10 Statistical Coding

The topic of this chapter is the *statistical coding* of sequences of symbols (aka *texts*) drawn from an alphabet Σ. Symbols may be characters, in this case the problem is named *text compression*, or they can be genomic-bases thus arising the Genomic-DB compression problem, or they can be bits and in this case we fall in the realm of classic data compression. If symbols are integers, then we have the Integer coding problem, addressed in the previous Chapter, which can be solved still with a statistical coder by just deriving statistical information on the integers occurring in the sequence S. In this latter case, the code we derive is an *optimal* prefix-free code for the integers of S, but its coding/decoding time is larger than the one incurred by the integer encoders of the previous Chapter, and indeed, this is the reason for their introduction.

Conceptually, statistical compression may be viewed as consisting of two phases: a *modeling* phase, followed by a *coding* phase. In the modeling phase the statistical properties of the input sequence are computed and a *model* is built. In the coding phase the model is used to compress the input sequence. In the first sections of this Chapter we will concentrate only on the second phase, whereas in the last section we will introduce a sophisticated modeling technique. We will survey the best known statistical compressors: Huffman coding, Arithmetic Coding, Range Coding, and finally Prediction by Partial Matching (PPM), thus providing a pretty complete picture of what can be done by statistical compressors. The net result will be to go from a compression performance that can be bounded in terms of 0-th order entropy, namely an entropy function depending on the probability of single symbols (which are therefore considered to occur i.i.d.), to the more precise k-th order entropy which depends on the probability of k-sized blocks of symbols and thus models the case e.g. of Markovian sources.

10.1 Huffman coding

First published in the early '50s, Huffman coding was regarded as one of the best methods for data compression for several decades, until the Arithmetic coding made higher compression rates possible at the end of '60s (see next chapter for a detailed discussion about this improved coder).

Huffman coding is based upon a *greedy algorithm* that constructs a binary tree whose leaves are the symbols in Σ, each provided with a probability $P[\sigma]$. At the beginning the tree consists only of its $|\Sigma|$ leaves, with probabilities set to the $P[\sigma]$s. These leaves constitute a so called *candidate set*, which will be kept updated during the construction of the Huffman tree. In a generic step, the Huffman algorithm selects the two nodes with the smallest probabilities from the candidate set, and creates their parent node whose probability is set equal to the sum of the probabilities of its two children. That parent node is inserted in the candidate set, while its two children are removed from it. Since each step adds one node and removes two nodes from the candidate set, the process stops after $|\Sigma| - 1$ steps, time in which the candidate set contains only the root of the tree. The Huffman tree has therefore size $t = |\Sigma| + (|\Sigma| - 1) = 2|\Sigma| - 1$.

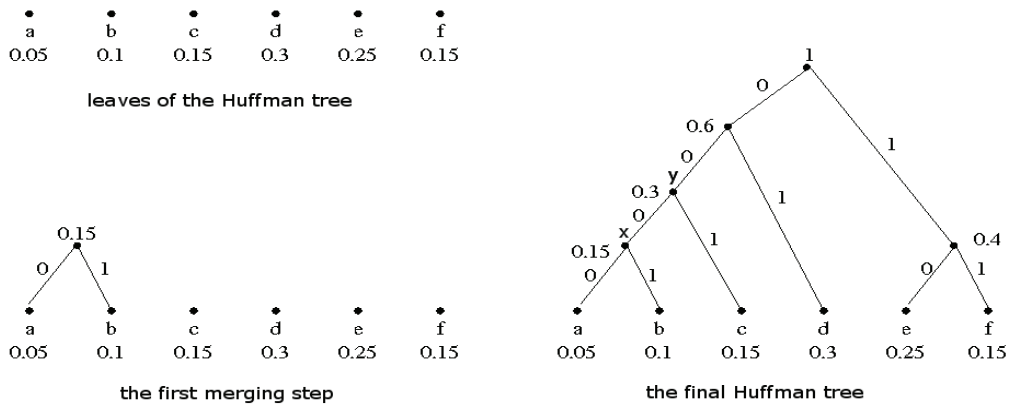


FIGURE 10.1: Constructing the Huffman tree for the alphabet $\Sigma = \{a, b, c, d, e, f\}$.

Figure 10.1 shows an example of Huffman tree for the alphabet $\Sigma = \{a, b, c, d, e, f\}$. The first merge (on the left) attaches the symbols a and b as children of the node x, whose probability is set to $0.05 + 0.1 = 0.15$. This node is added to the candidate set, whereas leaves a and b are removed from it. At the second step the two nodes with the smallest probabilities are the leaf c and the node x. Their merging updates the candidate set by deleting x and c, and by adding their parent node y whose probability is set to be $0.15 + 0.15 = 0.3$. The algorithm continues until there is left only one node (the root) with probability, of course, equal to 1.

In order to derive the Huffman code for the symbols in Σ, we assign binary labels to the tree edges. The typical labeling consists of assigning 0 to the left edge and 1 to the right edge spurring from each internal node. But this is one of the possible many choices. In fact a Huffman tree can originate $2^{|\Sigma|-1}$ labeled trees, because we have 2 labeling choices (i.e. 0-1 or 1-0) for the two edges spurring from each one of the $|\Sigma| - 1$ internal nodes. Given a labeled Huffman tree, the Huffman codeword for a symbol σ is derived by taking the binary labels encountered on the downward path that connects the root to the leaf associated to σ. This codeword has a length $L(\sigma)$ bits, which corresponds to the depth of the leaf σ in the Huffman tree. The Huffman code is *prefix-free* because every symbol is associated to a distinct leaf and thus no codeword is the prefix of another codeword.

We observe that the choice of the two nodes having minimum probability may be *not unique*, and the actual choices available may induce codes which are different in the structure but, nonetheless, they have all the same optimal average codeword length. In particular these codes may offer a *different maximum* codeword length. Minimizing this value is useful to reduce the size of the compression/decompression buffer, as well as the frequency of emitted symbols in the decoding process. Figure 10.2 provides an illustrative example of these multiple choices.

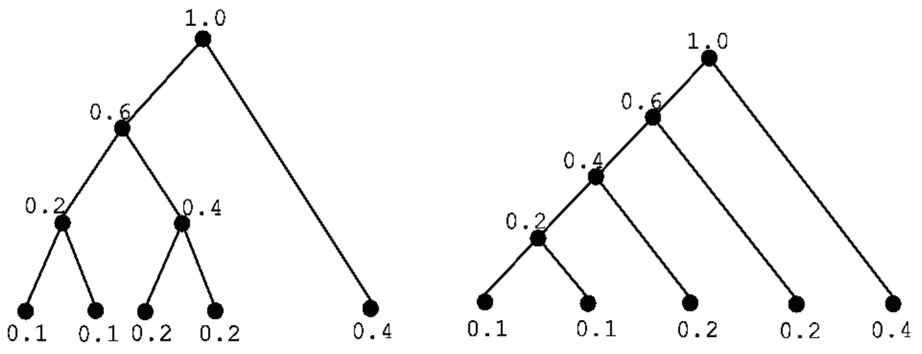


FIGURE 10.2: An example of two Huffman codes having the same average codeword length $\frac{22}{10}$, but different maximum codeword length.

A strategy to minimize the maximum codeword length is to choose the two *oldest nodes* among the ones having same probability and belonging to the current candidate set. Oldest nodes means that they are leaves or they are internal nodes that have been merged farther in the past than the other nodes in the candidate set. This strategy can be implemented by using two queues: the first one contains the symbols ordered by increasing probability, the second queue contains the internal nodes in the order they are created by the Huffman algorithm. It is not difficult to observe that the second queue is sorted by increasing probability too. In the presence of more than two minimum-probability nodes, the algorithm looks at the nodes in the first queue, after which it looks at the second queue. Figure 10.2 shows on the left the tree resulting by this algorithm and, on the right, the tree obtained by using an approach that makes an arbitrary choice.

The compressed file originated by Huffman algorithm consists of two parts: the *preamble* which contains an encoding of the Huffman tree, and thus has size $\Theta(|\Sigma|)$, and the *body* which contains the codewords of the symbols in the input sequence S. The size of the preamble is usually dropped from the evaluation of the length of the compressed file; even if this might be a significant size for large alphabets. So the alphabet size cannot be underestimated, and it must be carefully taken into account. In the rest of the section we will concentrate on the evaluation of the size in bits for the compressed body, and then turn to the efficient encoding of the Huffman tree by proposing the elegant *Canonical Huffman* version which offers space succinctness and very fast decoding speed.

Let $L_C = \sum_{\sigma\in\Sigma} L(\sigma)\, P[\sigma]$ be the average length of the codewords produced by a prefix-free code C, which encodes every symbol $\sigma \in \Sigma$ in $L(\sigma)$ bits. The following theorem states the *optimality* of Huffman coding:

THEOREM 10.1 *If C is an Huffman code, then L_C is the shortest possible average length among all prefix-free codes C', namely it is $L_C \leq L_{C'}$.*

To prove this result we first observe that a prefix-free code can be seen as a binary tree (more precisely, we should say binary trie), so the optimality of the Huffman code can be rephrased as the *minimality of the average depth* of the corresponding binary tree. This latter property can be proved by deploying the following key lemma, whose proof is left to the reader who should observe that, if the lemma does not hold, then a not minimum-probability leaf occurs at the deepest level of the binary tree; in which case it can be swapped with a minimum-probability leaf (therefore not occurring at the deepest level) and thus reduce the average depth of the resulting tree.

LEMMA 10.1 Let T be a binary tree whose average depth is minimum among the binary trees with $|\Sigma|$ leaves. Then the two leaves with minimum probabilities will be at the greatest depth of T, children of the same parent node.

Let us assume that the alphabet Σ consists of n symbols, and symbols x and y have the smallest probability. Let T_C be the binary tree generated by a code C applied onto this alphabet; and let us denote by R_C the *reduced* tree which is obtained by dropping the leaves for x and y. Thus the parent, say z, of leaves x and y is a leaf of R_C with probability $P[z] = P[x] + P[y]$. So the tree R_C is a tree with $n - 1$ leaves corresponding to the alphabet $\Sigma - \{x, y\} \cup \{z\}$ (see Figure 10.3).

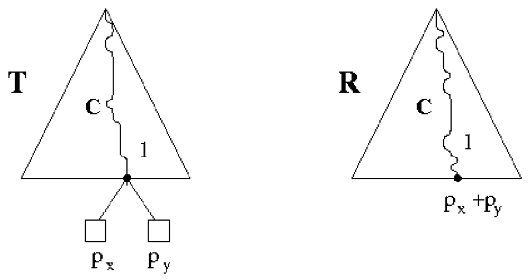


FIGURE 10.3: Relationship between a tree T and its corresponding reduced tree R.

LEMMA 10.2 The relation between the average depth of the tree T with the one of its reduced tree R is given by the formula $L_T = L_R + (P[x] + P[y])$, where x and y are the symbols having the smallest probability.

Proof It is enough to write down the equalities for L_T and L_R, by summing the length of all root-to-leaf paths multiplied by the probability of the landing leaf. So we have $L_T = \left(\sum_{\sigma \neq x,y} P[\sigma]\, L(\sigma)\right) + (P[x]+P[y])(L_T(z)+1)$, where z is the parent of x and y and thus $L_T(x) = L_T(y) = L_T(z)+1$. Similarly, we can write $L_R = \left(\sum_{\sigma \neq x,y} P[\sigma] L(\sigma)\right) + L(z)(P[x] + P[y])$. So the thesis follows. ■

The optimality of Huffman code (claimed in the previous Theorem 10.1) can now be proved by induction on the number n of symbols in Σ. The base $n = 2$ is obvious, because any prefix-free code must assign at least one bit to $|\Sigma|$'s symbols; therefore Huffman is optimal because it assigns the single bit 0 to one symbol and the single bit 1 to the other.

Let us now assume that $n > 2$ and, by induction, assume that Huffman code is optimal for an alphabet of $n - 1$ symbols. Take now $|\Sigma| = n$, and let C be an optimal code for Σ and its underlying distribution. Our goal will be to show that $L_C = L_H$, so that Huffman is optimal for n symbols too. Clearly $L_C \leq L_H$ because C was assumed to be an optimal code for Σ. Now we consider the two reduced trees, say R_C and R_H, which can be derived from T_C and T_H, respectively, by dropping the leaves x and y with the smallest probability and leaving their parent z. By Lemma 10.1 (for the optimal C) and the way Huffman works, this reduction is possible for both trees T_C and T_H. The two reduced trees define a prefix-code for an alphabet of $n - 1$ symbols; so, given the inductive hypothesis, the code defined by R_H is optimal for the "reduced" alphabet $\Sigma \cup \{z\} - \{x, y\}$. Therefore $L_{R_H} \leq L_{R_C}$ over this "reduced" alphabet. By Lemma 10.2 we can write $L_H = L_{R_H} + P[x] + P[y]$ and, according to Lemma 10.1, we can write $L_C = L_{R_C} + P[x] + P[y]$. So it turns out that $L_H \leq L_C$ which, combined with the previous (opposite) inequality due to the optimality of C, gives $L_H = L_C$. This actually means that Huffman is an optimal code also for an alphabet of n symbols, and thus inductively proves that it is an optimal code for any alphabet size.

We remark that this statement does not mean that $C = H$, and indeed do exist optimal prefix-free codes which cannot be obtained via the Huffman algorithm (see Figure 10.4). Rather, the previous statement indicates that the average codeword length of C and H is equal. The next fundamental theorem provides a quantitative upper-bound to this average length.

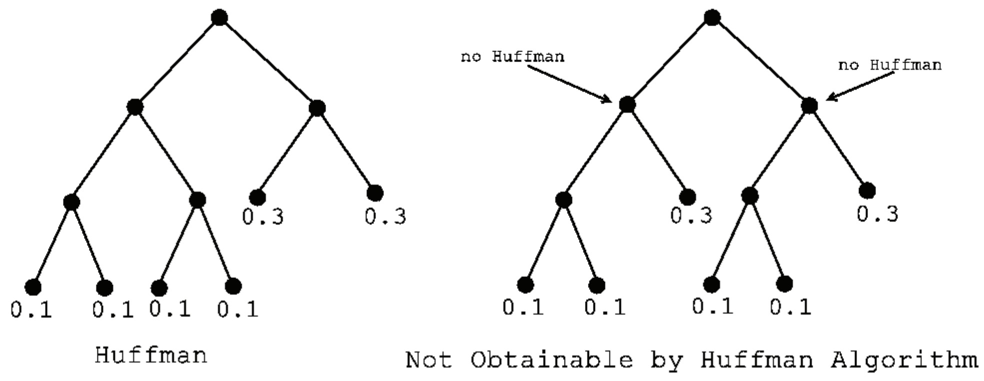


FIGURE 10.4: Example of an optimal code not obtainable by means of the Huffman algorithm.

THEOREM 10.2 *Let $\mathcal{H}$ be the entropy of the source emitting the symbols of an alphabet Σ, of size n, hence $\mathcal{H} = \sum_{i=1}^{n} P[\sigma_i] \log_2 \frac{1}{P[\sigma_i]}$. The average codeword length of the Huffman code satisfies the inequalities $\mathcal{H} \leq L_H < \mathcal{H} + 1$.*

This theorem states that the Huffman code can loose up to 1 bit per compressed symbol with respect to the entropy $\mathcal{H}$ of the underlying source. This extra-bit is a lot or a few depending on the value of $\mathcal{H}$. Clearly $\mathcal{H} \geq 0$, and it is equal to zero whenever the source emits just one symbol with probability 1 and all the other symbols with probability 0. Moreover it is also $\mathcal{H} \leq \log_2 |\Sigma|$, and it is equal to this upper bound for equiprobable symbols. As a result if $\mathcal{H} \gg 1$, the Huffman code is effective and the extra-bit is negligible; otherwise, the distribution is *skewed*, and the bit possibly lost by the Huffman code makes it inefficient. On the other hand Huffman, as any prefix-free code, cannot encode one symbol in less than 1 bit, so the best compression ratio that Huffman can obtain is $\geq \frac{1}{\log_2 |\Sigma|}$. If Σ is ASCII, hence $|\Sigma| = 256$, Huffman cannot achieve a compression ratio for any sequence S which is less than $1/8 = 12,57\%$.

In order to overcome this limitation, Shannon proposed in its famous article of 1948 a simple *blocking scheme* which considers an extended alphabet Σ^k whose symbols are substrings of k-symbols. This way, the new alphabet has size $|\Sigma|^k$ and thus, if we use Huffman on the symbol-blocks, the extra-bit lost is for a block of size k, rather than a single symbol. This actually means that we are loosing a fractional part of a bit per symbol, namely $1/k$, and this is indeed negligible for larger and larger values of k.

So why not taking longer and longer blocks as symbols of the new alphabet Σ^k? This would improve the coding of the input text, because of the blocking, but it would increase the encoding of the Huffman tree which constitutes the preamble of the compressed file: in fact, as k increases, the number of leaves/symbols also increases as $|\Sigma|^k$. The compressor should find the best trade-off between these two quantities, by possibly trying several values for k. This is clearly possible, but yet it is un-optimal; Section 10.2 will propose a provably optimal solution to this problem.

10.1.1 Canonical Huffman coding

Let us recall the two main limitations incurred by the Huffman code:

- It has to store the structure of the tree and this can be costly if the alphabet Σ is large, as it occurs when coding blocks of symbols, possibly words as symbols.
- Decoding is slow because it has to traverse the whole tree for each codeword, and every edge of the path (bit of the codeword) may elicit a cache miss.

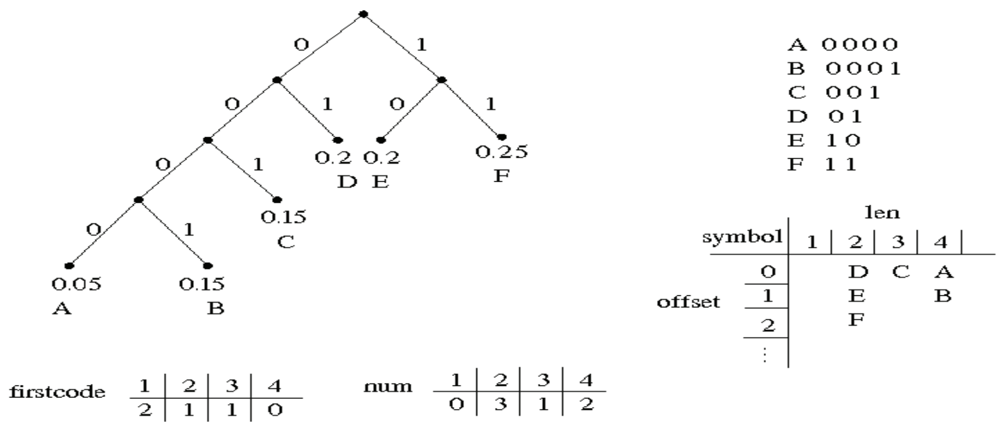
There is an elegant variant of the Huffman code, denoted as *Canonical Huffman*, that alleviates these problems by introducing a special restructuring of the Huffman tree that allows extremely fast decoding and a small memory footprint. This will be the topic of this subsection.

The Canonical Huffman code works as follows:

1. Compute the codeword length $L(\sigma)$ for each symbol $\sigma \in \Sigma$ according to the classical Huffman's algorithm.
2. Construct the array *num* which stores in the entry *num*$[\ell]$ the number of symbols having Huffman codeword of ℓ-bits.
3. Construct the array *symb* which stores in the entry *symb*$[\ell]$ the list of symbols having Huffman codeword of ℓ-bits.
4. Construct the array fc which stores in the entry $fc[\ell]$ the first codeword of all symbols encoded with ℓ bits;
5. Assign consecutive codewords to the symbols in *symb*$[\ell]$, starting from the codeword `fc`$[\ell]$.

Figure 10.5 provides an example of an Huffman tree which satisfies the Canonical property. The *num* array is actually useless, so that the Canonical Huffman needs only to store `fc` and `symb` arrays, which means at most $\texttt{max}^2$ bits to store `fc` (i.e. `max` codewords of length at most `max`), and at most $(|\Sigma| + \texttt{max}) \log_2 (|\Sigma| + 1)$ bits to encode table `symb`. Consequently the key advantage of Canonical Huffman is that we do not need to store the tree-structure via pointers, with a saving of $\Theta(|\Sigma| \log_2 (|\Sigma| + 1)$ bits.

The other important advantage of the Canonical Huffman algorithm is with the decoding procedure which does not need to percolate the Huffman tree, but it only operates on the two available arrays, thus inducing at most one cache-miss per decoded symbol. The pseudo-code is summarized in the following 6 lines:



firstcode	1	2	3	4
	2	1	1	0

num	1	2	3	4
	0	3	1	2

offset \ len	1	2	3	4
0		D	C	A
1		E		B
2		F		
⋮				

FIGURE 10.5: Example of canonical Huffman coding.

```
v = next_bit();
l = 1;
while( v < fc[l] )
    v = 2v + next_bit();
    l++;
return symb[ l, v-fc[l] ];
```

A running example of the decoding process is given un Figures 10.6–10.7. Let us assume that the compressed sequence is 01. The function `next_bit()` reads the incoming bit to be decoded, namely 0. At the first step (Figure 10.6), we have $\ell = 1$, $v = 0$ and $\texttt{fc}[1] = 2$; so the *while* condition is satisfied (because $v = 0 < 2 = \texttt{fc}[1]$) and therefore ℓ is incremented to 2 and v gets the next bit 1, thus assuming the value $v = 01 = 1$. At the second step (Figure 10.7), the *while* condition is no longer satisfied because $v = 1 < \texttt{fc}[2]$ is false and the loop has to stop. The decoded codeword has length $\ell = 2$ and, since $v - \texttt{fc}[2] = 0$, the algorithm returns the first symbol of $symb[2] = D$.

A subtle comment is in order at this point, the value $\texttt{fc}[1] = 2$ seems impossible, because we cannot represent the value 2 with a codeword consisting of one single bit. This is a *special value* for two reasons: first, it indicates that no codeword of length 1 does exist in this Canonical Huffman code; second, when $\ell = 1$, v will be surely ≤ 1 because it consists of only one bit. Therefore $v \leq \texttt{fc}[1]$, so that the decoding procedure will correctly fetch another bit.

The correctness of the decoding procedure can be inferred informally from Figure 10.8. The while-guard $v < \texttt{fc}[\ell]$ actually checks whether the current codeword v is to the left of $\texttt{fc}[\ell]$ and thus it is to the left of all symbols which are encoded with ℓ bits. In the figure this corresponds to the case $v = 0$ and $\ell = 4$, hence $v = \texttt{0000}$ and $\texttt{fc}[4] = \texttt{0001}$. If this is the case, since the Canonical Huffman tree is skewed to the left, the codeword to be decoded has to be longer and thus a new bit is fetched by the while-body. In the figure this corresponds to fetch the bit 1, and thus set $v = 1$ and $\ell = 5$, so $v = \texttt{00001}$. In the next step the while-guard is false, $v \geq \texttt{fc}[\ell]$ (as indeed $\texttt{fc}[5] = \texttt{00000}$), and thus v lies to the right of $\texttt{fc}[\ell]$ and can be decoded by looking at the symbols $symb[5]$.

The only issue it remains to detail is how to get a Canonical Huffman tree, whenever the underlying symbol distribution does not induce one with such a property. Figure 10.5 actually derived an Huffman tree which was canonical, but this is not necessarily the case. Take for example the distribution: $P[a] = P[b] = 0.05$, $P[c] = P[g] = 0.1$, $P[d] = P[f] = 0.2$, $P[e] = 0.3$, as shown in Figure 10.9. The Huffman algorithm on this tree generates a non Canonical tree, which can be turned into

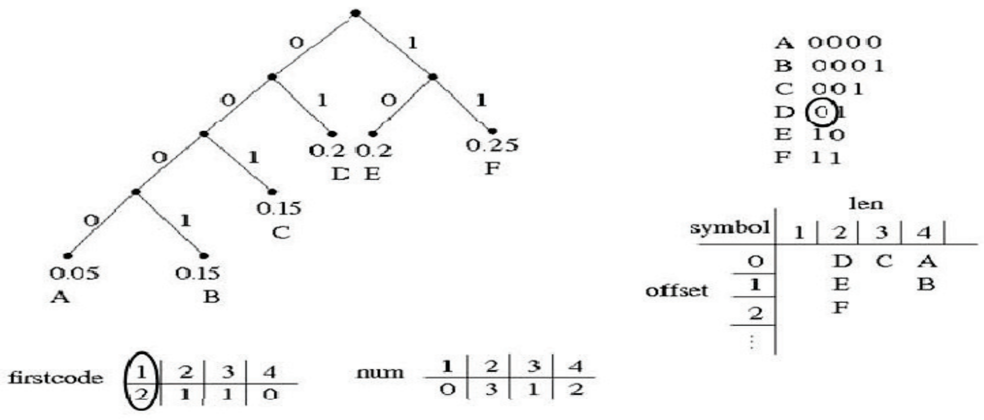


FIGURE 10.6: First Step of decoding 01 via the Canonical Huffman of Figure 10.5.

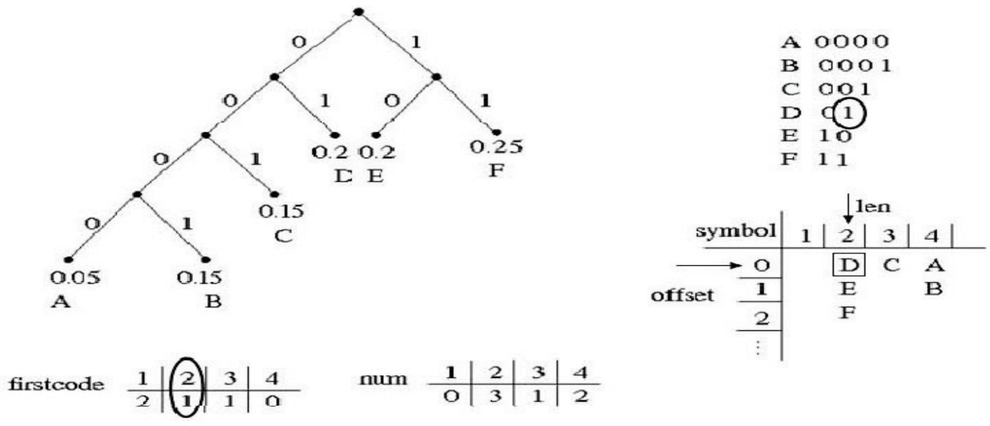


FIGURE 10.7: Second Step of decoding 01 via the Canonical Huffman of Figure 10.5.

a Canonical one by means of the following few lines of pseudo-code, in which max indicates the longest codeword length assigned by the Huffman algorithm:

```
fc[max]=0;
for(l= max-1; l>=1; l--)
    fc[l]=(fc[l+1] + num[l+1])/2;
```

There are two key remarks to be done before digging into the proof of correctness of the algorithm. First, fc[ℓ] is the value of a codeword consisting of ℓ bits, so the reader should keep in mind that fc[5] = 4 means that the corresponding codeword is 00100, which means that the binary representation of the value 4 is padded with zeros to have length 5. Second, since the algorithm sets fc[*max*] = 0, the longest codeword is a sequence of max zeros, and so the tree built by the Canonical Huffman is totally skewed to the left. If we analyze the formula that computes fc[ℓ] we can guess the reason of its correctness. The pseudo-code is reserving *num*[$\ell + 1$] codewords of length $\ell + 1$ bits to the symbols in *symb*[$\ell + 1$] starting from the value fc[$\ell + 1$]. The first *unused* codeword of $\ell + 1$

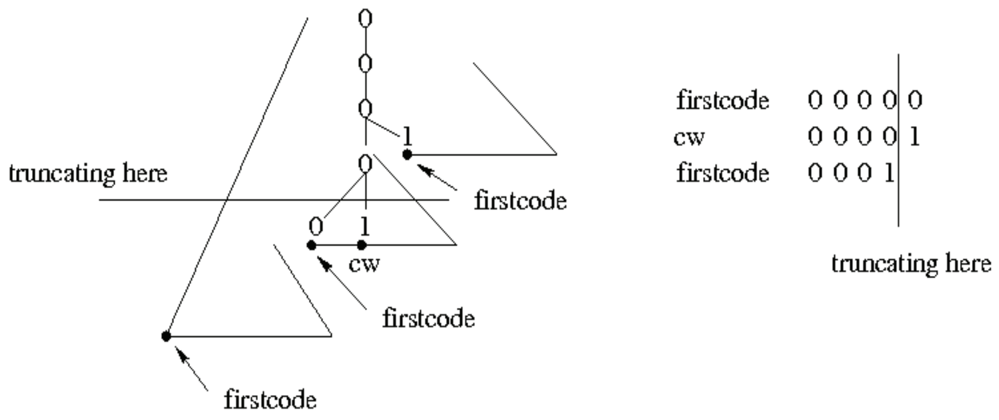


FIGURE 10.8: Tree of codewords.

bits is therefore given by the value $\texttt{fc}[\ell+1]+num[\ell+1]$. So the formula then divides this value by 2, which corresponds to dropping the last $(\ell+1)$-th bit from the binary encoding of that number. It can be proved that the resulting sequence of ℓ-bits can be taken as the first-codeword $\texttt{fc}[\ell]$ because it does not prefix any other codeword already assigned. The ”reason” can be derived graphically by looking at the binary tree which is being built by Canonical Huffman. In fact, the algorithm is taking the parent of the node at depth $\ell+1$, whose binary-path represents the value $\texttt{fc}[\ell+1]+num[\ell+1]$. Since the tree is a fully binary and we are allocating leaves in the tree from left to right, this node is always a left child of its parent, so its parent is not an ancestor of any $(\ell+1)$-bit codeword assigned before.

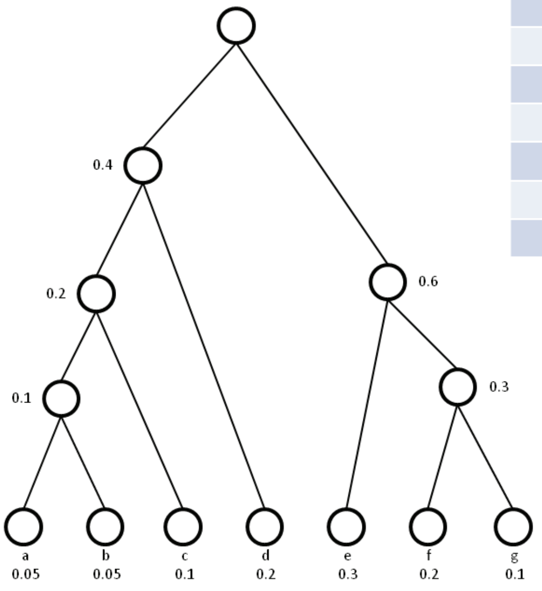
In Figure 10.9 we notice that $\texttt{fc}[1] = 2$ which is an impossible codeword because we cannot encode 2 in 1 bit; nevertheless this is the special case mentioned above that actually *encodes* the fact that no codeword of that length exists, and thus allows the decoder to find always $v < \texttt{fc}[1]$ after having read just one single bit, and thus execute `next_bit()` to fetch another bit from the input and thus consider a codeword of length 2.

10.1.2 Bounding the length of codewords

If the codeword length exceeds 32 bits the operations can become costly because it is no longer possible to store codewords as a single machine word. It is therefore interesting to survey how likely codeword overflow might be in the Huffman algorithm.

Given that the optimal code assigns a codeword length $L(\sigma) \approx \log_2 1/P[\sigma]$ bits to symbol σ, one could conclude that $P[\sigma] \approx 2^{-33}$ in order to have $L(\sigma) > 32$, and hence conclude that this bad situation occurs only after about 2^{33} symbols have been processed. This first approximation is an excessive upper bound, as the tree in Figure 10.10 allows to argue.

To obtain this tree, we construct the function $F(i)$ that gives the frequency of symbol i in an input sequence and induces the tree structure shown in Figure 10.10. Of course $F(i)$ has to be an increasing function and it should be such that $F(i+1) < F(i+2)$ and $\sum_{j=1}^{i} F(j) < F(i+2)$ in order to induce the Huffman algorithm to join $F(i+1)$ with x rather than with leaf $i+2$ (or all the other leaves $i+3, i+4, \ldots$). It is not difficult to observe that $F(i)$ may be taken to be the Fibonacci sequence, possibly with different initial conditions, such as $F(1) = F(2) = F(3) = 1$. The following two sequences show F's values and their cumulative sums for the modified Fibonacci sequence: $F = (1, 1, 1, 3, 4, 7, \ldots)$ and $\sum_{i=1}^{l+1} F(i) = (2, 3, 6, 10, 17, 28, \ldots)$. In particular it is $F(33) = 3.01 * 10^6$



Symb	depth
a	4
b	4
c	3
d	2
e	2
f	3
g	3

num = [0, 2, 3, 2]
fc = [**2**, 2 , 1, 0]
symb[2] = (d,e)
symb[3] = (c,f,g)
symb[4] = (a,b)

FIGURE 10.9: From Huffman tree to a Canonical Huffman Tree.

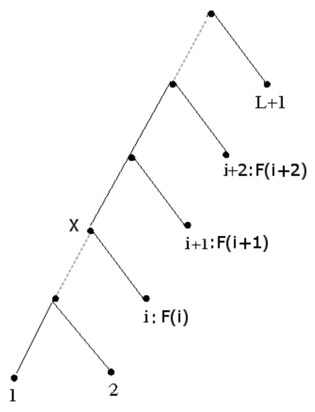


FIGURE 10.10: Example of a skewed Huffman tree.

and $\sum_{i=1}^{33} F(i) = 1.28 * 10^7$. The cumulative sum indicates how much the text has to be read in order to force a codeword of length l. Thus, the pathological case can occur just after 10 Mb; considerably less than the preceding estimation!

They do exist methods to reduce the codeword lengths still guaranteeing a good compression performance. One approach consists of *scaling the frequency counts* until they form a good arrangement. An appropriate scaling rule is

$$\hat{c}_i = \left\lceil c_i \; \frac{\sum_{i=1}^{L+2} F'(i) - 1 - |\Sigma|}{(\sum_{i=1}^{|\Sigma|} c_i)/c_{min}} \right\rceil$$

where c_i is the actual frequency count of the $i-th$ symbol in the actual sequence, c_{min} is the minimum frequency count, $\hat{c}_i$ is the scaled approximate count for i-th symbol, L is the maximum bit length permitted in the final code and $\sum_{i=1}^{L+2} F(i)$ represents the length of the text which may induce a code of length $L + 1$.

Although simple to implement, this approach could fail in some situations. An example is when 32 symbols have to be coded in codewords with no more than $L = 5$ bits. Applying the scaling rule we obtain $\sum_{i=1}^{L+2} F(i) - 1 - |\Sigma| = 28 - 1 - 32 = -5$ and consequently negative frequency counts $\hat{c}_i$. It is nevertheless possible to build a code with 5 bits per symbol, just take the fixed-length one! Another solution, which is more time-consuming but not subject to the previous drawback, is the so called *iterative scaling* process. We construct a Huffman code and, if the longest codeword is larger than L bits, all the counts are reduced by some constant ratio (e.g. 2 or the golden ratio 1.618) and a new Huffman code is constructed. This process is continued until a code of maximum codeword length L or less is generated. In the limit, all symbols will have their frequency equal to 1 thus leading to a fixed-length code.

10.2 Arithmetic Coding

The principal strength of this coding method, introduced by Elias in the '60s, is that it can code symbols arbitrarily close to the 0-th order entropy, thus resulting much better tha Huffman on skewed distributions. So in Shannon's sense it is optimal.

For the sake of clarity, let us consider the following example. Take an input alphabet $\Sigma = \{a, b\}$ with a skewed distribution: $P[a] = \frac{99}{100}$ and $P[b] = \frac{1}{100}$. According to Shannon, the *self information* of the symbols is respectively $i(a) = \log_2 \frac{1}{p_a} = \log_2 \frac{100}{99} \simeq 0,015$ bits and $i(b) = \log_2 \frac{1}{p_b} = \log_2 \frac{100}{99} \simeq 6,67$ bits. Hence the 0-th order entropy of this source is $\mathcal{H}_0 = P[a]\, i(a) + P[b]\, i(b) \simeq 0,08056$ bits. In contrast a Huffman coder, like any prefix-coders, applied to texts generated by this source must use at least one bit per symbol thus having average length $L_H = P[a]\, L(a) + P[b]\, L(b) = P[a] + P[b] = 1 \gg \mathcal{H}_0$. Consequently Huffman is far from the 0-th order entropy, and clearly, the more skewed is the symbol distribution the farthest is Huffman from optimality.

The problem is that Huffman replaces each input symbol with a codeword, formed by an integral number of bits, so the average length of a text T compressed by Huffman is $\Omega(|T|)$ bits. Therefore Huffman cannot achieve a compression ratio better than $\frac{1}{\log_2 |\Sigma|}$, the best case is when we substitute one symbol (i.e. $\log_2 |\Sigma|$) with 1 bit. This is $1/8 = 12.5\%$ in the case that Σ are the characters of the ASCII code.

To overcome this problem, Arithmetic Coding relaxes the request to be a prefix-coder by adopting a different strategy:

- the compressed output is *not* a concatenation of codewords associated to the symbols of the alphabet.
- rather, a bit of the output can represent *more than one* input symbols.