6.897: Advanced Data Structures Spring 2003

Lecture 13 (Compression) — April 9, 2003

Prof. Erik Demaine *Scribes: Percy Liang, David Malan*

1 Overview

The motivation for compression is the following: a sender S wishes to transmit some data—a string s of letters—through a channel to a receiver R. S encodes s into some code c (hopefully smaller than s) and sends c across the channel. R decodes c to recover s. We want to take advantage of knowledge about the probability distribution of s to minimize the expected number of bits in c.

First, we consider the case where the distribution of s is based on a stochastic source that generates a sequence of independent letters. We examine how Huffman coding (Section 2) and arithmetic coding (Section 3) attempt to achieve entropy.

Next, we see how we can do better if our data comes from a non-stochastic source. We generalize the notion of entropy (Section 4) and introduce the Burrows-Wheeler Transform (BWT) to achieve kth-order entropy (Section 5).

2 Huffman Coding

Assume that our stochastic source generates letters from a finite alphabet Σ such that letter i is generated with probability p_i. The probability distribution of letters is static.

2.1 Goals

The idea of Huffman coding [1] is to construct a *prefix code* for Σ: associate each letter i with a codeword (bit string) w_i. The goal is to minimize the average codeword length:

$$E\big[|w_i|\big] = \sum_{i=1}^{|\Sigma|} p_i |w_i|.$$

To be a prefix code, no w_i can be a prefix of another w_j. This means that, when the receiver is reading the incoming stream of codewords, reading w_i unambiguously refers to letter i. A binary prefix code corresponds to all the root-to-leaf paths of a binary tree. See an example in Figure 1.

The receiver does not need to know the probability distribution, but only the codewords. (Of course, given the probability distribution, one can recover the codewords if some canonical way of breaking ties between equal probabilities in the Huffman algorithm is established.)

Letter i	p_i	w_i
A	0.1	000
B	0.1	001
C	0.2	01
D	0.3	10
E	0.3	11

Table 1: $\Sigma = \{A, B, C, D, E\}$, and each letter's probability p_i is given in the table. The codewords w_i are an example of an optimal Huffman code. Figure 1 shows the corresponding binary tree of this code.

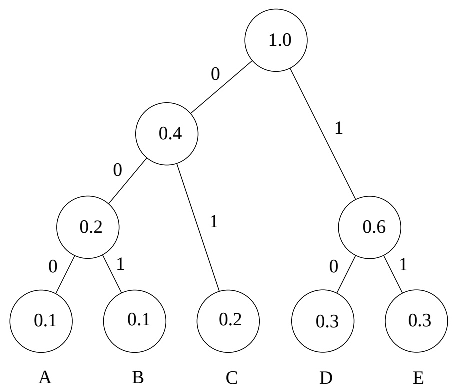

Figure 1: The binary tree of the example Huffman code in Table 1. Each node v is labelled with the sum of the probabilities of the leaves in the subtree rooted at v. The labels of the edges traversed in a root-to-leaf path determine the codeword for the letter corresponding to that leaf.

2.2 Algorithm

Now we present an algorithm for constructing the binary tree of a Huffman code. Note that this problem is similar to constructing an optimal BST, but there are two differences: (1) the order of the leaves is irrelevant and (2) only root-to-leaf paths matter with respect to optimality. Because the problem is less constrained, we can solve it very efficiently using a greedy algorithm.

The algorithm is as follows:

1. Start with the $|\Sigma|$ disconnected leaves, which are trivial subtrees.
2. Choose the two subtrees t_1 and t_2 with smallest p_{t_1} and p_{t_2}.
3. Create a new internal node t with the two children t_1 and t_2. t defines a subtree with probability $p_t = p_{t_1} + p_{t_2}$.
4. Until we have a single tree, repeat Steps 2–3. The root node is added last with probability 1. Steps 2–3 are repeated $|\Sigma| - 1$ times.

2.3 Properties

Theorem 1. *A Huffman code minimizes* $E\big[|w_i|\big] = E\big[\text{codeword length}\big] = E\big[\text{leaf depth}\big]$.

Proof. A formal proof is given in [1]. The main idea is that we first show that in an optimum tree, if a leaf v_a has a lower probability than v_b, then the depth of v_a is at most the depth of v_b. Then if the optimal code is a non-Huffman code, we can always greedily interchange two leaves to improve (or not worsen) the expected depth of a leaf. $\square$

Theorem 2. $E\big[\text{huffman codeword length}\big] \in [H, H+1)$, *where H is the entropy of the probability distribution of Σ.*

Proof. Consider an infinite complete binary tree. The claim is that you can, for each letter i, pick a node l_i in the binary tree such that $\text{depth}(l_i) = \lceil \lg \frac{1}{p_i} \rceil$ and l_i is not an ancestor of any l_j. Then, take the tree on those leaves $l_1, l_2, \ldots, l_{|\Sigma|}$, and we have

$$E\big[\text{leaf depth}\big] = \sum_{i=1}^{|\Sigma|} p_i \left\lceil \lg \frac{1}{p_i} \right\rceil < H + 1.$$

Suppose $p_1 \geq p_2 \geq \cdots \geq p_n$. Then pick $l_1, l_2, \ldots, l_n$ in order of increasing leaf depth. After picking l_i arbitrarily, there are still $\lceil \lceil \frac{1}{p_i} \rceil \rceil - 1$ nodes at that depth remaining, which is enough to support the $1 - p_i \leq 1 - \lfloor \lfloor p_i \rfloor \rfloor$ remaining probability. $\square$

We can also generalize Huffman codes to nonbinary codes (nonbinary trees), or to k-grams instead of letters, but we omit the details.

3 Arithmetic Coding

3.1 Motivation

Consider an example where $\Sigma = \{a, b\}$ with $p_a = \frac{1}{1024}$ and $p_b = 1 - \frac{1}{1024}$. One can verify that $H \approx 0.01$. However, a Huffman code is $w_a = 0$ and $w_b = 1$. The expected code length $E\big[|w_i|\big]$ is obviously 1, which is about 100 times greater than the entropy H!

In this example, Huffman coding is terrible in the multiplicative sense. The problem is that codewords can only be an integer number of bits, which forces us to round up to 1 no matter how small the entropy is. A clever idea is to aggregate several fractional bits into one bit. This is done in arithmetic coding [4]. Instead of losing 1 bit per letter, we lose 1 bit over the entire string. Thus, for long strings, we basically achieve entropy.

3.2 The Code

The idea is to encode each string as a unique real number in $[0, 1]$. We divide the $[0, 1]$ interval into $|\Sigma|$ first-level subintervals, with the length of each subinterval i equal to p_i. Then, we recursively divide each subinterval in the same way, scaling accordingly. Given a real number r, the first-level subinterval of $[0, 1]$ containing r determines the first letter of the decoded string. Which subinterval of the second-level subinterval contains r determines the second letter, and so on. The sender must include the length of the string (an additional $O(\lg n)$ bits) with the code so that the receiver knows when to stop recursing.

For infinite strings, the code produces a real number, but for finite strings, there is a interval of acceptable real numbers $(L, R]$. (We use L and R to represent both the real numbers and their corresponding bit-string encodings.) When encoding a string, we arbitrarily pick the number $\text{lcp}(L, R)1$. Here $\text{lcp}(L, R)$ is the longest common prefix of the bit strings of L and R. If we think of the bit strings as having an infinite number of trailing zeros, then after $\text{lcp}(L, R)$, L reads $0\ldots$ and R reads $1\ldots$, so clearly, $\text{lcp}(L, R)1$ is in the interval $(L, R]$.

Specifically, we compute the code of a string as follows:

$$\text{code}(x) = \begin{cases} (0, 1] & \text{if } |x| = 0, \\ \sum_{i=1}^{k-1} p_i + p_k \text{code}(y) & \text{if } x = k \cdot y. \end{cases}$$

$\text{code}(x)$ returns an interval, but we actually pick a real number in the interval as discussed above.

Unlike Huffman coding, in arithmetic coding, the receiver needs to know the probability distribution in order to decode.

3.3 Example

Consider an example where $\Sigma = \{1, 2, 3\}$ with $p_1 = 1/2$, $p_2 = 1/4$, and $p_3 = 1/4$. Figure 2 shows the partitioning of $[0, 1]$. For example, the string 32 maps to $\frac{1}{2} + \frac{1}{4} + \frac{1}{4}(\frac{1}{2} + \frac{1}{4}) = \frac{15}{16}$.

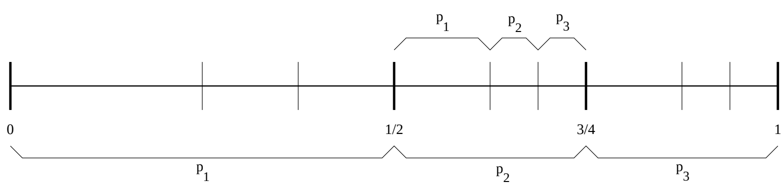

Figure 2: An example recursive partitioning of $[0, 1]$ based on the probability distribution.

3.4 Properties

Theorem 3. $E\big[\text{total code length}\big] \in [Hn, Hn + 1)$*, where H is the entropy of the probability distribution of Σ and n is the length of the string.*

Proof. Suppose the string $x = x_1 x_2 \cdots x_n$. We can compute code length exactly:

$$\begin{aligned} & \left\lceil \lg \frac{1}{p_{x_1}} + \lg \frac{1}{p_{x_2}} + \cdots + \lg \frac{1}{p_{x_n}} \right\rceil \\ = & \left\lceil \lg \frac{1}{\underbrace{p_{x_1} p_{x_2} \cdots p_{x_n}}_{\text{length of final interval}}} \right\rceil \\ = & \lg \frac{1}{\underbrace{\lfloor \lfloor p_{x_1} p_{x_2} \cdots p_{x_n} \rfloor \rfloor}_{\text{dividing into this power of 2 leaves a notch in the interval}}} \\ < & 1 + \lg \frac{1}{p_{x_1}} + \lg \frac{1}{p_{x_2}} + \cdots + \lg \frac{1}{p_{x_n}} \end{aligned}$$

Thus, $E\big[\text{total code length}\big] < 1 + Hn$. [Recall $\lfloor\lfloor x\rfloor\rfloor$ denotes the *hyperfloor* of x, which is equal to $2^{\lfloor \lg x\rfloor}$ (round down to the nearest power of 2).] $\square$

There are variations of arithmetic coding to speed up computation by using less precision and avoiding floating-point altogether, but again we omit the details. See e.g. [3].

4 Nonstochastic Sources

Thus far, we've been assuming stochastic strings, the frequency of whose characters is given only by some distribution. But what if there are correlations among characters? What if characters aren't independent? (Consider, for instance, the likelihood with which 'u' follows 'q' in English.)

4.1 Empirical 0th-order entropy

The *empirical* 0th-order entropy of a (finite) string x measures the frequency f_i (number of occurrences of each letter i) in x and assigns $p_i := f_i/n$ as "probabilities" (really relative frequencies), where $n = |x|$. Similar to before,

$$H(x) = H_0(x) = \sum_{i=1}^{|\Sigma|} p_i \log \frac{1}{p_i}.$$

Unfortunately, the empirical 0th order entropy of some string x fails to capture any correlation among x's characters. Some other analysis is neeeded.

4.2 Empirical kth order entropy

Let's instead compute the empirical kth order entropy of some string x, in order that we may analyze x's characters in the context of others.

$$H_k(x) = \sum_{|w|=k} \Pr[w \text{ occurring}] \times H_0(\text{successors}(w)),$$

where $|w| = k$ is assumed to be asymptotically smaller than n, $\Pr[w \text{ occurring}]$ is given by (# occurences of w)$/n$, and successors(w) represents the string of next letter after each occurrence of w (the order of which is irrelevant).

$|x|H_k(x)$, therefore, is a lower bound on compression by a kth-order code—a code depending on a string's last k letters. Accordingly, $H_{k+1} \leq H_k \forall k \geq 0$.

4.2.1 Realistic?

But is this bound realistic, particularly in light of the fact that $H_k(aaaaaaaaa)$, for instance, is 0?! Let us not forget that kth-order entropy does not compute length, essentially because, in information theory, length is assumed infinite. Let us therefore offer a minimal fix [2] and define *modified entropy* as follows:

$$H_0^*(x) = \begin{cases} 0 & \text{if } |x| = 0 \\ (1 + \log|x|)/|x| & \text{if } |x| \neq 0 \text{ but } H_0(x) = 0 \\ H_0(x) & \text{otherwise.} \end{cases}$$

But wait: now depending on more characters can actually hurt! (Consider a long string of a's.) We should therefore define H_k^* to allow dependence on shorter strings—anything $\leq k$; we can then take the overall minimum. Only then will it be the case that $H_{k+1}^*(x) \leq H_k^*(x)$, so that (rather intuitively) more information about a string can only help us. We won't bother with a more formal definition; it is messy. See [2].

5 Burrows-Wheeler Transforms (BWT)

Let us now consider an algorithm that aspires to give us H_k—for all k! (For the curious, this is the algorithm used by bzip2!)

5.1 Transforming a String

In order to attain more effective compression for a substring w, BWT clusters successors(w) together, effectively partitioning w into successors(w_1)|successors(w_1)$|\cdots$, where $w_1, w_2, \ldots$ are all strings of length k.

The transformation is as follows:

1. Append \$ to the end of the string.
2. Sort all n rotations of the string *in reverse order* (i.e., in the same order as suffixes of reverse(string)).
3. Write down the first letter in each rotation in order.

Let's consider our favorite word: *banana*.[1]

$$\begin{array}{ccccccc} n & a & n & a & \$ & b & a \\ n & a & \$ & b & a & b & a \\ \$ & b & a & n & a & n & a \\ a & n & a & n & a & \$ & b \\ a & n & a & \$ & b & a & n \\ a & \$ & b & a & n & a & n \\ b & a & n & a & n & a & \$ \end{array}$$

According to BWT, the encoding of *banana* is given by the leftmost column above: *nn\$aaab*. Notice the structure of this string: similar characters are clustered together! Moreover, consider the structure of the table itself: common contexts (i.e., previous letters) are clustered together, as in the case of *"ana"* in the table's second and third lines. Similarly, all successors of *"ana"* (i.e., n and \$) are clustered together in some order (in the leftmost column).

[1] "I know how to spell banana; I just don't know when to stop!" — a friend of Erik's

5.2 Reconstructing a String

But can we reconstruct a string from its BWT form? Yes. Consider the design of this table:

1. Every column is a permutation of the original string.
2. Accordingly, the rightmost column = sort(first column) = sort(BWT).
3. The successor of (row in last column) = (row in first column).
4. The ith occurence of a character in the first column is also the ith occurence of the character in the last column (because of lexical sorting).

The implication of this design is that we can unwind BWT's first character, followed by its second character, and so on, in $O(n)$ time using hashing.

How might we reconstruct *banana* from *nn\$aaab*? Well, we can derive the original string's first character (b) by locating the first character after \$. We can then find the original string's second character by locating the first character after b: fortunately, we have but one option in the table, a. Finding the third character is trickier (but doable!). We need to find the character after a: unfortunately, we seem to have three (non-distinct) choices, per the table's top three rows. However, bear in mind that the table has maintained a sorted order. Hence, just as we chose a from the leftmost column of the table's fourth row (that a being the first in a sequence of three in that column), so should we choose the first candidate for the original string's third character: n! The remainder of the string can be derived similarly.

5.3 Performance

Let's examine the performance of BWT. We note again that, for any substring w, BWT puts successors(w) together. That is, it partitions the string into successors(w_1)|successors(w_1)| $\cdots$. If we could encode $x = x_1 x_2 \cdots x_r$ in $\sum_{i=1}^{r} |x_i| H_0(x_i)$, then encoding with BWT would give us H_k—for all k!

Let's do just that, albeit roughly. But first, a definition.

MTF Transform: Instead of writing a letter, write the index of that letter in a list and move it to the front. (The resulting string, of course, will have the same length and the same alphabet size.)

We now put forth the following theorem without proof.

Theorem 4. [2] *BWT + MTF + arithmetic coding* $\leq 8 \cdot |T| \cdot H_k(T) + \frac{2}{25} \cdot |T| + O(\Sigma^{k+1} \log \Sigma)$, $\forall k$. *(The* $O(\Sigma^{k+1} \log \Sigma)$ *term is like* k*th-order Huffman.)*

Unfortunately, this $\frac{2}{25}$ for every letter is annoying; it seems artificial, so let's try to get rid of it with the following alternative.

Runlength Transform: Replace a string of 0's with a binary string encoding the length using special $\boxed{0}$ / $\boxed{1}$ symbols for clarity.

In other words,

$$1\ 2\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 5\ 3\ \ldots$$

becomes

$$1\ 2\ \boxed{1}\ \boxed{1}\ \boxed{0}\ 5\ 3\ \ldots.$$

We now offer a final theorem, also without proof.

Theorem 5. *BWT + MTF + runlength + arithmetic coding* $\leq 5 \cdot |T| \cdot H_k^*(T) + f(k)$ *[2].*

Exactly how much these constants can be improved remains an open question.

References

[1] David A. Huffman. A Method for The Construction of Minimum Redundancy Codes. *Proceedings of the IRE*, 40(9):1098–1101, 1952.

[2] Giovanni Manzini. An analysis of the Burrows-Wheeler Transform. *Journal of the ACM*, 48(3):407–430, 2001.

[3] Alistair Moffat, Radford Neal, Ian H. Witten. Arithmetic Coding Revisited. *ACM Transactions on Information Systems*, 16(3):256-294, July 1998.

[4] Ian H. Witten, Radford M. Neal, and John G. Cleary. Arithmetic Coding for data Compression. *Communications of the ACM*, 30(6):520–540, 1987.