Available at
www.ElsevierMathematics.com
POWERED BY SCIENCE @ DIRECT•

An International Journal
computers & mathematics
with applications

ELSEVIER Computers and Mathematics with Applications 46 (2003) 1581–1587
www.elsevier.com/locate/camwa

Block Huffman Coding

M. ABDUL MANNAN AND M. KAYKOBAD
Department of Computer Science and Engineering
Bangladesh University of Engineering and Technology (BUET)
Dhaka-1000, Bangladesh
mannan_74@hotmail.com
kaykobad@cse.buet.ac.bd

(Received January 2001; revised and accepted December 2002)

Abstract—Dynamic or adaptive Huffman coding, proposed by Gallager [1] and extended by Knuth [2], can be used for compressing a continuous stream. Our proposal for accomplishing the same task is termed here as block Huffman coding. This is an easy and simple solution to compress continuous data by applying simple Huffman coding in blocks of data. For each block, a different header is stored. This header is shipped with each block of compressed data. However, to keep the header overhead low, we have used the proposed storage efficient header [3]. © 2003 Elsevier Ltd. All rights reserved.

1. INTRODUCTION

Huffman coding is not the best coding method now, but it may be the most-cited coding method until today. Huffman published his paper on coding in 1952 [4], and it instantly became the most imperative work in information theory. Huffman's original work spawned many variations. And it dominated the world of coding until the early 1980s.

But there are some practical problems in using original Huffman coding. One of the most prominent problems is that we have to read the whole stream prior to coding. This is a major problem when

(i) the file size is too large;
(ii) the source stream is continuous.

When the file size is large it will take much a longer time to build the Huffman header for compression. This happens because we have to read the whole file twice from the source stream, that is, in most cases hard disks. In the first read pass, we have to build the Huffman tree to build codes for individual character. In the second read pass, we do the real work of compression. If the file size is small enough to be stored in the main memory, reading in the second pass can be done from the main memory instead of the hard disk. This will reduce the overhead time of reading from a slow speed device twice. But when the file size is large, we cannot store it in the main memory. And thus, we are unable to avoid the second time reading from the hard disk drive. With the original method of Huffman coding, there is no way to avoid it.

When the source stream is continuous, the original Huffman coding is simply inapplicable. Because without knowledge of the total stream, we cannot build the Huffman tree to compress the input stream. However, there is a remedy for this problem, which is known as dynamic Huffman coding [1,2].

Given nonnegative weights $(w_1 \cdots w_n)$, the Huffman algorithm can be used to construct a binary tree with n external nodes and $n-1$ internal nodes, where the external nodes are labeled with weights $(w_1 \cdots w_n)$ in some order. Huffman's tree has the minimum value of $(w_1 l_1 \cdots w_n l_n)$ over all such binary trees, where l_i is the level at which w_i occurs in the tree. But in the case of a continuous string, the weights w_i are not given. In this case, encoding of the i^{th} character c_i is based on a Huffman tree for the frequencies of the previously encoded portion $c_1 c_2 \cdots c_{i-1}$. The encoding process thereby learns the frequencies of the encoded string as it proceeds. The decoding process also learns how the code is evolving in exactly the same way as the encoding process does. Thus, by continually updating a Huffman code, both sender and receiver of a continuous stream can keep synchronized with each other.

This outline of dynamic Huffman coding was described in [2]. However, this principle of adaptive Huffman coding was discovered by Gallager [1].

The problem of this method is that it simply hampers the original simplicity of Huffman coding and puts some overhead of rearranging the code tree every time the sender sends a character and a receiver receives a character. However, the most common case of continuous stream—the internet—sends data chunk by chunk, rather than character by character. Keeping in mind these things, we have proposed an alternative method of Huffman coding for a continuous stream, which is termed block Huffman coding.

2. PROPOSED ALTERNATIVE METHOD

Our proposed method is pretty simple. The main idea is to break the input stream into blocks and compress each block separately. We choose block size in such a way that we can store one full single block in main memory. Our proposed method is termed block Huffman coding. We have used BHC for block Huffman coding and PHC for pure Huffman coding. The proposed methods for static and continuous data are outlined below.

Algorithm for Static Data

1. Read a block from the stream into the main memory.
2. Build the Huffman tree and code for this block.
3. Compress this block by reading it from the main memory.
4. Put the header and compressed data to the output stream.
5. If there is more data in the input stream, go to Step 1. Otherwise coding is ended.

How It Solves Our Problem

This method can handle both the drawbacks mentioned in the previous section. In the first case, as we are reading the file from the hard disk only once, compression speed increases significantly, because the second pass reading is done from the main memory that is much faster than the hard disk. Now the file size may be as large as we can imagine without suffering double penalties for reading two times from the hard disk drive. So our first problem is practically solved.

Algorithm for the Continuous Stream

Now let us consider the second shortcoming. In information transfer in a network environment, we have to face a continuous stream quite often. For this case, we can modify the above idea in the following way.

1. Read data from the stream into the main memory.
2. If the block is not completed, then go to Step 1.
3. Build the Huffman tree and code for this block.

4. Compress this block by reading it from the main memory.
5. Put the header and compressed data to the output stream.
6. Go to Step 1.

However, during Steps 3–5, we have to store the incoming data in parallel mode. In this way, we can solve the problem in the case of a continuous stream.

Multiple Header Storage

If we investigate our above methods, it may seem that there is a potential problem of increase in size of compressed data due to the storage space of a multiple header for a single file. The problem is shown pictorially in Figure 1.

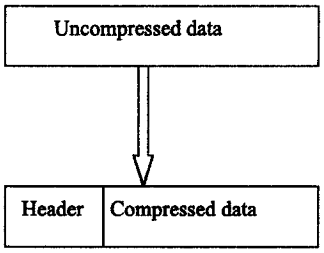

Figure 1. Original method.

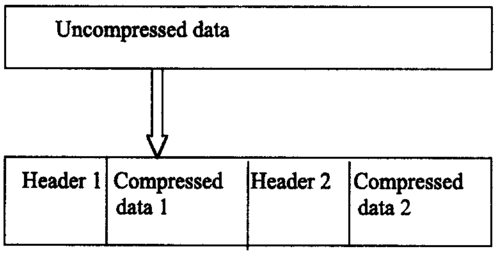

Figure 2. Proposed method.

As the number of data blocks increases, the overhead for storing multiple headers becomes significant. This causes penalty in compression ratio. That is why we have redesigned the storage method of headers. Here we cannot reduce number of blocks much. But if we can reduce the size of the header, the overhead may not be that significant. Actually, we did just that. The size of the proposed method of header structure takes much less space than the common method of storing a header. This is discussed in a separate section.

Block Size

Another thing we have to consider in this proposed method is the block size. The main limiting factor here is the size of the usable main memory. If we take block size to be as small as 1 Kbyte, there would be a large number of blocks. However, this will incur much less memory overhead. But as for each block, we have to store a header, the storage overhead for headers is significant. This decreases with the increasing size of the block. But increase in block size must put up with the usable size of physical memory. We can use a block size as moderate as 5 Kbytes, 10 Kbytes, or 12 Kbytes.

Locality

In original Huffman coding, the code tree is built after reading the whole file. The algorithm assumes that the probability of every character is almost the same in the entire file. Hence, the

Table 1. Compression ratio in pure and block Huffman coding.

File Name	Size in Bytes	Compression Ratio in PHC (%)	Compression Ratio in BHC (%)
CDPLAYER.EXE	106496	32.83	36.44
D_COMP2.EXE	123609	67.67	68.95
D5.EXE	37251	41.63	42.75
D7.EXE	41145	41.53	43.13
DIALER.EXE	63240	16.59	19.22
EMM386.EXE	125495	39.34	42.57
AUXFUNCS.BAK	52852	37.02	37.17
01.BMP	118448	84.66	84.60
02.BMP	66704	83.61	83.47
03.BMP	733216	85.96	85.84
04.BMP	307514	37.02	42.45
05.BMP	66146	20.27	25.59
CC.LIB	280064	18.44	18.73
DATA.DOC	106496	55.16	54.92
HELP.EXE	35541	17.53	20.24
JVIEW.EXE	169232	45.53	48.44
NET.EXE	356134	23.14	27.27
VISIO32.EXE	903680	52.61	63.63
PAINT.C	50900	36.89	37.61
SH1.JPG	137509	8.74	7.84
GRAPHICS.LIB	29263	16.85	16.32
FE.HTM	14569	36.77	36.41
FINAL.DOC	401408	48.28	54.52
ISD1.DOC	61440	53.49	57.71
JURAIN.DOC	35328	67.79	67.80
NFPE.DOC	349184	39.00	48.41
TEST.TXT	46186	39.55	40.92

code tree is built for global data. But in practical cases, we have seen that characters are not scattered randomly in the entire file. For example, the ASCII character zero (null character) is found in groups as large as 10 Kbytes in some executable (.exe) files. This type of locality is also prominently present in bitmap (.bmp) files. In fact, it is almost always a better idea to compress a data block with the local code tree for that block. And this consideration of locality is an inherent feature of our proposed method. So we can expect that our proposed method will perform better than the original method.

Finding the Middle Ground

If we increase the block size, we can minimize the overhead of storing multiple headers. However, by taking a small size block, we can acquire greater advantage of locality feature. So we have to find the middle ground. This may vary from file to file as it is related to inherent characteristics of the input file.

The Program

We have implemented the program in C language. In the program, we have also shown the overhead incurred by each block. To increase the speed of compression, no dynamic memory allocation is used. All operations are implemented with array. Our findings for the program are given in the following section.

The Results

Results of our findings are listed in the following tables. In Table 1, we have compared the results from original Huffman coding and block Huffman coding. Here we have used a block size of 10 Kbytes.

In Table 2, we have shown the results of the block Huffman coding method in different block sizes. We have also listed the overhead caused by storing multiple headers.

Table 2. Table listing compression and overhead ratio for different block sizes.

File Name	Original Size (Bytes)	Compression Ratio (%) (for Block Size)			Overhead Ratio (%) (for Block Size)		
		5 Kbytes	8 Kbytes	12 Kbytes	5 Kbytes	8 Kbytes	12 Kbytes
CDPLAYER.EXE	106496	36.83	37.26	37.12	4.92	3.42	2.56
D_COMP2.EXE	123609	68.57	68.71	68.85	1.97	1.36	0.96
D5.EXE	37251	43.88	44.06	43.57	4.73	3.32	2.72
D7.EXE	41145	43.30	43.91	43.38	4.55	3.19	3.06
DIALER.EXE	63240	18.35	18.92	19.79	5.80	3.85	2.69
EMM386.EXE	125495	42.15	42.07	41.87	4.58	3.48	2.44
HELP.EXE	35541	19.98	21.98	19.77	6.07	3.86	2.74
JVIEW.EXE	169232	47.93	48.38	47.88	3.66	2.64	1.92
NET.EXE	356134	26.26	26.97	27.10	5.02	3.40	2.35
VISIO32.EXE	903680	63.92	63.75	63.21	1.85	1.33	1.05
FE.HTM	14569	37.05	36.70	36.23	2.04	1.48	1.48
01.BMP	118448	84.44	84.56	84.61	0.62	0.47	0.39
02.BMP	66704	83.34	83.38	83.53	0.95	0.79	0.57
03.BMP	733216	85.71	85.80	85.86	0.40	0.28	0.21
04.BMP	307514	42.26	42.43	42.33	1.35	0.88	0.62
05.BMP	66146	25.73	25.59	25.37	2.77	1.94	1.35
PAINT.C	50900	36.79	37.26	37.40	1.99	1.40	1.02
SH1.JPG	137509	5.90	7.15	7.94	6.27	4.07	2.65
CC.LIB	280064	17.11	18.33	18.89	5.47	3.73	2.57
GRAPHICS.LIB	29263	16.09	16.75	17.07	6.36	4.39	3.29
AUXFUNCS.BAK	52852	37.07	37.05	37.18	1.97	1.36	1.01
TEST.TXT	46186	40.75	41.02	40.64	1.83	1.27	0.89
EKUSHEY.DOC	27136	59.71	59.69	59.05	3.41	2.69	2.28
FINAL.DOC	401408	54.27	54.66	53.98	3.34	2.32	1.72
ISD1.DOC	61440	58.18	58.39	57.58	3.54	2.82	1.99
JURAIN.DOC	35328	67.54	67.49	67.77	3.95	3.25	2.31
NFPE.DOC	349184	47.71	48.24	48.35	4.28	2.93	2.13
DATA.DOC	106496	54.44	54.66	54.85	4.94	3.48	2.60

Discussion on Results in PHC and BHC

We have said earlier that our proposed block Huffman coding is advantageous over pure Huffman coding when we consider a large file and continuous stream. From the results of Table 1 and Figure 3, we have found another criteria of block Huffman coding—that is, in some cases it has a better compression ratio than original Huffman coding. This better efficiency in the compression ratio is the outcome of locality characteristics of the proposed method as it compresses locally rather than globally. Though the difference in compression ratio is insignificant in many cases, BHC is advantageous over PHC in terms of reading time from secondary storage.

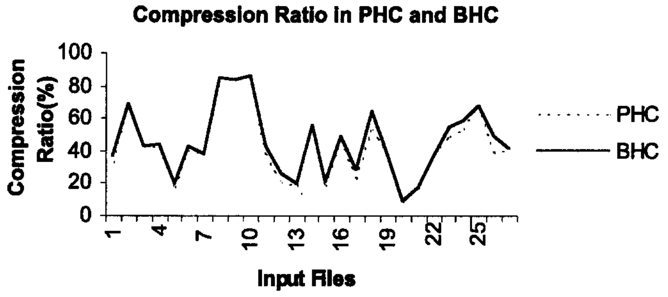

Figure 3. Compression ratio of PHC and BHC.

Discussion on Results for Different Block Sizes

From the results of Table 2 and Figure 4, it is clear that the compression ratio has not changed significantly with the increase of block size. Even in some cases, the compression ratio deteriorates with increasing block size. The reason may be attributed to locality criteria as we have discussed earlier. The more we have taken the block size, the less we have taken the advantages of proximity characteristics of the input stream.

Another comparison we have made in Table 2 and Figure 5 is among overhead ratios in different block sizes. The overhead mainly consists of the bitmap representation of the code tree and the sorted character list for every block. The overhead ratio has decreased as we have increased the block size. With the increase of block size, we have reduced the total number of blocks to compress. As with every block, there is a header to store, so the reduction in number of blocks has always caused reduction of the overhead ratio.

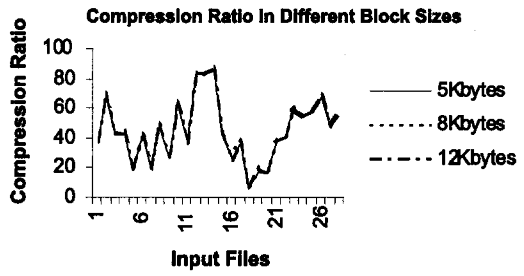

Figure 4. Compression ratios in different block sizes.

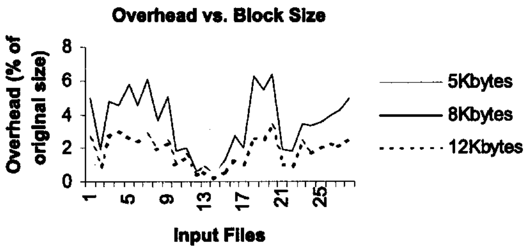

Figure 5. Overhead ratio in different block sizes.

3. CONCLUSIONS

So we can conclude from the discussion above that to obtain better efficiency from our proposed block Huffman coding, a moderate sized block is better. Another thing we may notice is that the block size does not depend on file types. So we can use this proposed method as a general method of coding.

REFERENCES

1. R.G. Gallager, Variations on a theme by Huffman, *IEEE Trans. Inform. Theory* **IT-24**, 668–674, (1978).
2. D.E. Knuth, Dynamic Huffman coding, *Journal of Algorithms* **6**, 163–180, (1985).
3. M.A. Mannan, R.A. Chowdhury and M. Kaykobad, A storage efficient header for Huffman coding, *Proceedings ICCIT*, 57–59, (2001).
4. D.A. Huffman, A method for the construction of minimum-redundancy codes, *Proceedings of the IRE* **40** (9). 1098–1101, (September 1952).