A Simple Dictionary-Based Compression Algorithm

By Robert Jaskuła (`robak_1983@o2.pl`), September 2007.

A simple, two-pass dictionary-based compression method is described. The method replaces the original data symbols with two-part tokens of the form (length, code). The length field is a prefix code (1 or 3 bits long) that specifies the length of the code field. The method is simple but not very efficient and its only "claim to fame" is its elegant encoding.

The first pass counts the number of occurrences of each data symbol and the second pass converts each symbol to a token. In-between the passes, the data symbols are sorted in descending order of their frequencies of occurrence and are written on the output in that order.

The first pass employs a 2-row table. The data symbols are stored in row `SYMB` and their occurrences are counted and stored in row `FREQ`. Thus, if the data symbols are bytes (they can be anything else), the table will have 256 columns and row `SYMB` will be initialized to the integers 0 through 255.

Example. If the input file is `ihggfffeeeeddddccccccbbbbbbbaaaaaaaa`, then the first pass generates table 1

`SYMB`	0	1	...	97	98	99	100	101	102	103	104	105	106	...	255
`FREQ`	0	0		8	7	6	5	4	3	2	1	1	0		0

which, after sorting, becomes table 2

`Index`	0	1	2	3	4	5	6	7	8	9	...
`SYMB`	97	98	99	100	101	102	103	104	105	...	
`FREQ`	8	7	6	5	4	3	2	1	1	0	...

(followed by symbols 0 through 96 and 106 through 255, all with 0 frequencies). The upper row of table 2 (row `SYMB`) is now written, as a string of 256 bytes, on the output, for the use of the decoder. This row is the dictionary of our method.

The second pass reads the input file and replaces its data symbols with tokens according to the following table 3

	Token	
Index	length	code
0 to 7	000	000 to 111
8 to 23	001	0000 to 1111
24 to 55	010	00000 to 11111
56 to 127	011	0000000 to 1111111
128 to 255	1	0000000 to 1111111

Thus, the symbol 104 that's located at index 7 of table 2 becomes the token (000, 111) and the symbol at table 2 index 8 (105) is converted to the pair (001, 0000). It is clear that common symbols become short tokens, and it is also obvious that decoding is easy. The decoder starts by reading the first 256 bytes (the dictionary) from the compressed file and storing them in a table D. It then reads tokens from the compressed file, computes an index from each token, and uses the index to locate the corresponding data symbol in D.

Decoding is unique because the lengths listed in table 3 are prefix codes. When the decoder reads 001, it knows that this corresponds to a 4-bit symbol code, so it reads the next four bits and adds 8 in order to obtain an index to D. If the next four bits are 0000, adding 8 results in 8, and location 8 in D contains byte 105.

This also explains how the encoder works. It reads the next data symbol and encodes it by finding it in table 2 and converting its index to a token in one or two steps. If the index is between 0 and 7, it becomes the symbol code (the second field of the token) and it is preceded by three zeros. Thus, the token is simply the index, written on the output as a 6-bit integer. If the index is between 8 and 23, the encoder generates a 7-bit token by subtracting 8 from the index. If the index is between 24 and 55, the encoder generates an 8-bit token by subtracting 24 from the index. If the index is between 56 and 127, the encoder generates a 10-bit token by subtracting 56 from the index. Finally, If the index is between 128 and 255, the encoder generates the 8-bit token as the index itself. This way of encoding is fast and elegant and is the chief "selling point" of this algorithm.

This method is simple, but not very efficient. The worst case is where the data symbols occur with the same frequency F (a flat distribution). If the symbols are bytes, the size of the original file is $8\times256\times F = 2048F$ bits. Eight of those symbols will be encoded in 6 bits, 16 will be encoded in 7 bits, and so on, for a total size of

$$(6\times8 + 7\times16 + 8\times32 + 10\times72 + 8\times128)F = 2160F \text{ bits.}$$

The compression ratio is therefore $2160/2048 \approx 1.05$; a slight expansion!

In the best case, the original file consists of occurrences of only eight different bytes, so each is encoded in 6 bits. Each byte is converted from 8 to 6 bits, for a compression ratio of 0.75, not very impressive.

This method is an improvement of the simple algorithm by Ismail Mohamed that's described in section 3.2 of *Data Compression: The Complete Reference* (3rd and 4th editions).

Robert Jaskuła (`robak_1983@o2.pl`), Gryfino, Poland, September 2007.

There are a thousand thoughts lying within a man that
he does not know until he takes up a pen to write.
—William Makepeace Thackeray