CHAPTER 4

Huffman Coding

STEVEN PIGEON

4.1 INTRODUCTION

Codes may be characterized by how general they are with respect to the distribution of symbols they are meant to code. Universal coding techniques assume only a non-increasing distribution. Golomb coding assumes a geometric distribution [1]. Fiala and Greene's (*start, step, stop*) codes assume a piecewise uniform distribution function with each part distributed exponentially [2]. A detailed look at universal coding techniques is presented in Chapter 3. Huffman codes are more general because they assume nothing particular about the distribution, only that all probabilities are non-zero. This generality makes them suitable not only for certain classes of distributions, but for all distributions, including those where there is no obvious relation between the symbol number and its probability, as is the case with text. In text, letters go from "a" to "z" and are usually mapped onto a contiguous range, such as, for example, from 0 to 25 or from 97 to 122, as in ASCII, but there is no direct relation between the symbol's number and its frequency rank.

There are various ways of dealing with this situation. One way could be that we build a permutation function that maps the arbitrary symbol numbers to numbers corresponding to their ranks. This would effectively give us a non-increasing distribution. Once this distribution is obtained, we could use one of the aforementioned codes to do the actual coding. Since it is unlikely that the distribution obtained matches exactly one of the distributions assumed by the simpler codes, a certain inefficiency is introduced, in addition to the storage of the permutation information. This inefficiency may lead to an unacceptable loss of compression. Huffman was the first to give an exact, optimal algorithm to code symbols from an arbitrary distribution [3]. In this chapter, we will see how this algorithm manages to produce not only efficient but optimal codes. We will also see some adaptive algorithms that will change the codebook as symbol statistics are updated, adjusting themselves as probabilities change locally in the data.

4.2 HUFFMAN CODES

Huffman codes solve the problem of finding an optimal codebook for an arbitrary probability distribution of symbols. Throughout this chapter, we will use the following conventions. The uncompressed string of symbols will be the *message* and its alphabet will be the original or message alphabet. The compressed output will be composed of output symbols. We will call a *code* a string of output symbols associated with a message symbol, and a *codebook* will be the set of all codes associated with all symbols in the message alphabet. A codebook can be seen as a function F mapping symbols in A, the message alphabet, to a subset of B^+, the set of all (non-empty) strings composed from the output alphabet B, in a non-ambiguous way.

There are, of course, many different ways of devising codebooks. However, Huffman's algorithm produces optimal codebooks, in the sense that although there may exist many equivalent codebooks, none will have a smaller average code length. Note that this is true when we respect the implicit assumption that each code is composed of an integer number of output symbol, and that each symbol receives a distinct code, which may not be true for all types of codes. Think of arithmetic coding, where the algorithm is able to assign a fractional number of output symbols to a code. Therefore, in the context of Huffman coding, "variable-length codes" really means variable-*integer*-length codes. In this section, we will be concerned with the construction of Huffman codes and their efficiency. We will also see that while we generally intend the output alphabet to be $B = \{0, 1\}$, the only requirement is that the output alphabet contains at least two symbols.

4.2.1 Shannon–Fano Coding

Shannon and Fano (see [4]) separately introduced essentially the same algorithm to compute an efficient codebook given an arbitrary distribution. This algorithm is seductively simple: First, sort all the symbols in non-increasing frequency order. Then, split this list in a way that the first part's sum of frequencies is as equal as possible to the second part's sum of frequencies. This should give you two lists where the probability of any symbol being a member of either list is as close as possible to one-half. When the split is done, prefix all the codes of the symbols in the first list with 0 and all the codes of the symbols of the second list with 1. Repeat recursively this procedure on both sublists until you get lists that contain a single symbol. At the end of this procedure, you will have a uniquely decodable codebook for the arbitrary distribution you input. It will give a codebook such as that shown in Table 4.1a.

There are problems with this algorithm. One problem is that it is not always possible to be sure (because we proceed greedily) how to split the list. Splitting the list in a way that minimizes the difference of probability between the two sublists does not always yield the optimal code! An example of a flawed codebook is given in Table 4.1b. While this algorithm is conceptually attractive because it is simple, it sometimes results in codebooks that are much worse than those given by Huffman's algorithm for the same distribution.

4.2.2 Building Huffman Codes

Huffman in his landmark 1952 paper [3] gives the procedure to build optimal variable-length codes given an arbitrary frequency distribution for a finite alphabet. The procedure is built upon a few conditions that, if satisfied, make the codebook optimal. The first conditions are as follows:

(a) No code is a prefix of another code.
(b) No auxiliary information is needed as delimiter between codes.

Table 4.1 Examples of Shannon–Fano Codes for Eight Symbols

(a) Symbol	Frequency	Code	(b) Symbol	Frequency	Code
a	38,416	00	a	34,225	000
b	32,761	01	b	28,224	001
c	26,896	100	c	27,889	01
d	14,400	101	d	16,900	100
e	11,881	1100	e	14,161	101
f	6,724	1101	f	4,624	110
g	4,225	1110	g	2,025	1110
h	2,705	1111	h	324	1111

Note: (a) A codebook with codes of average length 2.67. The entropy of this distribution is 2.59 bits/symbol. Discrepancy is only 0.08 bits/symbol. (b) An example of a Shannon–Fano codebook for eight symbols exhibiting the problem resulting from greedy cutting. The average code length is 2.8, while the entropy of this distribution is 2.5 bits/symbol. Here, discrepancy is 0.3 bits/symbol. This is much worse than the discrepancy of the codes shown in (a).

The first requirement implies that, for a series of bits, there is a unique way to decode it. This leaves no ambiguity as to the encoded message. The second requirement tells us that we do not need markers, or special symbols (such as the "space" symbol in the seemingly binary Morse code),[1] to mark the beginning or the end of a code.

One way to satisfy both requirements is to build a *tree-structured codebook.* If the codebook is tree-structured, then there exists a full binary tree (a binary tree is *full* when each node either is a leaf or has exactly two children) with all the symbols in the leaves. The path from the root to a leaf is the code for that leaf. The code is obtained by walking down the path from the root to the leaf and appending a 0 to the code when we go down to the left or a 1 when we go down to the right. Reaching a leaf naturally delimits the code, thus satisfying the second requirement. Were we allowed to address internal nodes, there would be codes that are prefixes of other codes, violating requirement (a). The first requirement is therefore met because no codes are assigned to internal nodes; codes designate only leaves.

Let $A = \{a_1, a_2, \ldots, a_n\}$ be our alphabet for which we want a code. We rearrange the alphabet A so that symbols are listed in order of non-increasing frequency. This convention prevents us from using a permutation function $\pi(i)$ that gives the index in A of the symbol that occupies rank i. This would needlessly complicate the notation. This rearranged alphabet satisfies

$$P(a_1) \geq P(a_2) \geq \cdots \geq P(a_n), \tag{4.1}$$

where $P(a)$ is a shorthand for $P(X = a)$, and X is our random source emitting the symbols. For this alphabet, we want to build an optimal set of codes with lengths

$$L(a_1) \leq L(a_2) \leq \cdots \leq L(a_n). \tag{4.2}$$

It turns out that we will have to slightly modify Eq. (4.2). Let L_n be the length of the longest code. Let us suppose that there are more than two codes of length L_n that do not share prefixes of length $L_n - 1$ among themselves. Since by hypothesis we already have an optimal code, there are no other codes that are prefixes of the codes of length L_n. Since the codes of length L_n share no prefix between them and there are no other codes that are prefixes to any of the length L_n codes, it

[1] In Morse code, one uses strings of dots (·) and dashes (—) to encode letters, but the codes are separated by spaces. The duration of a dot or a space is a third of that of a dash.

means that these codes of length L_n have at least one extra bit, making them too long. Therefore, L_n should be $L_n - 1$, a contradiction! What this means is that we can drop the last bit on all these length L_n codes. This in turn implies that at least two of the longest codes of length L_{max} must share a prefix of length $L_{max} - 1$. The same kind of argument will prevent an algorithm that builds an optimal codebook from giving extra, useless bits to codes. Equation (4.2) becomes

$$L(a_1) \leq L(a_2) \leq \cdots \leq L(a_{n-1}) = L(a_n), \tag{4.3}$$

where the last $\leq$ is now $=$. We now have the following conditions to satisfy:

(a) No code is a prefix of another code.
(b) No auxiliary information is needed as delimiter between codes.
(c) $L(a_1) \leq L(a_2) \leq \cdots \leq L(a_{n-1}) = L(a_n)$.
(d) Exactly two of the codes are of length L_{max} and are identical except for their last bit.
(e) Every possible code of lengths $L_{max} - 1$ is either already used or has one of its prefixes used as a code.

Surprisingly enough, these requirements will allow a simple algorithm to fulfill them. The key requirement is requirement (c). While Huffman codes generalize to any number of coding digits greater than 1, as we will see in Section 4.2.3, let us restrict ourselves for now to the case of a binary output alphabet, the usual {0, 1}.

The algorithm will proceed iteratively. The process is illustrated in Fig. 4.1. At the start, all symbols are given a tree node that is the root of its own subtree. Besides the symbol and its probability, the node contains pointers to a right and a left child. They are initialized to null,

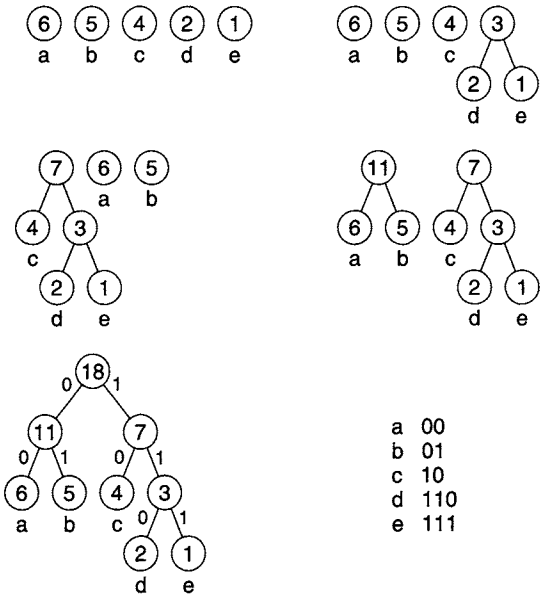

FIGURE 4.1
How Huffman's algorithm proceeds to build a code. At the beginning, all nodes are roots of their own degenerate tree of only one leaf. The algorithm merges the trees with the least probability first (note that in the figure, the *frequencies* are shown) and repeats this procedure until only one tree is left. The resulting code is $\{a \rightarrow 00, b \rightarrow 01, c \rightarrow 10, d \rightarrow 110, e \rightarrow 111\}$, for an average length of 2.17 (while the entropy is 2.11 bits/symbol, a discrepancy of 0.06 bits/symbol).

symbolized here by $\perp$. All the roots are put in a list L. Requirement (c) asks for the two symbols with lowest probability to have codes of the same length. We remove the two roots having the smallest probabilities from L; let them be a and b. We create a new root c having probability $P(a) + P(b)$ and having children a and b. We add c to L. This will cause a and b to share a common prefix, the code for c. We find ourselves with one less tree in L. If we repeat this until only one tree is left in L, we will have completed the tree-structured code satisfying *all* the requirements. The algorithm in pseudo-code looks like this:

$L = \{(a_1, P(a_1), \perp, \perp), (a_2, P(a_2), \perp, \perp), \ldots, (a_n, P(a_n), \perp, \perp)\}$
While $|L| > 1$
{
  $a = \min_P L$
  $L = L - \{a\}$
  $b = \min_P L$
  $L = L - \{b\}$
  $c = (\perp, P(a) + P(b), b, a)$
  $L = L \cup \{c\}$
}.

From there, one has the tree that describes the codebook, but the codes per se are not assigned yet. We know, however, how to assign the codes, as we already have described the procedure earlier. The codes are obtained by walking down the path from the root to the leaves and appending a 0 if we go down to the left or a 1 if we go down to the right.[2] Reaching a leaf naturally determines the end of the code. We only have to copy the code that has been accumulated as a bit string in an array indexed by the symbol in the leaf we reached.

The encoding process is straightforward. We emit the bit string contained in the table, at the address indexed by the symbol. Decoding is just a bit more complicated. Since we do not know beforehand the length of the code that we are about to read, we will have to walk the tree as we read bits one by one until we reach a leaf which will correspond to the decoded symbol. This general procedure can be improved in many ways, as we will see in Section 4.5. We will also see in a subsequent section that not only is the procedure given by Huffman simple, it also generates optimally efficient codes.

4.2.3 *N*-ary Huffman Codes

Although binary coding is the most prevalent form of coding, there are cases where more than two coding symbols are preferable. In the context of electromechanical devices, one often has a channel that allows for more than two voltage values: It would be a waste not to use this possibility to our advantage! Communications can be made a lot more efficient if we have $m \gg 2$ coding symbols at our disposal. The first thing that comes to mind is simply to generalize requirement (c) by picking not only the two nodes with the smallest probability but the m nodes with the smallest probability. While this seems like a good idea, we soon encounter a problem. Let us say that we have $m = 3$ coding symbols and a source alphabet size of $n = 6$. We apply the algorithm as we described earlier, but rather than picking the two symbols with the smallest probability of occurrence, we pick three. After the second iteration, we find ourselves with only two nodes left! We cannot just pick the three less probable subtrees.

[2] Note that this is arbitrary. We could as well choose a 1 when going down to the left and a 0 when going down to the right.

Fortunately, there is a rather simple fix. The first time, instead of picking m nodes, pick $2 \leq m' \leq m$ nodes. In order to make sure that all subsequent steps merge m nodes, we first compute $a = n \bmod (m-1)$ and then find $m' \equiv a \pmod{m-1}$. As $2 \leq m' \leq m$, we test the different values of m' until we find one that satisfies $m' \equiv a \pmod{m-1}$.

For example, for $n = 6, m = 3$ we find $m' = 2$. For $n = 7, m = 3, m' = 3$. For $n = 8, m = 3$, we find $m' = 2$ again. The generalized algorithm will be based on generalized requirements (c) and (d), which now read

(c′) At least two, and not more than m, codes will be of length L_{max}.

(d′) At least two, and at most m, codes of length L_{max} are identical except for the last m-ary digit.

4.2.4 Canonical Huffman Coding

The reader may have noticed that the codes assigned to symbols by Huffman's algorithm described in Section 4.2.2 are in numerical order. It generates a code such that if $P(a_i) \leq P(a_j)$, then the code for a_i numerically precedes the code for a_j. The algorithm systematically assigns the right child to the subtree of least probability and the left child to the other. If we use the convention that going down to the left means 0 and down to the right means 1, we get a *canonical codebook*, where $code(a_i) < code(a_j)$ iff $P(a_i) \leq P(a_j)$.

The use of canonical codes is not self-evident, but it turns out that fast decoding algorithms, as we will see in Section 4.5, can exploit this property. If the codes are assigned somewhat randomly, there is no easy way to know the length of the current code before the end of decoding. For example, take the code obtained in Fig. 4.1. From the code {00, 01, 10, 110, 111}, we can see that we can read 2 bits to find out if the code represents a, b, or c or if we must read another bit to find whether it is d or e. We cut the number of bit extractions and comparison by $\frac{1}{3}$. This may not seem that spectacular because in fact we skipped only 1 bit, but in larger codebooks, we tend to get larger numbers of codes of the same length, thus leading to possibly greater savings in decoding time.

4.2.5 Performance of Huffman Codes

Let us now derive the bounds on the efficiency of Huffman codes. We will first consider the average code length of Huffman codes for some source X,

$$L(X) = \sum_{a_i \in A} P(a_i) L(a_i), \tag{4.4}$$

with $P(a_i)$ being shorthand for $P(X = a_i)$. We know, since the codes are uniquely decodable, that they must satisfy the Kraft–McMillan inequality; that is,

$$\sum_{a_i \in A} 2^{-L(a_i)} \leq 1. \tag{4.5}$$

Knowing this, we want to show that

$$\mathcal{H}(x) \leq L(X) < \mathcal{H}(X) + 1, \tag{4.6}$$

where $\mathcal{H}(X)$ is the entropy for source X. The entropy of a random source X is defined as

$$\mathcal{H}(X) = -\sum_{a_i \in A} P(a_i) \lg P(a_i),$$

where $\lg(x)$ is $\log_2(x)$. We will prove the bounds in two parts. The first part will be that, if the codes are uniquely decodable, $L(X) \geq \mathcal{H}(X)$. We will then prove that given a Huffman codebook, $L(X) < \mathcal{H}(X) + 1$.

The difference between entropy and average code length is

$$\begin{aligned}\mathcal{H}(X) - L(X) &= -\sum_{a_i \in A} P(a_i) \lg P(a_i) - \sum_{a_i \in A} P(a_i) L(a_i) \\ &= \sum_{a_i \in A} P(a_i) \left(-\lg P(a_i) - L(a_i)\right) \\ &= \sum_{a_i \in A} P(a_i) \left(-\lg P(a_i) + \lg 2^{-L(a_i)}\right) \\ &= \sum_{a_i \in A} P(a_i) \lg \frac{2^{-L(a_i)}}{P(a_i)} \\ &\leq \lg\left(\sum_{a_i \in A} 2^{-L(a_i)}\right) \leq 0.\end{aligned} \tag{4.7}$$

This last inequality is an application of Jensen's inequality. It states that if a function is concave (convex down, often written convex $\cap$), then $\mathrm{E}[f(X)] \leq f(\mathrm{E}[X])$. $\mathrm{E}[X]$ is the expectation of random variable X, $\lg(\cdot)$ is a convex $\cap$ function, and the part of Eq. (4.7) before the $\leq$ part is exactly that: the expectation of the number of bits wasted by the code. If $\mathcal{H}(X) - L(X) \leq 0$, it means that $L(X) \geq \mathcal{H}(X)$. The length is therefore never shorter than the entropy, as one could have guessed from the fact that a Huffman codebook satisfies the Kraft–McMillan inequality.

To show that the upper bound holds, we will show that an optimal code is always less than 1 bit longer than the entropy. We would like to have codes that are exactly $-\lg P(a_i)$ bits long, but this would ask for codes to have any length, not only integer lengths. The Huffman procedure obviously produces codes that have an integer number of bits. Indeed, the length of the code for symbol a_i is

$$L(a_i) = \lceil -\lg P(a_i) \rceil,$$

where $\lceil x \rceil$ is the ceiling of x, the smallest integer greater than or equal to x. We can express $\lceil x \rceil$ as $\lceil x \rceil = x + \varepsilon(x)$. We have

$$-\lg P(a_i) \leq L(a_i) < 1 - \lg P(a_i). \tag{4.8}$$

The strict inequality holds because $\lceil x \rceil - x = \varepsilon(x) < 1$. It also means that $2^{-L(a_i)} \leq P(a_i)$, which gives

$$\sum_{a_i \in A} 2^{-L(a_i)} \leq \sum_{a_i \in A} P(a_i) = 1.$$

This means that a codebook with code lengths $L(a_i)$ satisfies the Kraft–McMillan inequality. This implies that *there must exist* a code with these lengths! Using the right inequality in Eq. (4.8), we finally get

$$L(X) = \sum_{a_i \in A} P(a_i) L(a_i) < \sum_{a_i \in A} P(a_i)(-\lg P(a_i) + 1) = \mathcal{H}(X) + 1,$$

which concludes the proof on the bounds $\mathcal{H}(X) \leq L(X) < \mathcal{H}(X) + 1$. This means that a codebook generated by Huffman's procedure always has an average code length less than 1 bit longer than what entropy would ask for. This is not as tight a bound as it may seem. There are proofs, assuming something about the distribution, that give much tighter bounds on code lengths. For example, Gallager [5] shows that if the distribution is dominated by $p_{max} = \max_{a_i \in A} P(a_i)$, the expected code length is upper-bounded by $\mathcal{H}(X) + p_{max} + \sigma$, where $\sigma = 1 - \lg e + \lg \lg e \approx 0.0860713320\ldots$. This result is obtained by the analysis of internal node probabilities defined

as the sum of the probabilities of the leaves it supports. Capocelli *et al.* [6] give bounds for a code where $\frac{2}{9} \leq p_{max} \leq \frac{4}{10}$ with no other information on the other p_i, as well as bounds for $\frac{1}{3} \leq p_{max} \leq \frac{4}{10}$ and for distributions for which only p_{max} and p_{min} are known. Buro [7], under the assumption that $P(a_i) \neq 0$ for all symbols and that Eq. (4.1) holds, characterizes the maximum expected length in terms of ϕ, the golden ratio!

Digression 4.1. Although we have shown that the average length of Huffman codes are generally quite close to the entropy of a given random source X, we have not seen how long an individual code can get. We can see that an alphabet with $P(a_i) = 2^{-\min(i,n-1)}$ will produce a code of length $L(a_i) = \min(i, n-1)$. Campos [8] pointed out another way of producing maximally long codes. Suppose that for n symbols, we have the probability function

$$P(X = a_i) = \frac{F_i}{F_{n+2} - 1},$$

where F_i is the ith Fibonacci number. The sum of the first n Fibonacci numbers is

$$\sum_{i=1}^{n} F_i = F_{n+2} - 1.$$

This set of probabilities also generates a Huffman code with a maximum code length of $n-1$. To avoid ludicrously long codes, one could consider limiting the maximum length of the codes generated by Huffman's algorithm. There are indeed papers where such algorithms are presented, as we will see in Section 4.3.4.

Katona and Nemetz [9] show that using Fibonacci-like probabilities causes the Huffman algorithm to produce not only maximally long codes but also the largest possible differences between assigned code lengths and what would be strictly needed according to the entropy. They use the notion of self-information, $I(a)$, defined as $I(a) = -\lg P(X = a)$, to show that the ratio of the produced code length to the self-information is $1/\lg \phi = 1.44\ldots$. Here, $\phi = \frac{1}{2}(1 + \sqrt{5})$.

Digression 4.2. We noted earlier that the choice of assigning a 0 when going down to the left and a 1 when going down to the right in the tree was arbitrary but led to canonical codebooks. If we did not care about the canonicity of the codebook, we could as well reverse the use of 0 and 1. This leads to the observation that for a given tree, multiple codebooks are possible. Suppose, for example, that each time a branching occurs, we decide randomly on which branch is designated by 0 or 1. This would still result in a valid Huffman code tree. Recall that a Huffman tree is *full*, and this property ensures that if there are n leaves (one for each of the symbols in the alphabet), then there are $n-1$ internal nodes. Since each internal node has the choice on how to assign 0 or 1 to its children, it is the same as saying that each internal node has two possible states: ($0 \rightarrow$ left, $1 \rightarrow$ right) or ($0 \rightarrow$ right, $1 \rightarrow$ left). Because each of the $n-1$ nodes has two possible states, there are 2^{n-1} possible different equivalent Huffman code trees, all optimal.

4.3 VARIATIONS ON A THEME

Because Huffman codebooks are very efficient, it is not surprising that they would appear in a number of places and in numerous guises. They do not, however, satisfy everybody as is. Some will need very large sets of symbols, and others will want them to be length constrained. There are also cases where basic Huffman codes are not as efficient as we may hope. In this section, we will consider a few common modifications to Huffman's algorithm to produce slightly different codes.

We will get a look at modified Huffman codes, Huffman prefixed codes, and extended Huffman codes.

4.3.1 Modified Huffman Codes

In facsimile (a.k.a. fax) transmission, the document to be transmitted is scanned into lines of 1728 pixels, for up to 3912 lines. Each line of the document image is translated into runs of white and black pixels, which are coded, in accordance with CCITT Group 3 recommendations, by *modified Huffman* codes. This modified Huffman scheme allocates two sets of codes, one for the white runs and one for the black runs. The reason for this is that the distribution of the number of successive white pixels in a scanned text document differs substantially from the distribution of the number of successive black pixels since there are usually black pixels only where there is text. Using a unique codebook for both distributions would be inefficient in terms of compression.

The codes are "modified Huffman" codes because instead of allocating a code for each possible run length $0 \leq l < 1728$, codes are allocated for all $0 \leq l < 63$, then for every l that is a multiple of 64, up to 2560. The codes for $0 \leq l < 63$ are termed Termination Codes because they always encode the number of pixels at the end of a run, while the codes for the runs of lengths that are multiples of 64 are termed Make-Up Codes, since they encode the body of the runs. The reader interested in the intricacies of facsimile coding is referred to the chapter on Facsimile Compression.

4.3.2 Huffman Prefixed Codes

Another way to devise a codebook for a very large number of symbols is to use probability classes and to compute the codebook for the classes only. For example, let us pretend that we have an alphabet A with a very large number of symbols, something like 2^{32}. We can partition A into equivalence classes in a way such that all symbols that have roughly the same probability end up in the same equivalence class. The idea is to create a codebook in two parts. The first part, or prefix, is a Huffman code for equivalence classes, and the second part, or suffix, is simply an index within a given equivalence class to the desired symbol.

Let us define the equivalence classes and the concept of "having roughly the same probability as" more rigorously. Two symbols a and b, will be equivalent $a \equiv_p b$ iff $\lceil -\lg P(a) \rceil = \lceil -\lg P(b) \rceil$. Let C_{p_i} be the equivalence class for probability p_i. All $a \equiv_{p_i} b$ are in class C_{p_i}. Instead of computing individual codes for each of the symbols $a_i \in A$, using $P(a_i)$, we will build the codes for the classes C_{p_i} with probabilities $P(C_{p_i}) = \sum_{a \in C_{p_i}} P(a)$.

The resulting prefix codebook for the equivalence classes satisfies Eq. (4.6). This means that the codes for the prefix part will always be less than 1 bit longer than the entropy. The suffix part is only the index of a_i into C_{p_i}. Using a natural code for the index within an equivalence class, $\lceil |C_{p_i}| \rceil$ bits suffice. The resulting code is therefore always less than 2 bits longer than the entropy. If we use a *phase-in* code [10], the length suffix of the suffix code is essentially $\lg |C_{p_i}|$ when $|C_{p_i}|$ is large enough and given that symbols within an equivalence class are chosen according to a uniform random variable. Table 4.2 shows such a code.

4.3.3 Extended Huffman Codes

If the source alphabet is rather large, p_{max} is likely to be comparatively small. On the other hand, if the source alphabet contains only a few symbols, the chances are that p_{max} is quite large compared to the other probabilities. Recall that we saw in Section 4.2.5 that the average code length is upper-bounded by $\mathcal{H}(X) + p_{max} + 0.086$. This seems to be the assurance that the code is never very bad.

Table 4.2 An Example of a Large Number of Symbols Represented by a Small Number of Probability Classes, and the Resulting Huffman Prefixed Code

Probability	Prefix	Suffix Bits	Range
$P(0 \leq X < 4) = \frac{1}{2}$	0	xx	0–3
$P(4 \leq X < 11) = \frac{1}{4}$	10	xxx	4–11
$P(12 \leq X < 27) = \frac{1}{16}$	1100	$xxxx$	12–27
$P(28 \leq X < 59) = \frac{1}{16}$	1101	$xxxxx$	28–59
$P(60 \leq X < 128) = \frac{1}{16}$	1110	$xxxxxx$	60–123
$P(124 \leq X < 251) = \frac{1}{16}$	1111	$xxxxxxx$	124–251

It is unfortunately not so. Consider the following example. Say that source alphabet is $A = \{0, 1\}$ and that output alphabet $B = \{0, 1\}$. Regardless of the probabilities, the average code length is still $\mathcal{H}(X) \leq L(X) < \mathcal{H}(X) + p_{max} + 0.086\ldots$, but *absolutely no compression is achieved*, since the resulting Huffman codebook will be $\{0 \rightarrow 0, 1 \rightarrow 1\}$.

How do we get out of this tar pit? One solution is to use extended alphabets. The extended alphabet of order m generated from A is given by

$$A^m = \underbrace{A \times A \times \cdots \times A}_{m \text{ times}} = \{a_1a_1 \ldots a_1a_1, \underbrace{a_1a_1 \ldots a_1a_2}_{m \text{ symbols}}, \ldots, a_na_n \ldots a_n\},$$

where $A \times A$ is to be interpreted as the Cartesian product of A with itself. The probability of a symbol a_i^m of A^m is simply (if one assumes that the original symbols of A are independent and identically distributed) given by

$$P(a_i^m) = \prod_{j=1}^{m} P(a_{ij}^m).$$

This results in n^m symbols and in the same number of probabilities. The resulting Huffman codebook satisfies, under the independent identically distributed (i.i.d.) hypothesis,

$$\mathcal{H}(X^m) \leq L(X^m) < \mathcal{H}(X^m) + 1$$

and that gives, for the original symbols of A, an average code length

$$\frac{1}{m}\mathcal{H}(X^m) \leq \frac{1}{m}L(X^m) < \frac{1}{m}(\mathcal{H}(X^m) + 1)$$

or, after simplification,

$$\mathcal{H}(X) \leq L(X) < \mathcal{H}(X) + \frac{1}{m}.$$

Therefore, we can correctly conclude that packing symbols lets us get as close as we want to the entropy, as long as we are willing to have a large m. The problem with a large m is that the number of symbols (and probabilities) for alphabet size n is n^m, which gets unmanageably large except for trivially small values of both n and m.

In such a situation, other coding techniques may be preferable to Huffman coding. For example, we could build a k-bit Tunstall code [11]. A Tunstall code associates variable-length strings of symbols to a set of fixed-length codes rather than associating a code to every symbol. We can build a codebook with the $2^k - n$ first codes associated with the $2^k - n$ most *a priori* likely strings, under i.i.d assumptions, and keep n codes for literals, because not all possible strings will

be present in the the dictionary. We could also consider using another algorithm altogether, one that naturally extracts repeated strings from the input. One such algorithm is Welch's variation [12] on the Ziv–Lempel dictionary-based scheme [13], known as LZW. We could also consider the use of arithmetic coding and estimate the probabilities with a small-order probabilistic model, like an order m Markov chain or a similar mechanism.

4.3.4 Length-Constrained Huffman Codes

One may want to limit in length the codes generated by Huffman's procedure. They are many possible reasons for doing so. One possible reason for limiting the length of Huffman-type codes is to prevent getting very long codes when symbol probabilities are underestimated. Often, we get approximated probability information by sampling part of the data. For rarer symbols, this may lead to symbols getting a near zero probability, despite their occurring much more frequently in the data. Another reason could be that we want to limit the number of bits required to be held in memory (or CPU register) in order to decode the next symbol from a compressed data stream. We may also wish to put an upper bound on the number of steps needed to decode a symbol. This situation arises when a compression application is severely constrained in time, for example, in multimedia or telecommunication applications, where timing is crucial.

There are several algorithms to compute length-constrained Huffman codes. Fraenkel and Klein [14] introduced an exact algorithm requiring $O(n \lg n)$ steps and $O(n)$ memory locations to compute the length-constrained code for a size n alphabet. Subsequently, Milidiú *et al.* [15] presented an approximated fast algorithm asking for $O(n)$ time and memory locations, which, much like Moffat's and Katajainen's algorithm [16], computes the code lengths *in situ* using a dynamic programming-type algorithm. Finally, Larmore and Hirschberg [17] introduced another algorithm to compute length-constrained Huffman codes. This last paper is extensive as it describes both mathematical background and implementation details.

4.4 ADAPTIVE HUFFMAN CODING

The code generated by the basic Huffman coding algorithm is called the *static Huffman Code*. The static Huffman code is generated in two steps. The first step is the gathering of probability information from the data. This may be done either by exhaustively scanning the data or by sampling some of it. If we opt for exhaustively scanning the data, we may have to deal with large quantities of data, if it is even possible to scan the complete data set at all. In telecommunications, "all the data" does not mean much because the data may not yet be available as it arrives in small packets from a communication channel. If we opt for sampling some of the data, we expose ourselves to the possibility of very bad estimates of the probabilities, yielding an inefficient code.

Fortunately, there are a number of ways to deal with this problem. The solutions bear the common name *dynamic* or *adaptive Huffman coding*. In dynamic Huffman coding, we will update the codes as we get better estimates of the probability, either locally or globally. In this section, we will get a look at the various schemes that have been proposed to achieve adaptive compression.

4.4.1 Brute Force Adaptive Huffman

One naive way to do adaptive Huffman coding is the brute force approach. One could recompute the tree each time the probabilities are updated. As even an efficient implementation (such as, for example, Moffat's and Katajainen's astute method [16], Section 4.5) asks for at least $O(n)$ steps, for alphabet size n, it seems impractical to update the codes after each symbol.

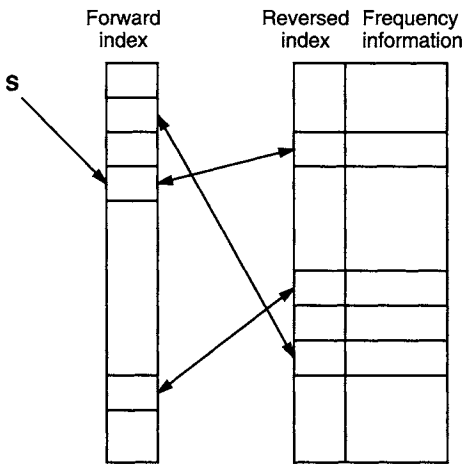

FIGURE 4.2
A simple structure to keep symbol rank information. The forward index uses the symbol directly to find where in the rank table the frequency information is stored. The reversed index is used to update the forward index when the symbol rank changes up or down.

What if we compute the new codes every now and then? One could easily imagine a setting where we recompute the codes only after every k symbols have been observed. If we update the codes only every 1000 symbols, we divide the cost of this algorithm by 1000, but we pay in terms of compression efficiency. Another option would be to compute the code only when we determine that the codes are significantly out of balance. Such would be the case when a symbol's rank in terms of frequency of occurrence changes.

This is most easily done with a simple ranking data structure, described in Fig. 4.2. A forward index uses the symbol as the key and points at an entry in the table of frequencies. Initially, all the frequencies are set to 1 (zero could lead to problems). Each time a symbol is observed, we update its frequency count, stored in the memory location pointed to by the forward index. The frequency is then bubbled up so as to maintain the array sorted in non-increasing frequency order. As in bubble sort, we swap the current frequency count with the previous frequency count while it is larger. We also update the forward index by swapping the forward index pointers (we know where they are by the reversed index). This procedure is expected $O(1)$ amortized times since it is unlikely, at least after a good number of observations, that a symbol must be bubbled up from the bottom of the array to the top. However, it can be $O(n)$ in the beginning when all symbols have a frequency count of 1. After, and if, a swap occurred in the table we recompute the codes using any implementation of Huffman's procedure. Experimental evidence shows that this algorithms performs slightly better than Vitter's algorithm (which we will see in Section 4.4.3) and that the number of tree recomputations is low for text files. Experiments on the Calgary Corpus text files show that this algorithm recomputes the tree for about 2–5% of the symbols, giving an acceptable run time even for large files.

This algorithm is not quite as sophisticated as the algorithms we will present in the next few subsections. Indeed, it does a lot more work than seems necessary. With this simple algorithm, we recalculate the whole codebook even when only a few symbols exchange rank. Moreover, exchanging rank does not mean that the symbols also have new code lengths or even that they must change codes.

4.4.2 The Faller, Gallager, and Knuth (FGK) Algorithm

Faller [18] and Gallager [5] independently introduced essentially the same algorithm, a few years apart. It was later improved upon by Cormack and Horspool [19]. The same improvements appeared again 1 year later, this time proposed by Knuth [20]. The adaptive algorithm proposed by Gallager requires some introductory material before it is introduced.

A tree has the *sibling property* if every internal node other than the root has a sibling and if all nodes can be enumerated in nondecreasing probability order. In such a tree, there are exactly $2n - 2$ nodes. A useful theorem introduced by Gallager states that a full tree is a Huffman tree if, and only if, it has the sibling property. Let us define the depth of a node as the number of edges that separate it from the root. The root has a depth of 0. Internal node probabilities are defined as the sum of the probabilities of the leaves that they support.

We will maintain a tree such that all nodes at a given depth d have a probability that is less than or equal to all the probabilities of nodes at depth $d - 1$ and above. A tree that has this property for all depths is said to be ordered if the 0-bit goes to the most likely of the two children for each node. Furthermore, a tree is said to be *lexicographically ordered* if, for any depth $d \geq 1$, all the nodes at depth d have smaller probabilities than nodes at depth $d - 1$, and all nodes at depth d can be enumerated in non-decreasing order, from left to right, in the tree. Finally, a binary prefix code tree is a Huffman tree if, and only if, it is lexicographically ordered.

The update procedure proposed by Gallager requires that the tree remain lexicographically ordered at all times. We will not discuss how the data structures are used to maintain efficient access to data within the tree, as Knuth's paper [20] describes in a complete way a very efficient set of data structures and subalgorithms to update the tree.

When a new symbol is observed, its old code (here also obtained by walking down the tree from the root to the leaf that contains this symbol, exactly as in the basic Huffman algorithm) is transmitted, so that the decoder remains synchronized with the encoder. After the code is emitted, the update procedure begins. The frequency count associated with the leaf is incremented by 1. Let us say that this leaf, a, is at depth d. After its counts have been incremented, we check if all the nodes at depth d are still in nondecreasing order. If not, we exchange this leaf with the rightmost node with a count lower than a's count. Let this node be b. Node b does not have to be at depth d; it can be at depth $d - 1$ or above. After the exchange, we update b's parent frequency count recursively up to the root. Since b's frequency is less than or equal to a's frequency before the update, no other exchange will be provoked by the recursive frequency updates. On the other hand, while we recursively update a's ancestors up to the root, we may have to do more exchanges. Figures 4.3 and 4.4 illustrate the process. The initial state of the tree is adapted from the example given by Gallager in [5].

We have seen that this algorithm takes only positive unit increments on the frequency count and that it does not deal explicitly with symbols with zero frequency. The problem with only being able to increment the frequency is that the algorithm captures no local trend in the data; it only converges slowly on the long-term characteristics of the distribution. Gallager proposes a "frequency decay" method where, every N symbols, all frequencies are multiplied by some constant $0 < \alpha \leq 1$ to keep them relatively small so that an increment of 1 has a good chance to provoke an update in the tree. There are problems with this method, such as, do we round or truncate the result after multiplication with α? Another problem is how to choose N and α in an adaptive way.

Cormack and Horspool proposed an algorithm to update the frequency by any positive amount. This procedure was generalized by Knuth to allow negative integers as well. The advantage with negative integer increments is that we can keep a size M window over the source to compress, incrementing the frequencies according to the symbols that are coming in and decrementing them according to the symbols that are leaving the window. This method provides a better mechanism to

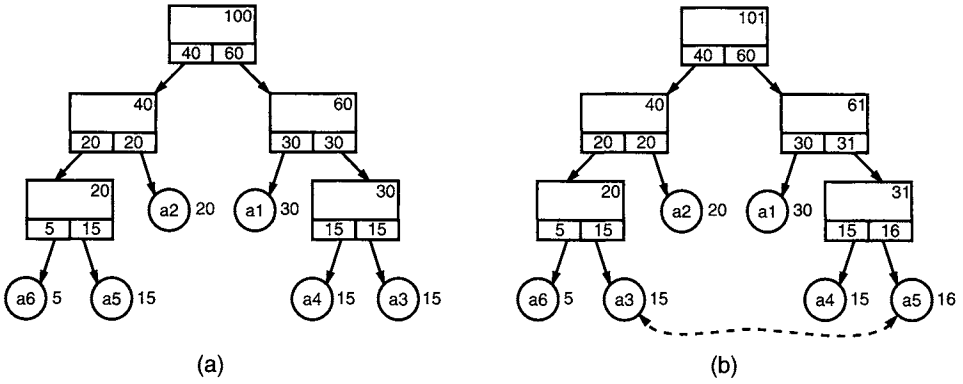

FIGURE 4.3
A single update. (a) The tree before the update; (b) the tree after the observation of symbol a_5. The dotted arrow shows where the exchanges occurred.

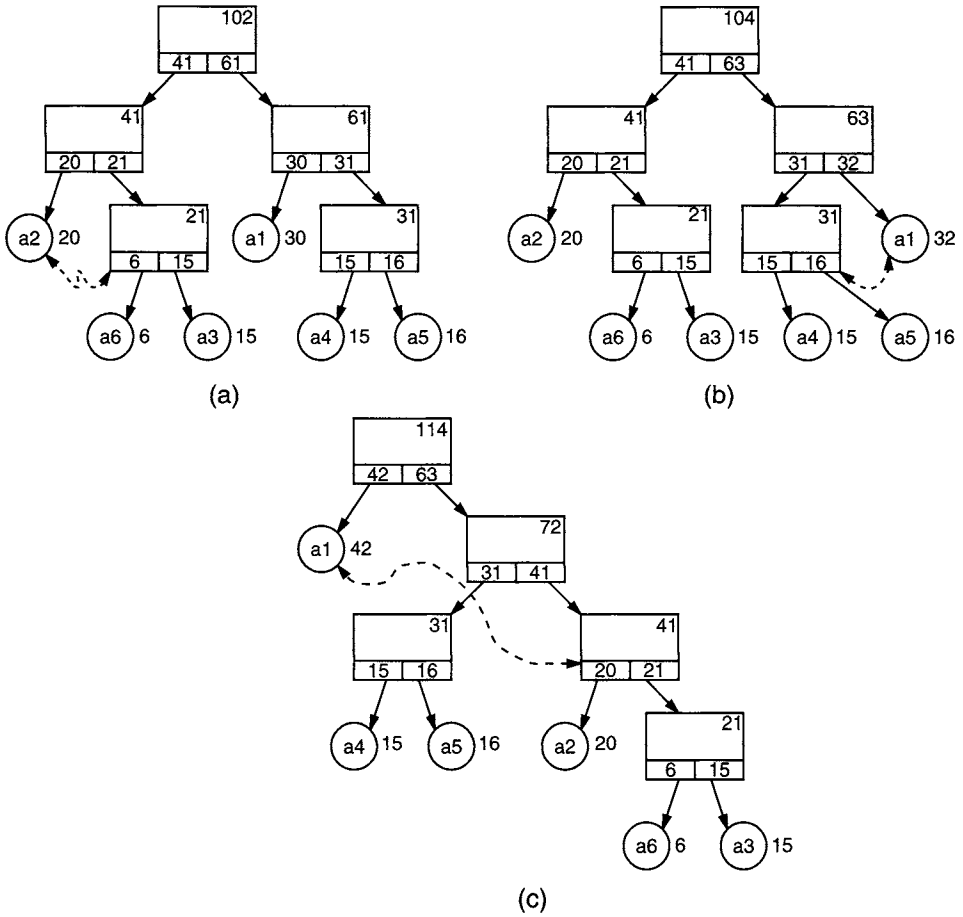

FIGURE 4.4
After multiple updates. (a) The tree after the observation of symbol a_6. (b) The tree after 2 observations of symbol a_1. At this point, symbol a_1 is as far right as it can get on this level, at depth 2. If it is updated again, it will climb to the previous level, depth 1. (c) The updated tree after 10 observations of a_1. Note that a complete subtree has been exchanged, as is shown by the dotted arrow.

capture non-stationarity in the source. The hypothesis of stationarity may make sense for certain sources, but not for all. There are many cases where statistics clearly change according to the source's history.

Gallager's algorithm starts with all symbols as leaves and the tree is originally *complete*. A tree is said to be complete if all leaves lie on at most two different depths. If n, the alphabet size, is of the form $n = 2^k$, then all leaves are at depth k. Knuth further modified Gallager's algorithm to include an escape code used to introduce symbols that have not been observed yet. Since it is possible that for a given source, only a subset of the symbols will be generated for a given message, it is wasteful to allocate the complete tree as soon as compression starts. Knuths idea is the following. The tree starts as a single leaf, the special zero-frequency symbol. This leaf will be the special escape code. When a new symbol is met, we first emit the code for the special leaf and then emit the new symbol's number among the remaining unobserved symbols, from 0 to $n-r-1$, if r symbols have been observed so far. This asks for $\lceil \lg(n-r) \rceil$ bits. We create the new symbol's leaf as the sibling of the zero-frequency leaf, creating a parent where the zero-frequency leaf was. This maintains exactly 1 zero-frequency leaf at all times. When the last unobserved symbol is encountered, the special zero-frequency leaf is simply replaced by this symbol, and the algorithm continues as Gallager described. The obvious advantage of using this technique is that the codes are very short right from the start.

Knuth gives the complexity of the update and "downdate" (negative increment) procedures as being $O(-\lg P(X = a_i))$ given we observe symbol a_i. This gives a total encoding complexity, for a sequence of length m, of $O(m\mathcal{H}(X))$, since the expected value of $-\lg P(X = a_i)$ is $\mathcal{H}(X)$. This algorithm is suitable for adaptive Huffman coding even for low-computational-power environments. The memory requirement is $O((2n-1)c)$, where c is the cost (in bytes or bits) of storing the information of a node in memory.

4.4.3 Vitter's Algorithm: Algorithm Λ

Vitter's algorithm Λ [21, 22] is a variation of algorithm FGK. The main difference between algorithm FGK and Vitter's algorithm Λ is the way the nodes are numbered. Algorithm Λ orders the nodes the same way as algorithm FGK, but also asks for leaves of a given weight to always precede internal nodes of the same weight and depth, while in algorithm FGK it was sufficient to order weights of nodes at a given depth from left to right, with no distinction of node type. This modification (as shown in [21]) is enough to guarantee that algorithm Λ encodes a sequence of length s with no more than s bits more than with the static Huffman algorithm, without the expense of transmitting the probability (or code) table to the decoder. In [22], Vitter gives a complete pseudo-Pascal implementation of his algorithm, where most of the data structures introduced in [20] are present. The code is clear enough that we should not have any major problem translating it to our favorite programming language.

4.4.4 Other Adaptive Huffman Coding Algorithms

Jones introduced an algorithm using splay trees [23]. A splay tree is a data structure where recently accessed nodes are promoted to a position near the root of the tree [24]. The original splay tree is in fact an ordinary search tree, with information in every node, and not necessarily full. In the case of a Huffman-type code tree, the symbols are only in the leaves, and the internal nodes contain no symbol information. Jones presents an algorithm that modifies the splay tree algorithm of Tarjan enough so that splaying maintains a valid Huffman tree. Splaying is also reduced. Instead of promoting a symbol directly to be one of the root's children, a leaf is allowed to climb only two

levels at a time, and depth is reduced by a factor of 2. This algorithm performs significantly worse than the other adaptive Huffman algorithms in most cases. In some cases the message contains long series of repeating symbols. In these cases the repeated symbol climbs rapidly and quickly gets the shortest code, thus achieving potentially better compression than the other algorithms on run-rich data. The splaying provides Jones' algorithm with what we may call short-term memory.

Pigeon and Bengio introduced algorithm M where splaying is controlled by a weight rule [25–27]. A node splays up one level only if it is as heavy as its parent's sibling. While algorithm Λ almost always beats algorithm M for small alphabets ($n = 256$), its results are within $\approx 5\%$ of those of algorithm Λ on the Calgary Corpus. The situation is reversed when we consider very large alphabets, say $n = 2^{16}$: Algorithm M beats algorithm Λ by $\approx 5\%$ on the Calgary Corpus files treated as being composed of 16-bit symbols. This suggests the possible usefulness of algorithm M in the compression of Unicode files. Another original feature of algorithm M is that not all symbols get a leaf. The leaves represent *sets* of symbols. A set is composed of all the symbols of the same frequency. The codes are therefore given to frequency classes rather than to individual symbols. The complete code is composed of a prefix, the Huffman-like code of the frequency class, and of a suffix, the number of the symbol within this frequency class. The suffix is simply the natural code for the index. The fact that many symbols share the same leaf also cuts down on memory usage. The algorithm produces codes that are never more than 2 bits longer than the entropy, although in the average case, the average length is very close to the average length produced by the static Huffman algorithm.

4.4.5 An Observation on Adaptive Algorithms

McIntyre and Pechura [28] advocate the use of static Huffman coding for small files or short strings, since the adaptation process needs a relatively large number of symbols to be efficient. Furthermore, since the number of observations for each symbol is small, it is unlikely that an adaptive algorithm can produce an efficient codebook. This situation arises when the number of symbols to be encoded is large but still relatively small compared to the alphabet size. Suppose we want an adaptive code for integers smaller than 2^{16}. One would expect to have to observe many times the number of symbols in the alphabet before getting a good code. If, on the contrary, the number of symbols to be encoded is small, say a few tens, then this code will do much worse than a static, preoptimized, code. We will then have to consider using an alternative, such as an adaptive Huffman prefix code, much like that discussed in Section 4.3.2, but where the codes for the equivalence classes are updated by an adaptive algorithm such as algorithm M.

4.5 EFFICIENT IMPLEMENTATIONS

The work on efficient implementations of Huffman **cod**er/**dec**oder pairs, or codecs, takes essentially two directions, which are almost mutually exclusive: memory efficient or speed efficient. Memory-efficient algorithms vie to save as much memory as possible in order to have a Huffman codec running with very little memory. This is achieved by alternative encoding and decoding algorithms, as much as alternative data structures. Speed-efficient algorithms are concerned only with the speed of coding or decoding data with a Huffman codebook. This also relies on alternative data structures, but there is not necessarily a concern about how much memory is spent.

4.5.1 Memory-Efficient Algorithms

If we opt for an explicit tree representation to maintain the Huffman code information in memory, we have only a few choices. One is a classical binary tree, with three pointers (one for the parent, two for the children), plus possibly extra information. Another possible choice is to use heap-type storage (where the tree is stored in an array organized in a way that the parent for a node $0 \leq k$ is at $k/2$, and its children are at $2k+1$ and $2k+2$). This kind of storage is potentially sparse because while the Huffman code tree is full, it is rarely complete. Let the reader ponder the amount of memory potentially wasted by this method.

Moffat and Katajainen [16] proposed a dynamic programming algorithm to compute the lengths of the codes, given that the probabilities are sorted in non-increasing order, in $O(n)$, using $O(1)$ working space, as it overwrites the frequencies with the lengths as the algorithm progresses. This algorithm saves working space, especially all the housekeeping data that a list and tree-based algorithm would need. This leads to substantial savings when the number of symbols in the alphabet is large. In companion papers, Moffat and Turpin [29, 30] described an algorithm that allows us to use only $O(L_{max})$ memory, using a few tables that describe not the entire tree, but only the different lengths of the codes and their number. Knowing how many codes there are of a given length is enough to compute the Huffman code of a symbol at encode time. Fortunately, it also works at decode time. The total memory needed by this algorithm is $O(L_{max})$ machine words. This algorithm was also proposed by Hirschberg and Lelewer [31], and it seems that it even predates this paper!

4.5.2 Speed-Efficient Algorithms

All algorithms presented so far proceed to encoding or decoding 1 bit at a time. To speed up decoding, Choueka *et al.* [32], Siemiński [33], and Tanaka [34] introduced various data structures and algorithms, all using the concept of finite-state automata. While the use of an automaton is not explicit in all papers, the basic idea is that rather than decoding bit by bit, decoding of the input will proceed by chunks of m bits, and the internal state of the decoder will be represented as the state of an augmented automaton.

An automaton, in the sense of the theory of computation, is composed of an input alphabet A, a set of states Σ, a set $F \subseteq \Sigma$ of accepting states, an initial state i, and a transition function $T : \Sigma \times A^* \rightarrow \Sigma$. The variety of automata considered here also has an output alphabet Ω, so the transition function becomes $T : \Sigma \times A^* \rightarrow \Sigma \times \Omega^*$. The transition function takes as input the current state and a certain number of input symbols, produces a certain number of output symbols, and changes the current state to a new state. The automaton starts in state i. The automaton is allowed to stop if the current state is one of the accepting states in F.

How this applies to fast Huffman decoding is not obvious. In this case, the input alphabet A is $\{0, 1\}$ and Ω is the original message alphabet. Since we are interested in reading m bits at a time, we will have to consider all possible strings of m bits and all their prefixes and suffixes. For example, let us consider the codebook $\{a \rightarrow 0, b \rightarrow 10, c \rightarrow 11\}$ and a decode block length of $m = 2$. This gives us four possible blocks, $\{00, 01, 10, 11\}$. Block 00 can be either aa or the end of b and a; block 11 either is the end of the code of c and the beginning of b or c or is the code for c, depending on what was read before this block, what we call the context. The context is maintained by the automaton in the current state. Knowing the current state and the current input block is enough to unambiguously decode the input. The new state generated by the transition function will represent the new context, and decoding continues until we reach an accepting state and no more input symbols are to be read. Figure 4.5 shows the automaton for the simple code, and Table 4.3 shows its equivalent table form.

Table 4.3 The Table Representation of the Automaton in Fig. 4.5

State	Read	Emit	Goto
0	00	*aa*	0
	01	*a*	1
	10	*b*	0
	11	*c*	0
1	00	*ba*	0
	01	*b*	1
	10	*ca*	0
	11	*c*	1

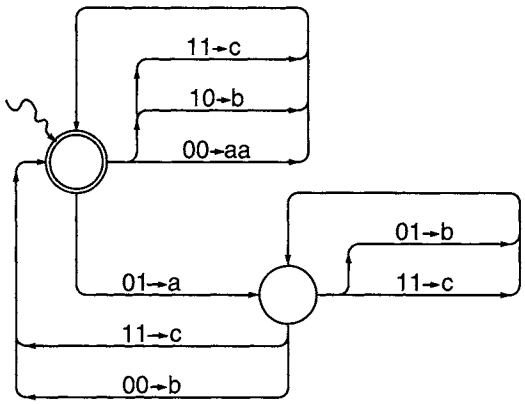

FIGURE 4.5
A fast decode automaton for the code $\{a \rightarrow 0, b \rightarrow 10, c \rightarrow 11\}$. The squiggly arrow indicates the initial state. The accepting state (where the decoding is allowed to stop) is encircled by a double line. The labels on the automaton, such as $b_1b_2 \rightarrow s$, read as "on reading b_1b_2, output string s".

As the reader may guess, the number of states and transitions grows rapidly with the number and length of codes, as well as the block size. Speed is gained at the expense of a great amount of memory. We can apply a minimizing algorithm to reduce the size of the automaton, but we still may have a great number of states and transitions. We also must consider the considerable computation needed to compute the automaton in a minimized form.

Other authors presented the idea of compressing a binary Huffman code tree to, say, a quaternary tree, effectively reducing by half the depth of the tree. This technique is due to Bassiouni and Mukherjee [35]. They also suggest using a fast decoding data structure, which is essentially an automaton by another name. Cheung *et al.* [36] proposed a fast decoding architecture based on a mixture of programmable logic, lookup tables, and software generation. The good thing about programmable logic is that it can be programmed on the fly, so the encoder/decoder can be made optimal with respect to the source. Current programmable logic technologies are not as fast as current VLSI technologies, but what they lack in speed they make up for in flexibility.

However, none of these few techniques takes into account the case where the Huffman codes are dynamic. Generating an automaton each time the code adapts is of course computationally prohibitive, even for small message alphabets. Reconfiguring programmable logic still takes a long time (it takes on the order of seconds), so one cannot consider reprogramming very often.

4.6 CONCLUSION AND FURTHER READING

The literature on Huffman coding is so abundant that an exhaustive list of papers, books, and articles is all but impossible to compile. We note a large number of papers concerned with efficient implementations, in the contexts of data communications, compressed file systems, archival, and search. For that very reason, there are a great number of topics related to Huffman coding that we could not present here, but we can encourage the reader to look for more papers and books on this rich topic.

Among the topics we have not discussed is *alphabetical coding*. An alphabetical code is a code such that if a precedes b, noted $a \prec b$, in the original alphabet, then $code(a)$ lexicographically precedes $code(b)$ when the codes are not considered as numerical values but as strings of output symbols. Note that here, lexicographically ordered does not have the same meaning as that given in Section 4.4.2, where we discussed algorithm FGK. This property allows, for example, direct comparison of compressed strings. That is, if $aa \prec ab$, then $code(aa) \prec code(ab)$. Alphabetical codes are therefore suitable for compressing individual strings in an index or other search table. The reader may want to consult the papers by Gilbert and Moore [37], by Yeung [38], and by Hu and Tucker [39], where algorithms to build minimum-redundancy alphabetical codes are presented.

Throughout this chapter, we assumed that all output symbols had an equal cost. Varn presents an algorithm where output symbols are allowed to have a cost [40]. This situation arises when, say, the communication device spends more power on some symbols than others. Whether the device uses many different, but fixed, voltage levels or encodes the signal differentially, as in the LVDS standard, there are symbols that cost more in power to emit than others. Obviously, we want to minimize both communication time and power consumption. Varn's algorithm lets us optimize a code considering the relative cost of the output symbols. This is critical for low-powered devices such as handheld PDAs, cellular phones, and the like.

In conclusion, let us point out that Huffman codes are rarely used as the sole means of compression. Many more elaborate compression algorithms use Huffman codes to compress pointers, integers, and other side information that they need to store efficiently to achieve further compression. We only have to think of compression algorithms such as the celebrated sliding window Ziv–Lempel technique that encodes both the position and the length of a match as its mean of compression. Using naive representations for these quantities leads to diminished compression, while using Huffman codes give much more satisfactory results.

4.7 REFERENCES

1. Golomb, S. W., 1966. Run length encodings. *IEEE Transactions on Information Theory*, Vol. 12, No. 3, pp. 399–401.
2. Fiala, E. R., and D. H. Greene, 1989. Data compression with finite windows. *Communications of the ACM*, Vol. 32, pp. 490–505, April 1989.
3. Huffman, D., 1952. A method for the construction of minimum redundancy codes. *Proceedings of the I.R.E.*, Vol. 40, pp. 1098–1101.
4. Fano, R. M., 1944. The Transmission of Information, Technical Report 65, Research Laboratory of Electronics, MIT, Cambridge, MA.
5. Gallager, R. G., 1978. Variations on a theme by Huffman. *IEEE Transactions on Information Theory*, Vol. 24, pp. 668–674, November 1978.
6. Capocelli, R. M., R. Giancarlo, and I. J. Taneja, 1996. Bounds on the redundancy of Huffman codes. *IEEE Transactions on Information Theory*, Vol. 32, No. 6, pp. 854–857.
7. Buro, M., 1993. On the maximum length of Huffman codes. *Information Processing Letters*, Vol. 45, pp. 219–223.

8. Campos, A. S. E., 2000. When Fibonacci and Huffman Met. Available at http://www.arturocampos.com/ac_fib_Huffman.html.
9. Katona, G. O. H., and T. O. H. Nemetz, 1976. Huffman codes and self-information. *IEEE Transactions on Information Theory*, Vol. 22, No. 3, pp. 337–340.
10. Bell, T. C., J. G. Cleary, and I. H. Witten, 1990. *Text Compression*. Prentice-Hall, New York. [QA76.9 T48B45].
11. Tunstall, B. P., 1967. *Synthesis of Noiseless Compression Codes*. Ph.D. thesis, Georgia Institute of Technology, Atlanta, GA, September 1967.
12. Welch, T. A., 1894. A technique for high performance data compression. *Computer*, pp. 8–19, June 1894.
13. Ziv, J., and A. Lempel, 1978. Compression of individual sequences via variable-rate coding. *IEEE Transactions on Information Theory*, Vol. 24, pp. 530–536, September 1978.
14. Fraenkel, A. S., and S. T. Klein, 1993. Bounding the depth of search trees. *The Computer Journal*, Vol. 36, No. 7, pp. 668–678.
15. Milidiú, R. L., A. A. Pessoa, and E. S. Laber, 1998. In-place length-restricted prefix coding. In *String Processing and Information Retrieval*, pp. 50–59.
16. Moffat, A., and J. Katajainen, 1995. In place calculation of minimum redundancy codes. In *Proceedings of the Workshop on Algorithms and Data Structures*, pp. 303–402, Springer-Verlag, Berlin/New York. [LNCS 955].
17. Larmore, L. L., and D. S. Hirschberg, 1990. A fast algorithm for optimal length-limited Huffman codes. *Journal of the ACM*, Vol. 37, pp. 464–473, July 1990.
18. Faller, N., 1973. An adaptive system for data compression. In *Records of the 7th Asilomar Conference on Circuits, Systems and Computers*, pp. 393–397.
19. Cormack, G. V., and R. N. Horspool, 1984. Algorithms for adaptive Huffman codes. *Information Processing Letters*, Vol. 18, pp. 159–165.
20. Knuth, D. E., 1983. Dynamic Huffman coding. *Journal of Algorithms*, Vol. 6, pp. 163–180.
21. Vitter, J. S., 1987. Design and analysis of dynamic Huffman codes. *Journal of the ACM*, Vol. 34, pp. 823–843, October 1987.
22. Vitter, J. S., 1989. Algorithm 673: Dynamic Huffman coding. *ACM Transactions on Mathematics Software*, Vol. 15, pp. 158–167, June 1989.
23. Jones, D. W., 1988. Application of splay trees to data compression. *Communications of the ACM*, Vol. 31, pp. 996–1007.
24. Tarjan, R. E., 1983. *Data Structures and Network Algorithms*, No. 44, Regional Conference Series in Applied Mathematics, SIAM.
25. Pigeon, S., and Y. Bengio, 1997. A Memory-Efficient Huffman Adaptive Coding Algorithm for Very Large Sets of Symbols, Technical Report 1081, Département d'Informatique et Recherche Opérationnelle, Université de Montréal, Montréal, Québec, Canada.
26. Pigeon, S., and Y. Bengio, 1997. A Memory-Efficient Huffman Adaptive Coding Algorithm for Very Large Sets of Symbols, Revisited, Technical Report 1095, Département d'Informatique et Recherche Opérationnelle, Université de Montréal, Montréal, Québec, Canada.
27. Pigeon, S., and Y. Bengio, 1998. Memory-efficient adaptive Huffman coding. *Doctor Dobb's Journal*, No. 290, pp. 131–135.
28. McIntyre, D. R., and M. A. Pechura, 1985. Data compression using static Huffman code–decode tables. *Communications of the ACM*, Vol. 28, pp. 612–616, June 1985.
29. Moffat, A., and A. Turpin, 1997. On the implementation of minimum redundancy prefix codes. *IEEE Transactions and Communications*, Vol. 45, pp. 1200–1207, October 1997.
30. Moffat, A., and A. Turpin, 1996. On the implementation of minimum redundancy prefix codes [extended abstract]. In *Proceedings of the Data Compression Conference* (M. C. James A. Storer, Ed.), pp. 170–179, IEEE Comput. Soc., Los Alamitus, CA.
31. Hirschberg, D. S., and D. A. Lelewer, 1990. Efficient decoding of prefix codes. *Communications of the ACM*, Vol. 33, No. 4, pp. 449–459.
32. Choueka, Y., S. T. Klein, and Y. Perl, 1985. Efficient variants of Huffman codes in high level languages. In *Proceedings of the 8th ACM-SIGIR Conference, Montreal*, pp. 122–130.

33. Siemiński, A., 1988. Fast decoding of the Huffman codes. *Information Processing Letters*, Vol. 26, pp. 237–241, January 1988.
34. Tanaka, H., 1987. Data structure of Huffman codes and its application to efficient coding and decoding. *IEEE Transactions on Information Theory*, Vol. 33, pp. 154–156, January 1987.
35. Bassiouni, M. A., and A. Mukherjee, 1995. Efficient decoding of compressed data. *Journal of the American Society of Information Science*, Vol. 46, No. 1, pp. 1–8.
36. Cheung, G., S. McCanne, and C. Papadimitriou, 1999. Software synthesis of variable-length code decoder using a mixture of programmed logic and table lookups. In *Proceedings of the Data Compression Conference* (M. C. James A. Storer, Ed.), pp. 121–139, IEEE Comput. Soc., Los Alanitus, CA.
37. Gilbert, E. N., and E. F. Moore, 1959. Variable length binary encodings. *Bell Systems Technical Journal*, Vol. 38, pp. 933–968.
38. Yeung, R. W., 1991. Alphabetic codes revisited. *IEEE Transactions on Information Theory*, Vol. 37, pp. 564–572, May 1991.
39. Hu, T. C., and A. C. Tucker, 1972. Optimum computer search trees and variable length alphabetic codes. *SIAM*, Vol. 22, pp. 225–234.
40. Varn, B., 1971. Optimal variable length codes (arbitrary symbol cost and equal word probability). *Information and Control*, Vol. 19, pp. 289–301.