10

Statistical Coding

10.1 Huffman coding 10-1
Canonical Huffman coding • Bounding the length of codewords
10.2 Arithmetic Coding 10-11
Bit streams and dyadic fractions • Compression algorithm • Decompression algorithm • Efficiency • Arithmetic coding in practice • Range Coding$^\infty$
10.3 Prediction by Partial Matching$^\infty$ 10-22
The algorithm • The principle of exclusion • Zero Frequency Problem

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).

© Paolo Ferragina, 2009-2016

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 $t = |\Sigma| + (|\Sigma| - 1) = 2|\Sigma| - 1$.

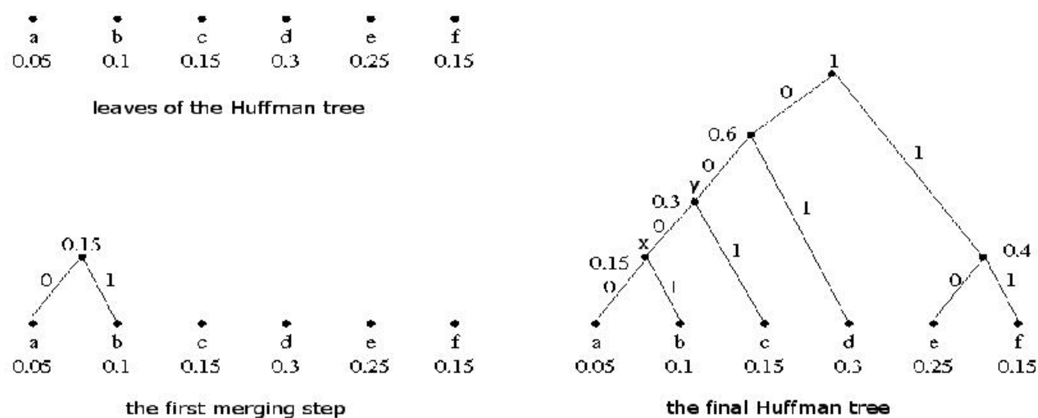


FIGURE 10.1: Constructing the Huffman tree for the alphabet $\Sigma = \{a, b, c, d, e, f\}$.

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.

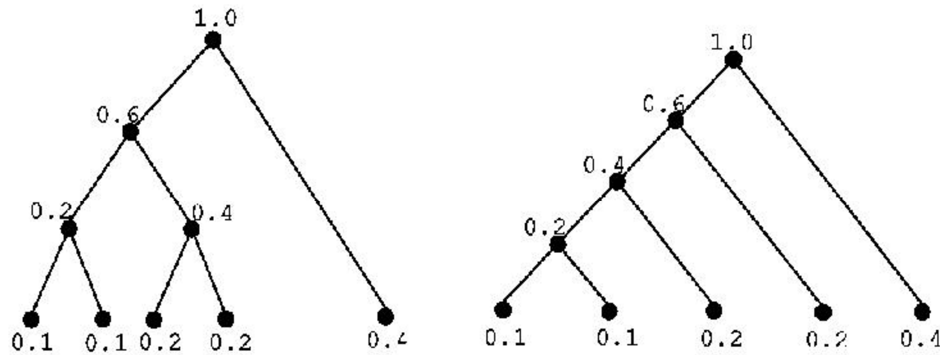


FIGURE 10.2: An example of two Huffman codes having the same average codeword length $\frac{22}{10}$, but different maximum codeword length.

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).

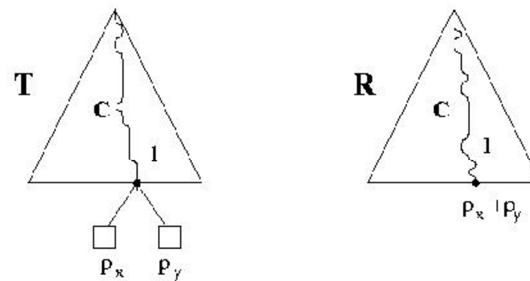


FIGURE 10.3: Relationship between a tree T and its corresponding reduced tree R.

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.

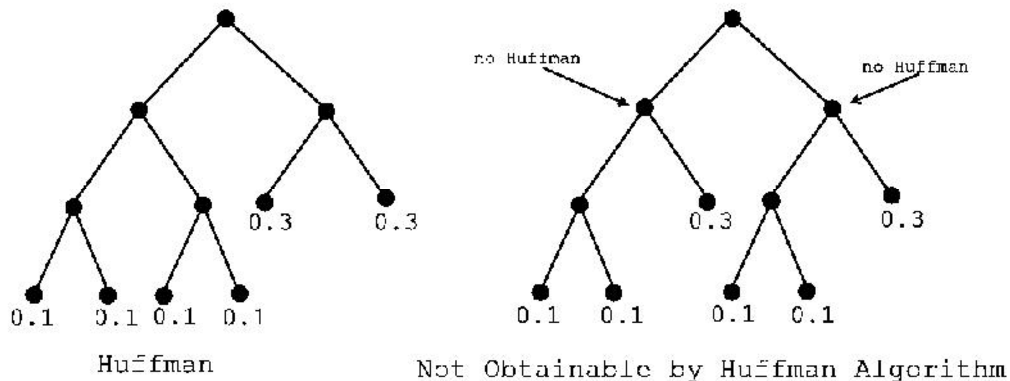


FIGURE 10.4: Example of an optimal code not obtainable by means of the Huffman algorithm.

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.
- 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. Thus the total number of cache misses could be equal to the total number of bits constituting the compressed file.

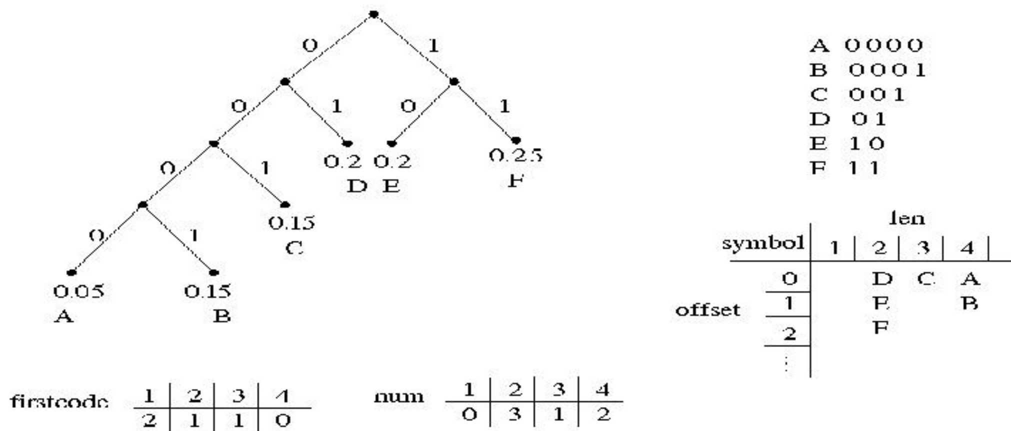
There is an elegant variant of the Huffman code, denoted as *Canonical Huffman*, that alleviates these problems by introducing a special restructuring of the Huffman tree that allows extremely fast decoding and a small memory footprint. This will be the topic of this subsection.

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 $\texttt{fc}[\ell]$ the first codeword of all symbols encoded with ℓ bits;
5. Assign consecutive codewords to the symbols in $symb[\ell]$, starting from the codeword $\texttt{fc}[\ell]$.

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` each), 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 Canonical Huffman resides in its decoding procedure which does not need to percolate the Huffman tree, but it only operates on the two available arrays, thus



firstcode	1	2	3	4
	2	1	1	0

num	1	2	3	4
	0	3	1	2

symbol / offset	1	2	3	4
0		D	C	A
1		E		B
2		F		
⋮				

FIGURE 10.5: Example of canonical Huffman coding.

inducing at most two cache-misses per decoded symbol.[1] The pseudo-code is summarized in the following 6 lines:

```
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 $\mathtt{fc}[1] = 2$; so the *while* condition is satisfied (because $v = 0 < 2 = \mathtt{fc}[1]$) 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 $v = 1 < \mathtt{fc}[2]$ is false and the loop has to stop. The decoded codeword has length $\ell = 2$ and, since $v - \mathtt{fc}[2] = 0$, the algorithm returns the first symbol of $symb[2] = D$.

A subtle comment is in order at this point, the value $\mathtt{fc}[1] = 2$ seems impossible, because we cannot represent the value 2 with a codeword consisting of one single bit. This is a *special value* because this way $\mathtt{fc}[1]$ will be surely larger than any codeword of one bit, hence the Canonical Huffman algorithm will surely fetch another bit in the while-cycle.

The correctness of the decoding procedure can be inferred informally from Figure 10.8. The while-guard $v < \mathtt{fc}[\ell]$ actually checks whether the current codeword v is to the left of $\mathtt{fc}[\ell]$ 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 $v = \mathtt{0000}$ and $\mathtt{fc}[4] = \mathtt{0001}$. 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 $v = \mathtt{00001}$. In the next step the while-guard is false, $v \geq \mathtt{fc}[\ell]$ (as indeed $\mathtt{fc}[5] = \mathtt{00000}$), and thus v lies to the right of $\mathtt{fc}[\ell]$ and can be decoded by looking at the symbols $symb[5]$.

[1]It is reasonable to assume that the number of cache-misses is just 1 because the array `fc` is small and can be fit in cache.

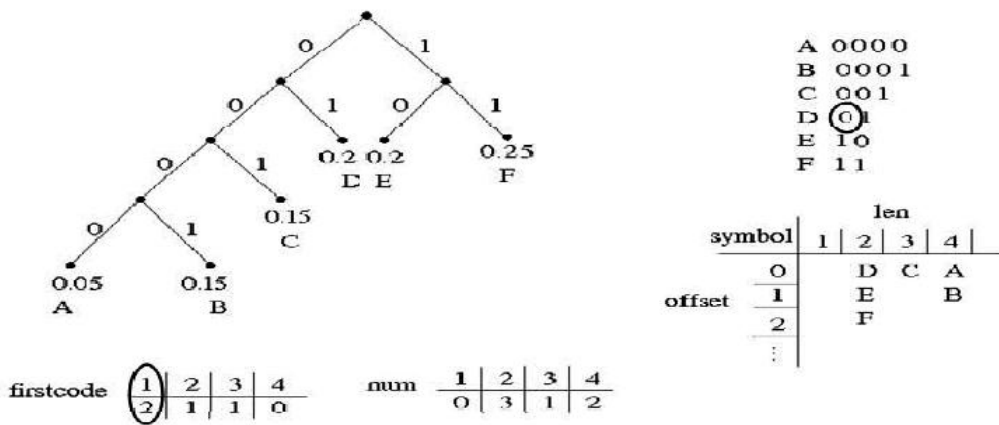


FIGURE 10.6: First Step of decoding 01 via the Canonical Huffman of Figure 10.5.

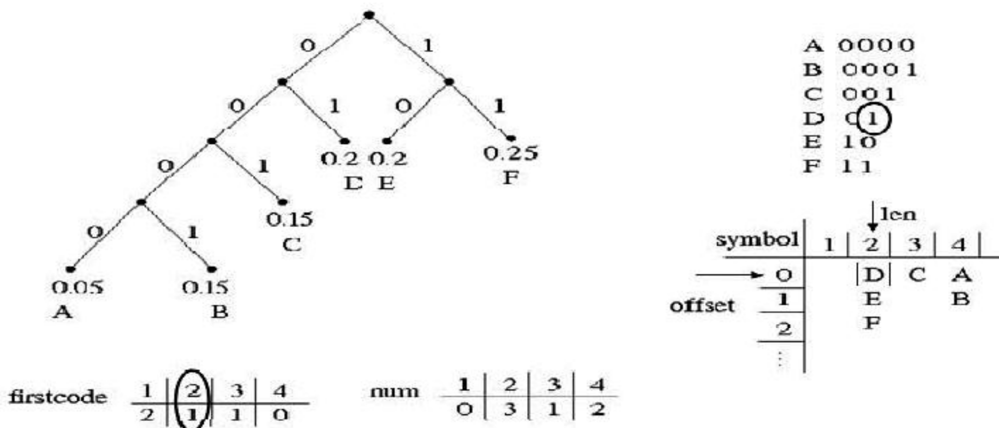


FIGURE 10.7: Second Step of decoding 01 via the Canonical Huffman of Figure 10.5.

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 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`$[\ell]$ 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

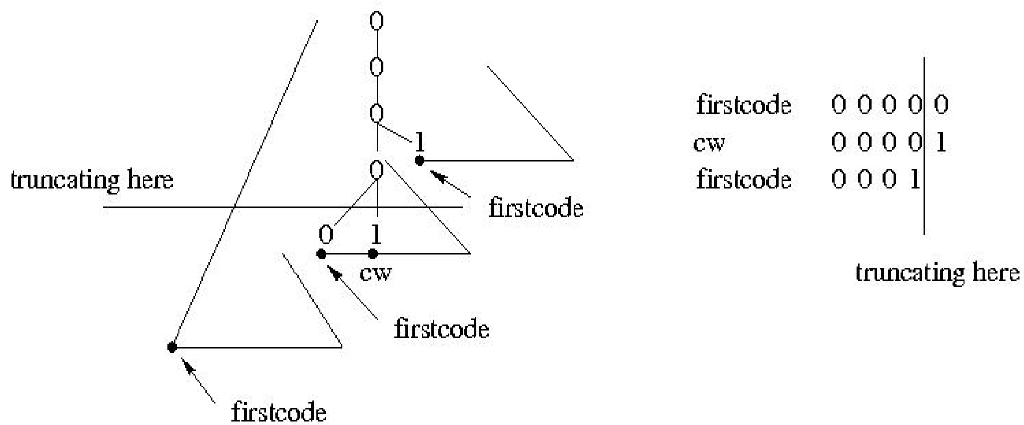


FIGURE 10.8: Tree of codewords.

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 $\ell+1$ bits is therefore given by the value 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 fc[ℓ] 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 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 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 $v <$ 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.

It is enough to consider a Huffman tree which has the structure of a binary tree, skewed to the left, whose leaf i has frequency $F(i)$ which is an increasing function, hence $F(i+1) < F(i+2)$. Moreover we assume that $\sum_{j=1}^{i} F(j) < F(i+2)$ in order to induce the Huffman algorithm to join $F(i+1)$ with

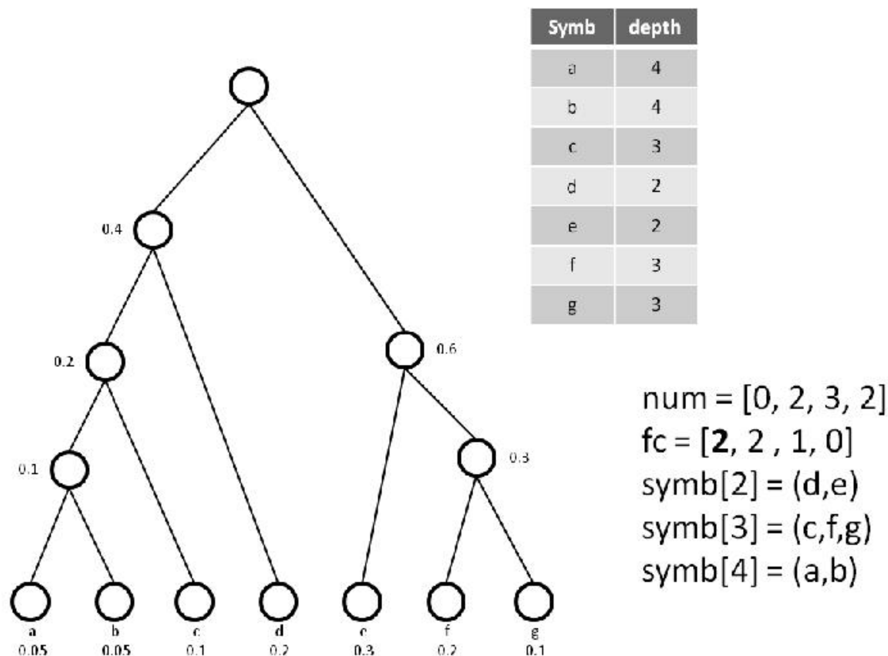


FIGURE 10.9: From Huffman tree to a Canonical Huffman Tree.

the last created internal node 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 $F(33) = 3.01 * 10^6$ and $\sum_{i=1}^{33} F(i) = 1.28 * 10^7$. The cumulative sum indicates how much 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 (encoded plainly with $\log_2 |\Sigma|$ bits) with just 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.

This results in a better compression, at the cost of slowing down the algorithm and of loosing the capability to access/decode the compressed output from any position.

Another interesting feature of Arithmetic coding is that it works easily also in the case of a dynamic model, namely a model in which probabilities $P[\sigma]$ are updated as the input sequence S is processed. It is enough to set $P[\sigma] = (\ell_\sigma + 1)/(\ell + |\Sigma|)$ where ℓ is the length of the prefix of S processed so far, and ℓ_σ is the number of occurrences of symbol σ in that prefix. The reader can check that this is a sound probability distribution, initially set to the uniform one. Easily enough, these dynamic probabilities can be also kept updated by the decompression algorithm, so that both compressor and decompressor look at the same input distribution and thus decode the same symbols.

10.2.1 Bit streams and dyadic fractions

A bit stream $b_1 b_2 b_3 \ldots b_k$, possibly $k \to \infty$, can be interpreted as a real number in the range $[0, 1)$ by prepending "0." to it:

$$b_1 b_2 b_3 \ldots b_k \to 0.b_1 b_2 b_3 \ldots b_k = \sum_{i=1}^{k} b_i \cdot 2^{-i}$$

A real number x in the range $[0, 1)$ can be converted in a (possibly infinite) sequence of bits with the algorithm `Converter`, whose pseudocode is given below. This algorithm consists of a loop where

the variable *output* is the output bitstream and where :: expresses concatenation among bits. The loop has to end when the condition *accuracy* is satisfied: we can decide a level of accuracy in the representation of x, we can stop when we emitted a certain number of bits in output or when we establish that the representation of x is periodic.

Algorithm 10.1 Converter (x)
Require: A real number $x \in [0, 1)$.
Ensure: The string of bits representing x.

```
1: repeat
2:     x = 2 * x
3:     if x < 1 then
4:         output = output :: 0
5:     else
6:         output = output :: 1
7:         x = x - 1
8:     end if
9: until accuracy
```

A key concept here is the one of *dyadic fraction*, namely a fraction of the form $\frac{v}{2^k}$ where v and k are positive integers. The real number associated to a finite bit stream $b_1 b_2 b_3 \ldots b_k$ is indeed the dyadic fraction $\frac{val(b_1 b_2 b_3 \ldots b_k)}{2^k}$, where $val(s)$ is the value of the binary string s. Vice versa a fraction $\frac{v}{2^k}$ can be written as $.bin_k(v)$, where $bin_k(v)$ is the binary representation of the integer v as a bit string of length k (eventually padded with zeroes).

In order to clarify how `Converter` works, we apply the pseudocode at the number $\frac{1}{3}$:

$$\frac{1}{3} \cdot 2 = \frac{2}{3} < 1 \rightarrow output = 0$$

$$\frac{2}{3} \cdot 2 = \frac{4}{3} \geq 1 \rightarrow output = 01$$

In this second iteration x is greater than 1, so we have concatenated the bit 1 to the output and, at this point, we need to update the value of x executing the line 7 in the pseudocode:

$$\frac{4}{3} - 1 = \frac{1}{3}$$

We have already encountered this value of x, so we can stop the loop and output the periodic representation $\overline{01}$ for $\frac{1}{3}$.
Let us consider another example; say $x = \frac{3}{32}$.

$$\frac{3}{32} \cdot 2 = \frac{6}{32} < 1 \rightarrow output = 0$$

$$\frac{6}{32} \cdot 2 = \frac{12}{32} < 1 \rightarrow output = 00$$

$$\frac{12}{32} \cdot 2 = \frac{24}{32} < 1 \rightarrow output = 000$$

$$\frac{24}{32} \cdot 2 = \frac{48}{32} \geq 1 \rightarrow output = 0001$$

$$\left(\frac{48}{32} - 1\right) \cdot 2 = 1 \geq 1 \rightarrow output = 00011$$

$$(1 - 1) \cdot 2 = 0 < 1 \rightarrow output = 000110$$

$$0 \cdot 2 = 0 < 1 \rightarrow output = 0001100$$

and so on. The binary representation for $\frac{3}{32}$ is $00011\bar{0}$.

10.2.2 Compression algorithm

Compression by Arithmetic coding is iterative: each step takes as input a subinterval of $[0, 1)$, representing the prefix of the input sequence compressed so far, and the *probabilities* and the *cumulative probabilities* of alphabet symbols,[2] and consumes the next input symbol. This subinterval is further subdivided into smaller subintervals, one for each symbol σ of Σ, whose lengths are proportional to their probabilities $P[\sigma]$. The step produces as output a new subinterval that is the one associated to the consumed input symbol, and is contained in the previous one. The number of steps is equal to the number of symbols to be encoded, and thus to the length of the input sequence.

More in detail, the algorithm starts considering the interval $[0, 1)$ and, consumed the entire input, produces the interval $[l, l + s)$ associated to the last symbol of the input sequence. The tricky issue here is that the output is not the pair $\langle l, s \rangle$ (hence two real numbers) but it is just one real $x \in [l, l+s)$, chosen to be a dyadic fraction, plus the length of the input sequence.

In the next section we will see how to choose this value in order to minimize the number of output bits, here we will concentrate on the overall compression stage whose pseudocode is indicated below: the variables l_i and s_i are, respectively, the starting point and the length of the interval encoding the i-th symbol of the input sequence.

Algorithm 10.2 AC-Coding (S)

Require: The input sequence S, of length n, the probabilities $P[\sigma]$ and the cumulative f_σ.
Ensure: A subinterval $[l, l + s)$ of $[0, 1)$.

```
1: s_0 = 1
2: l_0 = 0
3: i = 1
4: while i ≤ n do
5:     s_i = s_{i-1} * P[S[i]]
6:     l_i = l_{i-1} + s_{i-1} * f_{S[i]}
7:     i = i + 1
8: end while
9: output = ⟨x ∈ [l_n, l_n + s_n), n⟩
```

As an example, consider the input sequence $S = abac$ with probabilities $P[a] = \frac{1}{2}$, $P[b] = P[c] = \frac{1}{4}$ and cumulative probabilities $f_a = 0$, $f_b = P[a] = \frac{1}{2}$, and $f_c = P[a] + P[b] = \frac{3}{4}$. Following the pseudocode of `AC-Coding (S)` we have $n = 4$ and thus we repeat the internal loop four times.

[2]We recall that the cumulative probability of a symbol $\sigma \in \Sigma$ is computed as $\sum_{c<\sigma} P[c]$ and it is provided by the statistical model, constructed during the modeling phase of the compression process. In the case of a dynamic model, the probabilities and the cumulative probabilities change as the input sequence is scanned.

In the first iteration we consider the first symbol of the sequence, $S[1] = a$, and compute the new interval $[l_1, l_1 + s_1)$ given $P[a]$ and f_a from the static model:

$$s_1 = s_0\, P[S[1]] = 1 \times P[a] = \frac{1}{2}$$

$$l_1 = l_0 + s_0\, f_{S[1]} = 0 + 1 \times f_a = 0$$

In the second iteration we consider the second symbol, $S[2] = b$, the (cumulative) probabilities $P[b]$ and f_b, and determine the second interval $[l_2, l_2 + s_2)$:

$$s_2 = s_1\, P[S[2]] = \frac{1}{2} \times P[b] = \frac{1}{8}$$

$$l_2 = l_1 + s_1\, f_{S[2]} = 0 + \frac{1}{2} \times f_b = \frac{1}{4}$$

We continue this way for the third and the fourth symbols, namely $S[3] = a$ and $S[4] = c$, so at the end the final interval is:

$$[l_4, l_4 + s_4) = \left[\frac{19}{64}, \frac{19}{64} + \frac{1}{64}\right) = \left[\frac{19}{64}, \frac{20}{64}\right) = \left[\frac{19}{64}, \frac{5}{16}\right)$$

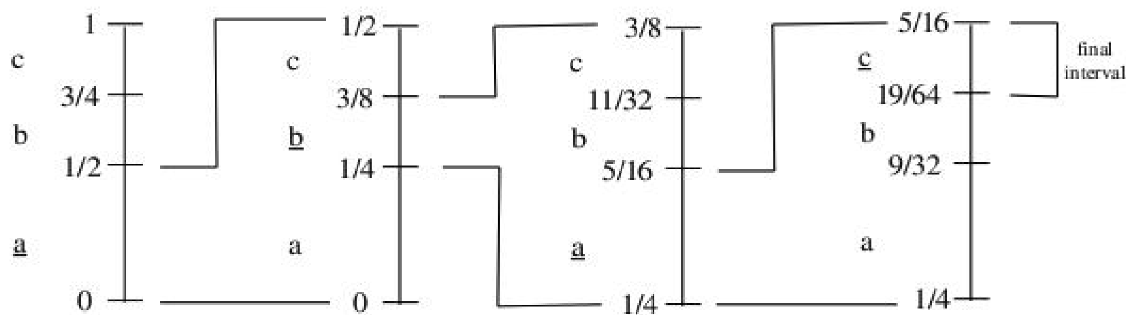


FIGURE 10.10: The algorithmic idea behind Arithmetic coding.

In Figure 10.10 we illustrate the execution of the algorithm in a graphical way. Each step zooms in the subinterval associated to the current symbol. The last step returns a real number inside the final subinterval, hence this number is *inside all* the previously generated intervals too. This number together with the value of n is sufficient to reconstruct the entire sequence of input symbols S, as we will show in the following subsection. In fact, all input sequences of a fixed length n will be associated to distinct sub-intervals which do not intersect and cover $[0, 1)$; but sequences of different length might be nested.

10.2.3 Decompression algorithm

The input consists of the stream of bits resulting from the compression stage, the length of the input sequence n, the symbol probabilities $P[\sigma]$, and the output is the original sequence S given that Arithmetic coding is a *lossless compressor*. The decompressor works also in the case of a dynamic model:

Algorithm 10.3 AC-Decoding (b, n)

Require: The binary representation b of the compressed output, the length n of S, the probabilities $P[\sigma]$ and the cumulative f_σ.
Ensure: The original sequence S.

1: $s_0 = 1$
2: $l_0 = 0$
3: $i = 1$
4: **while** $i \leq n$ **do**
5: subdivide the interval $[l_{i-1}, l_{i-1} + s_{i-1})$ into subintervals of length proportional to the probabilities of the symbols in Σ (in the predefined order)
6: take the symbol σ corresponding to the subinterval in which $0.b$ lies
7: $S = S :: \sigma$
8: $s_i = s_{i-1} * P[\sigma]$
9: $l_i = l_{i-1} + s_{i-1} * f_\sigma$
10: $i = i + 1$
11: **end while**
12: $output = S$

- at the first iteration, the statistical model is set to the uniform distribution over Σ;
- at every other iteration, a symbol is decoded using the current statistical model, which is then updated by increasing the frequency of that symbol.

Decoding is correct because encoder and decoder are synchronized, in fact they use the same statistical model to decompose the current interval, and both start from the interval $[0, 1)$. The difference is that the encoder uses symbols to choose the subintervals, whereas the decoder uses the number b to choose (the same) subintervals to zoom in.

As an example, take the result $\langle \frac{39}{128}, 4\rangle$ and assume that the input distribution is $P[a] = \frac{1}{2}$, $P[b] = P[c] = \frac{1}{4}$ (i.e. the one of the previous section). The decoder executes the decompression algorithm starting with the initial interval $[0, 1)$, we suggest the reader to parallel the decompression process with the compression one in Figure 10.10:

- in the first iteration the initial range will be subdivided in three subintervals one for each symbol of the alphabet. These intervals follow a predefined order; in particular, for this example, the first interval is associated to the symbol a, the second to b and the last one to c. The size of every subinterval is proportional to the probability of the respective symbol: $[0, \frac{1}{2})$, $[\frac{1}{2}, \frac{3}{4})$ and $[\frac{3}{4}, 1)$. At this point the algorithm will generate the symbol a because $\frac{39}{128} \in [0, \frac{1}{2})$. After that it will update the subinterval executing the steps 8 and 9, thus synchronizing itself with the encoder.
- in the second iteration the new interval $[0, \frac{1}{2})$ will be subdivided in the subintervals $[0, \frac{1}{4})$, $[\frac{1}{4}, \frac{3}{8})$, $[\frac{3}{8}, \frac{1}{2})$. The second symbol generated will be b because $\frac{39}{128} \in [\frac{1}{4}, \frac{3}{8})$.
- continuing in this way, the third and the fourth iterations will produce respectively the symbols a and c, and the initial sequence will be reconstructed correctly. Notice that we can stop after generating 4 symbols because that was the original sequence length, communicated to the decoder.

10.2.4 Efficiency

Intuitively this scheme performs well because we associate large subintervals to frequent symbols (given that the interval size s_i decreases as $P[S[i]] \leq 1$), and a large final interval requires fewer

bits to specify a number inside it. From step 5 of the pseudocode of `AC-Coding (S)` is easy to determine the size s_n of the final interval associated to an input sequence S of length n:

$$s_n = s_{n-1} \times P[S[n]] = s_{n-2} \times P[S[n-1]] \times P[S[n-1]] = \ldots =$$

$$= s_0 \times P[S[1]] * \cdots * P[S[n]] = 1 \times \prod_{i=1}^{n} P[S[i]] \quad (10.1)$$

The formula 10.1 is interesting because it says that s_n depends on the symbols forming S but not on their ordering within S. So the size of the interval returned by Arithmetic coding for S is the same whichever is that ordering. Now, since the size of the interval impacts onto the number of bits returned in the compressed output, we derive that the output size is *independent of* the permutation of S's characters. This does not contradicts with the previous statement, proved below, that Arithmetic coding achieves a performance close to the 0th order empirical entropy $\mathcal{H}_0 = \mathrm{Sum}_{i=1}^{|\Sigma|} P[\sigma_i] \log_2(1/P[\sigma_i])$ of the sequence S, given that entropy's formula is independent of S's symbol ordering too.

We are left with the problem of choosing a number inside the interval $[l_n, l_n + s_n)$ that has the form of a dyadic fraction $\frac{v}{2^k}$ and can be encoded with the fewest bits (i.e. smallest k). The following lemma is crucial to establish the performance and correctness of Arithmetic coding.

LEMMA 10.3 Take a real number $x = 0.b_1 b_2 \cdots$. If we truncate it to its first d bits, we obtain a real number $trunc_d(x) \in [x - 2^{-d}, x]$.

Proof The real number $x = 0.b_1 b_2 \cdots$ differs from its truncation, possibly, on the bits that follow the position d. Those bits have been reset to 0 in $trunc_d(x)$. Therefore we have:

$$x - \mathrm{trunc}_d(x) = \sum_{i=1}^{\infty} b_{d+i} 2^{-(d+i)} \leq \sum_{i=1}^{\infty} 1 \times 2^{-(d+i)} = 2^{-d} \sum_{i=1}^{\infty} \frac{1}{2^i} = 2^{-d}$$

So we have

$$x - \mathrm{trunc}_d(x) \leq 2^{-d} \Longleftrightarrow x - 2^{-d} \leq \mathrm{trunc}_d(x)$$

On the other hand, it is of course $\mathrm{trunc}_d(x) \leq x$ because we have reset possibly few bits to 0. ∎

COROLLARY 10.1 The truncation of $l + \frac{s}{2}$ to its first $\left\lceil \log_2 \frac{2}{s} \right\rceil$ bits falls in the interval $[l, l + s)$.

Proof It is enough to set $d = \left\lceil \log_2 \frac{2}{s} \right\rceil$ in Lemma 10.3, and observe that $2^{-d} \leq \frac{s}{2}$. ∎

At this point we can specialize `AC-Coding(S)` in order to return the first $\left\lceil \log_2 \frac{2}{s_n} \right\rceil$ bits of the binary representation of the value $l_n + \frac{s_n}{2}$. Nicely enough, algorithm `Converter` allows to incrementally generate these bits.

For the sake of clarity, let us resume the previous example taking the final interval $[l_4, l_4 + s_4) = [\frac{19}{64}, \frac{20}{64})$ found in the compression stage. We know that $l_4 = \frac{19}{64}$ and $s_4 = \frac{1}{64}$, hence the value to output is

$$l_4 + \frac{s_4}{2} = \frac{19}{64} + \frac{1}{64} \cdot \frac{1}{2} = \frac{39}{128}$$

truncated at the first $\left\lceil \log_2 \frac{2}{s_4} \right\rceil = \log_2 128 = 7$ bits. The resulting stream of bits associated to this value is obtained by executing the algorithm `Converter` for seven times, in this way:

$$\frac{39}{128} \cdot 2 = \frac{78}{128} < 1 \rightarrow output = 0$$

$$\frac{78}{128} \cdot 2 = \frac{156}{128} \geq 1 \rightarrow output = 01$$

$$\left(\frac{156}{128} - 1\right) \cdot 2 = \frac{56}{128} < 1 \rightarrow output = 010$$

$$\frac{56}{128} \cdot 2 = \frac{112}{128} < 1 \rightarrow output = 0100$$

$$\frac{112}{128} \cdot 2 = \frac{224}{128} \geq 1 \rightarrow output = 01001$$

$$\left(\frac{224}{128} - 1\right) \cdot 2 = \frac{192}{128} \geq 1 \rightarrow output = 010011$$

$$\left(\frac{192}{128} - 1\right) \cdot 2 = 1 \geq 1 \rightarrow output = 0100111$$

At the end the encoder sends the pair $\langle 0100111_2, 4\rangle$ and the statistical model to the decoder given by $\Sigma = \{a, b, c\}$ and the symbol probabilities $P[a] = \frac{1}{2}$, $P[b] = \frac{1}{4}$, $P[c] = \frac{1}{4}$.

We are ready now to prove the main theorem of this section which relates the compression ratio achieved by Arithmetic coding with the 0-th order entropy of S.

THEOREM 10.3 *The number of bits emitted by Arithmetic Coding for a sequence S of length n is at most $2 + n\mathcal{H}_0$, where $\mathcal{H}_0$ is the 0-th order entropy of the input source.*

Proof By Corollary 10.1, we know that the number of output bits is:

$$\left\lceil \log_2 \frac{2}{s_n} \right\rceil \leq 2 - \log_2 s_n = 2 - \log_2 \left(\prod_{i=1}^{n} P[S[i]] \right) = 2 - \sum_{i=1}^{n} \log_2 P[S[i]]$$

If we let n_σ be the number of times a symbol σ occurs in S, and assume that n is sufficiently large, then we can estimate $P[\sigma] \simeq \frac{n_\sigma}{n}$. At this point we can rewrite the summation by iterating not over the positions i in S, but rather by grouping the same symbols and thus iterating over the symbols σ:

$$2 - \sum_{\sigma \in \Sigma} n_\sigma \log_2 P[\sigma] = 2 - n \left(\sum_{\sigma \in \Sigma} P[\sigma] \log_2 P[\sigma] \right) = 2 + n\mathcal{H}_0$$

∎

We can draw some considerations from the result just proved:

- there is a waste of only two bits on an entire input sequence S, hence $\frac{2}{n}$ bits per symbol. This is a vanishing lost as the input sequence becomes longer and longer.
- the size of the output is a function of the set of symbols constituting S with their multiplicities, but not of their order.

In the previous section we have seen that Huffman coding requires $n + n\mathcal{H}_0$ bits for compressing a sequence of n symbols, so Arithmetic Coding is much better. Another advantage is that it calculates the representation on the fly thus it can easily accommodate the use of dynamic modeling. On the other hand, it must be said that (Canonical) Huffman is faster and can decompress any portion of the compressed file provided that we known its first codeword. This is however impossible for Arithmetic Coding which allows only the whole decompression of the compressed file. This property justifies the frequent use of Canonical Huffman coding in the context of compressing Web collections, where Σ consists of words/tokens. Such a variant is known as Huffword [7].

10.2.5 Arithmetic coding in practice

The implementation of Arithmetic coding presents two main problems that we comment below.

The number x produced by the coding phase is known only when the entire input is processed: this is a disadvantage in situations like digital communications, in which for the sake of speed, we desire to start encoding/decoding before the source/compressed string is completely scanned; some possible solutions are:

1. the text to be compressed is subdivided into blocks, which are compressed individually; this way, even the problem of specifying the length of the text is relieved: only the length of the last block must be sent to permit its decompression, or the original file can be padded to an integral number of blocks, if the real 'end of file' is not important.
2. the two extremes of the intervals produced at each compression step are compared and the binary prefix on which their binary representation coincides is emitted. This option does not solve the problem completely, in fact it can happen that they don't have any common prefix for a long time, nonetheless this is effective because it happens frequently in practice.

More significantly, the encoding and decoding algorithms presented above require *arithmetic with infinite precision* which is costly to be approximated. There are several proposals about using *finite precision arithmetic* (see e.g. [3, 8]), which nonetheless penalizes the compression ratio up to $n\mathcal{H}_0 + \frac{2}{100}n$. Even so Arithmetic coding is still better than Huffman: $\frac{2}{100}$ vs. 1 bit loss.

The next subsection describes a practical implementation for Arithmetic coding proposed by Witten, Neal and Clearly [8]; sometimes called *Range Coding* [5]. It is mathematically equivalent to Arithmetic Coding, which works with finite precision arithmetic so that subintervals have integral extremes.

10.2.6 Range Coding$^{\infty}$

The key idea is to make some approximations, in order to represent in finite precision real numbers:

- for every symbol σ in the sorted alphabet Σ the probability $P[\sigma]$ is approximated by an integer count $c[\sigma]$ of the number of occurrences of the symbol in the input sequence, and the cumulative probability f_σ with a cumulative count $C[\sigma]$ which sums the counts of all symbols preceding σ in Σ, hence $C[\sigma] = \sum_{\alpha<\sigma} c[\alpha]$. So we have

$$P[\sigma] = \frac{c[\sigma]}{C[|\Sigma| + 1]} \qquad f_\sigma = \frac{C[\sigma]}{C[|\Sigma| + 1]}$$

- the interval $[0, 1)$ is mapped into the integer interval $[0, M)$, where $M = 2^w$ depends on the length w in bits of the memory-word.

- during the i-th iteration of the compression or decompression stages the current subinterval (formerly $[l_i, l_i + s_i)$) will be chosen to have integer endpoints $[L_i, H_i)$ such that

$$L_i = L_{i-1} + \lfloor f_{S[i]} (H_{i-1} - L_{i-1}) \rfloor$$

$$H_i = L_i + \lfloor P[S[i]] (H_{i-1} - L_{i-1}) \rfloor$$

These approximations induce a compression loss empirically estimated (by the original authors) as 10^{-4} bits per input symbol. In order to clarify how it works, we will first explain the compression and decompression stages, and then we will illustrate an example.

Compression stage. In order to guarantee that every interval $[L_i, H_i)$ has non-empty subintervals, at each step we must have

$$H_i - L_i \geq \frac{M}{4} + 2 \qquad (10.2)$$

In fact, if we compute the integer starting point of the subinterval L_{i+1} by

$$L_{i+1} = L_i + \lfloor f_{S[i+1]} \cdot (H_i - L_i) \rfloor = L_i + \left\lfloor \frac{C[S[i+1]]}{C[|\Sigma| + 1]} \cdot (H_i - L_i) \right\rfloor$$

since $C[i]$s are strictly increasing, in order to guarantee that we do not have empty subintervals, it is sufficient to have $\frac{H_i - L_i}{C[|\Sigma|+1]} \geq 1$ that can be obtained, from Equ. 10.2, by keeping

$$C[|\Sigma| + 1] \leq \frac{M}{4} + 2 \leq H_i - L_i \qquad (10.3)$$

This means that an adaptive Arithmetic Coding, that computes the cumulative counts during compression, must reset these counts every $\frac{M}{4} + 2$ input symbols, or rescale these counts every e.g. $\frac{M}{8} + 1$ input symbols by dividing them by 2.

Rescaling. We proved that, if in each iteration the interval $[L_i, H_i)$ has size $\geq \frac{M}{4} + 2$, we can subdivide it in non-empty subintervals with integer endpoints and sizes proportional to the probabilities of the symbols. In order to guarantee this condition, one can adopt the following *expansion rules*, which are repeatedly checked before each step of the compression process:

1. $[L_i, H_i) \subseteq \left[0, \frac{M}{2}\right) \rightarrow$ output '0', and the new interval is:

$$[L_{i+1}, H_{i+1}) = [2 \cdot L_i, 2 \cdot (H_i - 1) + 2)$$

2. $[L_i, H_i) \subseteq \left[\frac{M}{2}, M\right) \rightarrow$ output '1', and the new interval is

$$[L_{i+1}, H_{i+1}) = \left[2 \cdot \left(L_i - \frac{M}{2}\right), 2 \cdot \left(H_i - 1 - \frac{M}{2}\right) + 2\right)$$

3. if $\frac{M}{4} \leq L_i < \frac{M}{2} < H_i \leq \frac{3M}{4}$ then we cannot output any bit, even if the size of the interval is less than $\frac{M}{4}$, so we have a so called *underflow condition* (that is managed as indicated below).
4. otherwise, it is $H_i - L_i \geq \frac{M}{4} + 2$ and we can continue the compression as is.

In the case of underflow, we cannot emit any bit until the interval falls in one of the two halves of $[0, M)$ (i.e. cases 1 or 2 above). If we suppose to continue and operate on the interval $[\frac{M}{4}, \frac{3M}{4})$ as we did with $[0, M)$, by properly rewriting conditions 1 and 2, the interval size can fall below $\frac{M}{8}$ and thus the same problem arises again. The solution is to use a parameter m that records the number of times that the underflow condition occurred, so that the current interval is within $[\frac{M}{2} - \frac{M}{2^{m+1}}, \frac{M}{2} + \frac{M}{2^{m+1}})$; and observe that, when eventually the interval will not include $\frac{M}{2}$, we will output 01^m if it is in the first half, or 10^m if it is in the second half. After that, we can expand the interval around its halfway point and count the number of expansions:

- mathematically, if $\frac{M}{4} \leq L_i < \frac{M}{2} < H_i \leq \frac{3M}{4}$ then we increment the number m of underflows and consider the new interval

$$[L_{i+1}, H_{i+1}) = \left[2 \cdot \left(L_i - \frac{M}{4}\right), 2 \cdot \left(H_i - 1 - \frac{M}{4}\right) + 2\right)$$

- when expansion 1 or 2 are operated, after the output of the bit, we output also m copies of the complement of that bit, and reset m to 0.

End of the input sequence. At the end of the input sequence, because of the expansions, the current interval satisfies at least one of the inequalities:

$$L_n < \frac{M}{4} < \frac{M}{2} < H_n \quad \text{or} \quad L_n < \frac{M}{2} < \frac{3M}{4} < H_n \tag{10.4}$$

It may be the case that $m > 0$ so that we have to complete the the output bit stream as follows:

- if the first inequality holds, we can emit $01^m1 = 01^{m+1}$ (if $m = 0$ this means to codify $\frac{M}{4}$)
- if the second inequality holds, we can emit $10^m0 = 10^{m+1}$ (if $m = 0$ this means to codify $\frac{3M}{4}$)

Decompression stage. The decoder must mimic the computations operated during the compression stage. It maintains a shift register v of $\lceil \log_2 M \rceil$ bits, which plays the role of x (in the classic Arithmetic coding) and thus it is used to find the next subinterval from the partition of the current interval. When the interval is expanded, v is modified accordingly, and a new bit from the compressed stream is loaded through the function `next_bit`:

1. $[L_i, H_i) \subseteq \left[0, \frac{M}{2}\right) \rightarrow$ consider the new interval

$$[L_{i+1}, H_{i+1}) = [2 \cdot L_i, 2 \cdot (H_i - 1) + 2), \quad v = 2v + \texttt{next_bit}$$

2. $[L_i, H_i) \subseteq \left[\frac{M}{2}, M\right) \rightarrow$ consider the new interval

$$[L_{i+1}, H_{i+1}) = \left[2 \cdot \left(L_i - \frac{M}{2}\right), 2 \cdot \left(H_i - 1 - \frac{M}{2}\right) + 2\right),$$

$$v = 2 \cdot \left(v - \frac{M}{2}\right) + \texttt{next_bit}$$

3. if $\frac{M}{4} \leq L_i < \frac{M}{2} < H_i \leq \frac{3M}{4}$ consider the new interval

$$[L_{i+1}, H_{i+1}) = \left[2 \cdot \left(L_i - \frac{M}{4}\right), 2 \cdot \left(H_i - 1 - \frac{M}{4}\right) + 2\right),$$

$$v = 2 \cdot \left(v - \frac{M}{4}\right) + \texttt{next_bit}$$

4. otherwise it is $H_i - L_i \geq \frac{M}{4} + 2$ and thus we can continue the decompression as is.

In order to understand this decoding process, let us resume the example of the previous sections with the same input sequence $S = abac$ of length $n = 4$, ordered alphabet $\Sigma = \{a, b, c\}$, probabilities $P[a] = \frac{1}{2}$, $P[b] = P[c] = \frac{1}{4}$ and cumulative probabilities $f_a = 0$, $f_b = P[a] = \frac{1}{2}$, and $f_c = P[a] + P[b] = \frac{3}{4}$. We rewrite these probabilities by using the approximations that we have seen

above, hence $C[|\Sigma|+1]=4$, and we set the initial interval as $[L_0, H_0) = [0, M)$, where M is chosen to satisfy the inequality 10.3:

$$C[|\Sigma|+1] \leq \frac{M}{4}+2 \Longleftrightarrow 4 \leq \frac{M}{4}+2$$

so we can take $M = 16$ and have $\frac{M}{4} = 4$, $\frac{M}{2} = 8$ and $\frac{3M}{4} = 12$ (of course, this value of M is not based on the real machine word length but it is useful for our example). At this point, we have the initial interval

$$[L_0, H_0) = [0, 16)$$

and we are ready to compress the first symbol $S[1] = a$ using the expressions for the endpoints seen above:

$$L_1 = L_0 + \lfloor f_a \cdot (H_0 - L_0) \rfloor = 0 + \lfloor 0 \cdot 16 \rfloor = 0$$

$$H_1 = L_1 + \lfloor P[a] \cdot (H_0 - L_0) \rfloor = 0 + \left\lfloor \frac{2}{4} \cdot 16 \right\rfloor = 8$$

The new interval $[L_1, H_1) = [0, 8)$ satisfies the first expansion rule $[L_1, H_1) \subseteq \left[0, \frac{M}{2}\right)$, hence we output '0' and we consider the new interval (for the expansion rules we do not change index):

$$[L_1, H_1) = [2 \cdot L_1, 2 \cdot (H_1 - 1) + 2) = [0, 16)$$

In the second iteration we consider the second symbol $S[2] = b$, and the endpoints of the new interval are:

$$L_2 = L_1 + \lfloor f_b \cdot (H_1 - L_1) \rfloor = 8$$

$$H_2 = L_2 + \lfloor P[b] \cdot (H_1 - L_1) \rfloor = 12$$

This interval satisfies the second expansion rule $[L_2, H_2) \subseteq \left[\frac{M}{2}, M\right)$, hence we concatenate at the output the bit '1' (obtaining 01) and we consider the interval

$$[L_2, H_2) = \left[2 \cdot \left(L_2 - \frac{M}{2}\right), 2 \cdot \left(H_2 - 1 - \frac{M}{2}\right) + 2\right) = [0, 8)$$

that satisfies again one of the expansion rules, hence we apply the first rule and we obtain the result $output = 010$ and:

$$[L_2, H_2) = [2 \cdot L_1, 2 \cdot (H_2 - 1) + 2) = [0, 16)$$

For the third symbol $S[3] = a$, we obtain

$$L_3 = L_2 + \lfloor f_a \cdot (H_2 - L_2) \rfloor = 0$$

$$H_3 = L_3 + \lfloor P[a] \cdot (H_2 - L_2) \rfloor = 8$$

and this interval satisfies the first rule, hence $output = 0100$ and

$$[L_3, H_3) = [2 \cdot L_3, 2 \cdot (H_3 - 1) + 2) = [0, 16)$$

We continue this way for the last symbol $S[4] = c$:

$$[L_4, H_4) = [L_3 + \lfloor f_c \cdot (H_3 - L_3) \rfloor, L_4 + \lfloor P[c] \cdot (H_3 - L_3) \rfloor) = [12, 16)$$

so get the output sequence $output = 01001$ and

$$[L_4, H_4) = \left[2 \cdot \left(L_4 - \frac{M}{2}\right), 2 \cdot \left(H_4 - 1 - \frac{M}{2}\right) + 2\right) = [8, 16)$$

so we expand again and get the output sequence $output = 010011$ and

$$[L_4, H_4) = \left[2 \cdot \left(L_4 - \frac{M}{2}\right), 2 \cdot \left(H_4 - 1 - \frac{M}{2}\right) + 2\right) = [0, 16)$$

At the end of the input sequence the last interval should satisfy the Equ. 10.4. This is true for our interval, hence, as we know, at this point the encoder sends the pair $\langle 010011_2, 4\rangle$ and the statistical model $P[a] = \frac{1}{2}$, $P[b] = P[c] = \frac{1}{4}$ to the decoder (we are assuming a static model).

As far as the decoding stage is concerned, the first step initializes the shift register v (of length $\lceil \log_2 M \rceil = \lceil \log_2 16 \rceil = 4$) with the first $\lceil \log_2 16 \rceil = 4$ bits of the compressed sequence, hence $v = 0100_2 = 4_{10}$. At this point the initial interval $[L_0, H_0) = [0, 16)$ is subdivided in three different subintervals, one for every symbol in the alphabet, according to the probabilities: $[0, 8)$, $[8, 12)$ and $[12, 16)$. The symbol generated at the first iteration will be a because $v = 4 \in [0, 8)$. At this point we apply the first expansion rule of the decompression process because $[L_1, H_1) = [0, 8) \subseteq [0, \frac{M}{2})$, obtaining:

$$[L_1, H_1) = [2 \cdot L_1, 2 \cdot (H_1 - 1) + 2) = [0, 16)$$

$$v = 2 \cdot v + \texttt{next_bit} = \texttt{shift}_{sx}(0100_2) + 1_2 = 1000_2 + 1_2 = 1001_2 = 9_{10}$$

In the second iteration the interval $[0, 16)$ is subdivided another time in the same ranges of integers and the generated symbol will be b because $v = 9 \in [8, 12)$. This last interval satisfies the second rule, hence:

$$[L_2, H_2) = \left[2 \cdot \left(L_2 - \frac{M}{2}\right), 2 \cdot \left(H_2 - 1 - \frac{M}{2}\right) + 2\right) = [0, 8)$$

$$v = 2 \cdot \left(v - \frac{M}{2}\right) + \texttt{next_bit} = \texttt{shift}_{sx}(1001_2 - 1000_2) + 1_2 = 0010_2 + 1_2 = 0011_2 = 3_{10}$$

We apply now the first rule (the function next_bit returns '0' if there are not more bits in the compressed sequence):

$$[L_2, H_2) = [2 \cdot L_2, 2 \cdot (H_2 - 1) + 2) = [0, 16)$$

$$v = 2 \cdot v + \texttt{next_bit} = \texttt{shift}_{sx}(0011_2) + 0_2 = 0110_2 = 6_{10}$$

This interval is subdivided again, in the third iteration, obtaining the subintervals $[0, 8)$, $[8, 12)$ and $[12, 16)$. Like in the previous steps, we find a as generated symbol because $v = 6 \in [0, 8)$, and we modify the interval $[L_3, H_3) = [0, 8)$ and the shift register v as the first rule specifies:

$$[L_3, H_3) = [2 \cdot L_3, 2 \cdot (H_3 - 1) + 2) = [0, 16)$$

$$v = 2 \cdot v + \texttt{next_bit} = \texttt{shift}_{sx}(0110_2) + 0_2 = 1100_2 = 12_{10}$$

The last generated symbol will be c because $v = 12 \in [12, 16)$, and the entire input sequence is exactly reconstructed. The algorithm can stop because it has generated 4 symbols, which was provided as input to the decoder as length of S.

10.3 Prediction by Partial Matching$^{\infty}$

In order to improve compression we need better models for the symbol probabilities. A typical approach consists of estimating them by considering not just individual symbols, and thus assume that they occur independently of each other, but evaluating the *conditional probability* of their occurrence in S, given few previous symbols, the so called *context*. In this section we will look at a particular *adaptive* technique to build a context–model that can be combined very well with Arithmetic Coding because it generates skewed probabilities and thus high compression. This method

is called *Prediction by Partial Matching* (shortly, PPM), it allows to move from 0-th order entropy coders to k-th order entropy coders. The implementation of PPM suffers two problems: (i) they need proper data structures to maintain updated in an efficient manner all conditional probabilities, as S is scanned; (ii) at the beginning the estimates of the context-based probabilities are poor thus producing an inefficient coding; so proper adjustments have to be imposed in order to quicker establish good statistics. In the rest of the section we will concentrate on the second issue, and refer the reader to the literature for the first one, namely [4, 7].

10.3.1 The algorithm

Let us dig into the algorithmic structure of PPM. It uses a suite of finite contexts in order to predict the next symbol. Frequency counts of these contexts are updated at each input symbol, namely PPM keeps counts for each symbol σ and for each context α of length ℓ, of how many times the string $\alpha\,\sigma$ occurs in the prefix of S processed so far. This way, at step i, PPM updates the counts for $\sigma = S[i]$ and its previous contexts $\alpha = S[i-\ell, i-1]$, for any $\ell = 0, 1, \ldots, K$ where K is the maximum context-model admitted.

In order to encode the symbol $S[i]$, PPM starts from the longest possible context of length K, namely $S[i-K, i-1]$, and then switches to shorter and shorter contexts until it finds the one, say $S[i-\ell, i-1]$, which is able to predict the current symbol. This means that PPM has a counting for $S[i]$ in $S[i-\ell, i-1]$ which is strictly larger than 0, and thus the estimated probability for this occurrence is not zero. Eventually it reaches the context of order -1 which corresponds to a model in which all symbols have the same probabilities. The key compression issue is how to encode the length ℓ of the model adopted to compress $S[i]$, so that also the decoder can use that context in the decompression phase. We could use an integer encoder, of course, but this would take an integral number of bits per symbol, thus vanishing all efforts of a good modeling; we need something smarter.

The algorithmic idea is to turn this problem into a symbol-encoding problem, by introducing an *escape* symbol (`esc`) which is emitted every time a context-switch has to be performed. So `esc` signals to the decoder to switch to the next shorter model. This escaping-process continues till a model where the symbol is not novel is reached. If the current symbol has never occurred before, $K+1$ escape symbols are transmitted to make the decoder to switch to the (-1)–order model that predicts all possible symbols according to the uniform distribution. Note that the 0–order context corresponds to a distribution of probabilities estimated by the frequency counts, just as in the classic Arithmetic coding. Using such strategy, the probability associated to a symbol is always the one that has been calculated in the longest context where the symbol has previously occurred; so it should be more precise than just the probabilities based on individual counts for the symbols in Σ.

We notice that at the beginning, some contexts may be missing; in particular, when $i \leq K$, the encoder and the decoder do not use the context with maximum order K, but the context of length $(i-1)$, hence the one read so far. When the second symbol is used, the 0–order context is used and, if the second symbol is novel, an `esc` is emitted. The longest context K is used when $i > K$: in this case K symbols have been read already, and the K–order context is available. To better understand how PPM works, we consider the following example.

Let the input sequence S be the string *abracadabra* and let $K = 2$ be the longest context used in the calculation of K–order models and its *conditional probabilities*. As previously said, the only model available when the algorithm starts is the (-1)–order model. So when the first symbol a is read, the (-1)–order assigns to it the uniform probability $\frac{1}{|\Sigma|} = \frac{1}{5}$. At the same time, PPM updates frequency counts in the 0–order model assigning a probability $P[a] = \frac{1}{2}$ and $P[\texttt{esc}] = \frac{1}{2}$. In this running example we assume that the escape symbol is given a count equal to the total number of different characters in the model. Other strategies to assign a probability to the escape symbol will be discussed in detail in section 10.3.3.

PPM then reads b and uses the 0–order model, which is currently the longest one, as explained above. An escape symbol is transmitted since b has never been read before. The (-1)–order model is so used to compress b and then both the 1–order and the 0–order models are updated. In the 0–order model we have $P[a] = \frac{1}{4}$, $P[b] = \frac{1}{4}$, $P[\texttt{esc}] = \frac{2}{4} = \frac{1}{2}$ (two distinct symbols have been read). In the 1–order model the probabilities are $P[b|a] = \frac{1}{2}$ and $P[\texttt{esc}|a] = \frac{1}{2}$ (only one distinct symbol is read). Now let us suppose that the current symbol is $S[6] = d$. Since it is the first occurrence of d in S, three escape symbols will be transmitted for switching from the 2–order to the (-1)–order model. Table 10.1 shows the predictions of the four models after that PPM, in its version C (see Sec. 10.3.3), has completed the processing of the input sequence.

We remark that PPM offers probability estimates which can then be used by an Arithmetic coder to compress a sequence of symbols. The idea is to use every model (either the one which offers a successful prediction or the one which leads to emit an `esc`) to partition the current interval of the Arithmetic coding stage and the current symbol to select the next sub-interval. This is the reason why PPM is better looked as a context-modeling technique rather than a compressor, the coding stage could be implemented via Arithmetic coding or any other statistical encoder.

Order $k = 2$			Order $k = 1$			Order $k = 0$			Order $k = -1$		
Predictions	c	p	Predictions	c	p	Predictions	c	p	Predictions	c	p
ab → r	2	$\frac{2}{3}$	a → b	2	$\frac{2}{7}$	→ a	5	$\frac{5}{16}$	→ $\forall\sigma[i]$	1	$\frac{1}{\|\Sigma\|} = \frac{1}{5}$
→ esc	1	$\frac{1}{3}$	→ c	1	$\frac{1}{7}$	→ b	2	$\frac{2}{16}$			
			→ d	1	$\frac{1}{7}$	→ c	1	$\frac{1}{16}$			
ac → a	1	$\frac{1}{2}$	→ esc	3	$\frac{2}{7}$	→ d	1	$\frac{1}{16}$			
→ esc	1	$\frac{1}{2}$				→ r	2	$\frac{2}{16}$			
			b → r	2	$\frac{2}{3}$	→ esc	5	$\frac{5}{16}$			
ad → a	1	$\frac{1}{2}$	→ esc	1	$\frac{1}{3}$						
→ esc	1	$\frac{1}{2}$									
			c → a	1	$\frac{1}{2}$						
br → a	2	$\frac{2}{3}$	→ esc	1	$\frac{1}{2}$						
→ esc	1	$\frac{1}{3}$									
			d → a	1	$\frac{1}{2}$						
ca → d	1	$\frac{1}{2}$	→ esc	1	$\frac{1}{2}$						
→ esc	1	$\frac{1}{2}$									
			r → a	2	$\frac{2}{3}$						
da → b	1	$\frac{1}{2}$	→ esc	1	$\frac{1}{3}$						
→ esc	1	$\frac{1}{2}$									
ra → c	1	$\frac{1}{2}$									
→ esc	1	$\frac{1}{2}$									

TABLE 10.1 Version C of PPM models after processing the whole $S = abracadabra$.

10.3.2 The principle of exclusion

It is possible to use the knowledge about symbol frequencies in k-order models to improve the compression rate when switching through escape symbols to low-order ones. Suppose that the whole input sequence S of the example above has been processed and that the following symbol

to be encoded is c. The prediction $ra \rightarrow c$ is used (since ra is the current 2–order context, with $K = 2$) and thus c is encoded with a probability of $\frac{1}{2}$ using 1 bit. Suppose now that, instead of c, the character d follows *abracadabra*. In the context ra an escape symbol is transmitted (encoded with a $\frac{1}{2}$ probability) and PPM switches to the 1–order model. In this model, the prediction $a \rightarrow d$ can be used and d can be encoded with probability $\frac{1}{7}$. This prediction takes into account the fact that, earlier in the processing, the symbol d has followed a. However the fact that ra was seen before followed by c implies that the current symbol cannot be c. In fact, if it were followed by c, PPM would not have switched to the 1–order model and would have used the longest context ra. The decoder can so *exclude* the case of an a followed by a c. This reduces the frequency count of the group $a \rightarrow$ by one and the character d could thus be encoded with probability $\frac{1}{6}$.

Suppose now that after *abracadabra* the novel symbol e occurs. In this case a sequence of escape symbols is transmitted to make the decoder switch to the -1–order model. Without *exclusion* the novel symbol would be encoded with a probability of $\frac{1}{|\Sigma|}$. However, if PPM is now using the (-1)–order model, it means that none of the previously seen symbols is the current one. The symbols already read can thus be excluded in the computation of the prediction since it is impossible that they are occurring at this point of the input scan. The probability assigned to the novel symbol by the (-1)–order model using the *exclusion principle* is $\frac{1}{|\Sigma|-q}$ where q is the total number of distinct symbols read in S. Performing this technique takes a little extra time but gives a reasonable payback in terms of extra compression, because all nonexcluded symbols have their probability increased.

10.3.3 Zero Frequency Problem

It may seem that the performance of PPM should improve when the length of the maximum context is increased. Fig 10.11 shows an experimental evidence of this intuition.

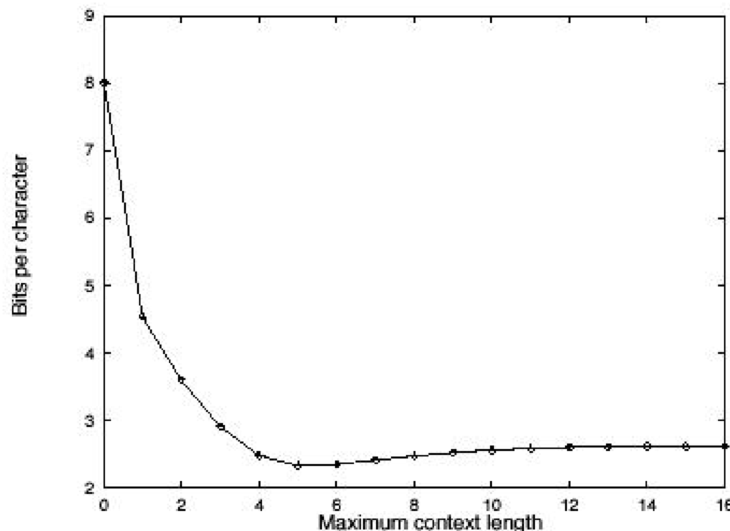


FIGURE 10.11: PPM compression ratio on increasing context length. Computed on Thomas Hardy's text *Far from the Madding Crowd*.

We notice that with contexts longer than 4–5 characters there is no improvement. The reason is that with longer contexts, there is a greater probability to use the escape mechanism, transmitting as many escape symbols as it is needed to reach the context length for which non-null predictions are available.

Anyway, whichever is the context length used, the choice of the encoding method for the escape symbol is very important. There is no sound theoretical basis for any particular choice to assign probabilities to the escape symbol if no a priori knowledge is available. A method is chosen according to the programmer experience only. The various implementations of PPM described below are identified by the escape method they use. For example, PPMA stands for PPM with escape method A. Likewise, PPMC or PPMD use escape methods C and D. The maximum order of the method can be included too. For example, PPMC5 stands for PPM with escape method C and maximum order-model $K = 5$. In the following, Cn will denote a context, σ a symbol in the input alphabet, $c(\sigma)$ the number of times that the symbol σ has occurred in context Cn, n the number of times the current context Cn has occurred, q the total number of distinct symbols read.

Method A [1, 6]. This technique allocates one count to the possibility that a symbol will occur in a context in which it has not been read before. The probability $P[\sigma]$ that σ will occur again in the same context is estimated by $P[\sigma] = \frac{c(\sigma)}{1+n}$. The probability that a novel symbol will occur in Cn, i.e. the escape probability $P[\mathtt{esc}]$, is then:

$$\begin{aligned} P[\mathtt{esc}] &= 1 - \sum_{\sigma\in\Sigma, c(\sigma)>0} P[\sigma] \;=\; 1 - \sum_{\sigma\in\Sigma, c(\sigma)>0} \frac{c(\sigma)}{1+n} \\ &= 1 - \frac{1}{1+n} \sum_{\sigma\in\Sigma, c(\sigma)>0} c(\sigma) \;=\; 1 - \frac{n}{1+n} = \frac{1}{1+n} \end{aligned}$$

`Method B [1, 6].` The second technique classifies a symbol as novel unless it has already occurred twice. The motivation is that a symbol that has occurred only once can be an anomaly. The probability of a symbol is thus estimated by $P[\sigma] = \frac{c(\sigma)-1}{n}$, and q is set as the number of distinct symbols seen so far in the input sequence. The escape probability now is:

$$\begin{aligned} P[\mathtt{esc}] &= 1 - \sum_{\sigma\in\Sigma, c(\sigma)>0} P[\sigma] \\ &= 1 - \sum_{\sigma\in\Sigma, c(\sigma)>0} \frac{c(\sigma)-1}{n} \\ &= 1 - \frac{1}{n} \sum_{\sigma\in\Sigma, c(\sigma)>0} (c(\sigma)-1) \\ &= 1 - \frac{1}{n} \left(\sum_{\sigma\in\Sigma, c(\sigma)>0} c(\sigma) - \sum_{\sigma\in\Sigma, c(\sigma)>0} 1 \right) \\ &= 1 - \frac{1}{n}(n-q) = \frac{q}{n} \end{aligned}$$

Method C [4]. It is an hybrid between the previous two methods. When a novel symbol occurs, a count of 1 is added both to the escape count and to the new symbol count. In this way the total count increases by 2. Thus it estimates the probability of a symbol σ as $P[\sigma] = \frac{c(\sigma)}{n+q}$ and the escape probability $P[\mathtt{esc}]$ as:

$$P[\mathtt{esc}] = 1 - \sum_{\sigma\in\Sigma, c(\sigma)>0} \frac{c(\sigma)}{n+q} = \frac{q}{n+q}$$

Method D. It is a minor modification of method C. It treats in a more uniform way the occurrence of a novel symbol: instead of adding 1, it adds $\frac{1}{2}$ both to the escape and to the new symbol. The probability of a symbol is then:

$$P[\sigma] = \frac{c(\sigma) - \frac{1}{2}}{n} = \left(\frac{2 \cdot c(\sigma) - 1}{2 \cdot n}\right)$$

The probability of the escape symbol is:

$$\begin{aligned}
P[\mathtt{esc}] &= 1 - \sum_{\sigma\in\Sigma, c(\sigma)>0} P[\sigma] \\
&= 1 - \sum_{\sigma\in\Sigma, c(\sigma)>0} \left(\frac{c(\sigma) - \frac{1}{2}}{n}\right) \\
&= 1 - \frac{1}{n}\left(\sum_{\sigma\in\Sigma, c(\sigma)>0} c(\sigma) - \sum_{\sigma\in\Sigma, c(\sigma)>0} \frac{1}{2}\right) \\
&= 1 - \frac{n}{n} + \frac{1}{n}\overbrace{\sum_{\sigma\in\Sigma, c(\sigma)>0} \frac{1}{2}}^{\frac{q}{2}} \\
&= \frac{q}{2n}
\end{aligned}$$

Method P [6]. The method is based on the assumption that symbols appear according to a Poisson process. Under such hypothesis, it is possible to extrapolate probabilities from an N symbol sample to a larger sample of size $N' = (1 + \theta)N$, where $\theta > 0$. Denoting with t_i the number of distinct symbols occurred exactly i times in the sample of size N, the number of novel symbols can be approximated by $t_1\theta - t_2\theta^2 + t_3\theta^3 - \cdots$. The probability that the next symbol will be novel equals the number of expected new symbols when $N' = N + 1$. In this case:

$$P[\mathtt{esc}] = t_1 \frac{1}{n} - t_2 \frac{1}{n^2} + t_3 \frac{1}{n^3} - \cdots$$

Method X [6]. This method approximates method P computing only the first term of the series since in most cases n is very large and t_i decreases rapidly as i increases: $P[\mathtt{esc}] = \frac{t_1}{n}$. The previous formula may also be interpreted as approximating the escape probability by counting the symbols that have occurred only once.

Method XC [6]. Both methods P and X break down when either $t_1 = 0$ or $t_1 = n$. In those cases the probability assigned to the novel symbol is, respectively, 0 or 1. To overcome this problem, method XC uses method C when method X breaks down:

$$P[\mathtt{esc}] = \begin{cases} \frac{t_1}{n} & 0 < t_1 < n \\ \frac{q}{n+q} & \text{otherwise} \end{cases}$$

Method X1. The last method described here is a simple variant of method X that avoids breaking down to a probability of 0 or 1. Instead of using two different methods, it adds 1 to the total count. The probability of a generic symbol σ is $P[\sigma] = \frac{c(\sigma)}{n+t_1+1}$, and the probability of the escape symbol is $P[\mathtt{esc}] = \frac{t_1+1}{n+t_1+1}$.

References

[1] John G. Clearly and Ian H. Witten. Data Compression Using Adaptive Coding and Partial String Matching. *IEEE Transactions on Communications*, 32:396–402, 1984.

[2] Peter M. Fenwick. A new data structure for cumulative frequency tables. *Software Practice and Experience*, 327–336, 1994.

[3] Paul Howard and Jeffrey S. Vitter. Arithmetic Coding for Data Compression. *Proceedings of the IEEE*, 857–865, 1994.

[4] Alistair Moffat. Implementing the PPM Data Compression Scheme. *IEEE Transactions on Communications*, 38:1917–1921, 1990.

[5] G. Nigel and N. Martin. Range Encoding: an algorithm for removing redundancy from a digitised message. Presented in March 1979 to the *Video & Data Recording Conference*, 1979.

[6] Ian H. Witten and Timoty C. Bell. The Zero-Frequency Problem: Estimating the Probabilities of Novel Events in Adaptive Text Compression. *IEEE Transactions on Information Theory*, 37:1085–1094, 1991.

[7] Ian H. Witten, Alistair Moffat, Timoty C. Bell. *Managing Gigabytes*. Morgan Kauffman, second edition, 1999.

[8] Ian H. Witten, Radford M. Neal and John G. Clearly. Arithmetic Coding for data compression. *Communications of the ACM*, 520-540, 1987.