



Algorithm dictionary compression

We've all used ZIP to compress files. But how does its compression work?

In this issue...

▶ WHAT'S COVERED

Sometimes it's convenient to ignore how a particular program does its job; so long as it works, we don't give it a moment's thought. But, equally, sometimes it's interesting to see the algorithm a program uses and, for sheer fascination, dictionary compression is one of those algorithms. Dictionary compression is used by pretty much all the compression formats out there, ZIP, 7ZIP, RAR, and so on, as well as several lossless image formats like GIF, certain TIFFs, etc.

Nugget

There is another wrinkle we can take advantage of when searching for matches in previously seen data: we can include data we haven't yet officially seen.

Look at the example of 'rat-tat-tat'. When we first saw a match on the second a, we matched 4 characters 4 bytes back, and encoded the match as <4,4>.

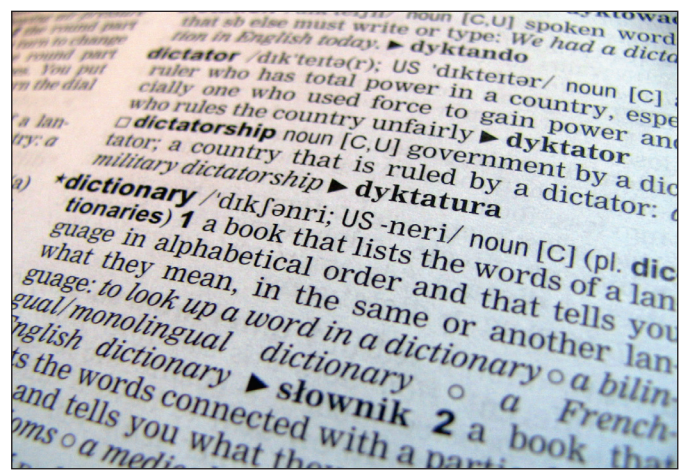
We are reading more characters from the message. Our sliding window is moving along as we match the characters. But why stop matching here? We can just continue, as in figure 3, and match up the rest, and encode the distance length pair as <4,10>. That is, go back four characters and then match up 10 characters. ■

Back in 1948, a seminal paper on information theory – in those days called communication theory, since it was geared towards the efficient transmission of data – was published in two parts in July and October. The paper was A Mathematical Theory of Communication by Claude Shannon and it introduced the concept of information entropy. Without going into too much detail, information entropy is the inverse of the amount of redundancy present in a message.

For example, a message consisting of a series of totally random bytes has high entropy and very low redundancy. Given a portion of the message, it's extremely hard – that is, impossible – to work out the rest. But, as a counter-example, given a message that just consists of the byte with say the value 2A hex repeated for the length of the message has very low entropy (zero, in fact) and very high redundancy. Given a part of the message, we can guess with high probability what the rest of the message is.

A new PC era

Together with computers, this new theory ushered in a new discipline in computer science: data compression. This discipline used information theory to reduce



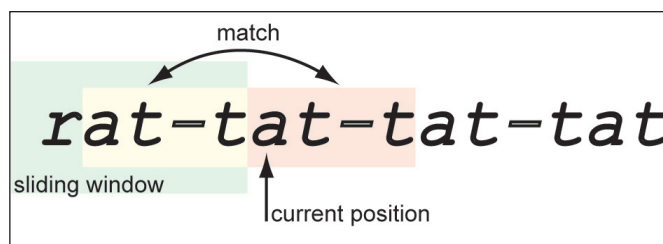
▲ LZ77 compresses the dictionary by matching strings of characters.

the amount of redundancy in a message (or equivalently to increase the information entropy in it). It wasn't long before Shannon himself and R.M. Fano devised, independently of each other, a way to encode data using a bit-encoding such that bytes in a message that occurred more frequently were encoded in a small number of bits, whereas others that were rarer were encoded using more bits. Overall, instead of using a fixed number of bits per byte, as we're used to, the average number of bits transmitted per byte in the message would be much reduced. (By the way, communication theory was all about transmitting messages efficiently, but if you

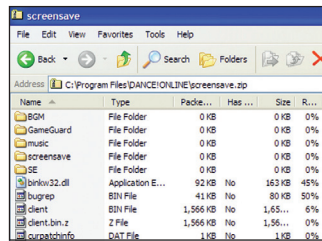
prefer you can certainly think about the message being transmitted as a file that must be stored in the smallest number of bits on a disk.)

The Shannon-Fano minimum redundancy compression algorithm didn't last long at the top however. Pretty soon, in 1952, David A. Huffman published a paper called A Method for the Construction of Minimum-Redundancy Codes that described an algorithm for generating the smallest codes for a given message and one that was mathematically provable to do so. It seemed that no sooner had data compression entered the scene that it had been completely solved. Huffman's compression ruled the roost.

Over the succeeding decades, many variants were described and researched in the literature. Of great importance were adaptive algorithms devised to counter a big problem of Huffman encoding, that of the transmission of the tree used to encode the message. Huffman encoding analyzes the data in the message and constructs a binary tree to



▲ Matching the 'at-t' string.



▲ The algorithm has affected how many of today's computer applications work.

► represent the frequencies of the bytes seen in the message. This tree is known as a Huffman tree, and is different for each message. Hence, for the receiver of the message to have a chance of decoding it, the tree must be transmitted with the message. It can be hard to devise a compression method to compress this tree since it generally has high entropy. Adaptive algorithms get around this by having the tree constructed at the same time as encoding the bytes in the message instead of before. The receiver constructs the same adaptive tree as it decompresses the message.

But, nevertheless, minimum redundancy encoding in whatever form it took continued as the compression method of choice. Until 1977.

Different algorithms

25 years after Huffman's original paper, two Israeli researchers, Jacob Ziv and Abraham Lempel, came up with a drastically new

Nugget

As with any complex computer algorithm devised from the seventies onwards, the LZ77 algorithm became embroiled in controversy when variants started to become patented.

The most famous of these was the LZW compression algorithm, invented by Terry Welch in 1983 from the LZ78 algorithm by Lempel and Ziv. Welch worked for Sperry Corporation (which later became Unisys), which patented the algorithm.

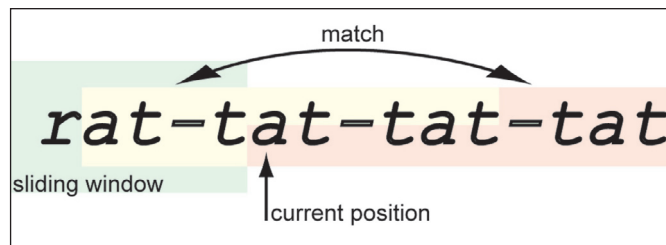
LZW became incorporated into the GIF image format, whose inventor, CompuServe, unfortunately didn't know about the patent. Unisys later decided to enforce the LZW patent on CompuServe and other software vendors and online sites, resulting in much condemnation of Unisys and leading to many people abandoning the GIF format. The patent expired in 2003.

way of looking at the problem and thereby opened up research in a completely new direction. Their idea was that, instead of trying to encode a message byte by byte, we should instead be encoding sequences or strings of bytes.

To see how this works, suppose you have a list of all the words in the English language, together with their plurals and conjugations. There'll be probably close to half a million entries. Number this list starting from one. Now take a block of English text. I'm sure you can see that you can encode this text by replacing each of the words by its number in the master list. Since each index value would take 2.5 bytes (that's 20 bits, which is enough to encode an integer up to one million), no matter how long the words were, you would be compressing the text overall. That's the essence of dictionary compression.

That example worked well for us because we had a previously built dictionary and we can assume that the decompressor also has the same dictionary. But what if you are compressing data for which you do not have a nice pre-built dictionary?

Ziv and Lempel's algorithm (known as LZ77) gets around this issue by building a dictionary as you go along. As you read through the original message, you attempt to match the string of characters (usually three characters or more) at your current position with something you've seen before. If you do, you output a distance-length pair. This is two numbers, one for the distance behind your current position where you've previously seen the current string, and the other for the length of characters that match. If there's no match, you just output the current character – it's known as a literal – and move on. To make the algorithm efficient, you limit how far back you go to search for matches. It's as if you had a small window on your previous data,



▲ Increasing the length of the match.

Spotlight on... A worked example of LZ77

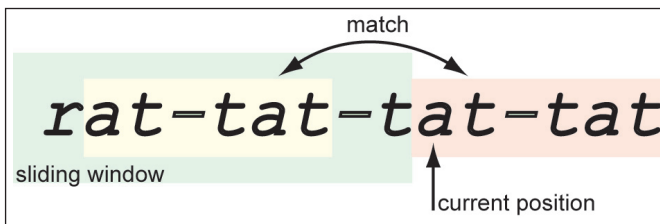
To see how LZ77 works, let's use it on a simple message to use as an example: rat-tat-tat-tat.

We start at the initial r. We haven't seen anything before, so we just output the r to the compressed message. Do the same for the next few characters.

In figure 1, on the second a, we notice that we can match 'at-t' from the previously seen message, so we output a distance-length pair of '<4,4>',

encoded in some way. At this point, our output message looks like this: rat-t<4,4>.

We advance past the matched string in our message to the third a. We see another possibility for a match in figure 2, and this time it's even longer. We've already seen 'at-tat' before, eight characters ago, and so we can encode another distance-length pair to give this end results: rat-t<4,4><8,6>.



▲ Matching the rest of the message.

and this window slides along behind you as you work your way through the message.

This sliding window technique, as it's known, takes advantage of a general quirk of real-life data. If you analyze a lot of files (documents, images, databases, etc), you'll notice that the data exhibits what's known as locality of reference. What this means is that any repetition in the data tends to occur close together. Look at this article for an example: the beginning has lots of references to Shannon and Huffman close together, whereas this later part has lots of references to Lempel and Ziv. By its very nature, the sliding window technique takes advantage of this clumping of repetition in the data.

In the early days of implementing this algorithm, a popular choice to encode the distance-length pair was to make the maximum distance value

4095 bytes and the maximum matched string length 15 bytes. That way the distance could be encoded in 12 bits (212 is 4096) and the length in 4 bits (24 is 16). The nice thing about this method was that everything naturally aligned on byte boundaries: there was no need to read bits separately. An example of this was Microsoft's help files in the early days of Windows.

Now, in our worked example, we haven't really used a sliding window and we certainly haven't used a dictionary, instead relying on our spotting previously seen data. The window we look back through is something like the previous 4KB or 32KB of the message, or sometime more. Also, rather than using simple search techniques to find strings, we add all three-byte strings we've seen to a hash table (or a dictionary, in computer science terms), and then search the hash table to find the longest match. The problem then becomes: how do we remove the entries in the hash table that are too far away and outside the current sliding window? That is left as an exercise for the reader.

Julian Bucknall has worked for some major companies and is now CTO for Developer Express. feedback@pcplus.co.uk