More on Huffman's Algorithm, Entropy, and Arithmetic Coding

We provide additional details on the analysis of Huffman's algorithm given in the textbook, we give a lower bound to the encoding length obtained by Huffman's algorithm as a function of the frequencies, and we prove an asymptotically tight upper bound by analysing arithmetic coding.

1 Huffman's algorithm's optimality

Given an "alphabet" of symbols V and a positive integer frequency $f(v)$ for every $v \in V$, we want to find a prefix-free way of mapping each symbol $v \in V$ to a bit string $C(v)$ of length $\ell(v)$, in such a way that we minimize the total encoding length

$$\sum_{v \in V} f(v) \cdot \ell(v)$$

A bit string s is a prefix of a bit string s' if s' is obtained by adding one or more bits at the end of s. For example 010 is a prefix of 01001. A mapping is prefix-free if all the encodings $C(v)$ are different and none of them is a prefix of any other.

For example if $V = \{a, b, c, d\}$, then

$$\begin{aligned} C(a) &= 0 \\ C(b) &= 10 \\ C(c) &= 110 \\ C(d) &= 111 \end{aligned}$$

is a prefix-free encoding, while

$$\begin{aligned} C(a) &= 0 \\ C(b) &= 10 \\ C(c) &= 101 \\ C(d) &= 111 \end{aligned}$$

is not prefix free because the encoding of b is a prefix of the encoding of c.

If all the encodings have the same length, then the mapping is always prefix-free. In a fixed-length mapping, the length of the encoding of any element of V must be at least $\log_2 V$, because if we use encodings of length ℓ we can represent at most 2^ℓ symbols. This means that the total encoding length achieved with fixed-length mappings is at least

$$\lceil \log_2 |V| \rceil \cdot \sum_v f(v)$$

With variable-length prefix-free mappings it is often possible to do better. As said above, we are interested in finding a prefix-free mapping that minimizes the total encoding length.

A prefix-free encoding of V can be represented as a binary tree in which each leaf is associated with an element of V. Each node of the binary tree is labeled by a binary string: the root is labeled with the empty string, and if s is the label of a node, then its left child, if it exists, is labeled $s0$, and the right child, if it exists, is labeled $s1$. The mapping defined by such a binary tree is to map each element $v \in V$ to the label of the leaf associated to v. Note that labelling is such that if a binary string s is a prefix of a binary string s', then the node labeled s' is a descendant of the node labeled s, so by associating elements of V to leaves we are guaranteed the prefix-free property, because leaves have no descendants.

We first note that, in an optimal solution represented as a binary tree, every non-leaf node has precisely two children. This is because if we had a node x with exactly one child y, we could "merge" x and y, and all the symbols that are descendants of y would have encodings one bit shorter than before, meaning that the original tree could not be an optimal solution.

Our starting point in reasoning about optimality is the following lemma.

Lemma 1 *Let V be a set of symbols, $f(v)$ the frequency of symbol $f(v)$, and let $v_1, \ldots, v_n$ be a sorting of V such that $f(v_1) \leq f(v_2) \leq f(v_3) \leq \cdots \leq f(v_n)$, where $n = |V|$.*

Then there is an optimal solution to the prefix-free encoding problem in which v_1 has an encoding at least as long as that of any other symbol.

PROOF: Consider any optimal solution C^*, in which each symbol v has length $\ell(v)$. If v_1 is associated to a deepest leaf of C^* there is nothing to prove. Otherwise, let x be a symbol associated to a deepest leaf of C^*, and consider a new mapping C' that is identical to C^* except that we switched the encodings of x and of v_1, and let us call $\ell'(v)$ the length of the encoding of each symbol v according to this new encoding.

Calling $cost(C)$ the total encoding length of a mapping C, the difference between the

total encoding length in C^* and the total encoding length in C' is

$$\begin{aligned} cost(C^*) - cost(C') &= \left(\sum_{v \in V} \ell(v) f(v)\right) - \left(\sum_{v \in V} \ell(v)' f(v)\right) \\ &= \ell(v_1) f(v_1) + \ell(x) f(x) - \ell'(v_1) f(v_1) - \ell'(x) f(x) \\ &= \ell(v_1) f(v_1) + \ell(x) f(x) - \ell(x) f(v_1) - \ell(v_1) f(x) \\ &= (\ell(x) - \ell(v_1)) \cdot (f(x) - f(v_1)) \\ &\geq 0 \end{aligned}$$

where we used the fact that the old and new encodings of all symbols are the same, except for v_1 and x, then we used the fact that the encoding lengths of x and v_1 are switched in the new encodings, then we used the distributivity property, and finally we used that $\ell(x) \geq \ell(v_1)$ and $f(v_1) \leq f(x)$ because x was a symbol with longest encoding and v_1 was defined as a symbol of smallest frequency.

This means that $cost(C') \leq cost(C^*)$, but C^* was assumed to be optimal so C' is also optimal and we conclude that there is an optimal solution in which v_1 is a symbol with longest encoding. □

Lemma 2 *Let V be a set of symbols, $f(v)$ the frequency of symbol $f(v)$, and let $v_1, \ldots, v_n$ be a sorting of V such that $f(v_1) \leq f(v_2) \leq f(v_3) \leq \cdots \leq f(v_n)$, where $n = |V|$.*

Then there is an optimal solution to the prefix-free encoding problem in which v_1 and v_2 are siblings and have an encoding at least as long as that of any other symbol.

PROOF: From the previous lemma we know that there is an optimal solution C' such that v_1 is associated to a deepest leaf in the binary tree representation of the encoding C'. Since all internal nodes must have exactly two children, the leaf associated to v_1 must have a sibling, and that sibling must also be a leaf (otherwise there would be a symbol with a longer encoding than v_1). If the symbol associated to that sibling leaf is v_2 then we are done. Suppose, otherwise, that it is some other symbol x. Then we have $f(v_2) \leq f(x)$ because of the definition of the sorted order and $\ell(x) \geq \ell(v_2)$ because $\ell(x) = \ell(v_1)$ and v_1 is a symbol with longest encoding in C'. So we can switch x and v_2 and, via the same reasoning of the previous lemma, observe that the solution C'' that we obtain has total encoding length smaller than or equal to the total encoding length of C', so that C'' is also an optimal solution, and v_1 and v_2 are siblings in C'', and are symbols of biggest encoding length. □

Now comes the main observation: if we are given an instance $V, f(\cdot)$ of the minimum-total-length prefix-free encoding problem, and $v_1, v_2, \ldots, v_n$ is a sorted order of V according to f, we can, without loss of generality, optimize among solutions in which

v_1 and v_2 are siblings. Now, the problem of finding the optimal encoding in which v_1 and v_2 are siblings can be transformed into an equivalent encoding problem over $n-1$ symbols. Once we explain how this works, it will become clear that we can repeat this transformation several times until we are left with a problem involving only two symbols, which is always optimally solved by encoding one symbol as 0 and the other as 1.

The transformation is as follows: we take out v_1 and v_2 from V, and replace them with a new symbol z of frequency $f(z) = f(v_1) + f(v_2)$. We do not make any change to the other symbols. The claim is that if C' is an optimal mapping for this new instance over the symbols $z, v_3, \ldots, v_m$, and if we define a mapping C such that $C(v_1) = C'(z)0$, $C(v_2) = C'(z)1$, $C(v_i) = C(v_i)$ for $i = 3, \ldots, n$, then C is optimal for our original problem.

First of all, note that C is a valid prefix-free encoding of our original symbol set, and if let $cost(\cdot)$ denote the total encoding length of a mapping we have

$$cost(C) = cost(C') + f(v_1) + f(v_2)$$

Now consider an optimal solution C^* for our original symbol set, such that v_1 and v_2 are siblings in the binary tree defined by C^* (by our previous lemma, we can assume that such a C^* exists). Define a possible solution C'' for the new problem, by letting $C''(z)$ be equal to the string associated to the parent of v_1 and v_2, and letting C'' be equal to C on $v_3, \ldots, v_n$. Then

$$cost(C'') = cost(C^*) - f(v_1) - f(v_2)$$

Putting things together we have

$$cost(C) = cost(C') + f(v_1) + f(v_2) \leq cost(C'') + f(v_1) + f(v_2) = cost(C^*)$$

where we used the fact that C' is optimal for the new problem. This means that

$$cost(C) \leq cost(C^*)$$

where C^* is optimal for the original problem, and so C is also optimal for the original problem, as claimed.

The algorithm at the end of section 5.2 of the book iteratively applies this transformation, and finds an optimal solution in time $O(n \log n)$.

2 Lower Bounds

We will make use of the following simple observation: the number of distinct bit strings of length L is 2^L, and the number of distinct bit strings of length $\leq L$ is

$$2^1 + 2^2 + \cdots + 2^L = 2^{L+1} - 2 < 2^{L+1}$$

This means, for example, that if we have an alphabet V that contains 2^k symbols, and we want to find a compression scheme for sequences of n elements of V, then, no matter what encoding scheme we develop, there will be sequences in V^n that map to bit strings of length $\geq kn$, because the number of bit strings of length $\leq kn-1$ is $2^{(kn-1)+1} - 2 < 2^{kn} = |V^n|$, and so we do not have enough bit strings of length $\leq kn-1$ to provide distinct encodings for all sequences in V^n. On the other hand, we can construct a simple fixed-length encoding of the elements of V using a k-bit string for every element of V, and so it is easy to encode every sequence in V^n using exactly kn bits.

Thus, from the point of view of worst-case encoding length, fixed-length encodings are optimal, and an analysis of compressibility needs to be done from the point of view of the average-case encoding length with respect to certain distributions of data, or by making some assumptions on the frequencies with which the elements of V appear in the sequence.

Even with an average-case analysis, however, compressibility is not always possible. In particular, random sequences with the uniform distribution are essentially incompressibile.

Theorem 3 (Incompressibility of uniformly random sequences) *Let V be a finite alphabet containing 2^k elements for some positive integer k. Let Enc be an arbitrary compression algorithm mapping sequences of elements of V to bit strings. Let n be any positive integer. Consider the uniform distribution of random sequences $x \in V^n$, that is, the distribution in which all sequences have the same probability $1/|V^n|$. Then*

$$\mathbb{P}[\textit{ length of } Enc(x) \geq kn - 10] \geq 0.999$$

PROOF: The number of sequences that can be encoded to binary strings of length $< kn - 10$ (that is, of length $\leq kn - 11$) is at most the number of distinct binary strings of length $\leq kn - 11$, which is equal to $2^{(n-11)+1} - 2 < 2^{n-10}$. Each of those sequence has probability $1/|V^k| = 1/2^{kn}$ of being generated, so the probability of generating such a string is at most

$$\frac{2^{kn-10}}{2^{kn}} = \frac{1}{2^{10}} < 0.001$$

□

In general, the probability that a random sequence is encoded to a length $\geq kn - t$ is at least $1 - 2^{-t}$.

In most interesting cases, however, we are dealing with data that is not uniformly distributed. Letters in languages, for example, occur with different frequencies, and pairs and triples of letters have even more non-uniform distributions. In such cases,

algorithms like Huffman's algorithm are able to substantially compress the data, and we will now provide some rigorous results bounding how significant this compression can be.

Suppose that we run Huffman's algorithm on a sequence s of n elements from an alphabet V. Call K the number of elements of V, which we do not assume any more to be a power of 2, and call them $v_1, \ldots, v_K$. Let us call $p_i n$ the frequency of the letter v_i in the sequence s. If Huffman's algorithm assigns, in a prefix-free way, a bit-string encoding of length ℓ_i to each letter v_i, then s is encoded to a bit string of length

$$L = \sum_{i=1}^{K} \ell_i p_i n$$

We now make an important observation: if we use the same Huffman code, all other sequences with $p_1 n$ occurrences of v_1, $p_2 n$ occurrences of v_2, etc., are also going to have an encoding of length L.

At this point we can ask: how many sequences of n elements of V are there such that, for $i = 1, \ldots, K$, the letter v_i appears $p_i n$ times? It can be proved that such a number is expressed by the *multinomial coefficient*

$$\frac{n!}{(p_1 n)!(p_2 n)! \cdots (p_K n)!}$$

This means that the above number must be $\leq 2^L$, since all such sequences are being injectively mapped to bit-strings of length L by the Huffman code, so we have

$$L \geq \log_2 n! - \sum_{i=1}^{K} \log_2 (p_i n)!$$

When we talked about the complexity of sorting, we already encountered the approximation

$$\log_2 n! = n \log_2 n - n \log_2 e + \Theta(\log n)$$

If we combine the above two expressions, we have

$$\begin{aligned}
L &\geq n\log_2 n - n\log_2 e - \left(\sum_{i=1}^{K} p_i n \log_2 p_i n\right) + \sum_{i=1}^{K} p_i n \log_2 e + \Theta(K\log n) \\
&= n\log_2 n - \left(\sum_{i=1}^{K} p_i n \log_2 p_i n\right) + \Theta(K\log n) \\
&= n\log_2 n - \left(\sum_{i=1}^{K} p_i n \log_2 p_i\right) - \left(\sum_{i=1}^{K} p_i n \log_2 n\right) + \Theta(K\log n) \\
&= -\sum_{i=1}^{K} p_i n \log_2 p_i + \Theta(K\log n) \\
&= n \cdot \left(\sum_{i=1}^{K} p_i \log_2 \frac{1}{p_i}\right) + \Theta(K\log n)
\end{aligned}$$

We went from the first to the second line by canceling out $-n\log_2 n$ with $\sum_i p_i n \log_2 n$, due to the fact that $\sum_i p_i n = n$. We went from the second to the third line by writing out $\log_2 p_i n$ as $\log_2 p_i + \log_2 n$. We went from the third to the fourth line by canceling out $-n\log_2 n$ with $\sum_i p_i n \log_2 n$. The last line is just a re-arrangement of terms.

The quantity $\sum_i p_i \log_2 \frac{1}{p_i}$ is called the *entropy* of $(p_1, \ldots, p_K)$, and it is a fundamental quantity in data compression and in information theory in general.

Notice that in this section we have proved the following result.

Theorem 4 (Lower bound to Huffman encoding length) *Let $V = \{v_1, \ldots, v_K\}$ be any finite alphabet, let $s \in V^n$ be any sequence of n elements of V, let p_i be the fraction of times that letter v_i appear in the sequence s, and let L be the length of a Huffman encoding of s. Then*

$$L \geq n \cdot \left(\sum_{i=1}^{K} p_i \log_2 \frac{1}{p_i}\right) + \Theta(K\log n)$$

3 Arithmetic coding

In this section we show that, up to some additive error, the length of the encoding found by Huffman algorithm matches the lower bound proved in the previous section.

Theorem 5 (Upper bound to Huffman encoding length) *Let $V = \{v_1, \ldots, v_K\}$ be any finite alphabet, let $s \in V^n$ be any sequence of n elements of V, let p_i be the*

fraction of times that letter v_i appear in the sequence s, and let L be the length of a Huffman encoding of s. Then

$$L \leq n \cdot \left(\sum_{i=1}^{K} p_i \log_2 \frac{1}{p_i} \right) + 2n$$

We will prove the above result by describing a prefix-free encoding of the letters of V that achieves the above bound. Since Huffman's algorithm is optimal among prefix-free encodings of letters of V, the above bound will apply to the output of Huffman's algorithm as well.

The encoding that we discuss is called *arithmetic coding*. To apply this method, we fix some arbitrary order $v_1, \ldots, v_K$ of the letters of V, and we consider, for every letter v_i, the interval of values between

$$p_1 + \cdots + p_{i-1}$$

and

$$p_1 + \cdots + p_{i-1} + p_i$$

For example, if we have three letters $\{a, b, c\}$, and we have $p_a = .2$, $p_b = .5$ and $p_c = .3$, then we associate to a the interval $[0, 0.2)$, to b the interval $[0.2, 0.7)$ and to c the interval $[0.7, 1)$.

The first step of the encoding is to choose, in each interval, a binary number that is contained in the interval and such that *appending any additional binary digits* to the number defines another number still in the interval. For example, the first interval, written in binary, is between 0 and $0.00110\cdots$. The number 0.000 has the desired property, because 0.000 is in the interval, and any number obtained by adding additional digits, for example 0.0001101 is also in the interval. The interval $[0.2, 0.7)$ is

$$[0.00110\cdots, 0.10110011\cdots)$$

in binary, and the number 0.01 has the required properties. The third interval $[0.7, 1)$ is

$$[0.10110011\cdots, 0.111111\cdots)$$

in binary, and the number 0.11 has the required properties.

After we find a number with the required properties in each interval, remove the "0." initial part of the binary representation from each of these numbers, and the remaining bit string will be the binary encoding of the respective element of V.

In the above example, we would have 000 as an encoding of a, 01 as an encoding of b, and 11 as an encoding of c.

Note that the properties enforce that each element of V is mapped to a different bit string, and that the encoding is prefix-free.

How do we find such numbers in practice?

The idea is to use the following fact: if r is a real number between 0 and 1, and we write the binary expansion of r and take only ℓ digits of precision, then we obtain a number $\hat{r}$ that is at most r and at least $r - 2^{-\ell}$. (Think about the analogous fact in base 10.) Furthermore, if we attach any additional digits at the end of $\hat{r}$, we obtain a number that is at most $\hat{r} + 2^{-\ell}$.

For example, 1/3 in binary is $0.010101\cdots$. If we take 4 digits of precision, we have the approximation $0.0101_2 = 0.315_{10}$, and the error is $0.018333\cdots_{10}$ which is less than $2^{-4} = 0.0625_{10}$. Furthermore, any number that we can obtain by appending digits at the end of 0.0101_2 is at most $0.0101111\cdots_2 = 0.0101_2 + 0.00001111\cdots_2 = 0.0101_2 + 2^{-4}$.

The trick is to write the midpoint of the interval, that is $p_1 + \cdots + p_{i-1} + \frac{1}{2}p_i$, in binary and to truncate the binary representation to have enough precision that the truncated binary number is still $> p_1 + \cdots + p_{i-1}$, and such that if we append any additional digits we are still below $p_1 + \cdots + p_{i-1} + p_i$. This is doable with ℓ_i digits of precision, provided that $2^{-\ell_i} < \frac{p_i}{2}$, that is, provided that

$$\ell_i > \log_2 \frac{1}{p_1} + 1$$

In particular, it is possible to encode each letter v_i using $\lceil \log_2 \frac{1}{p_i} \rceil + 1 \leq \log_2 \frac{1}{p_i} + 2$ bits, and the total length of the encoding is at most

$$\sum_i p_i n \cdot \left(\left\lceil \log_2 \frac{1}{p_i} \right\rceil + 1 \right) \leq n \cdot \left(\sum_i p_i \log_2 \frac{1}{p_i} \right) + 2n$$

To see how this works in the above example, the interval $[0, 0.2)$ has midpoint 1/10, whose binary representation is $0.0001100110011\cdots$ and we use the approximation 0.000 (the theory gives us that four digits after the dot suffice, but in this case even three are enough). The midpoint of the interval $[0.2, 0.7)$ is 0.45, whose binary representation is $0.011100110011\cdots$. In this case we take two digits of precision, as guaranteed by the theory. The midpoint of $[0.7, 1)$ is 0.85, whose binary representation is $0.11011001100110011\cdots$, and again two digits after the dot suffice.