Bricolage: Data Compression

Bricolage: Data Compression

You are probably familiar with Unix `compress`, `gzip`, or `bzip2` utilities, or the DOS `pkzip` utility. These programs all make files smaller; we say that such files are *compressed*. Compressed files take less disk space and less network bandwidth. The downside of compressed files is that it they are full of unreadable gibberish; you usually have to run another program to *uncompress* them before you can use them again. In this article we'll see how file compression works, and I'll show a simple module that includes functions for compressing and uncompressing files.

Morse Code

The idea behind data compression is very simple. In a typical file, say a text file, every character takes up the same amount of space: 8 bits. The letter `e` is represented by the 8 bits 01100101; the letter Z is represented by the 8 bits 010101010. But in a text file, `e` occurs much more frequently than `z`---maybe about 75 times as frequently. If you could give the common symbols short codes and the uncommon symbols long codes, you'd have a net gain.

This isn't a new idea. It was exploited by Samuel Morse in the Morse Code, a very early digital data transmission protocol. Morse Code was designed to send text files over telegraph wires. A telegraph is very simple; it has a switch at one end, and when you close the switch, an electric current travels through a wire to the other end, where there is a relay that makes a click. By tapping the switch at one end, you make the relay at the other end click. Letters and digits are encoded as sequences of short and long clicks. A short click is called a *dot*, and a long click is called a *dash*.

The two most common letters in English text are `E` and `T`; in Morse code these are represented by a single dot and a single dash, respectively. The codes for `I`, `A`, `N`, and `M`, all common letters, are `**`, `*-`, `-*`, and `--`. In contrast, the codes for the uncommon letters `Q` and `Z` are `--*-` and `--**`.

In computer file compression, we do a similar thing. We analyze the contents of the data, and figure out which symbols are frequent and which are infrequent. Then we assign short codes to the frequent symbols and long codes to the infrequent symbols. We write out the coded version of the file, and that usually makes it smaller.

Ambiguous codes

There's a problem with Morse Code: You need a third symbol, typically a long pause, to separate the dots and dashes that make up one letter from the dats and dashes that make up the next. Otherwise, if you get *-, you don't know whether it's the single letter A or the two letters ET---or it might be the first bit of the letter R or L. In a long message, all the dots and dashes run together and you get a big mess that can't be turned back into text. In Morse code, it can be hard to tell `Eugenia' from `Sofia': Without the interletter pauses, they're both:

```
***---**-****-
```

Those interletter spaces take up a lot of transmission time, and it would be nice if you didn't need them. It turns out that if you arrange the code properly, you don't. The ambiguity problem with Morse Code occurs because some codes are *prefixes* of others: There are some letters where the code for the first part of one letter is just the same as the code for the other letter, but with something extra tacked on. When you see the shorter code, you don't know if it's complete or if it should be combined with the following symbols. It turns out that if no code is a prefix of any other, then the code is unambiguous.

Suppose for simplicity that we only needed to send the letters A, C, E, and S over the telegraph. Instead of Morse code, we could use the following code table:

```
A        -
C        **
E        *-*
S        *--
```

Suppose we receive the message -*****-**--*-*-**--. What was the message? Well, the first symbol is -, so the first letter in the message must be A, because that's the only letter with a code that starts with a -. Then the next two symbols are **, so the second letter must be a C, because all the other codes that start with * have a - after the * instead of another *. Similar reasoning shows that the third letter is also C. After that, the code is *-*; it must be an E. We continue through the message, reading off one letter at a time, and eventually we get the whole thing this way.

It's so simple that a computer can decode it, if the computer is equipped with a decision tree like this one:
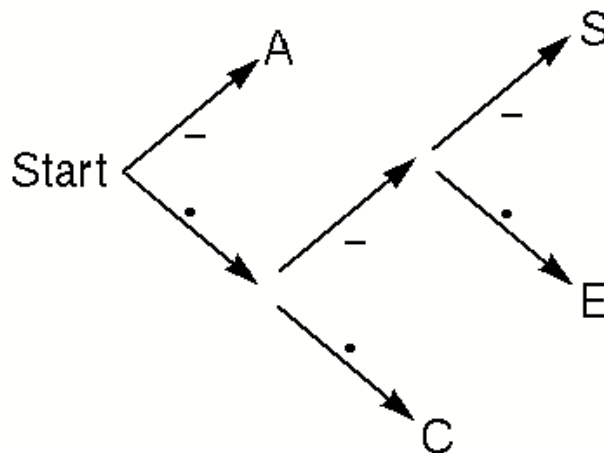

Start at *Start*, and look at the symbols in the message one by one. At each stage, follow the appropriate labelled branch to the next node. If there's a letter at that node, output the letter and go back to the start node. If there's no letter at the node, look at the next symbol in the input and continue down the tree towards the leaves.

Huffman Coding

Obviously, it's important to choose the right code. If Morse had made the * code for z and **-* the code for e, he wouldn't be famous.

Choosing the right code can be tricky. Consider the example of the previous section, where we only had to code messages that contain A, C, E, and S. The code I showed is good when we expect our messages to contain more A's than E's or S's. If S were very common, we clearly could have done better; less clearly, if all four letters were about equally common, then we could still have done better, by assigning each letter a code of the same length:
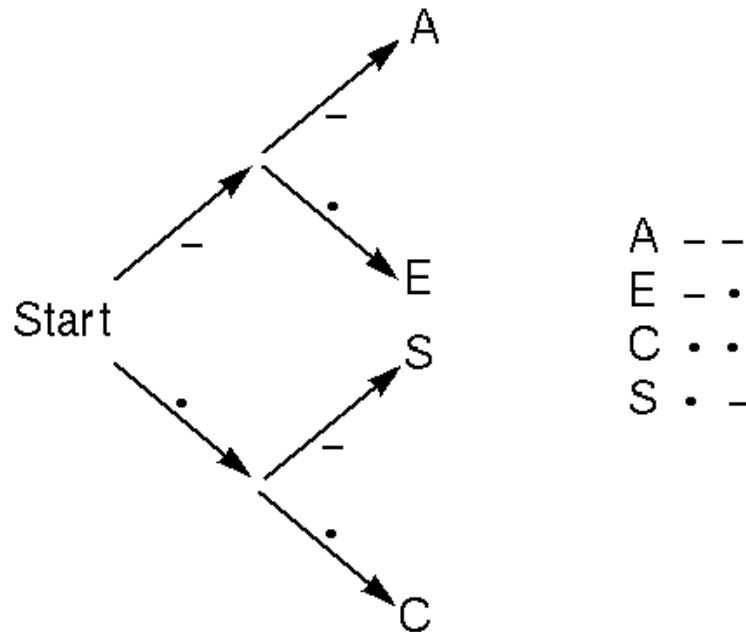
Suppose, for example, our message happened to contain 200 of each of the four letters. Then the first code would use 1800 symbols, and the second code would use only 1600.

In 1952, David Huffman discovered a method for producing the *optimal* unambigous code. For a given set of symbols, if you know the probability with which each symbol appears in the input, you can use Huffman's method to construct an unambiguous code that encodes the typical message with fewer *'s and -'s than any other code.

The method is very simple and ingenious. For concreteness, let's suppose that the (rather silly) message is

        THE_THIRSTIEST_SISTERS_TEETH_RESIST_THIS_STRESS

(I used _ instead of space so that it'll be easier to see.)

Start with the table of relative probabilities; you can get this by counting the number of occurrences of every symbol in the message. This is called *histogramming*. (A *histogram* is a bar chart; *histos* is Greek for a beam or a mast.) Here's the histogram for the symbols in our sample message:

```
_ 6 ████████
I 5 ██████
H 4 ████
R 4 ████
```

Now take the two least common entries in the table, that's H and R. They'll get the longest codes, because they're least common. We'll simplify this by pretending that H and R are the same, and lumping them together into one category, which we'll call HR. Then we'll assign codes to all the other letters and to HR. When we're done, we still have to distinguish between H and R. Now, HR has some code. We don't know what it is yet, so let's symbolize it with <HR>. We don't really need to use <HR> in our message, because the is no such thing as the letter HR, so we'll split it in two, and let the code for H be <HR>* and the code for R be <HR>-. As a result of this, the codes for H and R will be longer than the codes for the other letters, but if that has to appen, it's better for it to happen for H and R, because they are the least common letters in the message.

So we will lump H and R together and pretend temporarily that they are only one letter. Our table then looks like this:

```
S        11          R = <HR>-
T        10          H = <HR>*
HR        8
E         7
_         6
I         5
```

Now we repeat the process. The two least common symbols are I and _. We'll lump them together into a new `symbol' called I_, we'll assign finish assigning the codes to S, T, HR, E, and I_. When we're done, I will get the code <I_>* and _ will get the code <I_>-.

```
S        11          R = <HR>-
I_       11          H = <HR>*
T        10          _ = <I_>-
HR        8          I = <I_>*
E         7
```

Then we lump together HR and E:

```
HRE      15          R   = <HR>-
S        11          H   = <HR>*
I_       11          _   = <I_>-
T        10          I   = <I_>*
                     HR  = <HRE>-
                     E   = <HRE>*
```

Then we lump together T and I_:

```
I_T      21          R   = <HR>-
HRE      15          H   = <HR>*
S        11          _   = <I_>-
                     I   = <I_>*
                     HR  = <HRE>-
                     E   = <HRE>*
                     I_  = <I_T>-
                     T   = <I_T>*
```

Then we lump together S and HRE:

```
SHRE     25          R   = <HR>-
I_T      21          H   = <HR>*
                     I   = <I_>-
```

```
   _   = <I_>*
  HR   = <HRE>-
  E    = <HRE>*
  I_   = <I_T>-
  T    = <I_T>*
  S    = <SHRE>-
  HRE  = <SHRE>*
```

Now we only have two `symbols' left. There's only one way to assign a code to two symbols; one of them gets `*` and the other gets `-`. It doesn't matter which gets which, so let's say that SHRE gets `*` and I_T gets `-`.

Now the codes fall out of the table we've built up in the right-hand column:

```
  SHRE = *
         S = *-
         HRE = **
                 HR = **-
                        R = **--
                        H = **-*
                 E  = ***
  I_T = -
         I_ = --
                  _ = ---
                  I = --*
         T = -*
```

We throw away the codes for the fictitious compound symbols, and we're left with the real code:

```
  S = *-
  T = -*
  E = ***
  _ = ---
  I = --*
  H = **-*
  R = **--
```

As promised, the code is unambiguous, because no code is a prefix of any other code. Our original message encodes like this:

```
  -***-****----***-*--***--*--*--****--*---
  *---**--******--*-----*******-***-*---
  **--****---**--*----***-*--**----
  *--***--****-*-
```

For a total of 128 `*`'s and `-`'s, an average of 2.72 symbols per character, and a 9.3% improvement over the 141 symbols we would have had to use if we had given every letter a three-symbol code.

The Code

For this article, I implemented a demonstration module that compresses files. You can retrieve it from the perl Journal web site at http://www.tpj.com/ or from my Perl Paraphernalia web site at http://www.plover.com/~mjd/perl/Huffman/. The program htest.pl compresses an input and saves it to the file /tmp/htest.out; then it opens this file, reads in the compressed data, decompresses it, and prints the result to standard output.

Most of the real work is in the Huffman module that htest uses. Here are the important functions that htest calls:

```
  my $hist = Huffman::histogram(\@symbols);
```

DOCKET ALARM

Explore Litigation Insights

Docket Alarm provides insights to develop a more informed litigation strategy and the peace of mind of knowing you're on top of things.

Real-Time Litigation Alerts

Keep your litigation team up-to-date with **real-time alerts** and advanced team management tools built for the enterprise, all while greatly reducing PACER spend.

Our comprehensive service means we can handle Federal, State, and Administrative courts across the country.

Advanced Docket Research

With over 230 million records, Docket Alarm's cloud-native docket research platform finds what other services can't. Coverage includes Federal, State, plus PTAB, TTAB, ITC and NLRB decisions, all in one place.

Identify arguments that have been successful in the past with full text, pinpoint searching. Link to case law cited within any court document via Fastcase.

Analytics At Your Fingertips

Learn what happened the last time a particular judge, opposing counsel or company faced cases similar to yours.

Advanced out-of-the-box PTAB and TTAB analytics are always at your fingertips.

API

Docket Alarm offers a powerful API (application programming interface) to developers that want to integrate case filings into their apps.

LAW FIRMS
Build custom dashboards for your attorneys and clients with live data direct from the court.

Automate many repetitive legal tasks like conflict checks, document management, and marketing.

FINANCIAL INSTITUTIONS
Litigation and bankruptcy checks for companies and debtors.

E-DISCOVERY AND LEGAL VENDORS
Sync your system to PACER to automate legal marketing.

fastcase
Smarter legal research.