

Data Compression

Double Your Disk Space for \$49.95

by Winfried Gerum



Winfried Gerum

I won't try to sell you a bargain for \$49.95 as my hypothetical "come-on" headline would suggest. But if you look in various magazines you may find similar offers—such as hair restorers for bald people. They're tempting, but do they work? Don't dismiss bargain offers too fast. In the computer world, for example, we've grown accustomed to getting ever more memory and disk space cheaply. Old programming habits did make extremely clever use of available resources. But our newer habits lead us to use these resources wastefully. And because expectations grow faster than budgets, there is a greater need to curb this wastefulness. Compression and decompression become ever more feasible options, as powerful processors and better algorithms make the additional burden negligible.

M traditionally makes extremely efficient use of disk space. But if you have to deliver software to a large number of customers, it makes a difference whether you send one or three floppy disks. Making do with just one

floppy not only results in immediate savings of some postage and some floppies, it also avoids a lot of errors in handling the stuff. In UNIX there is the compress utility, and under MS-DOS there are several utilities, e.g., PKZIP. As long as all your customers have their systems running under these same host operating systems, you are best advised to use these utilities. M just cannot compete.

Long-time programmers talking about data compression will say, "Ah, this bit-shuffling stuff is impossible to do with M." Shannon coding, Huffman coding and the like are indeed bit-shuffling stuff, which is not quite impossible to do with M but cannot be done efficiently. Earlier times of computer science saw just character-oriented compression algorithms. Now there are dictionary-oriented compression algorithms.

Suppose you assign a number to each word in a Webster's dictionary. Allowing for future enhancements, we reserve a million numbers for a million words. To code these numbers we need 20 bits (2^{20} is 1 million). This is the equivalent of 2 1/2 bytes for every word!

You don't have Webster's online? There is no need to have it, at least not for data compression's sake. The trick is to compile the dictionary as you go! This implies that it may take processing some initial amount of characters before the compression becomes

effective. But there is no need to transmit or store a dictionary, except during compression or decompression.

There are several such algorithms. Most of them can be traced back to two basic algorithms devised by Ziv and Lempel. [1,2] Here is one of them that is impossibly complicated to implement in a typical computer language. [3] But you can write down the algorithm in plain M very straightforwardly and easily. The resulting program in figure 1 is amazingly short.

The decompression algorithm is presented in figure 2.

Despite efforts to present standard M code in this publication, there is a problem when it comes to I/O. I/O seems to be mostly irrelevant to programming—at least if you take the degree of standardization as a measure of relevance.

Yes, M has standardized commands to OPEN, USE, and CLOSE files/devices. But there is no portable way to OPEN a file, processing each character until end of file. In these examples, therefore, I assume simply that the necessary devices are already open by some implementation-specific magic.

But OPEN/USE/CLOSE is not everything that causes headaches: If I want to process every character, READ VAR may be terminated by some special characters that I have to process as well. I chose to use READ *VAR. This had the desired effect of processing

```

;Input params: IN=input dev, OUT=output dev, must be OPEN
CMPR(IN,OUT) NEW C,I,IO,OFS,X,Y
+1 SET IO=$IO
+2 KILL ^UTILITY($JOB)
+3 SET ^($JOB,0)=0 FOR C=0:1:255 SET ^C=C
+4 SET X=-1 FOR DO QUIT:X=-1
+5 . USE IN SET Y=$JOB IF $DATA(^UTILITY(Y))
+6 . IF X<0 READ *X IF $$$EndOfFile() QUIT
+7 . FOR I=0:1 QUIT:$DATA(^Y,X) SET OFS=^X,Y=X READ *X QUIT:I>31 IF $$$EndOfFile SET X=-1 QUIT
+8 . IF X>-1,I<32,C<32766,$DATA(^X)) SET C=C+1,^X=C
+9 . USE OUT WRITE *OFS\256+($TEST*128),*OFS256
+10 . IF WRITE *X SET X=-2
+11 USE IO KILL ^UTILITY($JOB)
+12 QUIT

```

Figure 1. The compression algorithm.

```

;Input params: IN=input dev, OUT=output dev, must be OPEN
DCMPR(IN,OUT) NEW C,I,O,OFS,X,Y,Z
+1 SET IO=$IO
+2 KILL ^UTILITY($JOB)
+3 SET ^($JOB,0)=0 FOR C=0:1:255 SET ^C=$CHAR(C)
+4 FOR DO QUIT:X<0
+5 . USE IN READ *X IF $$$EndOfFile() SET X=-1 QUIT
+6 . READ *Y IF X>127 SET X=X-128 READ *Z
+7 . USE OUT SET OFS=X*256+Y WRITE ^OFS)
+8 . IF WRITE *Z SET C=C+1,^C=^OFS)_CHAR(Z)
+9 USE IO KILL ^UTILITY($JOB)
+10 QUIT

```

Figure 2. The decompression algorithm.

each character as it is. But you should have a look at the standard. There you read that the implementor may define the values returned by READ *lvn in a device-dependent manner. This translates to "do not use this in portable programs." Fortunately, the above code works under all M implementations to which I have access. Testing for the end-of-file condition is also not possible in a standardized way. So please replace \$\$\$EndOfFile() with any code, testing for this condition in your M implementation. (But don't use globals in this test!)

Let's have a look at the code: In the line CMPR (DCMPR), all variables internal to the procedure are NEWed. In the line CMPR+1 (DCMPR+1), the current device is saved (what about NEW\$IO?).

CMPR+2 clears a scratch global.

CMPR+3 preloads the dictionary with characters 0 through 255. The dic-

tionary in compression and decompression has the same contents, but we use a simpler structure in the decompression procedure. During compression, strings are held on the index level, with the \$ASCII() value of each character treated as a separate index. The dictionary codes are held on the data level. During decompression, dictionary codes are held on the index level and the complete coded strings are held on the data level.

CMPR+5 sets up the naked indicator.

CMPR+6 reads a character, unless we have a surplus character from previous processing.

CMPR+7 scans the dictionary built up so far. If the new character has a dictionary entry, we read an additional character. Note that the first character we try in this loop always has a match in the dictionary due to preloading. Using naked reference with two sub-

scripts changes the value of the naked indicator, which is essential for this procedure to work. There is no need to store all the characters. The information is implicit in the naked indicator!

To avoid getting global references that are too long, there is a limit of thirty-two in the loop. On older systems you have to adjust that to lower values (same change in following line!). The normal termination of the loop is when the longest entry in the dictionary matching the current entry has been found (code in OFS).

CMPR+8 Normally each successful match of a string results in the definition of an additional entry into the dictionary. The exceptions are when there are no more characters (EndOfFile), or when the FOR is terminated by the length criterion, and, of

course, when we cannot make more entries into a full dictionary (2^{15} entries = 32768).

CMPR+9 The dictionary code is written as two bytes. The high bit of the high byte indicates whether a plain character follows.


CMPR+11 (DCMPR+9) Clean-up.

The decompression is somewhat simpler and faster.

DCMPR+5, and +6 Two bytes of the dictionary code are read. If the high bit of the first byte is set, a third character is read.

Figure 3 shows some examples of compression at work. In the first line, small.txt are the first 256 chars of this article. In the second line, naked.txt is a file with this article and the one which appeared in the November 1993 issue of *M Computing* as ASCII files. UMSDA.DAT is a file with fixed-length records, supplied by the German postal service for conversion to the new ZIP codes. CMPR is this algorithm, compress is the UNIX compress algorithm. File sizes are given in bytes. Compression ratio is shown in percentage $(1 - (\text{CompressedSize} / \text{UncompressedSize})) * 100$.

The second direction is experimenting with strategies of how to proceed when the dictionary is full. If the nature of the processed data changes at some point, it makes sense to throw away the dictionary if the change makes compression rates deteriorate. That can be done easily, but it is beyond the scope of this article.

There are many more ways to do data compression under M. I hope that my demonstration of this one example encourages many of you to experiment with this topic. It is possible to beat UNIX compress with an M-based solution! Meet the challenge! 

file	uncompressed	CMPR (M)	compress (UNIX)
small.txt	256	341 (-33.20%)	222 (13.28%)
naked.txt	18,883	14,970 (20.72%)	9,880 (47.67%)
UMSDA.DAT	9,110,247	2,177,467 (76.10%)	1,577,894 (82.68%)

Figure 3. Examples of compression at work.

DCMPR+7 writes the coded string from the dictionary.

DCMPR+8 If there has been a third character in the previous line, write it and define a new dictionary entry.

This algorithm is as portable as M software using I/O can be. It is easy to implement. If there are alternates in UNIX/DOS that can be used, just use them; they are probably faster.

The algorithm does not compress small files. It may even make very small files larger. This is because the first characters (8-bit) are coded by dictionary entries (15 plus 1-bit). But when the dictionary has built up somewhat, compression works the way you would expect.

On small files the performance of CMPR is ugly: A negative compression ratio means that the file is in fact increased. The larger a file, the better the performance. On megabyte files it approaches the compression ratios of UNIX compress.

The algorithm can be modified in two directions. First, use only N bits to code dictionary entries in the compressed file, as long as there are fewer than 2^{N+1} entries in the dictionary. This lowers the compression threshold and makes compression ratios better. That involves some bit operations, which are unusual, albeit not impossible, under M.

Winfried Gerum is president of Winner Software GmbH in Erlangen, Germany. His column appears regularly in *M Computing*. A similar article has appeared in the MTA-Europe newsletter.

Endnotes

1. J. Ziv and A. Lempel, "A Universal Algorithm for Sequential Data Compression," *IEEE Transactions on Information Theory* 23:3 (May 1977): 337-343.
2. J. Ziv and A. Lempel, "Compression of Individual Sequences via Variable-rate Coding," *IEEE Transactions on Information Theory* 24:5 (September 1978): 530-536.
3. Ziv and Lempel (1978).