Chapter 26

Compression, Information and Entropy – Huffman’s coding

CS 573: Algorithms, Fall 2013
December 3, 2013

26.1 Huffman coding

26.1.0.1 Codes...

(A) Σ: alphabet.
(B) binary code: assigns a string of 0s and 1s to each character in the alphabet.
(C) each symbol in input = a codeword over some other alphabet.
(D) Useful for transmitting messages over a wire: only 0/1.
(E) receiver gets a binary stream of bits...
(F) ... decode the message sent.
(G) prefix code: reading a prefix of the input binary string uniquely match it to a code word.
(H) ... continuing to decipher the rest of the stream.
(I) binary/prefix code is prefix-free if no code is a prefix of any other.
(J) ASCII and Unicode’s UTF-8 are both prefix-free binary codes.

26.1.0.2 Codes...

(A) Morse code is binary+prefix code but not prefix-free.
(B) ... code for S (⋯) includes the code for E (⋅) as a prefix.
(C) Prefix codes are binary trees...

(D) ...characters in leaves, code word is path from root.
(E) prefix tree!prefix tree or code trees.
(F) Decoding/encoding is easy.
26.1.0.3 Codes...

(A) Encoding: given frequency table: 
\[ f[1 \ldots n] \]

(B) \( f[i] \): frequency of \( i \)th character.

(C) \( \text{code}(i) \): binary string for \( i \)th character.

\( \text{len}(s) \): length (in bits) of binary string \( s \).

(D) Compute tree \( T \) that minimizes

\[
\text{cost}(T) = \sum_{i=1}^{n} f[i] \times \text{len}(	ext{code}(i)),
\]

26.1.1 Frequency table for...

26.1.1.1 “A tale of two cities” by Dickens

<table>
<thead>
<tr>
<th>char</th>
<th>frequency</th>
<th>code</th>
</tr>
</thead>
<tbody>
<tr>
<td>'A'</td>
<td>48165</td>
<td>1110</td>
</tr>
<tr>
<td>'B'</td>
<td>8414</td>
<td>101000</td>
</tr>
<tr>
<td>'C'</td>
<td>13896</td>
<td>00100</td>
</tr>
<tr>
<td>'D'</td>
<td>28041</td>
<td>0011</td>
</tr>
<tr>
<td>'E'</td>
<td>74809</td>
<td>011</td>
</tr>
<tr>
<td>'F'</td>
<td>13559</td>
<td>111111</td>
</tr>
<tr>
<td>'G'</td>
<td>12530</td>
<td>111110</td>
</tr>
<tr>
<td>'H'</td>
<td>38961</td>
<td>1001</td>
</tr>
<tr>
<td>'I'</td>
<td>1,174</td>
<td>0011</td>
</tr>
<tr>
<td>'J'</td>
<td>151</td>
<td>1111011010</td>
</tr>
<tr>
<td>'K'</td>
<td>4,782</td>
<td>0101</td>
</tr>
<tr>
<td>'L'</td>
<td>22,030</td>
<td>10101</td>
</tr>
<tr>
<td>'M'</td>
<td>15298</td>
<td>01000</td>
</tr>
<tr>
<td>'N'</td>
<td>16,492</td>
<td>61</td>
</tr>
<tr>
<td>'O'</td>
<td>42,380</td>
<td>1100</td>
</tr>
<tr>
<td>'P'</td>
<td>9,957</td>
<td>101001</td>
</tr>
<tr>
<td>'Q'</td>
<td>7,105</td>
<td>111101101</td>
</tr>
<tr>
<td>'R'</td>
<td>37,187</td>
<td>667</td>
</tr>
<tr>
<td>'S'</td>
<td>3,755</td>
<td>4001</td>
</tr>
<tr>
<td>'T'</td>
<td>54,024</td>
<td>14,113</td>
</tr>
<tr>
<td>'U'</td>
<td>16,726</td>
<td>12,177</td>
</tr>
<tr>
<td>'V'</td>
<td>5,199</td>
<td>1001</td>
</tr>
<tr>
<td>'W'</td>
<td>14,133</td>
<td>1101</td>
</tr>
<tr>
<td>'X'</td>
<td>724</td>
<td>1111010</td>
</tr>
<tr>
<td>'Y'</td>
<td>12,177</td>
<td>111100</td>
</tr>
<tr>
<td>'Z'</td>
<td>215</td>
<td>111101100</td>
</tr>
</tbody>
</table>

26.1.1.2 Computed prefix codes...

<table>
<thead>
<tr>
<th>char</th>
<th>freq</th>
<th>code</th>
</tr>
</thead>
<tbody>
<tr>
<td>'A'</td>
<td>1110</td>
<td></td>
</tr>
<tr>
<td>'B'</td>
<td>101000</td>
<td></td>
</tr>
<tr>
<td>'C'</td>
<td>00100</td>
<td></td>
</tr>
<tr>
<td>'D'</td>
<td>0011</td>
<td></td>
</tr>
<tr>
<td>'E'</td>
<td>011</td>
<td></td>
</tr>
<tr>
<td>'F'</td>
<td>111111</td>
<td></td>
</tr>
<tr>
<td>'G'</td>
<td>111110</td>
<td></td>
</tr>
<tr>
<td>'H'</td>
<td>1001</td>
<td></td>
</tr>
<tr>
<td>'I'</td>
<td>0011</td>
<td></td>
</tr>
<tr>
<td>'J'</td>
<td>1111011010</td>
<td></td>
</tr>
<tr>
<td>'K'</td>
<td>111101111</td>
<td></td>
</tr>
<tr>
<td>'L'</td>
<td>10101</td>
<td></td>
</tr>
<tr>
<td>'M'</td>
<td>01000</td>
<td></td>
</tr>
<tr>
<td>'N'</td>
<td>1100</td>
<td></td>
</tr>
<tr>
<td>'O'</td>
<td>1101</td>
<td></td>
</tr>
<tr>
<td>'P'</td>
<td>101001</td>
<td></td>
</tr>
<tr>
<td>'Q'</td>
<td>111101101</td>
<td></td>
</tr>
<tr>
<td>'R'</td>
<td>0101</td>
<td></td>
</tr>
<tr>
<td>'S'</td>
<td>1000</td>
<td></td>
</tr>
<tr>
<td>'T'</td>
<td>000</td>
<td></td>
</tr>
<tr>
<td>'U'</td>
<td>01001</td>
<td></td>
</tr>
<tr>
<td>'V'</td>
<td>1111010</td>
<td></td>
</tr>
<tr>
<td>'W'</td>
<td>00101</td>
<td></td>
</tr>
<tr>
<td>'X'</td>
<td>1111001</td>
<td></td>
</tr>
<tr>
<td>'Y'</td>
<td>111100</td>
<td></td>
</tr>
<tr>
<td>'Z'</td>
<td>1111011000</td>
<td></td>
</tr>
</tbody>
</table>
26.1.2 The Huffman tree generating the code

26.1.2.1 Build only on A-Z for clarity.

26.1.2.2 Mergeability of code trees

(A) two trees for some disjoint parts of the alphabet...
(B) Merge into larger tree by creating a new node and hanging the trees from this common node.

(C) \[ \begin{array}{c}
M \\ U
\end{array} \] \[ \Rightarrow \] \[ \begin{array}{c}
M \\ U
\end{array} \]

(D) ...put together two subtrees.

26.1.3 The algorithm to build Hoffman’s code

26.1.3.1 Building optimal prefix code trees

(A) take two least frequent characters in frequency table...
(B) ... merge them into a tree, and put the root of merged tree back into table.
(C) ...instead of the two old trees.
(D) Algorithm stops when there is a single tree.
(E) Intuition: infrequent characters participate in a large number of merges. Long code words.
(F) Algorithm is due to David Huffman (1952).
(G) Resulting code is best one can do.
(H) Huffman coding: building block used by numerous other compression algorithms.

26.1.4 Analysis

26.1.4.1 Analysis...

Lemma 26.1.1. (A) \( T: \) optimal code tree (prefix free!).
Then \( T \) is a full binary tree.

...every node of \( T \) has either 0 or 2 children.

If height of \( T \) is \( d \), then there are leafs nodes of height \( d \) that are sibling.

26.1.4.2 Proof...

(A) 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 \( \text{cost}(T) = \sum_{i=1}^{n} f[i] \ast \text{len(code}(i)) \).

(B) \( u \): leaf \( u \) with maximum depth \( d \) in \( T \). Consider parent \( v = \overline{p}(u) \).

(C) \( \implies \): \( v \) has two children, both leafs.

26.1.4.3 Infrequency characters are stuck together...

Lemma 26.1.2. 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.

26.1.4.4 Proof...

(A) Claim: \( \exists \) optimal code s.t. \( x \) and \( y \) are siblings + deepest.

(B) \( T \): optimal code tree with depth \( d \).

(C) By lemma... \( T \) has two leafs at depth \( d \) that are siblings.

(D) If not \( x \) and \( y \), but some other characters \( \alpha \) and \( \beta \).

(E) \( T' \): swap \( x \) and \( \alpha \).

(F) \( x \) depth inc by \( \Delta \), and depth of \( \alpha \) decreases by \( \Delta \).

(G) \( \text{cost}(T') = \text{cost}(T) - (f[\alpha] - f[x]) \Delta \).

(H) \( x \): one of the two least frequent characters.

...but \( \alpha \) is not.

(I) \( \implies \): \( f[\alpha] > f[x] \).

(J) Swapping \( x \) and \( \alpha \) does not increase cost.

(K) \( T \): optimal code tree, swapping \( x \) and \( \alpha \) does not decrease cost.

(L) \( T' \) is also an optimal code tree (\( f[\alpha] = f[x] \)).

(M) Swapping \( y \) and \( b \) must give yet another optimal code tree.

(N) Final opt code tree, \( x, y \) are max-depth siblings.

26.1.4.5 Huffman’s codes are optimal

Theorem 26.1.3. Huffman codes are optimal prefix-free binary codes.

26.1.4.6 Proof...

(A) If message has 1 or 2 diff characters, then theorem easy.

(B) \( f[1 \ldots n] \) be original input frequencies.

(C) Assume \( f[1] \) and \( f[2] \) are the two smallest.


(E) lemma \( \implies \exists \) opt. code tree \( T_{\text{opt}} \) for \( f[1] \ldots n \)

(F) \( T_{\text{opt}} \) has 1 and 2 as siblings.

(G) Remove 1 and 2 from \( T_{\text{opt}} \).

(H) \( T'_{\text{opt}} \): Remaining tree has \( 3, \ldots, n \) as leafs and “special” character \( n + 1 \) (i.e., parent 1, 2 in \( T_{\text{opt}} \)).
26.1.4.7 La proof continued...

(A) character \( n + 1 \): has frequency \( f[n + 1] \).

Now, \( f[n + 1] = f[1] + f[2] \), we have

\[
\text{cost}(T_{\text{opt}}) = \sum_{i=1}^{n} f[i] \text{depth}_{T_{\text{opt}}}(i)
\]

\[
= \sum_{i=3}^{n+1} f[i] \text{depth}_{T_{\text{opt}}}(i) + f[1] \text{depth}_{T_{\text{opt}}}(1) + f[2] \text{depth}_{T_{\text{opt}}}(2) - f[n + 1] \text{depth}_{T_{\text{opt}}}(n + 1)
\]

\[
= \text{cost}(T'_{\text{opt}}) + (f[1] + f[2]) \text{depth}(T_{\text{opt}}) - (f[1] + f[2])(\text{depth}(T_{\text{opt}}) - 1)
\]

\[
= \text{cost}(T'_{\text{opt}}) + f[1] + f[2].
\]

26.1.4.8 La proof continued...

(A) implies \( \text{min cost of } T_{\text{opt}} \equiv \text{min cost } T'_{\text{opt}} \).
(B) \( T'_{\text{opt}} \): must be optimal coding tree for \( f[3 \ldots n + 1] \).
(C) \( T'_H \): Huffman tree for \( f[3 \ldots , n + 1] \)

\( T_H \): overall Huffman tree constructed for \( f[1, \ldots , n] \).

(D) By construction: \( T'_H \) formed by removing leaves 1 and 2 from \( T_H \).
(E) By induction: Huffman tree generated for \( f[3, \ldots , n + 1] \) is optimal.
(F) \( \text{cost}(T'_{\text{opt}}) = \text{cost}(T'_H) \).

(G) \( \Rightarrow \text{cost}(T_H) = \text{cost}(T'_H) + f[1] + f[2] = \text{cost}(T'_{\text{opt}}) + f[1] + f[2] = \text{cost}(T_{\text{opt}}) \),

(H) \( \Rightarrow \) Huffman tree has the same cost as the optimal tree.

26.1.5 What do we get

26.1.5.1 What we get...

(A) A tale of two cities: 779,940 bytes.
(B) using above Huffman compression results in a compression to a file of size 439,688 bytes.
(C) Ignoring space to store tree.
(D) gzip: 301,295 bytes
    bzip2: 220,156 bytes!
(E) Huffman encoder can be easily written in a few hours of work!
(F) All later compressors use it as a black box...

26.1.6 A formula for the average size of a code word

26.1.6.1 Average size of code word

(A) input is made out of \( n \) characters.
(B) \( p_i \): fraction of input that is \( i \)th char (probability).
(C) use probabilities to build Huffman tree.
Q: What is the length of the codewords assigned to characters as function of probabilities?

special case...

Lemma 26.1.4. Let $1, \ldots, n$ be $n$ symbols, such that the probability for the $i$th 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 $l_i$.

26.1.6.2 Proof

(A) induction of the Huffman algorithm.
(B) $n = 2$: claim holds since there are only two characters with probability $1/2$.
(C) Let $i$ and $j$ be the two characters with lowest probability.
(D) Must be that $p_i = p_j$ (otherwise, $\sum p_k$ can not be equal to one).
(E) Huffman’s tree merges this two letters, into a single “character” that have probability $2p_i$.
(F) New “character” has encoding of length $l_i - 1$, by induction
   (on remaining $n - 1$ symbols).
(G) resulting tree encodes $i$ and $j$ by code words of length $(l_i - 1) + 1 = l_i$. ■

26.1.6.3 Translating lemma...

(A) $p_i = 1/2^{l_i}$
(B) $l_i = \lg 1/p_i$.
(C) Average length of a code word is
   \[
   \sum_i p_i \lg \frac{1}{p_i}.
   \]

(D) $X$ is a random variable that takes a value $i$ with probability $p_i$, then this formula is
   \[
   \mathbb{H}(X) = \sum_i \Pr[X = i] \lg \frac{1}{\Pr[X = i]},
   \]
which is the entropy of $X$. 