Squeeze, Please

This month we look at compression with the Burrows-Wheeler Transform

by Julian Bucknall

For the last month I've been rewriting the implementation of the Deflate compression algorithm we had in Abbrevia, ready for version 3 (Deflate is the compression algorithm used in zip files). There were three main reasons for doing this: the current code was a maintenance nightmare, we wanted to add support for Deflate64, PKWare's new compression algorithm, and I wanted to be able to tune the implementation to produce faster and better compression. I succeeded: the code is now easier to understand, it's liberally peppered with assertions and logging statements to make debugging and maintenance easier, and (after many sessions with Sleuth QA Suite) it's faster.

But that's not the point of this article. The big problem with the Deflate algorithm is that, although it is very particular in its definition of the final format of the compressed data, how you get to that point is left as an interesting exercise for the reader. And what an interesting exercise it is (can you spot the cynical tone?). Back in April and May 1999, I described Huffman encoding and LZ77 compression. Well, Deflate is essentially LZ77 compression into literal and length/distance tokens followed by encoding of those tokens using Huffman trees (and then Huffman encoding of those Huffman trees, but I'm sure you get the drift). Although that sounds pretty cut and dried, in reality finding the best encoding for a given uncompressed stream has been shown to be NP-complete, that is, of the same level of difficulty as the Traveling Salesman problem. I spent a lot of time tweaking here and tweaking there, rewriting this and that, trying to get the compression ratio down.

Over all this time, I was wishing for a better-defined algorithm for compression that could produce compression ratios as good as Deflate can produce. I then remembered something I'd read about in the comp.alt.compression Usenet newsgroup a while back: the Burrows-Wheeler block transform. I searched the internet for papers on it and read quite a bit, and came to the conclusion that, although it wouldn't help for Abbrevia's Deflate implementation (it's a different type of compression altogether), it would make a good article for *Algorithms Alfresco*, especially since the transform is only part of the total compression method and other algorithms are required.

The Burrows-Wheeler Transform was first thought of in 1983 but was only described in full in May 1994 in a paper called *A Block-Sorting Lossless Data Compression Algorithm* by M Burrows and DJ Wheeler, published by Digital's System Research Center. From the title we can glean a couple of tidbits of information straight away: it operates on sorted blocks and it's lossless. Delving into the paper we learn that the transform is only part of the overall compression algorithm, but that it is the most interesting part.

Quick Recap

Before we get ahead of ourselves, let's discourse a little on compression to re-familiarize ourselves. As I stated above the zip-based compression method starts off with LZ77 compression. This is a tokenizing type of compression: it takes the input uncompressed stream of data and converts it into a stream of tokens, or symbols. The tokens consist of literals (these are uncompressed and untouched single bytes from the input stream) and length/distance pairs (an indication of a repeat of some data we've seen before). So, for example, the phrase:

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

would be converted by the LZ77 algorithm to the sequence:

```
a cat is <14,9>
```

where the `<14,9>` token should be read as 'copy the 14 bytes of data from 9 bytes back'.

May 1999's *Algorithms Alfresco* has details on how to efficiently calculate the length/distance tokens and a simple method to encode them into bytes.

The Burrows-Wheeler Transform

The point to recognize here is that LZ77 compression works by finding sequences of bytes that are repeated. It employs the general assumption that the data we compress tends to be locally similar, or, to put it another way, when data repeats, it'll generally repeat within a smallish area of the overall stream. The Burrows-Wheeler Transform, on the other hand, is an algorithm that rearranges a block of data so that similar data sequences are physically brought together (to increase the local similarity) and hence can be efficiently compressed.

Let's take a look at how it works. Suppose we have the small block of data (11 bytes):

```
la habanera
```

We now form a matrix with all the cyclic rotations of that string (I've replaced the space with an underscore so that you can more easily see what's going on). By a rotation

I mean taking the first character of a particular row and putting it at the back to form the next row:

```
la_habanera
a_habaneral
_habanerala
habanerala_
abanerala_h
banerala_ha
anerala_hab
nerala_haba
erala_haban
rala_habane
ala_habaner
```

The next step is to sort the rows in this matrix. We just use a standard lexicographic order (in other words, sorting on the binary value of each character).

```
_habanerala
a_habaneral
abanerala_h
ala_habaner
anerala_hab
banerala_ha
erala_haban
habanerala_
la_habanera
nerala_haba
rala_habane
```

During the sorting process we keep track of the original string and save its index in the sorted matrix. In this case, the index is 9: the original string occurs at row 9 in this sorted matrix. (For the purposes of illustration we'll count from 1, although in practice we'll be counting from 0.)

We now take the *last* character of each row and form a string:

```
alhrban_aae
```

This string, together with the index of the original string in the sorted matrix, is the output of the Burrows-Wheeler Transform. Notice that the output string is merely an anagram of the original one: all the same characters are there, just rearranged in a different way. But where's the compression, I can hear you say. I just implemented some mumbo jumbo on a string, got back the original string rearranged, together with a numeric value. I've actually *increased* the size of the block, not compressed it!

Quite right, and we'll get back to this issue in a moment. For now, let's show how the original matrix can be regenerated from this output, and hence how to get the original string back.

Undoing The Transform

The decoder gets the string of characters formed by the transform operation. These characters form the last characters of each row of the sorted matrix. Because of the way the original matrix was formed, the characters in this string are also all of the characters in the original string, just rearranged in some bizarre cannot-be-easily-deduced fashion.

The first thing to do is sort the characters in the passed string. If you look back at the matrix, you can see that the result will be the first column of the matrix:

```
_.........a
a.........l
a.........h
a.........r
a.........b
b.........a
e.........n
h........._
l.........a
n.........a
r.........e
```

Rotate the rows in the matrix right one place. Each row in this matrix will map onto a row in the original sorted matrix.

```
a_.........
la.........
ha.........
ra.........
ba.........
ab.........
ne.........
_h.........
al.........
an.........
er.........
```

From this operation we can immediately ascertain from the first row that *a* is followed by a blank, from the second, *l* is followed by *a*, then *h* is followed by *a*, etc. The problem is that we get four different possibilities for the letter following an *a*: a blank, a *b*, an *l*, or an *n*. In which order should we use them?

Notice that this latest operation (moving the last character to the front) will have ordered the rows according to the second character. If we look at all the rows starting with an *a* in the latter matrix, they will be in sorted order (this is obvious: they all start with *a* and are sorted from the second character onwards). The rows in the original sorted matrix that started with *a* appear in the same order as those in this incomplete matrix. We can therefore easily devise an array of indexes that describe how to map a row of the original sorted matrix (here shown by the first incomplete matrix) onto this last rotated incomplete matrix. Row 1 is mapped onto row 8, row 2 onto row 1, 3 onto 6, 4 onto 9, 5 onto 10, 6 onto 5, and so on. The full array is this:

```
8, 1, 6, 9, 10, 5, 11, 3, 2, 7, 4
```

This array can now be used to generate the original string. From the output of the Burrows-Wheeler Transform, we know the original string appeared at row 9. Using the 9th element of the decoding array, the next character is the start of row 2, *a*. From the 2nd element we go to row 1, the blank. Using element 1 we go to row 8, *h*. Element 8 then says go to row 3, *a*, Element 3 points to row 6, *b*. And so on, until we get the original string, *la habanera*.

At the end of this first section, we can use the Burrows-Wheeler Transform to take a string, convert it by a strict algorithm into an anagram and an index value, from which we can reconstruct the original string.

This all looks very bizarre to say the least, and seems to be more of a party trick than a proper compression algorithm (and, come on, where *is* the compression anyway?). Hang on a moment, though. Figure 1 shows part of the sorted matrix obtained from applying the Burrows-Wheeler

Transform to the first paragraph of this article. You can see that the last characters of each row have an awful lot of repetition: *t* followed by *t* followed by *t*. Even without thinking too hard about it, you can see that we could use a type of run-length encoding on the string formed from the last characters in order to compress the string.

The Move-To-Front Algorithm

However, we shall do a little better than this with an algorithm that was invented prior to the Burrows-Wheeler Transform, but seems to have been created just for it. The algorithm I'm talking about is the *Move-To-Front* algorithm, first described by Jon Bentley *et al* in *A locally adaptive data compression algorithm*, published in *CACM*, April 1986.

This algorithm is simple to describe and implement. Prepare a 256-element encoding array containing all the character values, so that element 0 will contain #0, element 1 #1, and so on. To encode a character using the Move-To-Front algorithm you find the character in the encoding array, output its position as a code, and then move the character to the front of the array, pushing all the others along by one.

What this does is to make sure that the characters that are used the most often appear in the front of the array. The codes emitted by the algorithm, together known as a *position vector*, will therefore tend to be the low integers. Looking at Figure 1, you can mentally calculate the codes for this section yourself:

```
x, 0, 0, 0, y, 1, 0, 0, 0, 0,
0, 0, z, 1, 0, 0, 0
```

where x, y, and z are position values that are dependent on the state of the lookup array at the time. In other words, we find the first *t* in the encoding array, output its position x, move the *t* to the front of the array. The next character is also a *t*. Its position is 0 (we just put it there, remember). The next character is also a *t*. It's position is again 0. And so on, so forth.

The decoder is just as simple. Prepare a 256-element decoding array containing all the character values, so that element 0 will contain #0, element 1 #1, and so on, just like we did when encoding. For each byte read, it uses the byte as an index into the decoder array, outputs the character there, and then moves the character to the front of the decoder array.

You can see from this simple example that the codes we output from the Move-To-Front encoder have lots of zeros and ones. This situation is where a Huffman encoder will shine: the more repetition a token has, the shorter the Huffman tree that will encode it. You'll notice that we'll get better repetition by performing Move-To-Front than just using the characters themselves. Also, when we're compressing text streams, we'll find that the lower-case characters will tend to be at the front of the encoding array leading us to imagine that using this method to compress text will result in better compression ratios than we'd maybe expect.

```
h assertions and...peppered wit
h I've been rewr...he last mont
h QA Suite, it's...s with Sleut
h Sleuth QA Suit...sessions wit
had in Abbrevia,...lgorithm we
he code is now e...succeeded: t
he compression a...Deflate is t
he current code ...oing this: t
he Deflate compr...ntation of t
he implementatio... rewriting t
he implementatio...le to tune t
he last month I'...faster.For t
here were three ...ip files.) T
his: the current... for doing t
hm used in Zip f...sion algorit
hm we had in Abb...sion algorit
hm, and I wanted...sion algorit
```

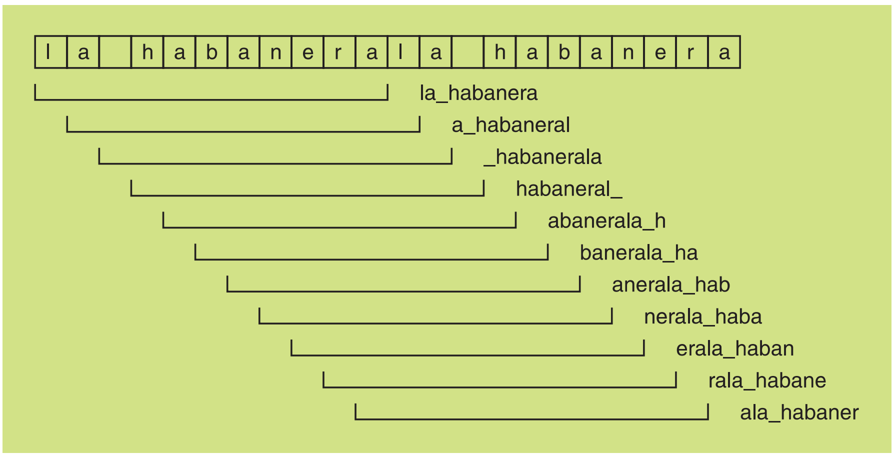
➤ *Figure 1: Part of a Burrows-Wheeler sorted matrix.*

The next stage, as I've hinted at, is to take the codes emitted by the Move-To-Front algorithm and compress them with a Huffman tree.

Implementing The Transform

After this blast of description, let's now look at coding it all. There are several distinct steps here. The first is to decide on the block size and then divide up the input stream into blocks of that size. The second is to (somehow) generate all the cyclic rotations of the block, sort them, and then peel off the last characters (this will be a block of data the same size as the original, and will be an anagram of it). This, together with the index value that describes the position of the original data in the matrix, is the output of the Burrows-Wheeler Transform. The third step is to take this output and encode it using the Move-To-Front (MTF) algorithm. Finally, in the last step, we take the output from the MTF algorithm and encode it using a Huffman tree.

For decompression, we have to decode the compressed stream using the Huffman tree into a position vector, decode this vector using the reverse of the Move-To-Front algorithm to give us the array of final characters, generate the array of initial characters, calculate the transformation vector, and then build the original block again.



➤ *Figure 2: Generating the pointers for a Burrows-Wheeler matrix.*

There's a lot to do, certainly, but at least it's in nice manageable chunks. We can write one part, test it and move onto the next.

The first step, then, is dividing up the uncompressed stream into equal-sized blocks. Easy enough: we'll use a `TStream` descendant and make the block size 16,384 bytes. The final block of the stream will be smaller in general, so we shall need to devise a method for passing the size of the block along for the decompressor, otherwise it won't know when to stop.

The next step is possibly the hardest. Conceptually, we have to create 16,384 rotations of 16,384 bytes, sort them, and then extract the final letters as another block. If we were to create the entire matrix it would be 256Mb in size: doable, just, but it smacks of the sledgehammer approach. Also the sorting would be horrendously slow, since we'd be swapping items that are 16Kb in size all the time.

No, it's better to think backwards from the sorting side of things. For efficient sorting we'd like to use a quicksort. We don't want to swap 16Kb items in doing so, so it makes sense to use an indirect quicksort: sorting pointers to the items we want sorted (an item being a rotation of the original block). That way, the items don't move, only the pointers get swapped. We could treat the block as a circular queue of characters, making the pointers to the rotations merely pointers to the individual characters, but that makes the comparisons difficult since we have to worry about wrapping around to the start of the block when we reach the end.

It seems that whenever I want to use a circular queue, I fake it by using a sliding window instead. I just hate having to worry about that wrap problem. In fact, this kind of scheme will help considerably. Take the block, double its size, copy the data from the first half over to the second half to duplicate the data. All the rotations can now be defined (without any wrapping problems) by pointers to the individual characters in the first half. I've illustrated this technique in Figure 2 by showing the different 'rotations' for *la habanera*.

We can now code up the first part of the Burrows-Wheeler compression method: the transform. To help, I grabbed the optimized quicksort from my book (you do have a copy, don't you? ☺) and modified it for my immediate needs. There was a minor problem, however: I needed a comparison method that would compare two blocks of bytes, up to some limit, and return less than, greater than, or equal. For some reason, I didn't have such a beast in my code library so I had to write one. Listing 1 shows this comparison routine: supply it with two pointers to blocks of data and it will return a negative number if the data in the first block is less than that in the second block, 0 if they're equal, or a positive number otherwise.

Listing 2 shows the Burrows-Wheeler Transform. It takes two pointers: the first is the block to which the transform is to be applied (this will eventually be supplied from a `TStream` descendant), the second is the block to which we should write the final characters. There's an integer parameter to define how big the blocks are, and the routine returns the index of the original block in the sorted 'matrix'. Since the quicksort makes no attempt to store the position of the original block in the sorted pointer array, we need to find it. Recognizing that the pointer array is sorted, we can considerably simplify the process of finding the original block by using a binary search.

➤ *Listing 1: Comparing two blocks.*

```
function CompareBlocks(aData1, aData2 : pointer; aSize : integer) : integer;
var
  Data1 : PChar;
  Data2 : PChar;
  i     : integer;
begin
  Data1 := aData1;
  Data2 := aData2;
  i := aSize;
  while (i > 0) and (Data1^ = Data2^) do begin
    dec(i);
    inc(Data1);
    inc(Data2);
  end;
  if (i = 0) then
    Result := 0
  else if (Data1^ < Data2^) then
    Result := -1
  else
    Result := +1;
end;
```

➤ *Listing 2: The Burrows-Wheeler Transform.*

```
function ApplyBWTransform(aInBlock : PChar;
  aOutBlock : PChar; aSize : integer) : integer;
var
  i : integer;
  DataBlock : PChar;
  PtrList   : PPointerList;
  TempPtr   : PChar;
begin
  {prepare for the try..finally}
  DataBlock := nil;
  PtrList := nil;
  try
    {allocate the data block and fill it with
     two copies of the input block}
    GetMem(DataBlock, aSize * 2);
    Move(aInBlock^, DataBlock^, aSize);
    Move(aInBlock^, DataBlock[aSize], aSize);
    {allocate the list of pointers and set the elements
     to the individual characters in the data block:
     these will be our rotations of the block}
    GetMem(PtrList, aSize * sizeof(pointer));
    TempPtr := DataBlock;
    for i := 0 to pred(aSize) do begin
      PtrList^[i] := TempPtr;
      inc(TempPtr);
    end;
    {sort the pointer list}
    Quicksort(PtrList, 0, pred(aSize), aSize);
    {calculate the output block}
    for i := 0 to pred(aSize) do
      aOutBlock[i] := PChar(PtrList^[i])[pred(aSize)];
    {find the original block in the list}
    Result := BinarySearch(PtrList, aSize, DataBlock);
  finally
    FreeMem(DataBlock);
    FreeMem(PtrList);
  end;
end;
```

Implementing The Compression

Now that we have the final character array, we need to encode it into a position vector by using the Move-To-Front algorithm. This doesn't take too much to implement, and Listing 3 shows the result. What's happening here is that we take the input block of final characters and then apply the MTF algorithm to produce a vector of position values. Because of the way the algorithm is structured, these position values are numbers from 0 to 255, and therefore can be stored in a byte. The MTF algorithm therefore encodes a block of bytes as another, equally sized, block of bytes and that's the way the routine is implemented.

Next is the Huffman encoding. This time I'll reuse the Huffman compression code I'd improved based on a reader's email in *Algorithms Alfresco* from August 2000 (pretty much the same code appears in the book as well). I've tweaked it a bit so that it works on a fixed size block, encoding the tree and the data to a bitstream. I won't reprint the Huffman tree code here: it's on this month's disk, after all, and to explain what's going on in it would require me to repeat an entire article.

Of course, now that we've seen the individual pieces, we have to join them up into a routine that takes an input stream and then compresses the data into another. The format of the output stream will consist of a small 4-byte signature to identify the stream as one of ours, a `longint` for the size of the original stream, a `word` value for the size of the block used, another `word` for the index of the original block in the sorted matrix after the Burrows-Wheeler Transform, and then the compressed data itself. Listing 4 shows this wrapping code.

➤ *Listing 3: The Move-To-Front algorithm.*

```
procedure MoveToFrontEncode(aInBlock : PChar; aOutBlock : PChar; aSize :
  integer);
var
  i, j, k : integer;
  Encoder : array [0..255] of char;
begin
  {initialize the encoder array}
  for i := 0 to 255 do
    Encoder[i] := char(i);
  {for all the characters in the input block...}
  for i := 0 to pred(aSize) do begin
    {find it in the encoder array}
    for j := 0 to 255 do
      if (Encoder[j] = aInBlock^) then
        Break;
    {output the position}
    aOutBlock^ := char(j);
    {move the character to the front of the encoder array}
    if (j > 0) then
      for k := j downto 1 do
        Encoder[k] := Encoder[k-1];
    Encoder[0] := aInBlock^;
    {advance the input and output pointers}
    inc(aInBlock);
    inc(aOutBlock);
  end;
end;
```

Implementing The Decompression

We are not done, of course. For our new compression code to have any usefulness at all we shall need to write a decompressor, otherwise it'll merely be a very long-winded one-way hash algorithm. So let's retrace our steps.

The first thing is to write the decompression wrapper. We know enough about the format of the input compressed stream, the way the various routines on the compression side worked, that we can write simple stubs for the Huffman decoder, the Move-To-Front decoder and the Burrows-Wheeler untransformer (have I just coined a new word?). Listing 5 shows the decompressor wrapping code. It firstly checks that the stream is one of ours by verifying the signature. It can then, with a certain amount of confidence, get the original size of the stream, the block size and the index value. It then launches into a loop where it decompresses a block, decodes it using the reverse MTF algorithm, and then applies the Burrows-Wheeler untransformer. The loop stops, of course, once all the data has been decompressed.

Like I did with the Huffman encoder, I won't go into details

➤ *Listing 4: The high-level compression routine.*

```
procedure AABWTCompress(aInStream, aOutStream : TStream);
const
  BufSize = 16*1024;
var
  InBuf     : PChar;
  BWTBuf    : PChar;
  MTFBuf    : PChar;
  BytesRead : integer;
  LongBuf   : longint;
  WordBuf   : word;
  Index     : integer;
begin
  {prepare for the try..finally}
  InBuf := nil;
  BWTBuf := nil;
  MTFBuf := nil;
  try
    {allocate the buffers}
    GetMem(InBuf, BufSize);
    GetMem(BWTBuf, BufSize);
    GetMem(MTFBuf, BufSize);
    {write the header information to the output stream}
    LongBuf := BWTSignature;
    aOutStream.WriteBuffer(LongBuf, sizeof(LongBuf));
    LongBuf := aInStream.Size;
    aOutStream.WriteBuffer(LongBuf, sizeof(LongBuf));
    WordBuf := BufSize;
    aOutStream.WriteBuffer(WordBuf, sizeof(WordBuf));
    {read the first buffer}
    BytesRead := aInStream.Read(InBuf^, BufSize);
    {while there is data to compress...}
    while (BytesRead <> 0) do begin
      {apply the BWT transform to this buffer}
      Index := ApplyBWTransform(InBuf, BWTBuf, BytesRead);
      {write the index to the output stream}
      WordBuf := Index;
      aOutStream.WriteBuffer(WordBuf, sizeof(WordBuf));
      {encode the BWT buffer with the Move-To-Front
        algorithm}
      MoveToFrontEncode(BWTBuf, MTFBuf, BytesRead);
      {compress the MTF buffer with Huffman}
      HuffmanCompressBlock(MTFBuf^, BytesRead, aOutStream);
      {read the next bufferful}
      BytesRead := aInStream.Read(InBuf^, BufSize);
    end;
  finally
    FreeMem(MTFBuf);
    FreeMem(BWTBuf);
    FreeMem(InBuf);
  end;
end;
```

```
procedure AABWTUncompress(aInStream, aOutStream : TStream);
var
  OutBuf     : PChar;
  BWTBuf     : PChar;
  MTFBuf     : PChar;
  BytesRead  : integer;
  LongBuf    : longint;
  WordBuf    : word;
  Index      : integer;
  Size       : longint;
  BufSize    : integer;
  BytesToRead : integer;
begin
  {prepare for the try..finally}
  OutBuf := nil;
  BWTBuf := nil;
  MTFBuf := nil;
  try
    {read the header information from the input stream}
    BytesRead := aInStream.Read(LongBuf, sizeof(LongBuf));
    if (BytesRead <> sizeof(LongBuf)) or
       (LongBuf <> BWTSignature) then
      raise Exception.Create('AABWTUncompress: input '+
        'stream is not a BWT compressed stream');
    aInStream.ReadBuffer(Size, sizeof(Size));
    aInStream.ReadBuffer(WordBuf, sizeof(WordBuf));
    BufSize := WordBuf;
    {allocate the buffers}
    GetMem(OutBuf, BufSize);
    GetMem(BWTBuf, BufSize);
    GetMem(MTFBuf, BufSize);
    {while there is still data to uncompress...}
    while (Size <> 0) do begin
      {read the index for the next buffer}
      aInStream.ReadBuffer(WordBuf, sizeof(WordBuf));
      Index := WordBuf;
      {read and decompress the next block}
      BytesToRead := Size;
      if (BytesToRead > BufSize) then
        BytesToRead := BufSize;
      HuffmanDecompressBlock(aInStream, MTFBuf^,
        BytesToRead);
      {decode using the Move-To-Front algorithm}
      MoveToFrontDecode(MTFBuf, BWTBuf, BytesToRead);
      {perform the reverse BWT transform}
      UnapplyBWTransform(BWTBuf, OutBuf, BytesToRead,
        Index);
      {write out the decompressed buffer}
      aOutStream.WriteBuffer(OutBuf^, BytesTORead);
      {update the loop variable}
      dec(Size, BytesTORead);
    end;
  finally
    FreeMem(MTFBuf);
    FreeMem(BWTBuf);
    FreeMem(OutBuf);
  end;
end;
```

➤ *Listing 5: The high-level decompression routine.*

about the Huffman decoder. You can easily check back with the relevant *Algorithms Alfresco* articles (get the back issues CD-ROM!) or just as easily buy my book. This month's disk has the relevant code, of course.

The implementation of the reverse of the Move-To-Front algorithm should hold no real surprises. I wrote Listing 6 very quickly by taking Listing 3 and rearranging the code a bit, not neglecting to change the word 'Encoder' to 'Decoder', of course.

Implementing The Reverse BWT

And, finally, we come to the reverse Burrows-Wheeler Transform. To remind you, we get the array of final characters and the index value of the original block in the sorted matrix and we have to regenerate that original block.

The first step is to sort the block of final characters to create the block of initial characters. What an ideal application of distribution sort from last month's *Algorithms Alfresco*. (You see: there is *some* method to my madness in choosing topics for my articles! And all along Our Esteemed Editor was under the impression I was jabbing the proverbial pin into Knuth's oeuvre on a monthly basis.) In fact, we can make a simple optimization of a distribution sort where the data for each key is the key and only the key. If you recall, distribution sort on bytes requires us to count the occurrence of each byte key value, calculate the cumulative numbers from that, and then copy the items over to an auxiliary array using the cumulative

```
procedure MoveToFrontDecode(aInBlock : PChar; aOutBlock : PChar; aSize :
  integer);
var
  i, j, k : integer;
  Decoder : array [0..255] of char;
begin
  {initialize the encoder array}
  for i := 0 to 255 do
    Decoder[i] := char(i);
  {for all the bytes in the input block...}
  for i := 0 to pred(aSize) do begin
    {output the character at that position in the decoder array}
    j := ord(aInBlock^);
    aOUtBlock^ := Decoder[j];
    {move the character to the front of the decoder array}
    if (j > 0) then
      for k := j downto 1 do
        Decoder[k] := Decoder[k-1];
    Decoder[0] := aOutBlock^;
    {advance the input and output pointers}
    inc(aInBlock);
    inc(aOutBlock);
  end;
end;
```

➤ *Listing 6: The reverse Move-To-Front algorithm.*

```
procedure DistSort(aInBlock, aOutBlock : PChar; aSize : integer;
  aStartPos : PIntArray);
var
  i, j : integer;
  Counter    : array [0..255] of longint;
  CumulCount : integer;
begin
  {clear the counter array}
  FillChar(Counter, sizeof(Counter), 0);
  {calculate the distribution of each key}
  for i := 0 to pred(aSize) do begin
    inc(Counter[ord(aInBlock^)]);
    inc(aInBlock);
  end;
  {copy over the byte values to the auxiliary list in sorted order,
   generating the start positions for each character as we go}
  CumulCount := 0;
  for i := 0 to 255 do begin
    aStartPos^[i] := CumulCount;
    inc(CumulCount, Counter[i]);
    for j := 0 to pred(Counter[i]) do begin
      aOutBlock^ := char(i);
      inc(aOutBlock);
    end;
  end;
end;
```

➤ *Listing 7: Distribution sorting a set of bytes.*

```
procedure UnapplyBWTransform(aInBlock  : PChar; aOutBlock :
PChar; aSize : integer; aIndex    : integer);
var
  i, j : integer;
  FirstCol     : PChar;
  Temp         : PChar;
  TransVector : PIntArray;
  StartPos     : PIntArray;
begin
  {prepare for the try..finally}
  FirstCol := nil;
  TransVector := nil;
  StartPos := nil;
  try
    {allocate the first column buffer and the
     transformation vector}
    GetMem(FirstCol, aSize);
    GetMem(TransVector, aSize * sizeof(integer));
    GetMem(StartPos, 256 * sizeof(integer));
    {sort the input block using distribution sort}
    DistSort(aInBlock, FirstCol, aSize, StartPos);
    {for each character in the unsorted block...}
    Temp := aInBlock;
    for i := 0 to pred(aSize) do begin
      {find the next occurrence of this character in
       the sorted block}
      j := StartPos[ord(Temp^)];
      inc(StartPos[ord(Temp^)]);
      {set the entry in the transformation vector}
      TransVector[j] := i;
      {advance to the next character in the unsorted block}
      inc(Temp);
    end;
    {we now have the transformation vector, so recreate
     the original data starting at the passed in index}
    j := aIndex;
    Temp := aOutBlock;
    for i := 0 to pred(aSize) do begin
      Temp^ := FirstCol[j];
      j := TransVector[j];
      inc(Temp);
    end;
  finally
    FreeMem(StartPos);
    FreeMem(TransVector);
    FreeMem(FirstCol);
  end;
end;
```

➤ *Listing 8: The reverse Burrows-Wheeler Transform.*

numbers and the items' keys. Well, if there are no items to copy and the item is the key, we can easily sort by regenerating the keys themselves in the destination array using the counts. If there are 42 zeros, generate 42 zeros in the destination array and then move onto the ones. It's like we are cloning the keys.

Listing 7 shows this distribution sort implementation. Because the reverse BWT algorithm is going to require an array of start positions for each character, I've added code to maintain this list, but this extra code does not obfuscate the simplified distribution sort code too much.

Now we get to the generation of the transformation vector. For each character in the unsorted block, let's say that this is indexed by *i*, we find the next equal character in the sorted block (this is where the array of start positions comes in handy), using index *j*. This simple step gives us that the *j*th entry in the transformation vector is equal to *i*. We iterate over the entire input block, building up the transformation vector.

Once we have the transformation vector, we can easily regenerate the original block starting off with the passed index. The first character is the entry in the sorted block given by the passed index. We then calculate the next index from the transformation vector and hence the next character. We continue like this until the entire original block has been calculated. Listing 8 shows the entire routine, which seems to take up less room than this explanation.

Conclusions

And so we reach the end of this article. Unfortunately, my tests with this implementation of the Burrows-Wheeler compression method show that we are not getting as good a compression ratio as Deflate does in zip files. This is entirely due to the Huffman encoder we are using (every block we compress needs a separate Huffman tree and this tree must be sent with the compressed data). Possibly an adaptive Huffman encoder, or another adaptive algorithm altogether, would be the way to go, since the tree does not have to be sent with an adaptive algorithm, but that's the stuff of another article.

On the plus side, the implementation does produce much smaller compressed files than plain Huffman on its own does. This is due to the effect of both the Burrows-Wheeler Transform and the Move-To-Front algorithm producing a very compressible intermediary stage.

Julian Bucknall can be reached at julianb@turbopower.com