Chapter 42

Entropy II

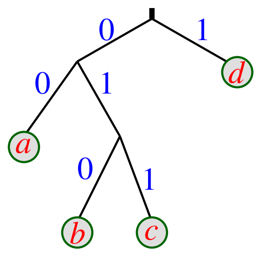
By Sariel Har-Peled, April 26, 2022[1]

> The memory of my father is wrapped up in white paper, like sandwiches taken for a day at work. Just as a magician takes towers and rabbits out of his hat, he drew love from his small body, and the rivers of his hands overflowed with good deeds.
>
> Yehuda Amichai, My Father

42.1. Huffman coding

A ***binary code*** assigns a string of 0s and 1s to each character in the alphabet. A code assigns for each symbol in the input a codeword over some other alphabet. Such a coding is necessary, for example, for transmitting messages over a wire, were you can send only 0 or 1 on the wire (i.e., for example, consider the good old telegraph and Morse code). The receiver gets a binary stream of bits and needs to decode the message sent. A prefix code, is a code where one can decipher the message, a character by character, by reading a prefix of the input binary string, matching it to a code word (i.e., string), and continuing to decipher the rest of the stream. Such a code is a ***prefix code***.

A binary code (or a prefix code) is ***prefix-free*** if no code is a prefix of any other. ASCII and Unicode's UTF-8 are both prefix-free binary codes. Morse code is a binary code (and also a prefix code), but it is not prefix-free; for example, the code for `S` ($\cdots$) includes the code for `E` ($\cdot$) as a prefix. (Hopefully the receiver knows that when it gets $\cdots$ that it is extremely unlikely that this should be interpreted as `EEE`, but rather `S`.



Any prefix-free binary code can be visualized as a binary tree with the encoded characters stored at the leaves. The code word for any symbol is given by the path from the root to the corresponding leaf; 0 for left, 1 for right. The length of a codeword for a symbol is the depth of the corresponding leaf. Such trees are usually referred to as ***prefix trees*** or ***code trees***.

The beauty of prefix trees (and thus of prefix odes) is that decoding is easy. As a concrete example, consider the tree on the right. Given a string '010100', we can traverse down the tree from the root, going left if get a '0' and right if we get '1'. Whenever we get to a leaf, we output the character output in the leaf, and we jump back to the root for the next character we are about to read. For the example '010100', after reading '010' our traversal in the tree leads us to the leaf marked with 'b', we jump back to the root and read the next input digit, which is '1', and this leads us to the leaf marked with 'd', which we output, and jump back to the root. Finally, '00' leads us to the leaf marked by 'a', which the algorithm output. Thus, the binary string '010100' encodes the string `"bda"`.

Suppose we want to encode messages in an n-character alphabet so that the encoded message is as short as possible. Specifically, given an array frequency counts $f[1 \ldots n]$, we want to compute a prefix-free binary code that minimizes the total encoded length of the message. That is we would like

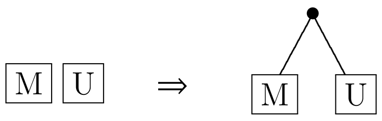
[1]This work is licensed under the Creative Commons Attribution-Noncommercial 3.0 License. To view a copy of this license, visit http://creativecommons.org/licenses/by-nc/3.0/ or send a letter to Creative Commons, 171 Second Street, Suite 300, San Francisco, California, 94105, USA.

to compute a tree T that minimizes

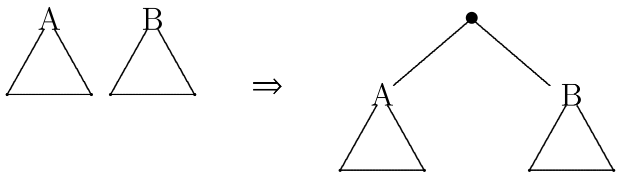
$$\operatorname{cost}(T) = \sum_{i=1}^{n} f[i] * \operatorname{len}(\operatorname{code}(i)), \tag{42.1}$$

where code(i) is the binary string encoding the ith character and len(s) is the length (in bits) of the binary string s.

A nice property of this problem is that given two trees for some parts of the alphabet, we can easily put them together into a larger tree by just creating a new node and hanging the trees from this common node. For example, putting two characters together, we have the following.



Similarly, we can put together two subtrees.



42.1.1. The algorithm to build Hoffman's code

This suggests a simple algorithm that takes the two least frequent characters in the current frequency table, merge them into a tree, and put the merged tree back into the table (instead of the two old trees). The algorithm stops when there is a single tree. The intuition is that infrequent characters would participate in a large number of merges, and as such would be low in the tree – they would be assigned a long code word.

This algorithm is due to David Huffman, who developed it in 1952. Shockingly, this code is the best one can do. Namely, the resulting code is *asymptotically* gives the best possible compression of the data (of course, one can do better compression in practice using additional properties of the data and careful hacking). This ***Huffman coding*** is used widely and is the basic building block used by numerous other compression algorithms.

42.1.2. Analysis

Lemma 42.1.1. *Let T be an optimal code tree. Then T is a full binary tree (i.e., every node of T has either 0 or 2 children). In particular, if the height of T is d, then there are leafs nodes of height d that are sibling.*

Proof: If there is an internal node in T that has one child, we can remove this node from T, by connecting its only child directly with its parent. The resulting code tree is clearly a better compressor, in the sense of Eq. (42.1).

As for the second claim, consider a leaf u with maximum depth d in T, and consider it parent $v = \overline{\mathrm{p}}(u)$. The node v has two children, and they are both leafs (otherwise u would not be the deepest node in the tree), as claimed. ■

Lemma 42.1.2. *Let x and y be the two least frequent characters (breaking ties between equally frequent characters arbitrarily). There is an optimal code tree in which x and y are siblings.*

Proof: More precisely, there is an optimal code in which x and y are siblings and have the largest depth of any leaf. Indeed, let T be an optimal code tree with depth d. The tree T has at least two leaves at depth d that are siblings, by Lemma 42.1.1.

Now, suppose those two leaves are not x and y, but some other characters α and β. Let $\mathcal{U}$ be the code tree obtained by swapping x and α. The depth of x increases by some amount Δ, and the depth of α decreases by the same amount. Thus,

$$\mathrm{cost}(\mathcal{U}) = \mathrm{cost}(T) - (f[\alpha] - f[x])\Delta.$$

By assumption, x is one of the two least frequent characters, but α is not, which implies that $f[\alpha] > f[x]$. Thus, swapping x and α does not increase the total cost of the code. Since T was an optimal code tree, swapping x and α does not decrease the cost, either. Thus, $\mathcal{U}$ is also an optimal code tree (and incidentally, $f[\alpha]$ actually equals $f[x]$). Similarly, swapping y and b must give yet another optimal code tree. In this final optimal code tree, x and y as maximum-depth siblings, as required. ■

Theorem 42.1.3. *Huffman codes are optimal prefix-free binary codes.*

Proof: If the message has only one or two different characters, the theorem is trivial. Otherwise, let $f[1 \ldots n]$ be the original input frequencies, where without loss of generality, $f[1]$ and $f[2]$ are the two smallest. To keep things simple, let $f[n+1] = f[1] + f[2]$. By the previous lemma, we know that some optimal code for $f[1..n]$ has characters 1 and 2 as siblings. Let $\mathcal{T}_{\mathrm{opt}}$ be this optimal tree, and consider the tree formed by it by removing 1 and 2 as it leaves. We remain with a tree $\mathcal{T}'_{\mathrm{opt}}$ that has as leafs the characters $3, \ldots, n$ and a "special" character $n+1$ (which is the parent of 1 and 2 in $\mathcal{T}_{\mathrm{opt}}$) that has frequency $f[n+1]$. Now, since $f[n+1] = f[1] + f[2]$, we have

$$\begin{aligned}
\mathrm{cost}\left(\mathcal{T}_{\mathrm{opt}}\right) &= \sum_{i=1}^{n} f[i]\mathrm{depth}_{\mathcal{T}_{\mathrm{opt}}}(i)\\
&= \sum_{i=3}^{n+1} f[i]\mathrm{depth}_{\mathcal{T}_{\mathrm{opt}}}(i) + f[1]\mathrm{depth}_{\mathcal{T}_{\mathrm{opt}}}(1) + f[2]\mathrm{depth}_{\mathcal{T}_{\mathrm{opt}}}(2) - f[n+1]\mathrm{depth}_{\mathcal{T}_{\mathrm{opt}}}(n+1)\\
&= \mathrm{cost}\left(\mathcal{T}'_{\mathrm{opt}}\right) + (f[1]+f[2])\mathrm{depth}\left(\mathcal{T}_{\mathrm{opt}}\right) - (f[1]+f[2])\left(\mathrm{depth}\left(\mathcal{T}_{\mathrm{opt}}\right) - 1\right)\\
&= \mathrm{cost}\left(\mathcal{T}'_{\mathrm{opt}}\right) + f[1] + f[2]. \qquad (42.2)
\end{aligned}$$

This implies that minimizing the cost of $\mathcal{T}_{\mathrm{opt}}$ is equivalent to minimizing the cost of $\mathcal{T}'_{\mathrm{opt}}$. In particular, $\mathcal{T}'_{\mathrm{opt}}$ must be an optimal coding tree for $f[3 \ldots n{+}1]$. Now, consider the Huffman tree $\mathcal{U}_H$ constructed for $f[3, \ldots, n{+}1]$ and the overall Huffman tree T_H constructed for $f[1, \ldots, n]$. By the way the construction algorithm works, we have that $\mathcal{U}_H$ is formed by removing the leafs of 1 and 2 from T. Now, by induction, we know that the Huffman tree generated for $f[3, \ldots, n+1]$ is optimal; namely, $\mathrm{cost}\left(\mathcal{T}'_{\mathrm{opt}}\right) = \mathrm{cost}(\mathcal{U}_H)$. As such, arguing as above, we have

$$\mathrm{cost}(T_H) = \mathrm{cost}(\mathcal{U}_H) + f[1] + f[2] = \mathrm{cost}\left(\mathcal{T}'_{\mathrm{opt}}\right) + f[1] + f[2] = \mathrm{cost}\left(\mathcal{T}_{\mathrm{opt}}\right),$$

by Eq. (42.2). Namely, the Huffman tree has the same cost as the optimal tree. ■

42.1.3. A formula for the average size of a code word

Assume that our input is made out of n characters, where the ith character is p_i fraction of the input (one can think about p_i as the probability of seeing the ith character, if we were to pick a random character from the input).

Now, we can use these probabilities instead of frequencies to build a Huffman tree. The natural question is what is the length of the codewords assigned to characters as a function of their probabilities?

In general this question does not have a trivial answer, but there is a simple elegant answer, if all the probabilities are power of 2.

Lemma 42.1.4. *Let $1, \ldots, n$ be n symbols, such that the probability for the ith symbol is p_i, and furthermore, there is an integer $l_i \geq 0$, such that $p_i = 1/2^{l_i}$. Then, in the Huffman coding for this input, the code for i is of length l_i.*

Proof: The proof is by easy induction of the Huffman algorithm. Indeed, for $n = 2$ the claim trivially holds since there are only two characters with probability $1/2$. Otherwise, let i and j be the two characters with lowest probability. It must hold that $p_i = p_j$ (otherwise, $\sum_k p_k$ can not be equal to one). As such, Huffman's merges this two letters, into a single "character" that have probability $2p_i$, which would have encoding of length $l_i - 1$, by induction (on the remaining $n - 1$ symbols). Now, the resulting tree encodes i and j by code words of length $(l_i - 1) + 1 = l_i$, as claimed. ■

In particular, we have that $l_i = \lg 1/p_i$. This implies that the average length of a code word is

$$\sum_i p_i \lg \frac{1}{p_i}.$$

If we consider X to be a random variable that takes a value i with probability p_i, then this formula is

$$\mathbb{H}(X) = \sum_i \mathbb{P}[X = i] \lg \frac{1}{\mathbb{P}[X = i]},$$

which is the ***entropy*** of X.

Theorem 42.1.5. *Consider an input sequence S of m characters, where the characters are taken from an alphabet set Σ of size n. In particular, let f_i be the number of times the ith character of Σ appears in S, for $i = 1, \ldots, n$. Consider the compression of this string using Huffman's code. Then, the total length of the compressed string (ignoring the space needed to store the code itself) is $\leq m(\mathbb{H}(X) + 1)$, where X is a random variable that returns i with probability $p_i = f_i/m$.*

Proof: The trick is to replace p_i, which might not be a power of 2, by $q_i = 2^{\lfloor \lg p_i \rfloor}$. We have that $q_i \leq p_i \leq 2q_i$, and q_i is a power of 2, for all i. The leftover of this coding is $\Delta = 1 - \sum_i q_i$. We write Δ as a sum of powers of 2 (since the frequencies are fractions of the form i/m [since the input string is of length m] – this requires at most $\tau = O(\log m)$ numbers): $\Delta = \sum_{j=n+1}^{n+\tau} q_j$. We now create a Huffman code T for the frequencies $q_1, \ldots, q_n, q_{n+1}, \ldots, q_{n+\tau}$. The output length to encode the input string using this code, by Lemma 42.1.4, is

$$L = m \sum_{i=1}^{n} p_i \lg \frac{1}{q_i} \leq m \sum_{i=1}^{n} p_i \left(1 + \lg \frac{1}{p_i}\right) \leq m + m \sum_{i=1}^{n} p_i \lg \frac{1}{p_i} = m + m\mathbb{H}(X).$$

One can now restrict T to be a prefix tree only for the first n symbols. Indeed, delete the τ "fake" leafs/symbols, and repeatedly remove internal nodes that have only a single child. In the end of this process, we get a valid prefix tree for the first n symbols, and encoding the input string using this tree would require at most L bits, since process only shortened the code words. Finally, let $\mathcal{V}$ be the resulting tree.

Now, consider the Huffman tree code for the n input symbols using the original frequencies $p_1, \ldots p_n$. The resulting tree $\mathcal{U}$ is a better encoder for the input string than $\mathcal{V}$, by Theorem 42.1.3. As such, the compressed string, would have at most L bits – thus establishing the claim. ■

42.2. Compression

In this section, we consider the problem of how to compress a binary string. We map each binary string, into a new string (which is hopefully shorter). In general, by using a simple counting argument, one can show that no such mapping can achieve real compression (when the inputs are adversarial). However, the hope is that there is an underling distribution on the inputs, such that some strings are considerably more common than others.

Definition 42.2.1. A compression function `Compress` takes as input a sequence of n coin flips, given as an element of $\{H, T\}^n$, and outputs a sequence of bits such that each input sequence of n flips yields a distinct output sequence.

The following is easy to verify.

Lemma 42.2.2. *If a sequence S_1 is more likely than S_2 then the compression function that minimizes the expected number of bits in the output assigns a bit sequence to S_2 which is at least as long as S_1.*

Note, that this is weak. Usually, we would like the function to output a prefix code, like the Huffman code.

Theorem 42.2.3. *Consider a coin that comes up heads with probability $p > 1/2$. For any constant $\delta > 0$, when n is sufficiently large, the following holds.*

(i) There exists a compression function `Compress` such that the expected number of bits output by `Compress` on an input sequence of n independent coin flips (each flip gets heads with probability p) is at most $(1+\delta)n\mathbb{H}(p)$; and

(ii) The expected number of bits output by any compression function on an input sequence of n independent coin flips is at least $(1-\delta)n\mathbb{H}(p)$.

Proof: Let $\varepsilon > 0$ be a constant such that $p - \varepsilon > 1/2$. The first bit output by the compression procedure is '1' if the output string is just a copy of the input (using $n+1$ bits overall in the output), and '0' if it is compressed. We compress only if the number of ones in the input sequence, denoted by X is larger than $(p-\varepsilon)n$. By the Chernoff inequality, we know that $\mathbb{P}[X < (p-\varepsilon)n] \leq \exp(-n\varepsilon^2/2p)$.

If there are more than $(p-\varepsilon)n$ ones in the input, and since $p - \varepsilon > 1/2$, we have that

$$\sum_{j=\lceil n(p-\varepsilon)\rceil}^{n} \binom{n}{j} \leq \sum_{j=\lceil n(p-\varepsilon)\rceil}^{n} \binom{n}{\lceil n(p-\varepsilon)\rceil} \leq \frac{n}{2} 2^{n\mathbb{H}(p-\varepsilon)},$$

by Corollary 42.3.1. As such, we can assign each such input sequence a number in the range $0 \ldots \frac{n}{2}2^{n\mathbb{H}(p-\varepsilon)}$, and this requires (with the flag bit) $1 + \lfloor \lg n + n\mathbb{H}(p-\varepsilon) \rfloor$ random bits.

Thus, the expected number of bits output is bounded by

$$(n+1)\exp\bigl(-n\varepsilon^2/2p\bigr) + \bigl(1 + \lfloor \lg n + n\mathbb{H}(p-\varepsilon)\rfloor\bigr) \leq (1+\delta)n\mathbb{H}(p),$$

by carefully setting ε and n being sufficiently large. Establishing the upper bound.

As for the lower bound, observe that at least one of the sequences having exactly $\tau = \lfloor (p+\varepsilon)n \rfloor$ heads, must be compressed into a sequence having

$$\lg \binom{n}{\lfloor (p+\varepsilon)n \rfloor} - 1 \geq \lg \frac{2^{n\mathbb{H}(p+\varepsilon)}}{n+1} - 1 = n\mathbb{H}(p-\varepsilon) - \lg(n+1) - 1 = \mu,$$

by Corollary 42.3.1. Now, any input string with less than τ heads has lower probability to be generated. Indeed, for a specific strings with $\alpha < \tau$ ones the probability to generate them is $p^\alpha(1-p)^{n-\alpha}$ and $p^\tau(1-p)^{n-\tau}$, respectively. Now, observe that

$$p^\alpha(1-p)^{n-\alpha} = p^\tau(1-p)^{n-\tau}\cdot\frac{(1-p)^{\tau-\alpha}}{p^{\tau-\alpha}} = p^\tau(1-p)^{n-\tau}\left(\frac{1-p}{p}\right)^{\tau-\alpha} < p^\tau(1-p)^{n-\tau},$$

as $1-p < 1/2 < p$ implies that $(1-p)/p < 1$.

As such, Lemma 42.2.2 implies that all the input strings with less than τ ones, must be compressed into strings of length at least μ, by an optimal compresser. Now, the Chenroff inequality implies that $\mathbb{P}[X \leq \tau] \geq 1 - \exp\bigl(-n\varepsilon^2/12p\bigr)$. Implying that an optimal compresser outputs on average at least $\bigl(1-\exp\bigl(-n\varepsilon^2/12p\bigr)\bigr)\mu$. Again, by carefully choosing ε and n sufficiently large, we have that the average output length of an optimal compressor is at least $(1-\delta)n\mathbb{H}(p)$. ■

42.3. From previous lecture

Corollary 42.3.1. *We have:* *(i)* $q \in [0,1/2] \Rightarrow \binom{n}{\lfloor nq \rfloor} \leq 2^{n\mathbb{H}(q)}$. *(ii)* $q \in [1/2,1]$ $\binom{n}{\lceil nq \rceil}$
*beginequation*0.1cm]* *(iii)* $q \in [1/2,1] \Rightarrow \frac{2^{n\mathbb{H}(q)}}{n+1} \leq \binom{n}{\lfloor nq \rfloor}$. *(iv)* $q \in [0,1/2] \Rightarrow$

42.4. Bibliographical Notes

The presentation here follows [MU05, Sec. 9.1-Sec 9.3].

References

[MU05] M. Mitzenmacher and U. Upfal. *Probability and computing – randomized algorithms and probabilistic analysis.* Cambridge, 2005.