Huffman Encoding

Tom Davis
tomrdavis@earthlink.net
http://www.geometer.org/mathcircles
November 12, 2010

1 An Easy Example: Telephone Numbers

Let's start with a familiar example: telephone numbers. Ideally, to make a phone call, you would like to have to dial as few numbers as possible. Most telephone numbers in the United States are seven digits long, partly because seven digits is about the most you can expect the average human to remember between the time he looks it up and the time he dials.

Of course, even if all possible 7-digit numbers were usable, that's only 10^7, or ten million numbers, and there are far more telephones than that in the United States. So what is done is that the country is divided into regions, and each region is assigned a 3-digit area code. If you are calling a number within your region, you don't need to include the area code, but if you want to call outside your region, you need 10 digits: the area code plus the 7-digit number identifying the phone inside the region.

But there's a problem: how does the phone company computer know that you are dialing a 7-digit or a 10-digit number? After all, you might be dialing 10 digits, but pause for a moment after dialing 7 of them. The answer is that no 7-digit numbers nor 3-digit area codes begin with the digits 0 or 1. If you want to dial outside your area code, you need to dial a 1 first, then the area code, then the 7-digit local phone number. (This may be handled differently in different parts of the country.) Since no area codes or local numbers begin with a 1, then if the first number you dial is 1, the phone company computer knows that the next three digits are the area code and the following 7 are local.

(Of course since we cannot have a 0 or a 1 as the initial digit of a local phone number, there are only $8 \times 10^6 = 8,000,000$ different possible local numbers instead of $10^7 = 10,000,000$ of them.)

It used to be that the local 7-digit numbers had a "prefix" and a 4-digit number (as in the Marvelettes' song, "Beechwood 4-5789"). The first two letters could be remembered as two letters on the phone where the digit 2 was ABC, digit 3 was DEF, and so on. Thus "Beechwood 4-5789" was 234-5789. Since neither the 0 nor 1 had any letters assigned to it, it was impossible to have a 0 or a 1 as the first or second digit of prefix. (I just know the system where I grew up, in Denver, and the "prefixes" were effectively three digits. The first two were letters, like "FR", for "Fremont", but it was *always* "FR-7"; there was never an "FR-6" or "FR-8". Maybe it was different in other places.)

Because of that, it was initially easy to determine whether the first three digits were an area code or not, since all the area codes had a 0 or a 1 as their second digit. So as soon as the telephone company's computer saw you dial "213. . . ", the 1 in the second position made it clear that you were dialing an area code, and it turned out that it was in the Los Angeles area[1]. This restriction has been dropped to make more local numbers, so the leading 1 is now required.

There are even more problems if you want to dial internationally: different countries have different systems (perhaps not 7-digit numbers nor area codes, but something else). The solution is that if you begin by dialing 011, the next information to dial is a country code, and after that, the US phone company, since it knows that the call is international, can interpret the next digits as a country code, and then just send the following numbers to that country's local phone system for it to figure out.

Notice how this system will tend to reduce the total number of digits that are dialed: most of your calls are to friends or associates in the local area, so most of your calls require only 7 digits. Less likely is for the calls to be in another area code, in which case you need to dial 11 digits (the leading 1, then three digits of area code, plus seven digits of local number). In what is usually the rarest case, international calls, you need to dial enough numbers to get out of the US system, and then you have to dial whatever is necessary to connect to the foreign telephone.

In fact, this system can even be simplified in some cases. Suppose you work for a company with fewer than, say, 900 employees. For some jobs, it is likely that most of your calls will be to others within the company, so you can dial three digits and that's enough to connect you to the correct person within the company. Of course, if you want an outside line, you dial, say, a 0, followed by whatever you would do to make the outside call, be it local, in another area code, or international. The reason for the limit of 900 is that none of the internal extensions can begin with 0, since a leading 0 indicates an outside line. If the company has fewer than 9000 employees, then a 4-digit extension will handle all of them, et cetera.

Let's consider the situation where all we're concerned with is calls within the United States. Suppose that 90% of the numbers you call are within your area code and 10% are outside it. That means that 90% of the time you'll have to dial 7 digits and 10% of the time, 11 digits. On average, the number of digits you need to dial is:

$$0.90 \times 7 + 0.10 \times 11 = 7.4$$

If you just required the area code every time, even for local calls, then you wouldn't need the leading 1, so every call would require 10 digits. When you call local numbers more often, you'll save on the number of digits required, and in this example, you save 2.6 digits per call, on average.

[1]With the old rotary-dial phones, it was faster to dial shorter numbers, so area codes for the more populated areas had easier-to-dial area codes. For example, 212 = New York, 312 = Chicago, 313 = Detroit.

2 Transmitting Messages

Now let's go to a much simpler system: you wish to send messages over a communication line, and you want to transmit the messages as fast as possible. Let's assume that you can only send your messages as a series of 1's and 0's (which is exactly the case for almost all digital transmissions these days). A single 0 or 1 is called a "bit" of information.

We will be very flexible at first in what we mean by a "message". Suppose, for example, you have an elevator, and there is a light that goes on if the elevator is there. Every second, the elevator needs to send one of two messages: "I am there" or "I am not there". We can imagine that the elevator sends a 0 if it is not there and a 1 if it is there. Just by looking at this single bit of information, you can tell whether to turn on the light or not. In this case, this is about the best you can do: every message is exactly one bit in length.

But let's suppose there are more than two messages: the two above, plus one more that means the elevator is locked on a floor and another that means it is broken. Now there are four possible messages, so one bit of information isn't sufficient, but we can easily encode the messages with two bits of information:

I am not there	00
I am there	01
Locked	10
Broken	11

To interpret the situation, you need to look at two bits at a time, and every message will be two bits long.

In a situation like this, however, the locked and broken situations are very rare, and if the building has a lot of floors, then the message "I am not there" will be far more common than the message "I am there". Just to have some numbers, let's assume that 90% of the time the message is "I am not there", 8% of the time the message is "I am there", and 1% of the time each of the other two messages are sent. Consider the following encodings for the messages:

I am not there	0
I am there	10
Locked	110
Broken	111

It may seem strange to have different-length encodings for the messages, but notice first that this is a bit like the telephone numbers: the most common message is sent with the least number of bits, and there is never a question about the message.

Can see why there is never any confusion about the message? The answer is in this footnote[2].

[2] If you see a 0 first, then the message is over and the elevator is not there. If you see a 1, you have to wait for the next bit. If that next bit is a 0, you know the message and you're done, but if not, you need to look at the third bit. Now, in any case, the message is over and you can start looking for the next message.

With this encoding mechanism, what is the length of the average message in bits? Well, it is:

$$0.90 \times 1 + 0.08 \times 2 + 0.01 \times 3 + 0.1 \times 3 = 1.12$$

With this encoding, on average, you save 0.88 bits per message.

3 A Real World Example

The example above with the elevator is fairly contrived, but exactly the same idea can be used for messages that are more like the ones you typically send. Let's look at messages that are just typed with normal text in English.

If you look at lots of English text, you will note that different letters have different probabilities of occurring. The letter “e” is by far the most common, and letters like “x”, “j” and “z” are very rare. There are a lot of standard encodings for letters and probably the most common one today is the ASCII encoding which assigns 8 bits to each letter. This allows for $2^8 = 256$ letters, and that is plenty for the 26 upper-case and 26 lower-case letters, all the punctuation, the space character, tabs, control characters brackets, braces, and other miscellaneous characters.

The ASCII encoding is very simple to work with: every chunk of 8 bits represents the next character in a string, but in terms of packing English text, it is very inefficient since the actual characters have wildly different frequencies. In fact, the space character is the most common, followed by “e”, and so on. It isn't hard to examine large chunks of English text to get a rough idea of the relative frequencies of the letters, and once we have those, we can find an encoding of the character set that is far more efficient in terms of the time it would take to transmit typical messages or the space that it would take to store them on a computer.

One question we would like to answer here is that given the set of relative frequencies of the letters, how do we find the most space-efficient encoding for them?

Rather than start with the full ASCII set that contains all 256 characters, as usual, it is best to begin with the simplest possible situations, and work up from there. Let's just look at tiny alphabets and see what we can do.

Note: In certain situations, we can do even more, and we will talk about those later, but for now, we'll assume that a bit sequence must be sent for each letter. We'll start with 2-character alphabets, then 3, and so on. To make the notation easier, suppose there are n characters and we call them $C_1, C_2, \ldots, C_n$, where C_1 is the most common character and they are listed in order of frequency, so C_n is the least common. Let's write p_1 for the probability that C_1 occurs, p_2 for the probability that C_2 occurs, and so on. If there are n characters, then:

$$p_1 + p_2 + \cdots + p_n = 1,$$

with

$$p_1 \geq p_2 \geq \cdots \geq p_n.$$

If we use b_1 bits to represent the first character, b_2 to represent the second and so on, then the average number of bits required to send a single character is:

$$p_1 b_1 + p_2 b_2 + \cdots + p_n b_n.$$

3.1 A 2-Character Alphabet

If there are only two characters, we can assign 0 to one and 1 to the other. The average number of bits sent for each character is $p_1(1) + p_2(1) = p_1 + p_2 = 1$.

3.2 A 3-Character Alphabet

We're going to need at least 2 bits for some of the characters since with just 0 and 1, only two characters can be distinguished. The easiest way would simply be to assign two bits to each, like 00, 10 and 11. The combination 01 would never occur. But this is clearly wasteful: once you see a leading 0, you know the next bit has to be zero in this encoding, so why bother to send it? A better encoding would be 0, 10 and 11. (Or equivalently, 1, 00 and 01 — there's nothing that makes a 0 better than a 1: both take the same amount of space and the same amount of time to transmit[3]).

Since we're trying to minimize the average number of bits sent, we should assign the 0 code to the most common letter, and the 10 and 11 codes to the other two. It doesn't matter how the last two are assigned; they both will require 2 bits.

Can you prove that this is the most efficient encoding? (Hint, work out the average bit count per letter under the different scenarios, keeping in mind that $p_1 \geq p_2 \geq p_3$.) The answer appears in Section 6.1

3.3 A 4-Character Alphabet

Again, the easiest encoding would be to use all the 2-bit patterns to cover the four possibilities: 00, 01, 10 and 11, although in the elevator example, we've already seen that this may be a bad idea.

At this point, things begin to get tricky. To see why, suppose all four characters are equally likely: that each has probability 25% of occurring (in other words, $p_1 = p_2 = p_3 = p_4 = 0.25$). Then the assignment to the four possibilities works best, since the average message size is 2 bits, and if we used the encodings we did for the elevator: 0, 10, 110 and 111, we would do worse. The average number of bits would be:

$$0.25 \times 1 + 0.25 \times 2 + 0.25 \times 3 + 0.25 \times 3 = 2.25,$$

which is, on average, 0.25 bits more per character, on average.

[3]Notice that this may not always be true: using Morse code as an example, it takes about three times as long to send a "dash" as a "dot".

When should you use each kind of encoding? Think about this for a while before reading on.

The answer is that if you add the two lowest probabilities together and obtain a number that is larger than the highest probability, then the best encoding is to use 2 bits for each; otherwise, assign them as 0, 10, 110, 111 for the highest probability to lowest probability character.

3.4 Alphabets With More Than 4 Characters

With 5 or 6 characters in the alphabet, it's still probably possible to work out the optimal binary encodings by brute force, but in the real world where alphabets are generally much larger, this method becomes far too tedious.

As an example, try to find as good an encoding as you can for the following set of characters with the given probabilities of occurrence. In other words, what would be an optimal encoding for the characters "A" through "H" below, and what would be the average number of bits per character in that encoding? Since there are 8 characters, we could clearly just assign to each one of the eight 3-bit encodings: 000, 001, 010, 011, 100, 101, 110, 111, and we would have an average of 3 bits per character. How much better than that can you do?

i	C_i	p_i
1	A	0.28
2	B	0.27
3	C	0.23
4	D	0.09
5	E	0.04
6	F	0.04
7	G	0.03
8	H	0.02

Try to work on this for a while and see how well you can do. The answer appears in Section 6.2

3.5 A Practical Example

Assuming you know how to construct an optimal encoding for any distribution of characters (which we will find later), how could we use this?

Suppose you have huge documents with millions or even hundreds of millions of characters. You would like to compress them so as to take up as little storage space as possible. One approach would be, for each document, to have the computer make a pass through and count the number of instances of each character, including spaces, punctuation, et cetera. Then it can generate, using rules we will discover later, an optimal Huffman encoding for that character distribution. Finally, use the first few hundred characters of the encoded document to list the particular encoding for that document, and this would be followed by the Huffman-encoding of the original text.

Doing this for each document has the advantage that if the documents have wildly-different distributions, each will be compressed optimally. For example, texts in Italian will have a lot more vowels than texts in English and would need a different encoding. If a document consisted of mostly mathematical tables, almost all the characters might be digits, so a Huffman encoding that gave digits priority would allow the document to be compressed far more efficiently, et cetera.

4 Huffman Encoding

It should be clear that whatever the best solution is, it ought to assign shorter codes to the more common characters and longer codes to the rarer characters. This is true since if there is a pair of characters C_1 and C_2 where C_1 is the most common but has a longer encoding than C_2, then you could just swap the encodings and have a more efficient final result.

We can think of the problem as a series of investments of bits. The longest strings of bits should go with the least common characters, so pick the two least-common characters and spend one bit on each (a 0 and a 1) to tell them apart. This will be the final bit in the code for each, with some unknown number of bits preceding them which are the same. But now every time you add one bit to that preceding part, the cost, in terms of number of total bits spent, will be the sum of the costs for the individual characters, since that extra bit has to be added to both codes. This sentence is a little vague, but it provides an insight into how an optimal encoding is derived and to why it is the best.

As an aid to calculation, notice that it is not necessary to use the true probabilities when you are trying to optimize an encoding; any multiple of all the probabilities will yield the same minimum. In other words, suppose there are five characters and you've done a count of the number of times each character occurs and you come up with the following counts:

A	123
B	82
C	76
D	40
E	11

To find the probabilities you need to add all the character counts: $123 + 82 + 76 + 40 + 11 = 332$ and then divide each of the numbers above by 332, so for example, the probability of obtaining an "A" is $123/332 = .3704\ldots$ — a bit more than 37% of the time. The same can be done for each of the numbers above, but notice that they all will be equal to 332 times the actual probability, so if you just use 123, 82, and so on, all your sums will be exactly the 332 times as large as what you'd get using the true probabilities.

So let's try to figure out logically an optimal encoding for the character set above.

There are 5 characters, so 2 bits will not be sufficient; at least one of them will require 3 bits of data (and perhaps more). We can't allow four of them to have 2 bits and the other 3 bits, since the 3-bit encoding will have to start the same way as one of the 2-bit encodings and there will be no way to tell whether the 2-bit character ended, or whether we need to wait one more bit for the 3-bit character. So at least two of the characters will require at least 3 bits. As we stated above, the letters "D" and "E" should be the ones chosen to have the longest string of bits in their codes, so we can arbitrarily say that D has a code of $x \ldots x0$ and E a code of $x \ldots x1$ where the earlier bits indicated by the x's (we don't know how many yet) are the same.

But each time we add a bit to replace the x's above for the D and E characters, we add one bit to the representation of 51 characters (both the 40 D's and the 11 E's). In a sense, we can treat the combination of D and E as a single character with weight 51 together with the other three letters, and we'd have a chart that looks something like this:

A	123
B	82
C	76
(DE)	$40 + 11 = 51$

Now the least common "letters" are C and (DE), where (DE) is the combination of D and E. We should use a bit to distinguish between them, and we obtain the following chart:

(C(DE))	$76 + 51 = 127$
A	123
B	82

The C combined with (DE) now is the most-likely "letter", so it moves to the top of the chart. The least likely of those left are A and B, which, when combined, yield:

(AB)	$123 + 82 = 205$
(C(DE))	$76 + 51 = 127$

Finally, one more bit needs to be added to distinguish between the two combinations above, and it could be indicated as follows, continuing the pattern above:

((AB)(C(DE)))	$205 + 127 = 332$

Each pair of parentheses contains two "subcodes" each of which is either a letter of a grouped pair of similar items. We can unwrap the parentheses as follows. Assign a leading 0 to one of the subcodes and a leading 1 to the other. Then continue recursively.

Thus the first bit tells us whether to go with (AB) or (C(DE)); arbitrarily use 0 for (AB) and 1 for (C(DE)). If we see a 0, we only need one bit to distinguish between A and B, so 00 means A and 01 means B. If we see a leading 1. then the next bit tells us whether we have a C or a (DE), et cetera, so the codes for the last three should be 10 for C, 110 for D and 111 for E:

Character	Frequency	Code
A	123	00
B	82	01
C	76	10
D	40	110
E	11	111

This optimal encoding requires only 3 bits for the least-common character, but there are situations with only 5 characters that 4 bits would be required for an optimal encoding. Here's an example:

Character	Frequency	Code
A	16	0
B	8	10
C	4	110
D	2	1110
E	1	1111

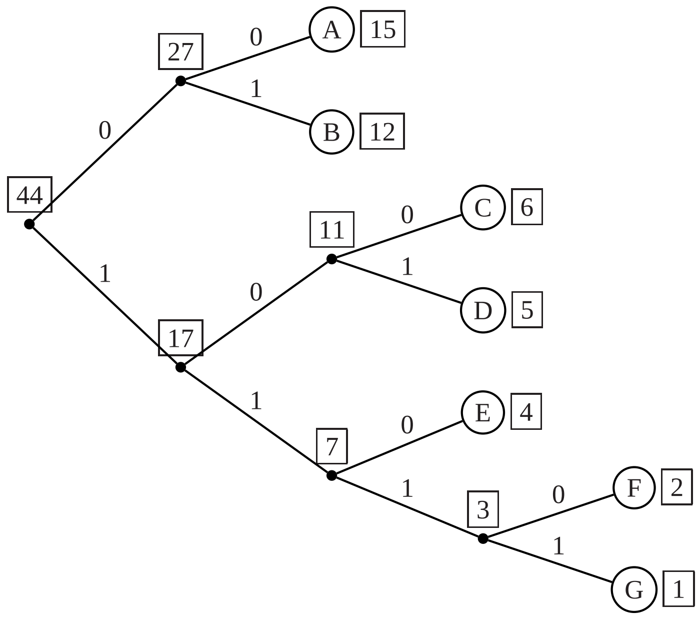
Follow the logic we used for the previous example to see that the structure for the encoding would be (A(B(C(DE)))).

5 Tree Formulation

Another way to visualize the Huffman encoding is as a tree. If we follow the same procedure as above for the following distribution of letters:

A	15
B	12
C	6
D	5
E	4
F	2
G	1

we obtain the encoding structure ((AB)((CD)(E(FG)))). This can be displayed as the binary tree below, where the letters are the final nodes, the numbers in boxes are the weights, both of the final and internal nodes and the 0's and 1's above the lines indicate the encoding. To find the code for any particular letter, just find the unique path to the letter from the root (numbered 44) and read off the numbers on the connecting edges.



For example, to get to D from the root we follow three edges labelled 1, 0 and 1, so the Huffman encoding for D in this example is 101. Here is the complete encoding:

A	15	00
B	12	01
C	6	100
D	5	101
E	4	110
F	2	1110
G	1	1111

6 Solutions

6.1 Optimal 3-character encoding

We want to show that if $p_1 \geq p_2 \geq p_3$, then assigning the code 0 to C_1 and the codes 10 and 11 to C_2 and C_3 (in either order) generates the most efficient encoding; namely, that:

$$p_1(1) + p_2(2) + p_3(2)$$

is the smallest possible value. In other words, if we assign 2 bits to C_1, the results will always be worse. Here are the average bit lengths corresponding to the other ways we could make that assignment:

$$p_2(1) + p_1(2) + p_3(2)$$
$$p_3(1) + p_1(2) + p_2(2)$$

We just have to show that:

$$p_1(1) + p_2(2) + p_3(2) \geq p_2(1) + p_1(2) + p_3(2)$$

and

$$p_1(1) + p_2(2) + p_3(2) \geq p_3(1) + p_1(2) + p_2(2).$$

Both are similar; let's just do the first. The following inequalities are all equivalent; we just add or subtract the same thing from both sides to move from one to the next:

$$\begin{aligned}
p_1(1) + p_2(2) + p_3(2) &\geq p_2(1) + p_1(2) + p_3(2) \\
p_1(1) + p_2(2) &\geq p_2(1) + p_1(2) \\
p_2(2-1) &\geq p_1(2-1) \\
p_2 &\geq p_1
\end{aligned}$$

Since the last line is true and they're all equivalent, we've shown the first inequality to be true. The second is similar.

6.2 Optimal 8-character encoding

Here is one optimal encoding:

i	C_i	p_i	encoding	b_i
1	A	0.28	00	2
2	B	0.27	01	2
3	C	0.23	10	2
4	D	0.09	110	3
5	E	0.04	11100	5
6	F	0.04	11101	5
7	G	0.03	11110	5
8	H	0.02	11111	5

and the average number of bits per character is:

$$p_1 \cdot b_1 + \cdots + p_8 \cdot b_8$$

$$= 0.28 \cdot 2 + 0.27 \cdot 2 + \cdots + 0.02 \cdot 5 = 2.48.$$

Of course any of the encodings with the same number of bits could be swapped around. You just need to find one where A, B and C are encoded in 2 bits, D in 3 bits, and E, F, G and H in 5 bits.