CS 493: Algorithms for Massive Data Sets	**Huffman and Arithmetic Coding**
DATE : Thursday, 2/7/2002	**Scribe: Chi Zhang**

1 Review of last class

For message set S, $s \in S$ has probability of $p(s)$. Entropy of S is given as

$$H(S) = \sum_{s \in S} p(s) \log_2 \frac{1}{p(s)}$$

And $\log_2 \frac{1}{p(s)}$ is the self information of s.

For any uniquely decodable code C for S, $l_a(C) \geq H(S)$.

Theorem 1.1 *Given S, $\exists$ code C for S, such that $l_a(C) \leq H(S) + 1$.*

Proof. Define $l(s) = \lceil \log_2 \frac{1}{p(s)} \rceil$.

$$\sum_{s \in S} 2^{-l(s)} = \sum_{s \in S} 2^{-\lceil \log_2 \frac{1}{p(s)} \rceil} \leq \sum_{s \in S} p(s) = 1$$

From Kraft-McMillan inequality, $\exists$ prefix code C',

$$l_a(C') = \sum_{s \in S} p(s) l(s) = \sum_{s \in S} p(s) \left\lceil \log_2 \frac{1}{p(s)} \right\rceil \leq \sum_{s \in S} p(s) \left(\log_2 \frac{1}{p(s)} + 1 \right) = H(S) + 1$$

■

2 Huffman Code

Given a set of messages with probabilities $p_1 \leq p_2 \leq ... \leq p_n$, the Huffman code tree is constructed by recursively combining subtrees:

1. Begin with n trees, each consists of a single node corresponding to one message word, with the weight of p_i
2. Repeat until there is only one tree
   - pick two subtrees with smallest weights
   - combine them by adding a new node as root, and make the two trees its children. The weight of the new tree is the sum of the weight of two subtrees

With a heap, each step of combining tree takes $O(\log n)$ time, and the total time is $O(n \log n)$.

Lemma 2.1 *Suppose C is the optimal code for S, p_1, p_2 and l_1, l_2 are the probabilities and code lengths of messages s_1 and s_2, respectively. Then $p_1 > p_2 \Rightarrow l_1 \leq l_2$.*

Proof. Suppose $p_1 > p_2$ and $l_1 > l_2$, we swap the code words for s_1 and s_2, and get a new code C'. The length of C' is $l_a(C') = l_a(C) + p_1(l_2 - l_1) + p_2(l_1 - l_2) = l_a(C) + (p_1 - p_2)(l_2 - l_1) < l_a(C)$. This contradicts the optimality of code C. ■

Lemma 2.2 *Without loss of generality, the two messages of smallest probability occur as siblings in the code tree for an optimal code.*

Proof. Given the code tree for an optimal code, we will show that it can always be modified without increasing the average code length so that the two smallest probability nodes are siblings. From Lemma 2.1, the smallest probability node must occur at the largest depth in the code tree. Note that the sibling of this node is also at the same depth. Now the sibling can be swapped with the second smallest probability node to obtain a code tree of the desired structure. This transformation does not increase the average code length. ■

Theorem 2.3 *The Huffman code is an optimal prefix code.*

Proof. The proof proceeds by induction on n, the number of messages.

Suppose the Huffman code is optimal for all sets of n proabablities $\{p_1 \leq p_2 \leq ... \leq p_n\}$. Look at the case of $\{p_1 \leq p_2 \leq ... \leq p_n \leq p_{n+1}\}$. We build a Huffman tree for this case by first combining p_1 and p_2. Then we look at the new nodes of $\{p_1 + p_2, p_3, ..., p_{n+1}\}$. We know that there is an optimal Huffman tree T' for it. Suppose the depth of the node for $p_1 + p_2$ in T' is d. So the code length of T is given as $l_a(T) = l_a(T') + (d+1)(p_1 + p_2) - d(p_1 + p_2) = l_a(T') + p_1 + p_2$.

From Lemma 2.2, there exists an optimal code tree with p_1 and p_2 as siblings. By the inductiev hypothesis, T' is optimal for the set of n probabilities obtained by combining p_1 and p_2. So the optimal tree must have code length at least $l_a(T') + p_1 + p_2$. Thus T is the optimal code tree. ■

Question: Where did we use the fact that we compared ourselves to the optimal prefix code ?

2.1 Prefix codes for larger alphabets

How do we build an optimal prefix code for the ternary alphabet {0, 1, 2} ?

Suppose we have the following set of probabilities: {0.1, 0.2, 0.2, 0.5}. The first attempt to generalize the Huffman algorithm would be to simply combine the smallest three probabilities at each stage. This generates the following tree: ((0.1) (0.2) (0.2)) (0.5)). This not optimal, because there is one empty slot wasted at the highest level.

Solution: We introduce dummy nodes of probability 0 to fill any empty slots. For a full ternary tree without any empty slot, the number of leaves should be $3 + 2k$ for some integer k. (This comes from the process of "growing" a ternary tree. We start from a 1 level ternary tree with 3 leaves. And each time we add 3 nodes to be subtree of one leaf, increasing the number of leaves by 2).

With this modification, we obtain the optimal tree which is (((0) (0.1) (0.2)) (0.2) (0.5)).

3 Problems with Huffman Coding

The Huffman code has the property that $H(S) \leq l_a(C) \leq H(S) + 1$. So up to one bit per character can be wasted.

Conside the case then $H(S) << 1$. e.g. alphabet $\{0, 1\}$. $p(0) = 0.9999$, p(1)=0.0001.$H(S) =$ 0.00147... If a Huffman code is used, each message takes at least one bit, which is much higher than $H(S)$.

One possible solution is to combine k consecutive messages into one message word. Then the entropy becomes $kH(S)$, while the average length of the Huffman code would be $kH(S)+$ 1, i.e. a worst case wastage of $1/k$ bits per message. However, if the original number of messages was m, this produces m^k messages (and hence m^k codewords).

How does this compare with the discussion of the entropy of English in the last class ? There too, we combined multiple characters to get an estimate of the entropy. However, there the combination of characters was done to exploit the dependence between consecutive characters and reduce the entropy estimate.

Consider another example:

Symbol	prob.
a	0.2
b	0.4
c	0.2
d	0.1
e	0.1

Here are two possible Huffman trees:

1. (((d, e), b), (a, c))
2. ((((d, e), a), c), b)

Which one is better? In certain applications, we would like to minimize the variance of code word length, defined as:

$$\sum_{c \in C} \{p(c)(l(c) - l_a(C))^2\}$$

This is achieved by an extension to the Huffman algorithm. When combining trees, break ties by picking the earliest produced subtrees with same smallest probability.

Question: Is this equivalent to breaking ties by picking "shortest" subtrees (for some appropriate definition of shortest) ?

4 Arithmetic Coding

One way to avoid the wastage in Huffman coding is to use arithmetic coding. Here the idea is to associate message sequences with intervals between 0 and 1.

Example: for alphabet $\{m_1, m_2, m_3\}$, with $p(m_i) = \frac{1}{3}$.

$$
\begin{array}{ccccccc}
0[& m_1 & | & m_2 & | & m_3 &]1 \\
 & & & \Downarrow & & & \\
 & \Longleftarrow & & & & \Longrightarrow & \\
\frac{1}{3}[& m_1 & | & m_2 & | & m_3 &]\frac{2}{3} \\
 & \Downarrow & & & & & \\
\Leftarrow & & & \Longrightarrow & & & \\
\frac{1}{3}[& m_1 & | & m_2 & | & m_3 &]\frac{4}{9} \\
 & \ldots & & & & &
\end{array}
$$

The above graph shows the encoding of message sequence $\{m_2, m_1, ...\}$. We begin from the whole interval of [0, 1]. With the first message m_1, we split the interval proportionally according to the probabilities of the messages. Then we use the sub-interval corresponding to m_1 as the current interval. With each consecutive messages, we repeat the above process to get smaller and smaller intervals. The final interval is the encoding for the whole message sequence.

Formally, for messages $\{m_1, m_2, \ldots, m_k\}$ with probabilities $\{p_1, p_2, \ldots, p_k\}$, $\sum p_i = 1$, we divide the interval $[0, 1]$ into k intervals. The ith interval corresponding to p_i spans from $\sum_{j<i} p_j$ to $\sum_{j\leq i} p_j$. Denote $\sum_{j<i} p_j$ as d_i.

The code for a sequence of x messages $m_{k_1}, m_{k_2}, ..., m_{k_x}$ is calculated by constructing a sequence of intervals $[l_i, l_i + s_i]$, where

$l_0 = 0, s_0 = 1;$

$l_1 = d_{k_1}, s_1 = p_{k_1};$

. . .

$l_{i+1} = l_i + s_i * d_{k_{i+1}}, s_{i+1} = s_i * p_{k_{i+1}}$

The final encoding for sequence $\{m_{k_1}, m_{k_2}, ..., m_{k_x}\}$ is interval $[l_x, l_x + s_x]$.

Now, if we construct the intervals corresponding to all message sequences of length l, they form a disjoint partition of $[0, 1]$. A given interval uniquely determines a message sequence of length l. The size of the interval for message sequence $m_1, m_2, ..., m_l$ is $\prod p(m_i)$. We will show how we can represent an interval $[l, l+s]$ using $\sim \log_2 \frac{1}{s}$ bits. Note that the sum of the self informations for the sequence $m_1, m_2, ..., m_l$ is $\sum \log_2 \frac{1}{p(m_i)} = \log_2 \frac{1}{\prod p(m_i)}$ Next, we show how we can obtain a prefix free code for representing intervals.

The idea is to represent the interval by a number inside the interval which has few bits in its binary representation. However, such a number has to be chosen carefully in order to ensure that the code is prefix free. In order to ensure the prefix free property, for any b bit binary number x, we associate the interval consisting of all numbers which agree with x in the first b bits. (The interval associated with x is exactly $[x, x + \frac{1}{2^b}]$. Now, for an interval

$[l, l+s]$ we will represent it by a number $x \in [l, l+s]$ such that the interval associated with x is completely contained in $[l, l+s]$. This ensures that the code is prefix free.

Theorem 4.1 *The interval* $[l, l+s]$ *can be represented by at most* $\lceil \log_2 \frac{1}{s} \rceil + 1$ *bits.*

Proof. In order to represent $[l, l+s]$, we consider the mid point $l + s/2$, take its binary fractional representation and truncate it to $\lceil \log_2 \frac{1}{s} \rceil + 1$ bits. We claim that the interval associated with this number is completely contained in $[l, l+s]$. First, note that the resulting value is still in the original interval, since truncating can reduce the number by at most $2^{-(\lceil \log_2 \frac{1}{s} \rceil + 1)} \leq s/2$. Also, the interval associated with the truncated number has length at most $s/2$, hence it lies completely in $[l, l+s]$. ■

The above theorem gives a way to represent intervals as binary codes. The interval corresponding to any two message sequences of length n do not overlap. So for an interval $[l, l+s]$, we can use the truncated binary fractional representation as the code word.

Theorem 4.2 *For a sequence of* n *messages with self informations* $s_1, s_2, \ldots, s_n$*, the length of the encoding produced by arithmetic coding is at most* $2 + \sum_{i=1}^{n} s_i$*.*

Proof. Let p_i be the probability of message m_i. Then the length of the arthmetic code is at most $\lceil \log_2 \frac{1}{s} \rceil + 1$ where $s = \prod_{i=1}^{n} p_i$. Now, $\lceil \log_2 \frac{1}{s} \rceil + 1 = \lceil \sum \log_2 \frac{1}{p_i} \rceil + 1 \leq \sum \log_2 \frac{1}{p_i} + 2$. Note that $\log_2 \frac{1}{p_i}$ is the self information of message m_i. ■

5 Drawbacks of Arithmetic Coding

1. Long sequences of messages need arithmetic with arbitrarily high precision.

2. In the worst case, we may have to see the whole message sequence to produce the first bit of the code. This could happen for example when the interval corresponding to the message sequence straddles $1/2$.

One way to alleviate both of these problems is to use arithmetic coding of blocks of k messages for a suitably chosen value of k. Also, there are integer based variants of arithmetic coding which do not produce the optimal code, but are easier to implement since they do not require arbitrary precision arithmetic.