# *Introduction to information theory and data compression*

*Adel Magra, Emma Gouné, Irène Woo*

*March 18, 2017*

This is the augmented transcript of a lecture given by Luc Devroye on March 9th 2017 for a Data Structures and Algorithms class (COMP 252).

Data compression involves encoding information using fewer bits than the original representation.

## *Information Theory*

Information theory[1] is the study of quantification, storage, and communication of information. Claude Shannon developed the mathematical theory that describes the basic aspects of communication systems. It is concerned with the construction and study of mathematical models using probability theory.

[1] Thomas and Cover [2006]

In 1948, Shannon published his paper "A Mathematical Theory of Communication" in the Bell Systems Technical Journal [2]. The paper provided a "blueprint for the digital age." Figure 1 illustrates a general communication system as Shannon proposed in his paper.

[2] Shannon [1948]

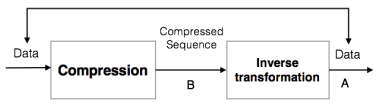


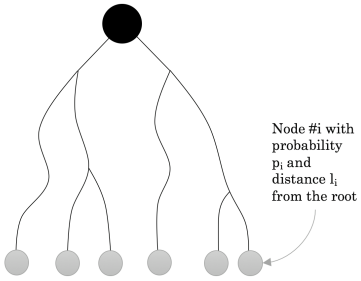

Figure 1: Communication system diagram

We can calculate the compression ratio C as: $C = \frac{length(B)}{length(A)}$

## *Shannon's Theory*

Shannon imagined that every possible input sequence that may have to be compressed has a given probability $p_i$, where the $p_i$'s sum to one. So if we transform the i-th input sequence into one having $\ell_i$ bits, the expected length of the output bit sequence is $\sum_i p_i \ell_i$.

One can reverse engineer a transformation (or compression) algorithm and construct a binary tree that maps every binary output back to an input. It is similar to the old decision tree we saw when arguing about lower bounds. Leaves correspond to possible inputs.




What matters is to find a compression method that minimizes $\sum_i p_i \ell_i$. Theoretically, this can be done by finding the Huffman tree using the Hu-Tucker algorithm (see section "Practice"). Since the number of possible inputs is incredibly large, one cannot possibly use Huffman to actually do it. In addition, $p_i$ is generally unknown.

However, one can still learn things from Shannon about $\sum_i p_i \ell_i$.

His main theorem is that:

$$\mathcal{E} + 1 \geq \min \sum_i p_i \ell_i \geq \mathcal{E}$$

where $\mathcal{E}$ is the binary entropy, $\sum_i p_i \log_2 \frac{1}{p_i}$, and the minimum is over all possible binary trees (and thus, all compression algorithms).

As a special case, putting $p_i = \frac{1}{n}$, where $n$ is the total number of possible answers for a particular algorithmic problem (such as sorting), we rediscover the decision tree lower bound seen earlier in the course: the expected number of binary oracle comparisons must be $\mathcal{E} = \log_2 n$.

We will prove Shannon's theorem in the next section.

*Entropy (symbol $\mathcal{E}$)*

In Information Theory, entropy ($\mathcal{E}$) is a number defined to be the measure of the average information content delivered by a message. It measures the unpredictability of the outcome. The binary entropy is defined by

$$\mathcal{E} = \sum_i p_i \log_2 \frac{1}{p_i} \geq 0,$$

where the $p_i$'s are the probabilities of the input sequences. We will prove that

$$\mathcal{E} + 1 \geq \min \sum_i p_i \ell_i \geq \mathcal{E},$$

where the minimum is over all binary trees.

Recall Kraft's inequality, which is valid for all binary trees:

$$\sum_i \frac{1}{2^{\ell_i}} \leq 1.$$

**Remark**: The converse of Kraft's inequality is also true, i.e., given numbers $\ell_1$, $\ell_2$, ... with $\sum_i \frac{1}{2^{\ell_i}} \leq 1$, there exists a binary tree such that its leaves have those depths.

We first show: $\sum_i p_i \ell_i \geq \mathcal{E}$.

Observe that:

$$\begin{aligned}
\sum_i p_i \ell_i &= \sum_i p_i \log_2 2^{\ell_i} \\
&= \sum_i p_i \log_2 \left( 2^{\ell_i} p_i \frac{1}{p_i} \right) \\
&= \sum_i p_i \log_2 (2^{\ell_i} p_i) + \sum_i p_i \log_2 \left( \frac{1}{p_i} \right) \\
&= \sum_i p_i \log_2 (2^{\ell_i} p_i) + \mathcal{E}.
\end{aligned}$$

Now, $\sum_i p_i \log \frac{1}{2^{\ell_i} p_i} \leq \sum_i (p_i(\frac{1}{2^{\ell_i} p_i} - 1)) = \sum_i \frac{1}{2^{\ell_i}} - 1 \leq 0$ since $\log x \leq x - 1$ and by Kraft's inequality. Thus, clearly $\sum_i p_i \ell_i \geq \mathcal{E}$.

Now we show: $\mathcal{E} + 1 \geq \sum_i p_i \ell_i$ for the so-called Shannon - Fano code. In this code, we take $\ell_i = \lceil (\log_2(\frac{1}{p_i})) \rceil$.
We have $\sum_i \frac{1}{2^{\ell_i}} \leq \sum_i p_i \leq 1$.

Thus, by the converse of Kraft's inequality, there exists a code that has length $\ell_i$ for input $i$. That is the Shannon - Fano code. Now, $\sum_i p_i \ell_i = \sum_i p_i \lceil (\log_2(\frac{1}{p_i})) \rceil \leq \sum_i p_i \log_2 \frac{1}{p_i} + \sum_i p_i = \mathcal{E} + 1$. So we are done.

In conclusion, $\mathcal{E}$, measured in "bits," corresponds to how well one can hope to compress a file given the assumption on the "$p_i$"'s.

## *Practice*

In practice, we will compress either symbols or small chunks of symbols. There is a separation problem on the part of the receiver. How do we separate a bit sequence if we transform each symbol in the input, symbol per symbol, into a small bit sequence? Indeed, when we send an encoded string, a concatenation of "codewords," the receiver might not be able to parse correctly the string in order to decode each string portion or codeword.

**Example 1.** Suppose we want to send a sequence of integers such as 18 23. Its standard bit representation is 10010 and 1011. Sending the sequence 1001010111 is not very useful. Where does the first portion start or end? How large is the portion? A bit number always starts with a 1, hence one can send a prefix that starts with 0 and indicates the length of codeword. We have then: $_{00101}10010_{00101}10111$ Another method is to add a prefix of 0's of the same length of the codeword. We have: $_{00000}10010_{00000}10111$. Still, this method is inconvenient and slightly wasteful.

Fixed width coding, as for example in the standard 8-bits per character coding, gives another solution. It can be vastly improved. A first such improvement is (variable width) prefix coding. A codeword assigns a sequence of bits to a symbol. A code is a set of codewords. One can picture a code as a trie (defined below). In a prefix code, each leaf uniquely corresponds to an input symbol. In this manner, the separation problem will be solved.

### *Tries*

A trie (pronounced "try") is a tree-based data structure for storing strings in order to support fast pattern matching. The main applica-

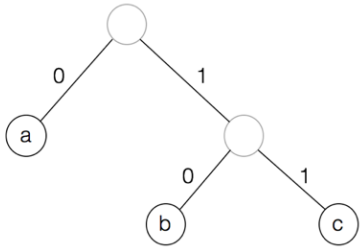
tion of a trie is information retrieval, thus it is not surprising that the name ”trie” comes from the word ”retrieval”.

In a binary trie, each edge represents a bit in a codeword, with an edge to a left child representing a ”0” and an edge to a right child representing a ”1”. Each leaf is associated with a specific character. Using a trie we can develop a strategy for the coder and decoder. The coder would search for a character and then go backwards to the root while recording the path. This step can be done using pointers to the parent node.

The decoder doesn't need to have the code trie. She/he only needs to process the code to find a leaf and go backward to the root. For efficiency we usually send the trie along with the coded message.

For prefix coding, the sender (or coder) has a prefix coding tree and uses either a table of codewords or parents pointers in the tree to do his coding. The receiver needs the tree (which is sent in some way), and with the tree, one can easily decode the sequence symbol by symbol as leaves correspond to input symbols.

**Example 2.** A simple example we can make is to encode the alphabet a,b,c with bits:

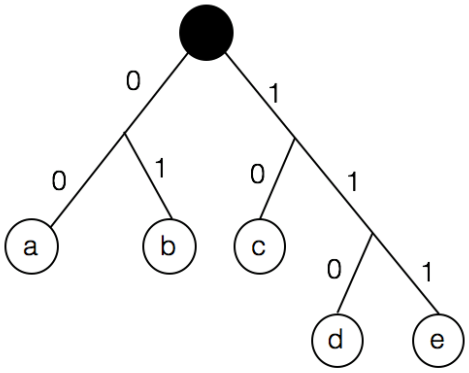



The leaves correspond to all the possible inputs. Here 0 maps to a, 10 maps to b and 11 maps to c.

*Prefix Coding*

**Definition 3.** Prefix coding is a coding system in which variables (words in English text) are differentiated by their prefix attribute.

We give an example with a trie and an alphabet composed of 5 letters a,b,c,d,e. Each letter is attributed a prefix code which is a proper binary sequence. Let $P$ be the prefix code: $P$ = a:00,b:01,c:10,d:110,e:111. We clearly see that P is a valid prefix code as no binary sequence is the prefix of another in $P$. We can view this prefix code as a code tree:




| The | code |
|---|---|
| a | 00 |
| b | 01 |
| c | 10 |
| d | 110 |
| e | 111 |

In this example, the string ”abbba” is transformed into ”0001010100”. This bit sequence can be uniquely interpreted and decoded back into

"abbba" — in other words, we have solved the separation problem quite elegantly.

If we assume a certain probability $p_i$ on symbol $i$ (possibly approximated by the relative frequency of symbols in general input sequences that we would like to compress), then the expected length of a codeword — which ultimately tells us about the expected length of the compressed sequence — is again $\sum_i p_i \ell_i$. We should first of all design the code by using the Huffman tree. Such codes are called Huffman codes. By Shannon's theorem, observe that for the Huffman code, $\sum_i p_i \ell_i \leq \mathcal{E} + 1$.

### *Huffman Codes*

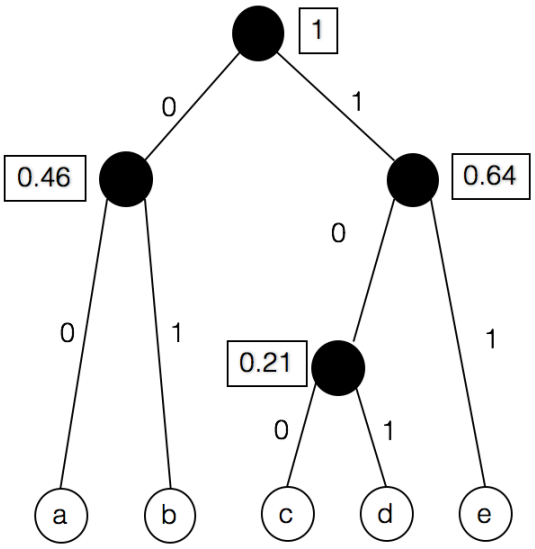
We can optimize a prefix code by taking into consideration the probability of different code words to occur. We could then construct a Huffman Tree [3]. The Huffman coding algorithm constructs a solution step by step by picking the locally optimal choice. It is called a greedy algorithm. Given a fixed tree with leaf distances $\ell_i$ and a certain assignment of symbols to the leaves, $\sum_i p_i \ell_i$ is minimized by placing the symbols i and j with smallest $p_i$ values furthest from the root. Therefore, since single child nodes are obviously suboptimal, the optimal tree has i and j as children of an internal node. This permits us to create one internal node and reduce the problem by one.

[3] Cormen et al. [2009]

The algorithm proceeds in a series of rounds.

**Algorithm**: First make each of the distinct characters of the string to encode the root node of a single-node binary tree. In each round, take the two binary trees with the smallest frequencies and merge them into a single binary tree. Repeat this process until only one tree is left.

**Example 4.** Let's show the algorithm with the following alphabet $i$ and probabilities $p_i$.




| i | $p_i$ |
|---|---|
| a | 0.20 |
| b | 0.26 |
| c | 0.06 |
| d | 0.15 |
| e | 0.33 |

As seen in the previous lecture (March 7, 2017), there is a complete algorithm for building a Huffman tree using binary heaps. Let's recall the method we used: The Huffman tree has n leaves and 2n-1 internal nodes. We build the Huffman Tree by filling the internal nodes with left and right children. We use the Hu-Tucker algorithm, which uses a priority queue (H).

HU-TUCKER(*n symbols with key i and probability $p_i$ are given*)

```
1  MAKE_EMPTY_PRIORITY_QUEUE(H)
2  For i from 1 to n do (to insert the leaves first)
3     LEFT[i] = 0
4     RIGHT[i] = 0
5     INSERT((p_i,i),H)
6  For i from n+1 to 2n-1 do (to implement the internal nodes)
7     (p_a,a) = DELETEMIN(H)
8     (p_b,b) = DELETEMIN(H)
9     LEFT[i] = a
10    RIGHT[i] = b
11    INSERT((p_a + p_r, i),H)
```

This algorithm outputs the Huffman tree. The root is node 2n - 1. Left and right children of nodes are stored in the arrays LEFT and RIGHT. The construction of a Huffman tree takes $O(n \log(n))$.

Finally, the entropy tells us about how well we can do. For example, $\mathcal{E}$ depends upon the language when the input consists of long texts.

## *Final Remarks*

Improvements are possible by grouping input symbols in groups of two, three, or more. Once can also employ adaptive Huffman coding, where the code is changed as a text is being processed (and the frequencies of the symbols change).

## *References*

Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. Introduction to Algorithms. 2009. Cambridge, MA.

Claude E. Shannon. A mathematical theory of communication. *The Bell System Technical Journal*, 27:379–423, 623–656, 1948.

Joy A. Thomas and Thomas M. Cover. *Elements of Information Theory*. Wiley Series, 2nd edition, 2006. New York.