positive integer ν, expression (4.59) must hold! Also note that we can choose ν freely here.

Note further that for any finite value of $l_{\max}$ one can show that

$$\lim_{\nu\to\infty} (\nu l_{\max})^{1/\nu} = 1. \tag{4.60}$$

Hence, by choosing ν extremely large (i.e. we let ν tend to infinity) we have

$$\sum_{u\in\mathcal{U}} 2^{-l(u)} \leq 1 \tag{4.61}$$

as we wanted to prove. □

4.8 Some historical background

David A. Huffman had finished his B.S. and M.S. in electrical engineering and also served in the U.S. Navy before he became a Ph.D. student at the Massachusetts Institute of Technology (MIT). There, in 1951, he attended an information theory class taught by Professor Robert M. Fano who was working at that time, together with Claude E. Shannon, on finding the most efficient code, but could not solve the problem. So Fano assigned the question to his students in the information theory class as a term paper. Huffman tried for a long time to find a solution and was about to give up when he had the sudden inspiration to start building the tree backwards from leaves to root instead from root to leaves. Once he had understood this, he was quickly able to prove that his code was the most efficient one. Naturally, Huffman's term paper was later published.

Huffman became a faculty member of MIT in 1953, and later, in 1967, he moved to the University of California, Santa Cruz, where he stayed until his retirement in 1994. He won many awards for his accomplishments, e.g. in 1988 the Richard Hamming Medal from the Institute of Electrical and Electronics Engineers (IEEE). Huffman died in 1998. See [Sti91] and [Nor89].

4.9 Further reading

For an easy-to-read, but precise, introduction to coding and trees, the lecture notes [Mas96] of Professor James L. Massey from ETH Zurich are highly recommended. There the interested reader will find a straightforward way to generalize the concept of binary codes to general D-ary codes using D-ary trees. Moreover, in [Mas96] one also finds the concept of *block codes*, i.e. codes with a fixed codeword length. Some of the best such codes are called *Tunstall codes* [Tun67].

References

[Mas96] James L. Massey, *Applied Digital Information Theory I and II,* Lecture notes, Signal and Information Processing Laboratory, ETH Zurich, 1995/1996. Available: http://www.isiweb.ee.ethz.ch/archive/massey_scr/

[Nor89] Arthur L. Norberg, "An interview with Robert M. Fano," Charles Babbage Institute, Center for the History of Information Processing, April 1989.

[Sti91] Gary Stix, "Profile: Information theorist David A. Huffman," *Scientific American (Special Issue on Communications, Computers, and Networks)*, vol. 265, no. 3, September 1991.

[Tun67] Brian P. Tunstall, "Synthesis of noiseless compression codes," Ph.D. dissertation, Georgia Institute of Technology, September 1967.