Cherchez La Femme

Continuing an investigation into data compression

by Julian Bucknall

In last month's article we looked at Huffman encoding as a way to compress data. Although I didn't really stress it at the time, Huffman encoding is the best you can do if you are trying to replace individual bytes with bit string codes. Huffman guarantees that the bit strings it generates for the bytes that appear in the original data stream are the minimal ones required for the job.

This month we'll briefly look at another byte encoding method, even older than Huffman, before moving onto an underlying algorithm used in the Zip file format.

Shannon-Fano Encoding

Another byte encoding method which had its heyday just before Huffman wrote his seminal paper was the Shannon-Fano encoding method. Invented separately and simultaneously by two researchers, Claude Shannon and RM Fano, it attempts to build the minimal encodings from the top down (unlike the Huffman method, which, if you think about it whilst looking at last's month's *Algorithms Alfresco*, is a bottom up method).

Just as for last month, we will use the song title *La Habanera* as an example chunk of data to compress. The words have the following distribution of letters (ignoring case sensitivity for now):

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

Draw a horizontal line to divide the letters up so that (roughly) the same distribution occurs above the line as below it:

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

There's a distribution value of 5 characters above the line and 6 characters below it. For each of the halves do the same, and repeat the algorithm until each individual letter is between two lines:

```
a        4
---------------2
b        1
----------------1
e        1
--------------3
h        1
-------------4
l        1
---------------2
n        1
--------------3
r        1
-------------4
space    1
```

I've tried to be clever in the example above and not only identified each line by a number denoting the level it was added, but also I have made the lines smaller each time. The reason why? Turn the magazine anti-clockwise by 90 degrees and look at the lines again. If you imagine links between the (level) 1 and the 2s, and also between the 2s and the 3s, and again between the 3s and the 4s, and finally between the lines and the actual characters, hey presto, you've got yourself a binary tree.

If we start at the root and take a move left as meaning we add a zero bit, and a move right as adding a one bit, we get the following encoding:

```
a        00
b        01
e        100
h        1010
l        1011
n        110
r        1110
space    1111
```

We can then encode *La Habanera* as the following bit string, comprising 32 bits in all:

```
10110011
11101000
01001101
00111000
```

Which is, if you recall from last month's article, pretty close to the optimal Huffman encoding of 31 bits.

New Beginnings

Up until 1977, the main thrust of compression research centered on either the Shannon-Fano or the Huffman algorithms, either in making them dynamic (so that the code table didn't have to be part of the compressed file), or in various speed, space and efficiency improvements.

Then, suddenly, two Israeli researchers, Jacob Ziv and Abraham Lempel, came up with a radically different way of compression and opened up research into a completely different direction.

Their main idea was not to try and encode single characters, but instead to encode *strings* of characters. Their idea was to use a *dictionary* of previously seen *phrases* from the file being compressed to help encode later phrases.

Suppose that you had a normal English dictionary. Every word you'd encounter in a given text file would appear in the dictionary. If the compressor and decompressor programs had access to an electronic version of this dictionary, they could encode individual words in the text file by finding them in the dictionary and outputting the page number and the number of the word on the page.

We could assume that a 2-byte integer would be able to hold the page number (there are not that many dictionaries with more than 65,536 pages) and a byte should be able to hold the word number on the page (again there are never usually more than 256 words defined on a page in a dictionary), hence each word in the text file, no matter how long, would be replaced by three bytes. Obviously small words like *a*, *in*, *up*, and so on, would grow in size instead of being compressed, but the majority of words are three or more letters long and so the overall size of the compressed file would tend to decrease.

Ziv and Lempel's idea followed these lines. Instead of having a static pre-built dictionary, though, their new algorithm generated a dictionary on the fly from the data that the compressor had already seen in the input file. And instead of using page numbers and word numbers on the page, they output *distance* and *length* values.

It works a little like this: as you read through the input file, you attempt to match up the set of characters at your current position with something you've already seen in the input file. If you do find a match you calculate the distance of the matching string from your current position and the number of bytes (the length) that match. If you manage to find several matches, you choose the longest one.

Dictionary Encoding

A small example would serve us here. Suppose we were compressing the sentence:

```
a cat is a cat is a cat
```

The first character, *a*, doesn't match with anything yet seen (well, duh, we haven't seen anything yet!), so we output it to the compressed stream. Similarly with the following *space* and *c*. The next *a* matches a previous *a* but that's all, we can't match anything more. Let us impose the rule that we only want to match three characters or more before we get excited. So, we output the *a* to the output stream. Similarly for the *t*, *space*, *i*, *s*, and *space*. We can visualize the current state of play as the following:

```
---------+
a cat is |a cat is a cat
---------+^
```

where the characters seen are in the box and the current position is marked with a caret.

Now it gets interesting. The set of characters '`a cat is `' at the current position matches something we've already seen before. The matching string occurs nine characters before our current position and we can match nine characters. So we can output a distance/length pair, represented by `<9,9>`, to the output file and then advance nine characters. The state of play is then

```
------------------+
a cat is a cat is |a cat
------------------+^
```

But, hey, again we can match the set of characters at the current position with something that's gone before. We have a choice in fact, we can either match five characters nine characters before, or five characters 18 characters before. Let's choose the first option, `<9,5>`. Our compressed stream looks like this

```
a cat is <9,9><9,5>
```

Decompressing this stream is quite easy too. As we decompress, we build up a buffer of decompressed characters so that we can decode the distance/length pairs or codes. Literal characters in the compressed stream are output to the decompressed stream as they are.

The first nine codes in the compressed stream are literal characters so we output them to the decompressed stream as is, and also we create a buffer (called a *sliding window*) at the same time. The buffer looks like this at this point:

```
---------+
a cat is |
---------+
```

The next code in the compressed stream is a distance/length pair, `<9,9>`. We decode this as, output the nine characters found at a distance of nine bytes back in the buffer. Those nine characters are '`a cat is `' and so we output those to our uncompressed stream and add them to our buffer, or sliding window:

```
------------------+
a cat is a cat is |
------------------+
```

Again the next code in the compressed stream is a distance/length pair, `<9,5>`, and I'm sure you can decode that using the buffer we have.

With this small example we haven't needed a dictionary *per se*, we just used brute force (or our expert eye!) to find the longest match in the set of previously seen characters.

Sliding Windows

By the way, in case you were wondering, defining the buffer of previously seen characters as a 'sliding window' means that we only consider the previous *n* bytes in trying to find a possible match; *n* is usually something like 4Kb or 8Kb (the Deflate algorithm in PKZIP can use a sliding window of up to 32Kb in size). As we advance the current position, so we slide the window forward on the data we've already seen. Why do we do this? Why not use the entirety of the previously seen text? The answer to this boils down to how text is generally structured. In general, text we read and write

obeys a rule called *locality of reference*. What this term means is that characters in a text file tend to match other characters close by rather than far away. In a novel, for example, the protagonists and locations the narrative is describing tend to be 'clumped' together in sections of chapters or chapters. The standard words and phrases like *and*, *the*, and *he said* occur throughout the novel.

Other text, like that in reference books or my articles, also exhibits locality of reference. Hence it makes sense to limit the amount of previously seen text we have to search through for a matching string: locality of reference tells us that it makes sense. Another strong reason is that the more text we have to search through, the slower the compression becomes.

Consider also how we are to encode the distance/length pair. I haven't talked about this yet, but it makes sense to pack them in as small a space as possible. If we have a sliding window over the last 4Kb of text, for example we can encode a distance value in 12 bits (2^{12} is 4Kb). If we limit the maximum length we try and match to 15 characters, we can encode that in 4 bits, and suddenly we see that we can manage to encode a distance/length pair in two bytes. We could also use an 8Kb window and a maximum of 7 matched characters and still fit into two bytes. Our compressed stream can be written and read in byte units, rather than having to mess around with writing and reading odd numbers of bits, as in the Huffman case. Also, if we limit the distance/length codes to two bytes, it means that strings of three characters or more which match with something prior can get compressed, whereas matches of one or two characters can be safely ignored since they won't compress.

Without being too rigorous, this is the essence of Ziv and Lempel's algorithm, usually known these days as LZ77 (ie, their initials and a non-Y2K-compliant year).

The above discussion does leave out a small implementation detail: how do you tell the difference between a literal character and a distance/length code as you are reading through the compressed data? After all, there is nothing intrinsically different between a literal character and the first byte of a distance/length pair. One simple answer is to output a single flag bit before the literal character or distance/length code: if the flag bit is on, the next code read will be a literal character, if the flag bit is off, the next code read will be a distance/length pair. But, stop a moment, this would mean that we'll be outputting odd bits again and our previously acclaimed advantage of reading/writing of entire bytes is thrown out of the window. What to do?

The general way around this disadvantage is to have a byte of eight flag bits that tells you what the next eight codes are going to be. The first bit denotes what the first code after the flag byte is going to be, the second bit what the second code is going to be, and so on for eight bits and codes, after which

there will be another flag byte. Using this scheme we can write (and read) the compressed stream as bytes.

Trouble Ahead

A similar scheme is used, for example, by the old Microsoft EXPAND.EXE program you used to get with MS-DOS or Windows 3.1 (nowadays, Microsoft products use Cabinet files instead). I'm sure the oldies amongst us would remember them, the files on the DOS diskettes used to have names like FILENAME.EX_ and the EXPAND.EXE program would decompress them and fix the final character of the extension in the decompressed file into the bargain. With Microsoft's version, the distance/length codes were always 2 bytes in size, 12 bits of which being the 'distance' value (in fact, the MS version used a circular queue of bytes and the distance value was an offset from the start of the queue), with the other 4 bits being the length value. Note that when perusing my old diskettes I noticed that the MS-DOS 6.22 diskettes used a different compression scheme, but my old Windows 3.1 diskettes used this LZ77 compression: the date when Microsoft switched compression methods seemed to have been in 1993. My research for this article also revealed that the old Windows 3.1 help files used this type of LZ77 compression for the text within them.

This article won't attempt to emulate what Microsoft used to do, though. The reason? Patents. I don't know how you feel, but in my view patents are the bane of computer algorithms. They simply should not be allowed *[Amen! Ed]*. The compression field is riddled with patented algorithms and negotiating it is like walking through a minefield. The original LZ77 was not patented. Microsoft, however, patented their LZ77 variant (the one used by EXPAND.EXE and Windows 3.1 help files). Ziv and Lempel went on to describe a different dictionary compression algorithm, known as LZ78: unpatented. Terry Welch of Unisys described a variant of this algorithm (known as LZW, essentially it preloaded the dictionary) which was patented and which formed the basis of the GIF file format. The result? Unisys collect a royalty for every GIF viewer sold. Whilst researching this article, I came across other patents for simple variants of LZ77, each of which sounded exactly like each other. It's a mess. US law states that ideas are not patentable, and yet someone somewhere managed to patent an algorithm and the gates from then on were opened. Imagine the mess we'd all be in if quicksort, the fastest general sort, or red-black binary trees, the basis for associative arrays in C++'s STL, were patented. Every time you used one in your program you'd have to pay a small sum to someone. Brrrr.

Our Own LZ77 Algorithm

Anyway, enough soapbox, what we'll do from now on is to describe the *Algorithms Alfresco* LZ77 algorithm. (Drum roll, fanfare.) As far as I know, this particular variant is not patented, mainly because it employs ideas and algorithms from other *Algorithms Alfresco* articles and borrows from the Deflate algorithm in PKZIP, which itself is not patented (Phil Katz, its inventor, was far-sighted enough to put it in the public domain). Of course, its main basis is still the LZ77 algorithm.

Let's define some variables. We'll employ the eight flag bits in a byte trick. If a flag bit is zero, the associated code will be a literal character; if it is one, the associated code will be a distance/length pair. This latter code will always be two bytes in length, as discussed, with 13 bits for the distance value and 3 bits for the length value. In fact, we shall use the upper 13 bits of the two bytes for the distance and the lower 3 bits for the length.

Because we have 13 bits for the distance value, in theory we can encode distances from 0 to 8,191 bytes, and so our sliding window will be 8Kb in size. Note that we can't have a distance of 0 bytes (we'd be matching with the current position) and so instead we'll interpret these 13 bits as being values from 1 to 8,192 by the simple expedient of adding one.

Consider the length value. In theory, using three bits, we can only encode lengths from 0 to 7. Remember, though, we only try

➤ *Listing 1: Sliding window methods under decompression.*

```
procedure TaaLZSlidingWindow.swAdvanceAfterAdd(
  aCount : integer);
begin
  {advance the start of the sliding window, if required}
  if ((swCurrent-swStart) >= aalzSlidingWindowSize)
    then begin
    inc(swStart, aCount);
    inc(swStartOffset, aCount);
  end;
  {advance the current pointer}
  inc(swCurrent, aCount);
  {check to see if we have advanced into the overflow zone}
  if (swStart >= swMidPoint) then begin
    {write some more data to the stream (from swBuffer
     to swStart)}
    swWriteToStream(false);
    {move current data back to the start of the buffer}
    Move(swStart^, swBuffer^, swCurrent - swStart);
    {reset the various pointers}
    dec(swCurrent, swStart - swBuffer);
    swStart := swBuffer;
  end;
end;
procedure TaaLZSlidingWindow.AddChar(aCh : char);
begin
  {add the character to the buffer}
  swCurrent^ := aCh;
  {advance the start of the sliding window}
  swAdvanceAfterAdd(1);
end;
procedure TaaLZSlidingWindow.AddCode(aDistance : integer;
  aLength : integer);
var
  FromChar : PChar;
  ToChar   : PChar;
  i        : integer;
begin
  {set up the pointers to do the data copy; note we cannot
   use Move since part of the data we are copying may be
   set up by the actual copying of the data}
  FromChar := swCurrent - aDistance;
  ToChar := swCurrent;
  for i := 1 to aLength do begin
    ToChar^ := FromChar^;
    inc(FromChar);
    inc(ToChar);
  end;
  {advance the start of the sliding window}
  swAdvanceAfterAdd(aLength);
end;
```

```
procedure AALZDecompress(aInStream, aOutStream : TStream);
type
  TModeState = (msGetFlagByte, msGetChar, msGetDistLen);
var
  SlideWin      : TaaLZSlidingWindow;
  BytesUnpacked : longint;
  TotalSize     : longint;
  ModeState     : TModeState;
  FlagByte      : byte;
  FlagMask      : byte;
  NextChar      : char;
  NextDistLen   : longint;
  CodeCount     : integer;
  Len           : integer;
begin
  SlideWin := TaaLZSlidingWindow.Create(aOutStream, false);
  try
    {read the uncompressed size from the stream}
    StreamRead(aInStream, TotalSize, sizeof(TotalSize));
    {prepare for the decompression}
    BytesUnpacked := 0;
    NextDistLen := 0;
    ModeState := msGetFlagByte;
    {while there are still bytes to decompress...}
    while (BytesUnpacked < TotalSize) do begin
      {read the next item}
      case ModeState of
        msGetFlagByte :
          begin
            StreamRead(aInStream, FlagByte, 1);
            CodeCount := 0;
            FlagMask := 1;
          end;
        msGetChar :
          begin
            StreamRead(aInStream, NextChar, 1);
            SlideWin.AddChar(NextChar);
            inc(BytesUnpacked);
          end;
        msGetDistLen :
          begin
            StreamRead(aInStream, NextDistLen, 2);
            Len := (NextDistLen and $7) + 3;
            SlideWin.AddCode((NextDistLen shr 3)+1, Len);
            inc(BytesUnpacked, Len);
          end;
      end;
      {calculate the next mode state}
      inc(CodeCount);
      if (CodeCount > 8) then
        ModeState := msGetFlagByte
      else begin
        if ((FlagByte and FlagMask) = 0) then
          ModeState := msGetChar
        else
          ModeState := msGetDistLen;
        FlagMask := FlagMask shl 1;
      end;
    end;
  finally
    SlideWin.Free;
  end;{try..finally}
end;
```

➤ *Listing 2: Algorithms Alfresco LZ decompression.*

and convert matching strings of three characters or more into distance/length pairs. Any less, and there would be no compression. So, it makes sense to interpret the 3 bits as being lengths of 3 to 10 bytes by simply adding 3.

Hence, to convert a distance and a length amount into a two byte value we'd write something like this:

```
Code := ((Distance-1) shl 3)+
        (Length-3);
```

And to get the distance and length values back, we'd code this:

```
Length := (Code and $7)+3;
Distance := (Code shr 3)+1;
```

Pretty simple stuff, I'm sure you'll agree.

Decompression

Let's look at the decompression method first, since conceptually it is the easiest to visualize.

With decompression, we read a flag byte and then use it to determine how we should read the next eight codes from the stream. If the current bit in the flag byte is zero, we read one byte from the stream and interpret it as a literal character to be written straight to the output stream. If, on the other hand, the current bit is one, we read two bytes from the input stream and split the value into distance and length values. We then use these with our current sliding window of previously decoded data to interpret what characters should be written to the output stream.

Every time we decode a single character, or a set of three to ten characters, not only do we have to write them to the output stream, as I have already indicated, but we have to add them onto the end of the sliding window buffer and advance the start of the sliding window by a commensurate amount so that the window size never exceeds 8,192 bytes.

In a lot of LZ77 implementations this is done with a circular queue. Well, just because we're different, we'll use a modification of the circular queue which I introduced in December 1998's *Algorithms Alfresco*. If you remember, this implemented a circular queue as a sliding buffer of data: when the head pointer reached the halfway mark of the buffer, the data was moved down to the start of the buffer again, and the pointers reset.

So, for a sliding window of 8,192 bytes, we'd need at least a 16,384 byte buffer and when the pointer to the start of the sliding window reached the halfway point, we'd move the window to the start of the buffer and reset the pointers. Since the compression phase would also use a similar sliding window (we'll discuss how in a moment) it makes sense to have a shareable class implementation.

Look-Ahead Buffer

Before we go on to describe the methods we'd need for this class, I want to describe a little trick that's employed by PKZIP's Deflate method.

Go back to the example sentence we were compressing above. At one stage in describing the algorithm we had the following position:

```
---------+
a cat is |a cat is a cat
---------+^
```

and we discovered the distance/length pair of <9,9>. However, there is a little trick we can use. Why stop at matching 9 characters? We can in fact match more than that by going *beyond* the right boundary of the sliding window, and continue matching with the current character and others to its right. We could in fact match 14 characters in all, to give a code of <9,14>, with the length being greater than the distance. All very well, and pretty clever, but what happens on decoding? At the point where we have to decode <9,14> we have this in our sliding window:

```
---------+
a cat is |
---------+
```

We go back nine characters into the window and start copying characters, one by one until we reach 14 of them. It turns out that we end up copying characters we have managed to set *as part of the same operation*. After copying nine characters we have

```
---------+
a cat is |a cat is
---------+^        ^
      from^      to^
```

with the places we are copying from and copying to shown. As you can see, we can easily copy the remaining five characters with no problem at all. Hence, it is simplicity itself to have a length value greater than a distance value (although, it must be admitted, we can't just use the `Move` procedure to copy the data).

What we shall do with the sliding window class during decompression is to pass it the output stream to which the data is to be written. That way, when the object determines that it needs to slide the active data in the buffer back to the start, it can first copy the data it is about to overwrite to the stream. There are two main methods we need for decompression: adding a single character, and converting a distance/length pair. Note that we make the sliding window class perform these actions since we need to update the sliding window and advance in both cases, and also the class is the best agent for converting the distance and length values. The code is shown in Listing 1.

That done, the decompression code is fairly simple to write. We'll code the main loop as a state machine with three states: read and process a flag byte, read and process a character, and, finally, read and process a distance/length code. The code is shown in Listing 2. Notice that we determine when to end the decompression by utilizing the fact that the compressor writes the number of bytes in the uncompressed stream to the start of the compressed stream.

Compression

So, compression then. It's the really meaty part of the algorithm, as it turns out. I'm sure that many of you are wondering how on earth we can manage to find the maximum matching string in the sliding window data. That is, apart from comparing the current set of characters against all 8,192 possible positions in the preceding data, which you'll agree would turn this algorithm into something slower than a three-toed sloth. So, given the current set of characters, how do we efficiently find the longest match in the previous 8,192 bytes?

Ziv and Lempel didn't suggest much in their original paper. Some people use a binary search tree built over the sliding window to store the (maximum length) strings previously seen (an example is Mark Nelson's implementation, see the references at the end). Instead, we'll make use of a hint presented in the internet document *Deflate Compressed Data Format Specification* (RFC1951): use a hash table.

Here's the plan. We get the three characters at the current position. (Why three? Because it is the minimal length match we can make.) Hash them using some routine to get a hash value. Use the hash value to access an element in a hash table. What should this

➤ *Listing 3: The hash table used by Algorithms Alfresco LZ compression.*

```
constructor TaaLZHashTable.Create;
begin
  inherited Create;
  htlArray := TList.Create;
  htlArray.Count := HashTableSize;
end;
destructor TaaLZHashTable.Destroy;
begin
  if (htlArray <> nil) then begin
    Empty;
    htlArray.Free;
  end;
  inherited Destroy;
end;
procedure TaaLZHashTable.Empty;
var
  Inx : integer;
  ChainHead : PaaLZHashNode;
begin
  for Inx := 0 to pred(HashTableSize) do begin
    ChainHead := PaaLZHashNode(htlArray[Inx]);
    if (ChainHead <> nil) then begin
      htlFreeChain(ChainHead, false);
      htlArray[Inx] := nil;
    end;
  end;
end;
function TaaLZHashTable.FindAll(const aKey : TaaLZKey;
  aCutOffset : longint; aAction : ThtLZKeyEnumProc;
  aExtraData : pointer) : boolean;
var
  Inx : integer;
  Temp : PaaLZHashNode;
  Dad  : PaaLZHashNode;
begin
  {assume we don't find any}
  Result := false;
  {calculate the hash table index for this key}
  Inx := (aKey.AsLong shr 8) mod HashTableSize;
  {wander along the chain at this index}
  Dad := nil;
  Temp := PaaLZHashNode(htlArray[Inx]);
  while (Temp <> nil) do begin
    {if this node has an offset that is less than the cutoff
     offset, then remove the rest of this chain and exit}
    if (Temp^.hnOffset < aCutOffset) then begin
      if (Dad = nil) then begin
        htlFreeChain(Temp, false);
        htlArray[Inx] := nil;
      end else
        htlFreeChain(Dad, true);
      Exit;
    end;
    {if the node's key matches our key, call action routine}
    if (Temp^.hnKey.AsLong = aKey.AsLong) then begin
      Result := true;
      aAction(aExtraData, aKey, Temp^.hnOffset);
    end;
    {advance to the next node}
    Dad := Temp;
    Temp := Dad^.hnNext;
  end;
end;
procedure TaaLZHashTable.Insert(const aKey : TaaLZKey;
  aOffset : longint);
var
  Inx     : integer;
  NewNode : PaaLZHashNode;
begin
  {calculate the hash table index for this key}
  Inx := (aKey.AsLong shr 8) mod HashTableSize;
  {allocate a new node and insert at the head of the chain
    at this index in the hash table; this ensures that the
    nodes in the chain are in reverse order of offset value}
  NewNode := snmAllocNode;
  NewNode^.hnKey := aKey;
  NewNode^.hnOffset := aOffset;
  NewNode^.hnNext := htlArray[Inx];
  htlArray[Inx] := NewNode;
end;
```

```
procedure TaaLZSlidingWindow.Advance(aCount : integer);
var
  ByteCount   : integer;
begin
  {advance the start of the sliding window, if required}
  if ((swCurrent-swStart) >= aalzSlidingWindowSize)
    then begin
    inc(swStart, aCount);
    inc(swStartOffset, aCount);
  end;
  {advance the current pointer}
  inc(swCurrent, aCount);
  {check to see if we have advanced into the overflow zone}
  if (swStart >= swMidPoint) then begin
    {move current data back to the start of the buffer}
    ByteCount := swLookAheadEnd - swStart;
    Move(swStart^, swBuffer^, ByteCount);
    {reset the various pointers}
    ByteCount := swStart - swBuffer;
    swStart := swBuffer;
    dec(swCurrent, ByteCount);
    dec(swLookAheadEnd, ByteCount);
    {read some more data from the stream}
    swReadFromStream;
  end;
end;
function TaaLZSlidingWindow.Compare(aOffset : longint;
  var aDistance : integer) : integer;
var
  MatchStr  : PChar;
  CurrentCh : PChar;
begin
  {Note: when this routine is called it is assumed that at
   least three characters will match between the passed
   position and the current position}
  {calculate the position in the sliding window for the
   passed offset and its distance from the current
   position}
  MatchStr := swStart + (aOffset - swStartOffset);
  aDistance := swCurrent - MatchStr;
  inc(MatchStr, 3);
  {calculate the length of the matching characters between
   this and the current position. Don't go above the
   maximum length. Have a special case for the end of the
   input stream}
  Result := 3;
  CurrentCh := swCurrent + 3;
  if (CurrentCh <> swLookAheadEnd) then begin
    while (Result < aalzMaxMatchLength) and
      (MatchStr^ = CurrentCh^) do begin
      inc(Result);
      inc(MatchStr);
      inc(CurrentCh);
      if (CurrentCh = swLookAheadEnd) then
        Break;
    end;
  end;
end;
procedure TaaLZSlidingWindow.GetNextKey(var aMS : TaaLZKey;
  var aOffset : longint);
var
  P : PChar;
  i : integer;
begin
  {calculate the length of the match string; usually it's 3,
   but at the end of the input stream it could be 2 or less}
  if ((swLookAheadEnd - swCurrent) < 3) then
    aMS.AsString[0] := char(swLookAheadEnd - swCurrent)
  else
    aMS.AsString[0] := #3;
  P := swCurrent;
  for i := 1 to length(aMS.AsString) do begin
    aMS.AsString[i] := P^;
    inc(P);
  end;
  aOffset := swStartOffset + (swCurrent - swStart);
end;
```

➤ *Listing 4: Sliding window methods under compression.*

element consist of? In my previous excursion into hash tables (*The Delphi Magazine*, February and March 1998) I used a linear probe hash table; this time, however, we shall use a hash table with chaining: all items that hash to the same value will form a linked list (a chain) at the required element. The items being stored in these linked lists will consist of the three character 'signature' as well as the offset in the input stream where the signature occurred.

All right then, we have the three character signature at the current position and we've hashed to a linked list. We follow the linked list and compare each item's signature in it to ours. If they're equal we go to the sliding window buffer using the item's offset value and then compare the characters in the sliding window with those at the current position. We do this with every item in the linked list that matches signatures, and keep a note of the largest match we find.

After this search, we'll need to add the current signature to the hash table so that we may find it with subsequent signatures. We add it to the front of the linked list, thereby ensuring that the linked list becomes sorted in reverse order by offset value.

But hang on a moment! With the description I've just given, the number of items in the hash table will just keep on growing. In fact, we don't need the items which no longer appear in our 8Kb sliding window. There's a choice here: we could remove the item that's just about to disappear out of the sliding window when we slide it along (ie, find the signature of the position just about to disappear from the back of the sliding window, hash it, follow the linked list at that position in the hash table until we find the relevant item, and then delete it), or we could be a little cleverer. Remember I said that the linked lists are sorted in descending order by offset value? Well, as we are stepping along a linked list trying to find a maximum match for the current position, if we should hit an item with a very 'old' offset value (ie one that no longer appears in the sliding window) we get rid of it *and all subsequent items in that linked list*. Brilliant! We defer removal of old items to our search through the linked list routine, when we're actually there in the middle of the linked list, in fact. This does mean that the hash table contains more items than it need to, but this is small potatoes compared with the benefit of a speedier algorithm.

Phew! What else? Well, the hash function needs to be decided on, for a start. Since the signatures are 3 characters long, we can pretend that they are the least significant 3 bytes of a longint, with the most significant byte being zero. Bingo, a hash value, with practically no calculation. As usual, the hash table size is a prime number: I chose 521, the smallest prime greater than 512. This means that, on average, 16 signatures from our 8Kb sliding window will map to the same element number: forming a reasonably sized linked list to step along during our search.

I can imagine that my regular readers are wondering whether I shall be using a node manager to perform the linked list node allocations and frees as I did in *Algorithms Alfresco* for February 1999. Well, of course! Think about it: as we travel through the input stream we'll be allocating exactly one node and, on average, freeing

one node for each character encountered. So for a reasonably sized file, say about 64Kb, we'll be allocating 65,000 nodes and freeing them all; all of them being the same size of course. Relying on the Delphi heap manager for this lot would certainly be inefficient.

The code for the hash table class is shown in Listing 3.

What happens to the sliding window during compression? Well, first of all, it would be nice if the sliding window class was given the responsibility for reading data from the input stream; that way the user of the class can forget about replenishing the data in its buffer. What else? Well, we'll need a routine that returns the three character signature at the current position, together with its offset value. We shall also need a routine that gets passed an offset value, that converts this offset value into a position into the sliding window, and that compares the characters there with the characters at the current position. It should return the number of characters that match (which will be at least three) and the distance value for that offset. Listing 4 has the details.

And now, finally, we can write the compressor routine (Listing 5). The routine is slightly complicated by the need to accumulate compression codes eight at a time, so that we can prepend them in the output stream with a flag byte; that's what the `Encodings` array is all about. However, since we have a lot of supporting code all worked out, the routine itself is not too hard to understand. As usual the full code is available on the accompanying disk.

Conclusion

Was all this worth it? Well, in theory, if we could compress all 10 byte strings in a file down to 2 (a maximal match every time) for every 80 bytes of the file we'd write out 17 (one flag byte and eight codes): a compression ratio of 22%. If, on the other hand, we could find *no* matches in the file, we'd actually write out nine bytes for every eight in the original file, a 'compression' ratio of 113%. Generally compressing files with this method would tend to fall somewhere between these two extremes (my favorite *Love's Labour's Lost* compresses down to 59%, for example).

And with that we come to the end of another fairly difficult, fairly complex, algorithm column. Congratulations if you made it all the way through only drinking one mug of coffee. Don't feel too despondent if you feel overwhelmed: before embarking on this article I'd never written an LZ77 sliding window implementation and it took around two weeks of lunchtime and weekend coding to get it right. Something simpler is in the works for next time!

Julian Bucknall is feeling squeezed dry. No more compression for a while! He can be reached at julianb@turbopower.com. The code that accompanies this article is freeware and can be used as-is in your own applications.

References

The Data Compression Book by Mark Nelson, Prentice Hall.

Microsoft's Compression File Format by Pete Davis in *Windows/DOS Developer's Journal*, July 1994.

Deflate Compressed Data Format Specification by P. Deutsch, RFC1951.

➤ *Listing 5: Algorithms Alfresco LZ compression.*

```
procedure AALZCompress(aInStream, aOutStream : TStream);
var
  HashTable : TaaLZHashTable;
  SlideWin  : TaaLZSlidingWindow;
  Key       : TaaLZKey;
  Offset    : longint;
  CodeCount : integer;
  Encodings : TEncodingArray;
  EnumData  : TEnumExtraData;
  LongValue : longint;
  i         : integer;
begin
  HashTable := TaaLZHashTable.Create;
  try
    SlideWin := TaaLZSlidingWindow.Create(aInStream, true);
    try
      {write uncompressed size of input stream to stream}
      LongValue := aInStream.Size;
      StreamWrite(aOutStream, LongValue, sizeof(LongValue));
      {prepare for the compression}
      CodeCount := 0;
      FillChar(Encodings, sizeof(Encodings), 0);
      {get the first key}
      SlideWin.GetNextKey(Key, Offset);
      {while the key is three characters long...}
      while (length(Key.AsString) = 3) do begin
        {find the longest match in the sliding window using
         the hash table to identify matches}
        EnumData.edSW := SlideWin;
        EnumData.edMaxLen := 0;
        if HashTable.FindAll(Key,
          Offset - aalzSlidingWindowSize, MatchLongest,
          @EnumData) then begin
          {we have a match: save the distance/length pair
           and advance the sliding window by the length}
          AddCodeToEncodings(aOutStream,
            EnumData.edDistMaxMatch, EnumData.edMaxLen,
            Encodings, CodeCount);
          SlideWin.Advance(EnumData.edMaxLen);
        end else begin
          {we don't have a match: save the current character
           and advance by 1}
          AddCharToEncodings(aOutStream, Key.AsString[1],
            Encodings, CodeCount);
          SlideWin.Advance(1);
        end;
        {now add this key to the hash table}
        HashTable.Insert(Key, Offset);
        {get the next key}
        SlideWin.GetNextKey(Key, Offset);
      end;
      {if the last key was two characters or less, save them
       as literal character encodings}
      if (length(Key.AsString) > 0) then begin
        for i := 1 to length(Key.AsString) do
          AddCharToEncodings(aOutStream, Key.AsString[i],
            Encodings, CodeCount);
      end;
      {make sure we write out the final encodings}
      if (CodeCount > 0) then
        WriteEncodings(aOutStream, Encodings, CodeCount);
    finally
      SlideWin.Free;
    end;{try..finally}
  finally
    HashTable.Free;
  end;{try..finally}
end;
```