Chapter 25

Huffman Coding

By Sariel Har-Peled, December 7, 2009[1]

25.1 Huffman coding

(This portion of the class notes is based on Jeff Erickson class notes.)

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 just reading a prefix of the input binary string, matching it to an original character, and continuing to decipher the rest of the stream. Such a code is known as 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***tree!code trees.

[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.

newline	16, 492
space	130,376
‘!’	955
‘"’	5,681
‘$’	2
‘%’	1
‘’’	1,174
‘(’	151
‘)’	151
‘*’	70
‘,’	13,276
‘–’	2,430
‘.’	6,769

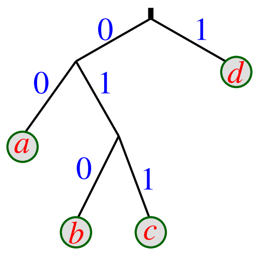
‘0’	20
‘1’	61
‘2’	10
‘3’	12
‘4’	10
‘5’	14
‘6’	11
‘7’	13
‘8’	13
‘9’	14
‘:’	267
‘;’	1,108
‘?’	913

‘A’	48,165
‘B’	8,414
‘C’	13,896
‘D’	28,041
‘E’	74,809
‘F’	13,559
‘G’	12,530
‘H’	38,961
‘I’	41,005
‘J’	710
‘K’	4,782
‘L’	22,030
‘M’	15,298

‘N’	42,380
‘O’	46,499
‘P’	9,957
‘Q’	667
‘R’	37,187
‘S’	37,575
‘T’	54,024
‘U’	16,726
‘V’	5,199
‘W’	14,113
‘X’	724
‘Y’	12,177
‘Z’	215

‘_’	182
’‘’	93
‘@’	2
‘/’	26

Figure 25.1: Frequency of characters in the book “A tale of two cities” by Dickens. For the sake of brevity, small letters were counted together with capital letters.



The beauty of prefix trees (and thus of prefix odes) is that decoding is very 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 to compute a tree $\mathcal{T}$ that minimizes

$$\text{cost}(\mathcal{T}) = \sum_{i=1}^{n} f[i] * \text{len}(\text{code}(i)), \tag{25.1}$$

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

As a concrete example, consider Figure 25.1, which shows the frequency of characters in the book “A tale of two cities”, which we would like to encode. Consider the characters ‘E’ and ‘Q’. The first appears $> 74,000$ times in the text, and other appears only 667 times in the text. Clearly, it would be logical to give ‘E’, the most frequent letter in English, a very short prefix code, and a very long (as far as number of bits) code to ‘Q’.

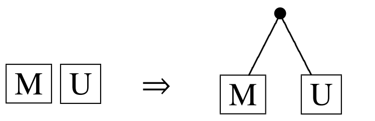
A nice property of this problem is that given two trees for some parts of the alphabet, we can easily put them together into a larger tree by just creating a new node and hanging the trees from this common node. For example, putting two characters together, we have the following.

char	frequency	code
‘A’	48165	1110
‘B’	8414	101000
‘C’	13896	00100
‘D’	28041	0011
‘E’	74809	011
‘F’	13559	111111
‘G’	12530	111110
‘H’	38961	1001

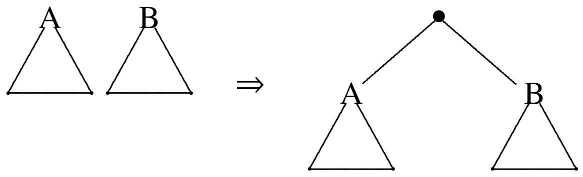
char	frequency	code
‘I’	41005	1011
‘J’	710	1111011010
‘K’	4782	11110111
‘L’	22030	10101
‘M’	15298	01000
‘N’	42380	1100
‘O’	46499	1101
‘P’	9957	101001
‘Q’	667	1111011001

char	frequency	code
‘R’	37187	0101
‘S’	37575	1000
‘T’	54024	000
‘U’	16726	01001
‘V’	5199	1111010
‘W’	14113	00101
‘X’	724	1111011011
‘Y’	12177	111100
‘Z’	215	1111011000

Figure 25.2: The resulting prefix code for the frequencies of Figure 25.1. Here, for the sake of simplicity of exposition, the code was constructed only for the A—Z characters.



Similarly, we can put together two subtrees.



25.1.1 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.

To see how such a resulting tree (and the associated code) looks like, see Figure 25.2 and Figure 25.3.

25.1.2 Analysis

Lemma 25.1.1 *Let $\mathcal{T}$ be an optimal code tree. Then $\mathcal{T}$ is a full binary tree (i.e., every node of $\mathcal{T}$ has either* 0 *or* 2 *children).*

In particular, if the height of $\mathcal{T}$ is d, then there are leafs nodes of height d that are sibling.

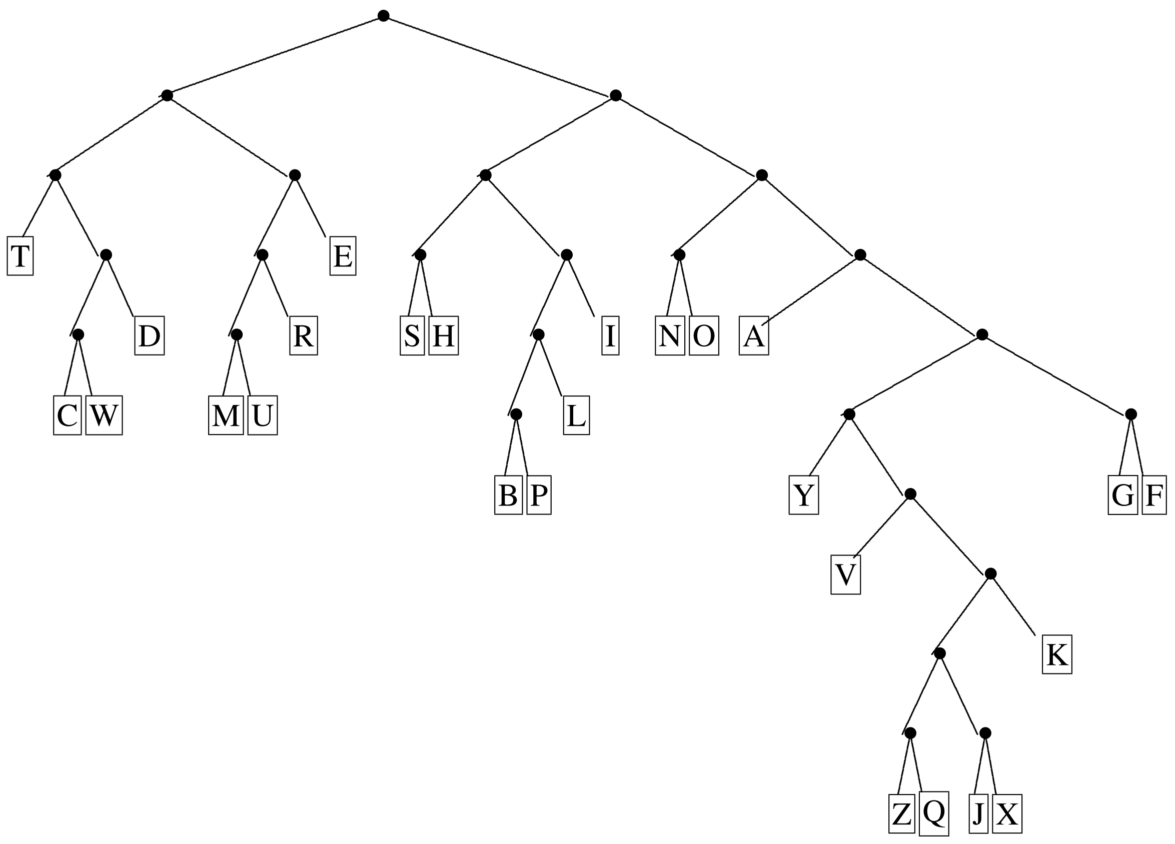


Figure 25.3: The Huffman tree generating the code of Figure 25.2.

Proof: If there is an internal node in $\mathcal{T}$ that has one child, we can remove this node from $\mathcal{T}$, by connecting its only child directly with its parent. The resulting code tree is clearly a better compressor, in the sense of Eq. (25.1).

As for the second claim, consider a leaf u with maximum depth d in $\mathcal{T}$, 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 25.1.2 *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: In fact, there is an optimal code in which x and y are siblings and have the largest depth of any leaf. Indeed, let $\mathcal{T}$ be an optimal code tree with depth d. The tree $\mathcal{T}$ has at least two leaves at depth d that are siblings, by Lemma 25.1.1.

Now, suppose those two leaves are not x and y, but some other characters α and β. Let $\mathcal{T}'$ 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,

$$\mathrm{cost}(\mathcal{T}') = \mathrm{cost}(\mathcal{T}) - \Big(f[\alpha] - f[x]\Big)\Delta.$$

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 $\mathcal{T}$ was an optimal code tree, swapping x and α does not decrease the cost, either. Thus, $\mathcal{T}'$ 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 25.1.3 *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

$$
\begin{aligned}
\mathrm{cost}\left(\mathcal{T}_{\mathrm{opt}}\right) &= \sum_{i=1}^{n} f[i]\mathrm{depth}_{\mathcal{T}_{\mathrm{opt}}}(i) \\
&= \sum_{i=3}^{n+1} f[i]\mathrm{depth}_{\mathcal{T}_{\mathrm{opt}}}(i) + f[1]\mathrm{depth}_{\mathcal{T}_{\mathrm{opt}}}(1) + f[2]\mathrm{depth}_{\mathcal{T}_{\mathrm{opt}}}(2) - f[n+1]\mathrm{depth}_{\mathcal{T}_{\mathrm{opt}}}(n+1) \\
&= \mathrm{cost}\left(\mathcal{T}'_{\mathrm{opt}}\right) + \left(f[1] + f[2]\right)\mathrm{depth}\left(\mathcal{T}_{\mathrm{opt}}\right) - \left(f[1] + f[2]\right)\left(\mathrm{depth}\left(\mathcal{T}_{\mathrm{opt}}\right) - 1\right) \\
&= \mathrm{cost}\left(\mathcal{T}'_{\mathrm{opt}}\right) + f[1] + f[2]. \qquad (25.2)
\end{aligned}
$$

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

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

by Eq. (25.2). Namely, the Huffman tree has the same cost as the optimal tree. ■

25.1.3 What do we get

For the book "A tale of two cities" which is made out of 779,940 bytes, and using the above Huffman compression results in a compression to a file of size 439,688 bytes. A far cry from what `gzip` can do (301,295 bytes) or `bzip2` can do (220,156 bytes!), but still very impressive when you consider that the Huffman encoder can be easily written in a few hours of work.

(This numbers ignore the space required to store the code with the file. This is pretty small, and would not change the compression numbers stated above significantly.

25.1.4 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 25.1.4 *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

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

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