Knocked Down, Made Small

In which we look at some elementary compression techniques

by Julian Bucknall

Just recently, the company I work for, TurboPower Software, decided that it should update its compression library, Abbrevia, for a version 2. Various enhancements were considered for inclusion in a planning meeting one day, one of them being optimizing the zip/unzip process, the actual engine of the library (there were other optimizations discussed but they needn't concern us here). The original developer of Abbrevia had left the company, so it really was time someone new took over the basic engine code. People looked around, weighing up each person in turn, looking for a heavy duty algorithms and data structures developer to review the code and suggest improvements. Finally, after five nanoseconds, all eyes were on me. Done deal, *fait accompli*.

I knew nothing concrete about the zip compression algorithm apart from some abstract understanding of how it worked. I soon had a lot more knowledge. Hence this initial article, where I'd like to share some of my findings with you.

What we'll do this month is discuss a couple of simple compression algorithms, including Huffman encoding. In a future article we'll look at sliding window and dictionary based algorithms. In all of them we'll be using some data structures we've been developing here at *Algorithms Alfresco*.

Professor Night

So, compression then. For a start, it has always fascinated me that data is actually *compressible*. I mean data is data, right? If we've got a bunch of data, how can we make it occupy less space? Well, as a first step, we need to recognize that we can separate the data into two parts: the actual 'information' conveyed by the data, and the method used to encode that data in some physical form. We can't do much about the information: by our definition it has no physical attributes, it's just some nebulous 'essence': let's get metaphysical here! But the method used to encode that information is something we can attack.

Of course, it goes without saying that we should be able to decompress the compressed data. It would be a pretty poor compression method which didn't have a decompression algorithm! For the moment we shall just be considering lossless compression; in other words, we get exactly the same data, once we've compressed it and decompressed it, as we had in the first place. Nothing gets lost through the process. This compares with lossy compression, such as the JPG image format, where the loss of some information isn't that noticeable because we are using our imperfect senses to view the image.

Let's take a simple example. Suppose our data consists of a list of the heights in metres of the people who are reading this column, collated for some statistical experiment. We have several thousand data points, for the sake of argument 10,000. How can we express this information in a file? One way is to create a text file containing the values, one measurement in metres per line:

```
1.83
1.77
1.78
...
```

If we assume that we'll always have 4 characters per line, the entire file will be 60,000 bytes long (remember each line is terminated by a carriage return and line feed character pair). Think about what this text file looks like. For a start, the second character of every line is a decimal point. This is totally redundant. We could instead impose the assumption that the measurements are quoted in centimeters and, at a stroke, we have a different encoding which is smaller in size than the original. We still have exactly the same *information*, but we exploited some redundancy in the original method to encode it so that we can store it in less space. Our file is now 50,000 bytes in size.

The next improvement? Well, we don't need the superfluous carriage return and line feed characters: just assume each measurement takes up three characters. We haven't altered the information, but the file is now 30,000 bytes long.

I'm sure that some of you are jumping up and down now saying 'encode the values in binary as bytes.' And, assuming that I don't have any readers over 2.55 metres tall, that's the final encoding we'll consider for this simple example. Same information, however the file is now 10,000 bytes in size. Without doubt, we could all easily write a simple program to compress the original data to 17% of its original size and decompress it into the same text file without even breaking into a sweat.

I'm sure you see the idea here. To compress data we exploit some redundancy in the original encoding and remove it to produce the compressed data.

Shake Your Head

Another example. Think about a text file containing some English text. Like me, you'd realize pretty quickly that it really only contains the twenty six letters of the alphabet, lower and upper case, plus the space character, plus a handful of punctuation characters, plus the carriage return/line feed character pair. At a guess, the text file contains 64 different characters, tops. Since 2^6 is 64, we have 6 bits per character. Yet, on the PC at least, it's encoded in ASCII, which uses 8 bits per character. You can see without any rigorous argument that we're 'wasting' 2 bits for every character. If we chose a different encoding scheme to ASCII we could get one and a third characters per byte, or four characters for every three bytes, instead of ASCII's three. We could be achieving a compression ratio of 75%.

A quick note on compression ratios is in order here. The ratios I shall be quoting are going to be expressed as percentages and will be calculated as compressed size / original size * 100%. Thus with the alternative character encoding above, for every character the compressed size is 6 bits, the original size is 8 bit and thus the compression ratio is 75%.

The compression method can be expressed as: read the next character from the input stream, encode it as six bits and output the six bits to the compressed stream (we'll see how to do this in a minute). The decompression method is: read six bits from the compressed data stream (again we'll see how to do this in a minute), convert it back to ASCII and output the resulting character to the uncompressed stream.

Of course, this simplistic encoding has a big problem. It assumes that the text file just contains the 64 characters we originally selected. For example, if the text file was written in French (*le texte est écrit en français*), we'd suddenly have characters with accents and cedillas embedded in the text. How do we deal with these? The usual method is to introduce the notion of an escape character. We select one of the 64 compressed encodings to represent a special character, the escape. When we come across a character in the input stream that's not one of the 63 we can now compress, we output the 6 bit escape followed by the 8 bit problematic character. The presence of this escape character in the compressed stream warns the person (or program) interpreting the data that something special is going on: instead of interpreting the next 6 bits as a character, it is to read a full 8 bits and simply assume that it is an ASCII character. This compression will make our compression ratio worse (these uncompressible characters would occupy 14 bits instead of their original eight, being the 6 bit escape character followed by the original 8 bit ASCII character, but one hopes they'll be rare enough not to affect the overall compression too much).

In fact, this brings up an important point. For some compression methods there will be some bytes or byte sequences that will 'compress' to a series of bits that is larger than the original series of bits for those bytes. If you think about it, this is obvious, otherwise we could go on compressing data time after time after time. All the way down to one byte!

It will be instructive to code this compression algorithm in Delphi. In doing so, we can investigate how to read and write individual bits from a stream. We won't get totally overwhelmed because the compression algorithm is relatively simple.

Walk The Dinosaur

Let's call the compression method SixBitPack. The compression algorithm is shown in pseudocode in Listing 1. The decompression algorithm is shown in pseudocode in Listing 2. If you read them, you'll see I've got a funny bit in the pseudocode. To signal the end of the compressed data we output a particular bit sequence to the compressed data stream (it's a 6 bit escape character, followed by an end-of-data character). Why? Well, consider compressing three letter As. We can output these as three lots of six bits, or 18 bits in all. Since our output stream is sized in bytes (a byte is the smallest physical representation of data in memory or on disk), we'd end up with three bytes.

On decompressing, we'd interpret these three bytes as 24 bits, or four lots of six bits. We'd decompress to four characters if we're not very careful. And that's bad news, compressing three characters and, on decompression, getting four back! The simplest way around this is to encode an end-of-data marker at the end of the compressed stream. Unless we remove another six bit encoding from our remaining 63 to serve this purpose, we need to use the escape character. But which ASCII

➤ *Listing 1: SixBitPack compression.*

```
while there are more chars in the input stream
  get next char from input stream
  if char can be compressed then
    convert char to 6 bit encoding
    output this encoding to output stream
  else
    output 6 bit escape encoding to output stream
    output char to output stream
output escape encoding to output stream
output end-of-data char to output stream
```

➤ *Listing 2: SixBitPack decompression.*

```
set Finished to false
while not Finished do
  get next 6 bits from input stream
  if this is the escape char
    read 8 bits from the input stream
    if this is the end-of-data char then
      set Finished to true
    else
      output char to output stream
  else
    convert this to an ASCII char
    output char to output stream
```

character should we use as an end-of-data marker? Well, how about a letter of the alphabet? If we encounter a letter of the alphabet in our original text stream we'll encode it as six bits. We won't ever encode it as an escape character followed by the ASCII letter. So we can use this peculiarity to signal the end of the data, we'll in fact use `Z` as the end-of-data marker.

Righty-ho then. With compression we're left with the problem of writing 6 or 8 bits to the compressed stream. With decompression we're left with the problem of reading 6 or 8 bits from the compressed stream. How is this done? Welcome to the world of bit-twiddling.

Let's consider writing a single bit. What we do is to use bit shifting operations and bit-masking methods to collect 8 bits into a byte and then write the byte to the output stream. Listing 3 shows a simple `WriteBit` routine. The bit we want to write is passed in as a `Boolean` value of `false` or `true` (in bit terms, `0` or `1`). We shift the value in the collector byte left by one bit to make room for the new value. We then `OR` in the new bit. If we've now collected 8 bits we've filled our collector byte and so we can write the whole byte to the output stream and reset our bit count.

In general, however, we'll be writing several bits in one go. Rather than continually call this `WriteBit` routine in a loop we can optimize it a little by writing a `WriteNBits` routine. Essentially, we just write the code for `WriteBit` in a loop. Reading a single bit and reading several bits works in the same way.

We can now write the SixBitPack compression and decompression routines. We'll make them generic routines that accept an input stream and an output stream as parameters, so we can compress memory, files or whatever, just so long as there is a stream representation. The disk has the details. I compressed *Love's Labour's Lost* with SitBitPack and achieved a compression ratio of 76%. Pretty close to the optimal 75% for this particular compression method.

```
procedure WriteBit(aBitValue : boolean; aStream : TStream;
  var aCollByte : byte; var aBitCount : integer);
begin
  aCollByte := aCollByte shl 1;  {make room for the new bit}
  if aBitValue then
    aCollByte := aCollByte or 1; {add the new bit}
  inc(aBitCount)                 {increment the count of bits};
  if (aBitCount = 8) then begin  {if we've filled a byte write it out}
    aStream.Write(aCollByte, sizeof(byte));
    aBitCount := 0;
  end;
end;
```

➤ *Listing 3: Writing a single bit.*

Carry Me Back To Old Morocco

The whole compression field started out with a seminal paper in 1952 by David Huffman (*A Method for the Construction of Minimum-Redundancy Codes*) in which he outlined the Huffman encoding or compression algorithm. Let's look at this file compression method.

In any file there will be a unique distribution of individual characters, with some characters represented more than others. For example in English text, the letter E is the most common character, followed by T, and so on. What Huffman realized was that if he represented the most common characters with a short bit string and uncommon ones with a long bit string, overall the encoding would save space, and he invented a method for creating the minimal codes required to do so.

Let's take an example. *La Habanera* is a track by Yello on their *One Second* CD. The words have the following distribution of letters (ignoring case sensitivity):

a	4
b	1
e	1
h	1
l	1
n	1
r	1
space	1

As if by magic, let me allocate the following bit strings to each letter.

a	0
b	1000
e	1001
h	1010
l	1011
n	1100
r	1101
space	111

We can then encode *La Habanera* as the following bit string, comprising 31 bits in all:

```
10110111
10100100
00110010
0111010
```

This is a compression ratio of about 36%.

Because of my magic (the *Algorithms Alfresco* corollary to Clarke's Law is: 'any sufficiently advanced algorithm is indistinguishable from magic'), decoding this bit string with the encoding table doesn't result in an ambiguous answer. There is no bit encoding for a character such that the first part of it is the encoding for another character. Try it and see. Peel off bits, one by one, and try to match them up against those in the table. 1 doesn't match, neither does 10, nor 101, but 1011 corresponds to 'l'. The next bit, 0, on its own is 'a'. We then have to peel off 3 bits for 111, which is space. And so on.

Cool. But how do we get the table of the characters and their codes? Before answering this let's view the table I miraculously produced earlier as a binary tree. Starting at the root node, we trace out the bits for each code as follows: a 0 bit means move down a link to the left and a 1 bit means a move down a link to the right. We can draw up the binary tree shown in Figure 1. Notice that the characters only appear at the leaves of the tree, and the value at each internal node is the sum of the counts of characters of its children. The count at the root is obviously the total number of characters in the text itself.

Because each character appears at a leaf, the code for one character *cannot* form the start of the code for another character.

Using this tree makes the decoding of our compressed bit string even easier. We start at the root, and then for each bit to peel off the bit string we follow the required link. When we hit a leaf, output the character that's found there and then start over at the root.

Out Come The Freaks

So how do we generate this tree? We first have to read through the entire file and create a table of characters and the number of occurrences of each character. That's the easy part.

Now we have to build a binary tree. The algorithm Huffman invented goes like this. First consider these character/count pairs as a pool of nodes for a tree. Secondly, remove the two nodes with the smallest counts from the pool. Next, join them to a new parent node, set the parent's count to the sum of its two children. If the pool is now empty, we're done and the new parent we created is the root of the final tree, otherwise, add the parent to the pool and continue at the second step.

Let's try the algorithm out on *La Habanera*. We have already calculated the character counts above. Figure 2 has the character/count pairs as preliminary nodes in a 'pool'. Find the two nodes with the smallest counts. We've got an abundance of nodes with counts of 1 (the minimum) so just choose any two, we'll take the *b* node and the *e* node. Join them to a new parent node, and add it to the 'pool'. We'll get the first step in Figure 3. Continue by finding the two smallest nodes again, the *h* and *l* nodes this time, join them and add the new parent node to the pool. Figure 3 shows the progress of the algorithm until we get to the final tree represented by Figure 1. Notice on the way that Figure 1 is not the only Huffman tree we could achieve, every time we have to make a choice between two or more nodes with the same count we could create another tree.

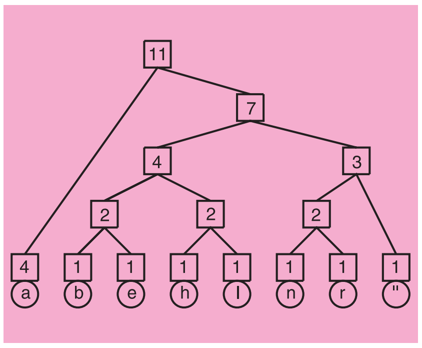
So just how do we code this? We could code it from the algorithm above: allocate a bunch of nodes, put them in a `TList` and then allocate new parent nodes etc, etc. I'm sure you'd agree that there are an awful number of individual node allocations going on. Is there anything we could do to reduce this?

Anything Can Happen

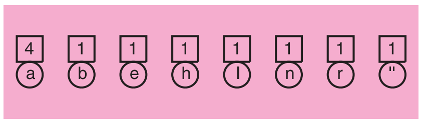
Consider a binary tree with *n* leaves. How many internal nodes must there be? Well, since a tree with two leaves would require a single internal node (the parent of both), and joining a single leaf to the root of an already formed tree would itself require a single parent (increasing the count of internal nodes by one), we can easily show by induction that *n* leaves would require *n*-1 internal nodes.

Using this information we can easily calculate that, for the 256 different byte values we could find in a generic file, we'll need a maximum of 511 nodes to create a fully populated Huffman tree. No more, maybe less; obviously, it depends on the file itself. So at the outset we can create an array of 511 nodes and use them in the creation of the Huffman tree. Since we have an array of nodes, we don't need to use pointers for the parent-child links, at least in the traditional sense. Instead, we can use array indexes to nodes.

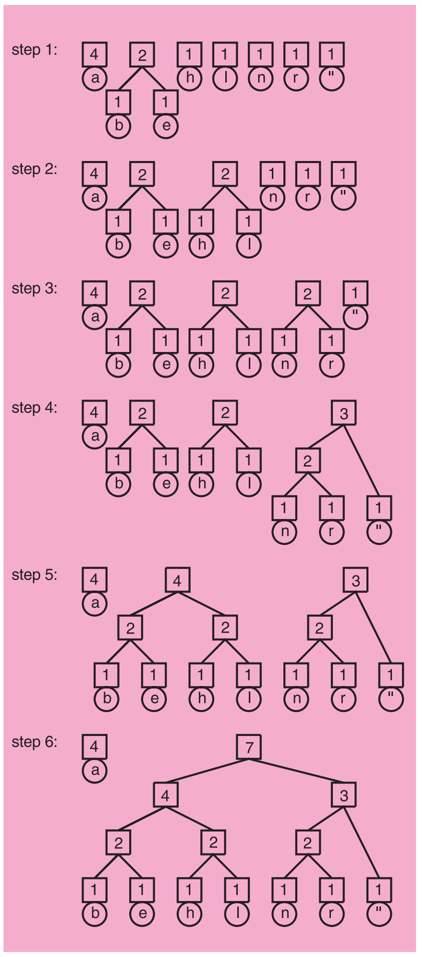
Internal nodes would require the following information: a count value, the index of the left child and the index of the right child. Leaf nodes (those with the actual character) would require the character as well, but not the left or right child indexes. We could save space by using some kind of variant record to accommodate both these types of nodes, but in the long run it isn't really necessary. In fact, we can be even cleverer and recognize that the leaf nodes will appear in the first 256 elements of the node array and that the index of a node in those 256 elements is the ordinal value of the character it represents. We won't need the character field at all. I shall use the node record structure shown in Listing 4.



➤ *Figure 1: Huffman tree for La Habanera.*



➤ *Figure 2: Preliminary nodes ready to build a Huffman tree.*



➤ *Figure 3: Stages in growing a Huffman tree.*

Great, so we've saved some time by using this allocation strategy and the design of the nodes. Looking at the algorithm again, is there any other design strategy we could choose right now to save some speed in the final code? If you are a regular reader of *Algorithms Alfresco*, I would hope that you are seriously considering the priority queue from the November 1998 issue for the part of the algorithm that requires us to remove the two nodes with the smallest counts, because that's indeed what we shall use.

The code that builds the Huffman tree is shown in Listing 5. For brevity I haven't shown the priority queue code since it would just obscure the workings of the Huffman tree algorithm and the code's on the disk anyway.

What Up Dog?

Now we have built the Huffman tree, we can start to apply it to compressing the original data. Again, let's consider some speed improvements before embarking on a brute force implementation. Consider what is about to happen: we are going to read the input file byte by byte and calculate the bit string for each byte and output it. What does the calculation for the bit string involve? In a simple sense we have to find the byte value in one of the leaves of the tree and then walk up the tree from that point until we find the root. The path we took is the reverse of the bit string we want. So it seems we'd need parent links as well so we can move up the tree. Also this would require some pretty intensive code to be executed for every character or byte we have to compress. Surely there's an easier way?

The best thing to do in the circumstances would be to calculate the bit strings for every character or byte *before* we start the compression. Then all we need to do is to pass the pre-calculated bit strings to the output routine for every byte in the input stream. Also, since we are going to calculate the bit strings for *every* byte value, we can just follow every path from the root in order, reaching every leaf and store the bit strings in some array, indexed on byte value.

There is one problem with this: how big are the bit strings? Or, to put it another way, since we'd like to pre-allocate a 256-element array of bit strings, one for each byte value, how big do we make the elements? One algorithm we could use is this: walk the tree twice, the first time to find the longest bit string in the tree, the second time to generate the individual bit strings. The first pass would enable us to calculate the amount of memory needed to store the bit strings and to allocate an array to hold them. Another algorithm, which takes the diametrically opposing viewpoint, is to recognize that the deepest Huffman tree we could form would have 255 levels (essentially it becomes a very long linked list); hence the longest bit string will fit in 32 bytes (there are 256 bits in 32 bytes). And 256 32-byte bit strings is 'only' 4Kb of memory, so just pre-allocate that and be done with it.

For this article, we'll take the latter course. We'll save some time in doing so, and in 32-bit programs we will generally have enough heap memory to play around in.

```
type
  PHuffmanNode = ^THuffmanNode;
  THuffmanNode = packed record
    hnCount    : longint;
    hnLeftInx  : longint;
    hnRightInx : longint;
  end;
  PHuffmanTree = ^THuffmanTree;
  THuffmanTree = array [0..510]
    of THuffmanNode;
```

➤ *Listing 4: Type definitions for the Huffman tree.*

Obviously if you are programming in an environment that does have memory limitations you'd go the opposite route, or even just walk the tree when compressing every character. We also have to ensure that we store the length of the bit string for each character, a `byte` would suffice, but in the interests of efficiency on 32-bit CPUs we'll want to align on a 4-byte boundary and hence we'll use a `longint`.

I won't show the code that generates the bit string array here. Although non-trivial, it is pretty simple to understand.

Look What's Back

So, are we done? Not quite. Imagine that we wrapped up the code we've just presented somehow into a compression program that compresses a file into another. How would we decompress the file we produced? Obviously we'd go roughly the same route as the compression: generate the table of characters and their counts, build the Huffman tree, and then use it to decompress the input bit stream. Stop right there! Reread that sentence and understand what it implies: we have to ship the original file with the compressed file, since without it we can't know the character count table and without that we can't generate the Huffman

➤ *Listing 5: Building the Huffman tree.*

```
procedure BuildHuffmanTree(aHTree : PHuffmanTree;
  var aLastParentInx : integer);
var
  i  : integer;
  PQ : THuffmanPriorityQueue;
  Node1Inx  : longint;
  Node2Inx  : longint;
  ParentInx : integer;
begin
  {create a priority queue}
  PQ := THuffmanPriorityQueue.Create(aHTree);
  try
    {add all the non-zero nodes to the queue}
    for i := 0 to 255 do
      if (aHTree^[i].hnCount <> 0) then
        PQ.Add(i);
    {while there is more than one item in the queue, remove
     the two smallest, join them to a new parent, and add
     the parent to the queue}
    while (PQ.Count > 1) do begin
      Node1Inx := PQ.Remove;
      Node2Inx := PQ.Remove;
      inc(aLastParentInx);
      with aHTree^[aLastParentInx] do begin
        hnLeftInx := Node1Inx;
        hnRightInx := Node2Inx;
        hnCount := aHTree^[Node1Inx].hnCount +
                   aHTree^[Node2Inx].hnCount;
      end;
      PQ.Add(aLastParentInx);
    end;
  finally
    PQ.Free;
  end;
end;
```

tree to be used in the decompression. I'm sure you'll agree that would defeat the purpose of the exercise!

Hence the compression phase of the Huffman algorithm *must* supply either the character count table or the Huffman tree as part of the compressed file. Which is best? The Huffman tree we create in memory is 511 elements of 12 bytes apiece, or 6,132 bytes in size. A little too large to tack onto a compressed file, I'm sure you'll agree. The better solution is to add the character count table. If we just add the 256 `longint` counts, then we'd add a guaranteed 1,024 bytes to the compressed file. An alternative is to add only those counts that are non-zero, and since we'd have to associate each count with a byte value, we'd be outputting 5 bytes for each non-zero count. We could even be clever and switch between the two: if there were 205 or more different byte values in the original file we would output the 256 counts as a single table; if there were less we'd use the byte value plus count method instead. That way we know we'd have the character count table as a 'header' to the compressed file and that it was 1,024 bytes or less.

Some of you who are familiar with the DFM file format in Delphi are no doubt aware that the streaming code outputs positive integer values in three different ways. If the value is between 0 and 255 inclusive, two bytes are output to the stream, the first identifying the next byte as a `byte` value between 0 and 255. If the value is between 256 and 65,535, then three bytes are output, the first identifying that the next two bytes are to be interpreted as a `word` value. Finally if the value is greater than 65,535, five bytes are output: an identifier and a `longint` value. We shall use the same technique for the Huffman character count table: it would certainly help reduce the size of the table in relation to the actual compressed data, especially for smaller files.

Decompressing? Well, we've seen the essential algorithm above when decompressing the compressed words *la habanera*.

1. Set the current node to the root node of the Huffman tree.

2. Read a bit from the input stream. If the bit is 0 follow the left hand link from the current node to get the new current node; if 1, the right hand link.

3. If the node reached is not a leaf, continue at step 2.

4. If the node reached *is* a leaf, output the character corresponding to that leaf to the output stream.

5. If we've output the required number of characters, stop. Otherwise continue at step 1.

So, at this point we are ready to write the compressor and the decompressor. The compressor, as we've seen, must output the character count table to the file in some form before the actual compressed data. It must output before the table one extra item of information: the number of elements in the character count table. From the latter the decompressor 'knows' how many character count items it needs to read from the compressed file and hence it can find out where the actual compressed data starts. How does it know when to stop decompressing? In other words, how many bytes must it end up with after decompression? The answer is hidden: once the decompressor has built the Huffman tree, the root node has the total count of characters in the original file. Listing 6 shows the code for the decompression algorithm.

Betrayal

There's a subtle point about Huffman compression that some of you may have spotted. I admit that it only hit me when I was actually coding the Huffman compression unit, none of my reference books mentioned it. Way back when, as I was describing building the Huffman tree, I said that you must remove the two smallest nodes from the pool of nodes. What happens if you are compressing a stream that just comprises repetitions of a single character? Calculating the character distribution gives you a single node. You can't build a tree out of a single node, at least not one that would make a lot of sense with regard to the Huffman algorithm (what Huffman code would you give the one and only character? The code is the path you take from the root after all, and if there are no paths...). The answer that I came up with was a form of RLE

➤ *Listing 6: Decompressing by walking the Huffman tree.*

```
procedure DoHuffmanDecompression(aInStream : TStream;
  aOutStream : TStream; aHTree : PHuffmanTree;
  aRoot : integer);
const
  Bit : array [0..7] of byte = {bit masks}
    ($01, $02, $04, $08, $10, $20, $40, $80);
var
  CharCount      : longint;
  TotalCharCount : longint;
  BitNum         : integer;
  CollectorByte  : byte;
  CurrNode       : integer;
  GoLeft         : boolean;
  Ch             : char;
begin
  {calculate the total number of characters to decompress;
   preset the loop variables}
  TotalCharCount := aHTree^[aRoot].hnCount;
  CharCount := 0;
  BitNum := 0;
  CurrNode := aRoot;
  {repeat until all the characters have been decompressed}
  while CharCount < TotalCharCount do begin
    {read the next bit}
    if (BitNum = 0) then begin
      aInStream.Read(CollectorByte, sizeof(CollectorByte));
      BitNum := 7;
    end else
      dec(BitNum);
    GoLeft := (CollectorByte and Bit[BitNum]) = 0;
    {walk down the Huffman tree}
    if GoLeft then
      CurrNode := aHTree^[CurrNode].hnLeftInx
    else
      CurrNode := aHTree^[CurrNode].hnRightInx;
    {if we have reached a leaf, output the character
     concerned, and reset the current node to the root}
    if (CurrNode < 256) then begin
      Ch := char(CurrNode);
      aOutStream.Write(Ch, sizeof(byte));
      CurrNode := aRoot;
      inc(CharCount);
    end;
  end;
end;
```

compression (run length encoding: where runs of characters are encoded as character and character count pairs) output the character distribution table only (that is, just the one character and its count). On decompression, if the decompressor notices just a single character in the distribution, it recreates the stream of single characters.

The disk accompanying this issue contains a unit to perform Huffman compression and decompression. To make the routines widely usable, I've coded them to operate on streams; thus you can compress data in memory or BLOBs just as easily as complete files. To make it even faster for certain streams, I've included on the disk a unit containing a `TStream` descendant that acts as a buffer for any other stream. It's a variation on an early article of mine for *The Delphi Magazine* (the January 1998 issue to be precise).

Using the Huffman compression routine I compressed the text for *Love's Labour's Lost* and achieved a 61% compression ratio. For comparison, WinZip achieved an overall ratio of 40% for the same file (measured as size of Zip file to size of original; note that WinZip quotes a 'reduction' ratio of 60%, which is equal to 100% less the compression ratio I'm quoting).

The Party Broke Up

There are a couple of points that must be stressed about Huffman encoding. Firstly, it is hampered by having to pass enough information with the compressed data to enable the decompressor to rebuild exactly the same tree the data was compressed with (in our case, the character count table). This makes it unsuitable for small files: the table will overwhelm the compressed data in that case, possibly resulting in a compressed file that is larger than the uncompressed file.

Secondly, the compressor must read the input file twice: once to generate the character count table and the other time to compress the data. For small files, that wouldn't be too bad, since the operating system will tend to have the file in its cache for the second pass, but for larger files, it will cause Huffman encoding to be slower than compression algorithms that only rely on a single pass through the file.

There is a modified Huffman compression algorithm called adaptive Huffman compression that avoids these two problems, but we must leave that for another time. Happy squeezing!

Julian Bucknall is highly compressible at the moment and needs to go on a diet and start exercising again, ready for the Bolder Boulder 10Km road race. He was ready, but was not before: was not was. He can be reached at julianb@turbopower.com. The code accompanying this article is freeware and can be used as-is in your own applications.

© Julian M Bucknall, 1999