Data Compression (Huffman Codes)

Lecture Notes by Raluca Budiu and Danny Sleator

March 16, 1999

The Problem

Given a set of data (i.e. a string of bits of length l), we want to be able to compress it to a string of smaller size through a compression algorithm and then decompress it and recover the initial data.

```
D ---> Compression ---> Compressed D ---> Decompression ---> D'
```

In general, if the input data D is random, you cannot expect to be able to compress it. Therefore, compression algorithms make certain assumptions about the data set. There exist specific algorithms for picture and sound compression, for instance (such compression algorithms put their output into special formats like "jpg" and "mpg").

If $D = D'$ the compression is called lossless; otherwise it is lossy. For some kinds of data (like picture or sound compression), lossless compression is acceptable, for others (e.g. text) it is not.

We will discuss four different data compression algorithms:

1. Huffman
2. Arithmetic codification
3. Lempel–Ziv
4. Wheeler–Burroughs

The first two are *non-adaptive* (they start with some a priori knowledge about the data source) and the others are "adaptive" (they build the source knowledge by themselves). The most naive implementation of Huffman coding and Arithmetic coding are non-adaptive, but they both have adaptive variants.

Non-adaptive Algorithms

Assumptions about the Data Set

Definition. An *Independent Identically Distributed (IID) source for data* is a source that produces a sequence of symbols $x_1, x_2 \ldots$, where each x_i

- is chosen from a finite alphabet $\alpha = (\alpha_1, \ldots, \alpha_n)$ of symbols,

- and is generated randomly according to some distribution of probability $p = (p_1, \ldots, p_n)$, with p_i being the probability of generating α_i.

An IID source is called "independent" because every symbol is generated independently of the others around it; it is called "identically distributed" because each symbol generated is chosen according to a distribution that is identical to the distribution used for all the others. (It does NOT mean that all symbols of the alphabet are equally likely.)

Example 1. The alphabet could be $\alpha = (a, b, c)$ and the distribution of probability $p = (1/2, 1/4, 1/4)$.

The first data compression problem is: how do we compress an IID source? Assume that p is known in advance.

(If we don't know p, we can estimate it by making an extra pass through the text and counting the frequency of each symbol. But the uncompressor needs p as well, so in this case we need to include p's description in the compressed file. In this way we can effectively make a non-adaptive algorithm adaptive.)

Using a set of binary codewords to compress an IID source

A *codeword* is a finite string of bits. A *code* is a finite set of codewords. A code is said to be *prefix-free* if no codeword is a prefix of any other codeword in the code. A code is said to be *uniquely deciferable* (or UD) if any word formed by concatenating codewords can be partitioned back in to codewords in a unique way.

Clearly any prefix-free code is uniquely deciferable – you can parse a string into codewords in a greedy fashion. Go from left to right, and the moment you find an initial section that is a codewords, you remove it and continue. This way of parsing the string is unique, since no codeword is a prefix of any other. Of course not all uniquely deciferable codes are prefix-free. For example, consider $\{1, 10, 00\}$. It's UD (because the reverse of all the codewords form a prefix-free code), but it's not prefix-free.

Theorem. Given a UD code with codewords of lengths $l_1, l_2 \ldots, l_n$. Then

$$\sum_{1 \leq i \leq n} \frac{1}{2^{l_i}} \leq 1$$

We will not prove this here. A proof can be found in chapter 3 of Shimon Evan's book on algorithms.

Our first compression algorithm is simply going to encode each symbol as a codeword. Our goal is to minimize the expected length of each encoded symbol. If our codewords have lengths $l_1, l_2 \ldots, l_n$. Then our goal is to minimize

$$\sum_{1 \leq i \leq n} p_i l_i$$

Theorem. Given a vector $p = (p_1, \ldots, p_n)$, of probabilities, the choice of $l_1, l_2 \ldots, l_n$ that minimizes $\sum_{1 \leq i \leq n} p_i l_i$ subject to the constraint that:

$$\sum_{1 \leq i \leq n} \frac{1}{2^{l_i}} \leq 1$$

is $l_i = \log 1/p_i$

Proof outline. Change variables by introducing $q_i = 1/2^{l_i}$. Now we're trying to choose the qs so as to minimize:

$$\sum_{1 \le i \le n} p_i \log \frac{1}{q_i}$$

subject to $0 \le q_i \le 1$ and $\sum q_i = 1$. For each pair i and j, if we keep the total of $q_i + q_j$ constant (and move weight between these two) it's easy to show by differentiation that the way to minimize $p_i \log(1/q_i) + p_j \log(1/q_j)$ is to choose $q_i/p_i = q_j/p_j$. Therefore if $q_i \ne p_i$ for some i then we can continue to move weights around until all the qs equal all the ps. Q.E.D.

So, given an IID source with probability distribution $p = (p_1, \ldots, p_n)$, the quantity

$$\sum_{1 \le i \le n} p_i \log \frac{1}{p_i}$$

We have proven that this is a lower bound on the expected number of bits that are required to send one symbol from the source. This quantity is known as the *entropy of the IID source.*

Entropy

Given a distribution of probability p, the previous section gives a lower bound on on how well you can compress an IID source. That bound is the entropy.

Definition. The *entropy of an IID source* of probability $p = (p_1 \ldots p_n)$ is $H(p) = \sum_{i=1}^{n} p_i \log(1/p_i)$.

Example 2. For the source in Example 1, we have:

$$H(p) = (1/2)\log 2 + (1/4)\log 4 + (1/4)\log 4 = 3/2.$$

Example 3. For a source whose alphabet has only 2 symbols with probabilities p and $1-p$, the entropy is:

$$H(p) = p\log(1/p) + (1-p)\log(1/(1-p)).$$

This function attains its maximum at $p = 1/2$: $H(1/2) = 1$.

Notice that the bitwise entropy of perfectly compressed data should be 1 (otherwise we may be able to recompress it). The entropy can be regarded as a measure of information in each symbol: the higher the entropy, the smaller the amount of information.

Entropy is important because of the following theorem, which is proved in the previous section:

Theorem. Given any algorithm for compressing an IID source, the expected number of bits per symbol compressed is at least the entropy.

We will actually construct an algorithm that can get as close as you want from this limit.

Example 4. How would we compress Example 1? We can make a binary code:

$$\begin{array}{rcl} a & \leftarrow & 0 \\ b & \leftarrow & 10 \\ c & \leftarrow & 11 \end{array}$$

The expected length per symbol would be $\sum p_i length(\alpha_i) = 1/2 \times 1 + 1/4 \times 2 + 1/4 \times 2 = 3/2$ – which is optimal!

Notice that one can draw a binary tree for this code:

```
      /\
    0/  \1
    /    \
   a     /\
       0/  \1
       /    \
      b      c
```

This tree has the property that all left edges are labeled with 0 and all right edges are labeled with 1. Moreover, all the leaves in the tree are alphabet symbols, and all the paths in the tree correspond to a symbol code.

If you can draw a tree like this for a code, the code is prefix-free (i.e. no codeword is the prefix of another). The reverse is true, as well.

Prefix-free codes can be parsed very easily – you can do it in a single pass, with no backtracking.

Example 5. With the code in the previous example, one can easily parse a text:

```
0|10|11|0|0|0|10|11|10
a b  c  a a a b  c  a
```

Example 6. For the following non prefix-free code it is harder to parse a text:

$$\begin{array}{rcl} a & \leftarrow & 10 \\ b & \leftarrow & 1 \\ c & \leftarrow & 01 \end{array}$$

If the text is 1010,

```
1|01|0
b c  fail

10|10
a  a
```

If the code is not prefix-free, you may still be able to use dynamic programming to decipher the text; or you may not be able to. In any case, a prefix-free code is much easier to decipher.

The next question is: can we build a prefix-free code that achieves the entropy? The answer is: no, we cannot quite do it, but we almost can: we can get within distance 1 of the entropy. And, by bootstrapping, we will be able to get arbitrarily close to the entropy (by considering a new source whose symbols will be pairs of original symbols).

We can reach the entropy if p_i are *dyadic numbers*: $p_i = 1/2^i$ (like in Example 4).

Dyadic Codes

Last time we saw that each prefix-free code corresponds to a binary tree: the leaves of the tree are the symbols and the paths form the codes.

Notice that sometimes the tree associated to a prefix-free code might have some gaps, like in the following example. In that case, we need to introduce dummy symbols to get rid of gaps.

Example 7. Let us consider the following code:

$$\begin{array}{ccc} a & \leftarrow & 01 \\ b & \leftarrow & 00 \\ c & \leftarrow & 10 \end{array}$$

The tree is:

```
     /\
   0/  \ 1
   /    \
  /\    /
0/  \1 / 0
b   a  c
```

Notice that c has no sibling: the place where its sibling should have been is called a gap. We can fill that gap by assigning the code 10 to a dummy symbol, say x.

Our goal is to build a prefix-free code whose average length comes as close as possible to the entropy.

To build that code, we will use the following algorithm. We are not claiming that this algorithm is *good*; we are introducing it solely for the purpose of analyzing the similar Huffman code we will describe later.

BuildDyadicCode(α, p)

1. round each of the probabilities p up to the next dyadic number: $p_i \leftarrow d_i$;

2. add extra symbols with dyadic weights such that $\sum_i d_i = 2$;

3. build a code tree for these symbols such that the leaf i has depth $1 + \log \frac{1}{d_i}$; to do it, keep gluing together nodes of equal dyadic weight. (The depth of a leaf is the number of edges on the path from the root, which is the number of bits in the codeword corresponding to the leaf.)

Remarks.

- After step 1, we get that $\sum_i d_i \leq \sum_i 2p_i = 2$.
- Step 2 allows us to build a tree with no gaps.
- For step 3, it easy too see that at any time there exist nodes with equal dyadic weight. Indeed, consider the smallest weight $1/2^k$; suppose there is no other equal weight. Then the sum of the remaining weights is of the form $a/2^{k-1}$ for an integer a. Hence,

$$\sum_i d_i = \frac{a}{2^{k-1}} + \frac{1}{2^k} = \frac{2a+1}{2^k} \neq 2,$$

since $2a + 1$ is an odd number and therefore is not divisible by a power of 2.

Example 8. Let $p = (1/5, 3/5, 1/10, 1/10)$. After rounding the probabilities (step 1), we get $d = (1/4, 1, 1/8, 1/8)$. Since $\sum_i d_i = 1/4 + 1 + 1/8 + 1/8 = 3/2$, we need to add an extra symbol x of weight $1/2$ (step 2). The tree is:

```
                 2 /\
                 0/  \1
                 /    \
              1 /\     b 1
              0/  \1
              /    \
         1/2 /\     x 1/2
           0/  \1
           /    \
      1/4 /\     a 1/4
        0/  \1
        /    \
   1/8 c      d 1/8
```

The code obtained is:

$$\begin{array}{lcl} b & \leftarrow & 1 \\ a & \leftarrow & 001 \\ c & \leftarrow & 0000 \\ d & \leftarrow & 0001 \end{array}$$

Theorem. The dyadic code (built with BuildDyadicCode) has an average length per symbol at most $1 + H(p)$, where $H(p)$ is the entropy of the original distribution.

Proof. Let A be the expected cost of encoding one symbol. Then

$$A = \sum_i p_i \times \text{ path length to the i-th symbol.}$$

Since the weight doubles when you go up into the tree, the path length for the i-th symbol is $\log(2/d_i)$, with d_i being the dyadic weight of that symbol. Hence:

$$\begin{aligned}
A &= \sum_i p_i \log(2/d_i) \\
&\leq \sum_i p_i \log(2/p_i) \\
&= \sum_i p_i (1 + \log(1/p_i)) \\
&= \sum_i p_i + \sum_i p_i \log(1/p_i) \\
&= 1 + H(p)
\end{aligned}$$

The dyadic code may not be optimal. Next, we will build a code (the Huffman code) that does not introduce any extra symbols and prove that it is the optimal prefix-free code. Since the dyadic code is prefix-free, it follows that the Huffman code uses at most $1 + H(p)$ bits per character.

Huffman Codes

We will work with the original probability distribution (no dyadic numbers). The algorithm is very simple:

BuildHuffman(α, p)

1. glue together the two symbols with the smallest probability;
2. repeat;

Example 9. Let us consider the following IID source: $\alpha = (a, b, c, d)$ and $p = (1/5, 3/5, 1/10, 1/10)$. We build the Huffman code for this distribution by interpreting what we have as four separate trees with probabilities 1/5, 3/5, 1/10, and 1/10. We now remove the least two and "glue" them together to get a new tree of total probability 1/5.

```
1/5 /\
  0/  \1
  /    \
 c      d 1/10
1/10
```

So we now have three trees with probabilities $1/5$, $3/5$, and $1/5$. We repeat, taking the least two again and gluing them together to get

```
   2/5 /\
     0/  \1
     /    \
1/5 /\     a 1/5
  0/  \1
  /    \
 c      d 1/10
1/10
```

This tree has weight $1/5 + 1/5 = 2/5$. This leaves us with two trees with probabilities $2/5$ and $3/5$. We finally remove these two and glue them together to get our final Huffman tree.

```
           /\
         0/  \1
         /    \
   2/5 /\     b 3/5
     0/  \1
     /    \
1/5 /\     a 1/5
  0/  \1
  /    \
 c      d 1/10
1/10
```

Which symbol we put left and which symbol we put right doesn't make any difference.

Theorem. The Huffman algorithm gives an optimal prefix-free code for an IID source.

Proof. First, let us prove that it is possible to build an optimal tree starting with the choice made in the first step of the algorithm.

Claim.. There is an optimal code tree in which the two smallest weight nodes are siblings.

Proof. Let a, b be the two smallest weight nodes. Consider a tree T for which a is not the deepest node. Then there is some node x which is the deepest. We can swap a and x in the tree T, thus obtaining the tree T'. We can show that by doing this we actually obtain a better code. Indeed, if we denote by A the expected code length per symbol and by L_i the length of the path to symbol i, then

$$\begin{aligned}
A_T &= \sum_i p_i L_i \\
&= \sum_{i,\alpha_i \neq a,x} p_i L_i + p(a)L(a) + p(x)L(x) \\
&\geq \sum_{i,\alpha_i \neq a,x} p_i L_i + p(a)L(x) + p(x)L(a) \\
&= A_{T'}
\end{aligned}$$

Thus, we can assume that the lightest symbol is the deepest. Now, let's look at the sibling of a – it cannot be a tree, since a is the deepest node. Therefore, it should be a leaf. By the same argument as before, if b, the next lightest symbol, is not the sibling of a, then you could swap the sibling of a and b and get a code at least as good as the original one.

Now, having proved this claim, we will prove the theorem by induction on the number of leaves in the tree. Suppose it's true for k leaves; then we will prove it for $k+1$.

We can replace the subtree made of the two lightest nodes in the original tree T by a single node x of probability $p(x) = p(a) + p(b)$. But, by the induction hypothesis, the tree thus obtained (let's call it T') is optimal. If T' is optimal, then T is optimal, since the expected code word length for T is $A_T = A_{T'} + p(x)$ ande since $p(x)$ is constant, independent on which tree you choose.

We know that Huffman codes are optimal and so they are better than dyadic codes. Since dyadic codes have average code word length within 1 of the entropy, Huffman codes are also within 1 of the entropy. But we can make things better.

Example 10. Consider a source with $p = (1/3, 1/3, 1/3)$. Its entropy is $H(p) = 3(1/3)\log 3 = \log 3 = 1.57$. The Huffman tree is:

```
          /\
        0/  \1
        /    \
  2/3 /\       1/3
     0/  \1
     /    \
   1/3    1/3
```

The average number of bits per symbol is $(1/3)2 + (1/3)2 + 1/3 = 1.67$. The Huffman tree is within 0.1 of the entropy. But we want to be able to come arbitrarily close.

Notice that getting within 1 of a larger entropy is better (getting within 1 of 100 as opposed to getting within 1 of 1.5). Hence, a good idea might be to try to create a new source with a larger entropy, by grouping together any two symbols and thus obtaining a new alphabet.

Example 11. Suppose you have a source with alphabet $\alpha = (a, b, c)$ and probabilities $p = (1/3, 1/3, 1/3)$. Then the new source uses the alphabet $\alpha' = (aa, ab, ac, ba, bb, bc, ca, cb, cc)$; the distribution of probability is uniform (each new symbol has probability 1/9). The entropy of the new source is:

$H(p') = 9(1/9)\log 9 = 2\log 3 = 2H(p)$.

Now, if we use Huffman encoding with the new source, we get an expected number of bits per symbol of $3(7/9) + 2(2/9) = 29/9 = 2(1.61)$.

By blocking in this way, you can use gluing to get as close to the entropy as you want by using longer blocks. The size of the block you want to use might depend on the application and on the memory available (as you need to store more for larger Huffman trees).