Analysis of Cascading Compression Algorithms

Gilbert H. Young and Kwok-Shing Cheng
Department of Computing
The Hong Kong Polytechnic University
Hung Hom, Kowloon
Hong Kong
csyoung@comp.polyu.edu.hk
csscheng@comp.polyu.edu.hk

ABSTRACT. Huffman coding scheme is a character-based algorithm in which every leaf node represents a character only. In this paper, we study three variations of the Huffman coding scheme for compressing 16-bit Chinese language. Although it is observed IDC can generate the shortest code length among the three variations, but its empirical compression ratio is below 1.8, which is unsatisfactory. In order to achieve higher compression performance, i.e., compression ratio over 2, word-based compression algorithms should be employed. A possible way to develop word-based algorithms is to use the technique of cascading. Two kinds of algorithms are chosen for cascading. They are LZ algorithms and the Huffman coding scheme. LZ algorithms are used for finding repeating phrases while the Huffman coding scheme is used for encoding the phrases instead of characters. The experimental results show that the cascading algorithm of LZSSPDC outperforms a famous UNIX cascading compressor GZIP by 5% on average.

1 Introduction

Since the compression results for 16-bit languages are not as good as English when the 8-bit character-based Huffman coding scheme [Huf52] is applied, some variations of the algorithm are developed in order to model 16-bit text better [CY97]. In this paper, we study three representative variations of the Huffman coding scheme for 16-bit languages. They are 16-bit Huffman coding scheme (16Huff) [CY97], predictive data coding scheme (PDC) [HT90] and indicator dependent Huffman coding scheme (IDC) [YC99].

For practical text compression, character-based Huffman coding scheme is seldom used because the compression results are unsatisfactory (see section 3). It is expected that word-based compression algorithms can achieve higher compression performance [HC92, CWY99]. Therefore, in the paper, LZ algorithms [Bel86, Wel84] are employed to find repeating phrases and three variations of Huffman coding scheme are applied to encode the phrases respectively. The compression results of the three cascading algorithms are provided and analyzed.

The rest of the paper is organized as follows. Section 2 introduces the measurement metric and benchmark files. Section 3 describes the compression performance of the three variations of Huffman coding scheme. The cascading models of LZSS and LZW with the three Huffman coding schemes are analyzed in Sections 4 and 5, respectively. Finally, Section 6 discusses the conclusions.

2 Measurement Metric and Benchmark Files

In this paper, *compression ratio* is used as the measurement for the performance of a compression algorithm. It is defined as:

$$C.R. = \frac{f_{s_o}}{f_{s_c}}$$

where C.R. is the compression ratio, f_{s_o} is the original file size and f_{s_c} is the compressed file size. In the measurement, a larger compression ratio means that a greater portion of the original file is reduced. In other words, better compression performance is obtained.

In our study, all programs are run on a Pentium 90MHz computer with 16M RAM. The operating system is Linux 2.0.1. Note that all agorithms presented in this paper can easily be applied to any 16-bit languages, but Chinese is chosen for testing because it is a 16-bit and widely-used language. We collect eight Chinese text files, namely $c1.gb, c2.gb, \ldots, c8.gb$. They are GB-coded and their sizes range from 236,321 to 15,619,863 bytes. The files are with different types of content. Table 1 summarizes the detailed information of the files.

Name	Size (in byte)	Types of content
c1.gb	236,321	Chinese Classical Issues and Essays
c2.gb	318,591	Chinese Classical Poetry
c3.gb	844,461	Chinese Modern Poetry
c4.gb	742,151	Chinese Miscellaneous
c5.gb	2,318,134	Chinese Classical Novels
c6.gb	2,475,726	Chinese Modern Novels
c7.gb	15,619,863	Chinese Electronic Magazine
c8.gb	8,163,166	Chinese Newspaper

Table 1: Details of the benchmark files

3 Variations of Huffman Coding Scheme

Three variations of Huffman coding scheme are presented in this section. They are 16-bit Huffman coding scheme (16Huff) [CY97], predictive data coding scheme (PDC) [HT90] and indicator dependent Huffman coding scheme (IDC) [YC99].

Name	16Huff	PDC	IDC (m=5,n=2)
c1.gb	1.57	1.58	1.71
c2.gb	1.36	1.37	1.47
c3.gb	1.67	1.67	1.83
c4.gb	1.54	1.54	1.60
c5.gb	1.72	1.72	1.76
c6.gb	1.67	1.67	1.71
c7.gb	1.69	1.68	1.71
c8.gb	1.68	1.68	1.74

Table 2: Compression ratios of 16Huff, PDC and IDC

16Huff can be classified as a single-tree algorithm while IDC is a double-tree algorithm, and PDC is a multi-tree algorithm. The compression ratios of the algorithms over the benchmark files are shown in Table 2. In the table, it is observed that the compression ratios obtained by IDC is highest since it uses correct 16-bit sampling technique and a tree-splitting method. Another observation is that PDC and 16Huff achieve similar compression ratios. For smaller files such as *c1.gb*, ... *c4.gb*, PDC compresses better than 16Huff. For larger files such as *c5.gb*, ... *c8.gb*, 16Huff outperforms PDC. If we simply focus on the Huffman code size without considering the size of frequency table (header size), 16Huff is better than PDC over all the benchmark files since 16Huff use a correct 16-bit sampling technique for Chinese while PDC still uses a 8-bit sampling technique. The code sizes and header sizes of the three algorithms are shown in Table 3.

File Name	16Huff		PDC		IDC	
	Code	Header	Code	Header	Code	Header
c1.gb	133,835	16,662	134,758	14,673	132,020	6,552
c2.gb	209,755	24,746	210,467	21,398	207,685	9,308
c3.gb	473,704	32,167	478,958	27,828	449,254	12,077
c4.gb	457,957	25,336	459,171	21,883	452,703	10,768
c5.gb	1,316,948	29,359	1,324,350	25,248	1,301,583	13,542
c6.gb	1,448,577	32,466	1,455,441	27,868	1,433,878	14,676
c7.gb	9,232,373	34,687	9,260,499	29,853	9,094,897	20,208
c8.gb	4,820,945	28,375	4,839,093	24,398	4,670,774	14,299

Table 3: Code sizes and header sizes of 16Huff, PDC and IDC

The two compression algorithms, 16Huff and PDC outperform each other for different ranges of input file size. Although 16Huff achieves smaller sizes of Huffman codes, it is not suitable for compressing small files because it would generate relatively larger header than PDC. In general, a header consists of the count of distinct characters, the distinct characters and their frequencies. Assume there are d distinct 16-bit characters with f distinct first bytes where $d \geq f$. In 16Huff, the header size is equal to $(2+2d+2d) = (4d+2)$ bytes. In PDC, the header size is equal to $(f+1)+(f+d)+(2f+2d) = (3d+4f+1)$ bytes. If the header size of PDC is not greater than that of 16Huff, the following inequality is observed:

$$\begin{aligned} 4d+2 &\geq 3d+4f+1 \\ \frac{d+1}{4} &\geq f \end{aligned} \tag{1}$$

Table 4 shows the values of d and f in the benchmark files. Since the inequality 1 can be satisfied for all benchmark files, so the header size in PDC should be smaller than that in 16Huff. The gain in the header size of PDC would cover the loss in the Huffman code size for smaller files. Thus, PDC should be used for compressing smaller files, and 16Huff should be used for compressing larger files.

File Name	Number of distinct 16-bit characters (d)	Number of distinct first bytes (f)	Is inequality 1 satisfied?
c1.gb	2,775	170	Yes
c2.gb	4,123	175	Yes
c3.gb	5,394	159	Yes
c4.gb	4,221	175	Yes
c5.gb	4,891	160	Yes
c6.gb	5,409	177	Yes
c7.gb	5,810	191	Yes
c8.gb	4,727	77	Yes

Table 4: The values of d and f in the benchmark files

Empirically, IDC is superior than the others because it obtains relatively higher compression ratios than the other two variations of Huffman coding scheme. Through the use of indicator splitting technique and the header reduction technique, both its code size and header size are smaller than 16Huff and PDC.

3.1 Time and Memory Consumption of PDC, 16Huff and IDC

In this part, PDC, 16Huff and IDC are chosen for empirical evaluation. We measure the time and memory usage consumed by each of the three algorithms. The information is listed in Table 5. In the table, there are two values in each entry. The value before a slash (/) is measured for compression while the value after the slash is measured for decompression. For example, under the columns of time usage, the first value is the compression time while the second value is the decompression time. Similarly, under the column of memory usage, the first value is the consumption of memory for compression while the second value is the consumption of memory for decompression. The units of time and memory are second (sec) and kilobyte (Kb) respectively.

Although the three algorithms share the same time complexity of $O(n \log n)$ [CLR90], their empirical performances are different. In Table 5, 16Huff uses less time than PDC and IDC because fewer operations such as comparisons and calculations are performed in 16Huff. In PDC, a structure of two dimensional array is used to store the first byte and second byte of a Chinese character. As there are multiple Huffman trees used in the scheme, more time is spent on computing the correct locations of the nodes for updating the frequencies. In IDC, an extra pass for the input text is used for finding indicators, and an extra comparison is taken to check whether each input character is an indicator. All these additional operations make PDC and IDC consume more time than 16Huff.

It is known that the memory consumption is proportional to the potential size of the Huffman trees. 16Huff uses only one tree for encoding, so the program size is smallest. PDC reserves one Huffman tree for the first bytes and a second-byte Huffman tree is built for each first byte. Since the frequencies of first bytes and second bytes are both kept, more memory is consumed. Similarly, IDC uses two Huffman trees to store 16-bit characters with duplication. It would consume more memory than 16Huff.

Name	PDC		16Huff		IDC	
	Time (sec)	Memory (Kb)	Time (sec)	Memory (Kb)	Time (sec)	Memory (Kb)
c1.gb	1.5/1.0	1,088/1,020	1.4/0.9	816/564	2.1/1.1	1,160/612
c2.gb	2.5/1.7	1,168/1,068	2.2/1.4	868/616	3.2/1.7	1,244/696
c3.gb	5.6/3.8	1,204/1,104	5.0/3.2	920/664	7.0/3.9	1,312/764
c4.gb	5.4/3.6	1,164/1,068	4.6/3.0	872/620	5.5/3.7	1,264/720
c5.gb	16.0/10.6	1,168/1,072	14.2/8.9	900/644	19.0/10.9	1,316/768
c6.gb	17.2/11.4	1,200/1,100	15.3/9.8	920/664	20.8/12.2	1,344/800
c7.gb	107.9/71.1	1,176/1,100	96.3/62.0	936/680	138.1/76.8	1,404/856
c8.gb	56.3/36.8	1,164/1,068	50.1/32.4	892/640	70.7/39.0	1,292/748

Table 5: Time and memory consumption of PDC, 16Huff and IDC

Overall, considering all factors (compression ratio, time and memory consumption), 16Huff is more favoured for practical 16-bit text compression.

4 Cascading LZSS with PDC, 16Huff and IDC

In this section, the cascading models of LZSS [Bel86] with PDC, 16Huff and IDC are studied. LZSS is one of the practical variations of LZ77[ZL77]. The basic idea of LZSS is that it uses previously seen text as a dictionary, and replaces phrases in the input text with pointers into the dictionary. There are two important structures in LZSS: the sliding window and the lookahead buffer. The input characters pass the buffer firstly, and then go into the window. Each time the algorithm tries to find a longest match from the buffer into the window. If a match is found, LZSS will output a bit 0 to indicate there is a match, and output the matched position followed by the matched length. If a match is not found, it will output a bit 1 and the unmatched character. All output elements, such as the matched position, the matched length and the unmatched character, are encoded in fixed length of bits. Figure 1 illustrates an example of LZSS.

The fixed-length bit representation assumes all values in the tokens of LZSS are in uniform distribution. However, this is not true in general. It is found out that some values occur very frequently while some occur rarely. If the frequent values can be represented in relatively smaller number of bits, and the infrequent values can be represented in relatively larger number of bits, the overall usage of bits would be decreased. Therefore, it would be better if we encode the values by variable-length bits basing on their frequencies. For this purpose, adaptive Huffman coding scheme is employed.

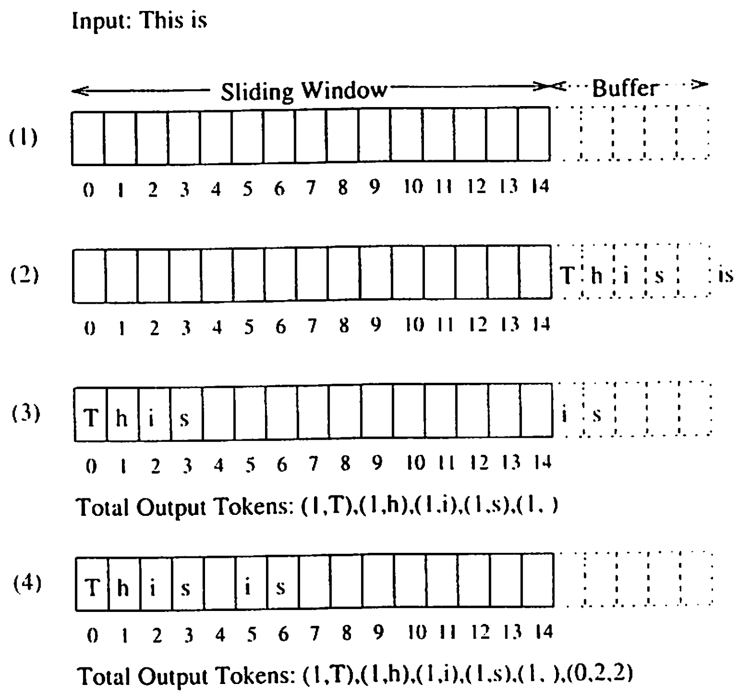

Figure 1: An example of LZSS

Two different adaptive Huffman trees are built during the cascading process. The PDC, 16Huff and IDC are used to encode the unmatched characters while a 8-bit Huffman coding scheme is used to encode the matched length. Fixed-length bits are used to represent the matched position because a wide range of matched position occurs, and the repetition of each matched position seems to be rare. Under this situation, variable-length bits would be useless for improving the compression ratios. The generalized cascading model of LZSS is shown in Figure 2.

LZSS is a one-pass compression algorithm which scans the input text once. However, 16Huff, PDC and IDC are multi-pass algorithms. In order to make the cascading algorithm suitable for on-line 16-bit text compression, the three semi-adaptive Huffman coding schemes are changed to adaptive versions, which scan and encode an input text in only one pass [Mar92].

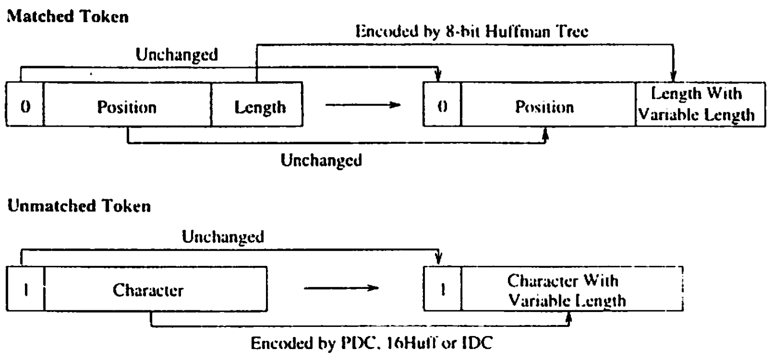
Matched Token
Encoded by 8-bit Huffman Tree
Unchanged
0
Position
Length
0
Position
Length With Variable Length
Unchanged
Unmatched Token
Unchanged
1
Character
1
Character With Variable Length
Encoded by PDC, 16Huff or IDC

Figure 2: The generalized cascading model of LZSS

4.1 Compression Results

In LZSS, there are two parameters: sliding window size (2^N) and lookahead buffer size (2^F). Before analyzing the cascading model, we have to find the best setting of the two parameters at first. The compressed file sizes of LZSS under different settings of N and F are shown in Figure 3. On average, when $N = 18$ or 19 and $F = 2$ or 3, the compression results seem to be the best. In other words, a large size of sliding window and a small size of lookahead buffer are favoured for compressing 16-bit text. Since a normal 16-bit passage contains many repeating phrases with length less than four characters, so a small size of lookahead buffer is enough. On the contrary, a large size of sliding window should be used for storing and matching more past phrases.

In the following paragraphs, the performance of different cascading algorithms will be compared. A default setting of LZSS ($N = 18$ and $F = 3$) is adopted for the cascading algorithms since the best compression results are obtained under this setting.

The cascading algorithms of LZSS with PDC, 16Huff and IDC are called LZSSPDC, LZSS16Huff and LZSSIDC respectively. The compression results of these algorithms are shown in Table 6. The value out of brackets is compressed file size while the value in the brackets is compression ratio. It is observed that LZSSPDC achieves higher compression ratios than the other two algorithms, and LZSSIDC performs the worst.

In the cascading model, the three variations of Huffman coding scheme are applied to unmatched characters only. Normally, the total number of unmatched characters are relatively small, comparing with the total number of matched characters. If all the unmatched characters are extracted to generate a new file, the size of the new file would also be small. In Section 3, it is explained that PDC is better than 16Huff when compressing small files since PDC can generate relatively smaller header. Undoubtly, LZSSPDC should outperform LZSS16Huff.

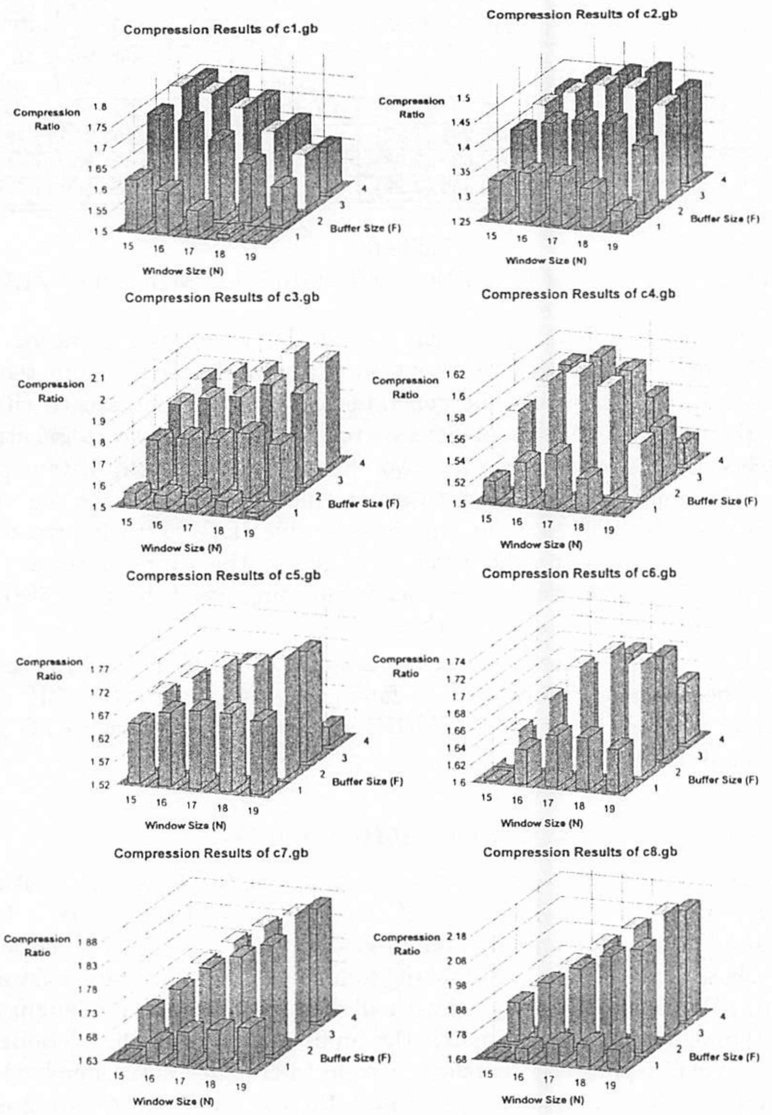

Figure 3:
Compression results of LZSS under different settings of N and F

Name	LZSSPDC	LZSS16Huff	LZSSIDC	GZIP
c1.gb	122,666 (1.93)	125,089 (1.89)	125,311 (1.89)	117,914 (2.00)
c2.gb	184,604 (1.73)	188,247 (1.69)	188,622 (1.69)	194,002 (1.64)
c3.gb	388,215 (2.18)	393,081 (2.15)	393,684 (2.15)	392,259 (2.15)
c4.gb	406,684 (1.82)	410,357 (1.81)	410,961 (1.81)	423,183 (1.75)
c5.gb	1,193,620 (1.94)	1,197,394 (1.94)	1,198,983 (1.93)	1,294,354 (1.79)
c6.gb	1,281,925 (1.93)	1,286,291 (1.92)	1,288,155 (1.92)	1,390,110 (1.78)
c7.gb	7,518,984 (2.08)	7,520,680 (2.08)	7,524,368 (2.08)	8,425,850 (1.85)
c8.gb	3,568,069 (2.29)	3,570,958 (2.29)	3,573,423 (2.28)	4,134,569 (1.97)

Table 6:
Compression results of LZSSPDC, LZSS16Huff, LZSSIDC and GZIP

On the other hand, LZSSIDC achieves the worst compression ratios. It is mainly due to the partition concept no longer works for the unmatched characters. Consider a new file generated only by the unmatched characters, there should have no obvious correlation between two consecutive unmatched characters because the two characters come from distant positions. Consequently, it is hard to choose indicators to cut the file into partitions clearly. Furthermore, in the adaptive IDC, extra information such as the indicators and the novel characters in the second tree have to be stored in the compressed file, so the overall compressed size of LZSSIDC is larger than the other two algorithms.

In order to compare the compression results with existing cascading compressor, the compression results of a famous UNIX compressor GZIP are also shown in Table 6. Again, LZSSPDC is the best. It outperforms GZIP by 5% on average.

5 Cascading LZW with PDC, 16Huff and IDC

In this section, the cascading models of LZW [Wel84] with PDC, 16Huff and IDC are studied. LZW is an effective variant of LZ78 [ZL78]. The basic idea is to use a potentially unlimited size of dictionary which contains previously seen phrases for encoding the coming phrases. The dictionary entries of LZW are initially assigned with all defined alphabets in a language. Then, the algorithm tries to match the input text against the dictionary entries. When a phrase is matched, a code (dictionary entry number) is generated. Each time a code is generated, the code and the following unmatched character are merged to form a new phrase and, the phrase will be appended in the dictionary. Table 7 illustrates an example of LZW [Mar92]. The example is for 8-bit ASCII codes, and the input string is "WED WE WEE WEB WET".

Character Input	Code Output	New code value & associated string
"W"	"	256 = "W"
"E"	W	257 = "WE"
"D"	E	258 = "ED"
" "	D	259 = "D"
"WE"	256	260 = "WE"
" "	E	261 = "E"
"WEE"	260	262 = "WEE"
"W"	261	263 = "E W"
"EB"	257	264 = "WEB"
" "	B	265 = "B"
"WET"	260	266 = "WET"
End of file	T	

Table 7: An example of LZW

In the example, LZW outputs a dictionary entry number each time. Some entries are referenced frequently while some are not. Therefore, the frequency distribution of the entries is not uniform, and Huffman coding scheme should be applied. The way of cascading LZW with an adaptive Huffman coding scheme is trivial. When LZW outputs an entry number in fixed-length bits, the entry number is further encoded by an adaptive Huffman tree in variable-length bits. The generalized cascading model of LZW is shown in Figure 4.

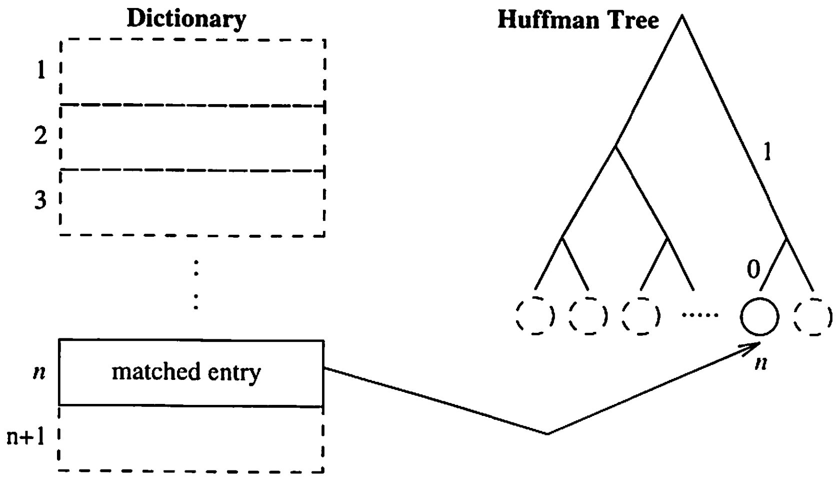

Figure 4: The generalized cascading model of LZW

The only parameter in LZW is the size of dictionary (2^N). If N is equal to 16, there are 65,536 entries in the dictionary, and each token is represented by 16 bits. Apart from tuning the parameter N, Mark [Mar92] proposed two modifications to increase the compression performance. They are the bumping and flushing mechanisms. Take a 8-bit ASCII input file as an example: assuming N is equal to 16. LZW loads the 256 defined characters into the dictionary, and the size of dictionary will be gradually increased. When the LZW encoding process starts, the number of filled entries is far less than 65,536, so it would be wasteful to output a 16-bit token for representing the beginning entries. Instead, a 9-bit token is generated at first. When the number of filled entries becomes 512, a special bumping code is output, and the length of token is increased to 10 bits. In other words, the bit length of a token is not fixed. It is gradually increased according to the number of filled entries in the dictionary. This is called the bumping mechanism. Once the dictionary is filled up, a special flushing code is output. Then, the dictionary is totally flushed and initialized again. This is called the flushing mechanism.

Before performing the cascading process, it is necessary to find the best size of the LZW dictionary so that it can achieve the minimum compressed file size. The bumping and flushing mechanisms are employed, and the parameter N is varied. It is found out that the token length would be very large if all predefined 16-bit characters are loaded in the dictionary. A simple modification is made to load the distinct characters appearing in an input text only. To achieve this effect, an extra pass is consumed for finding the distinct characters appearing in the text. It is worth using the extra pass to reduce the token length because the size of the distinct characters in the text would be far less than the size of predefined character set. Table 8 shows the compression results of LZW by varying the parameter N. Although when $N = 17$ or 18, the compression results of LZW are the best, $N = 16$ is chosen for the cascading model of LZW. The parameter setting will be explained later.

Name	$N = 15$	$N = 16$	$N = 17$	$N = 18$
cl.gb	129,419	130,174	130,539	130,539
c2.gb	219,986	219,396	215,171	215,171
c3.gb	426,891	424,530	414,173	410,810
c4.gb	448,193	446,521	442,577	439,343
c5.gb	1,324,362	1,297,368	1,272,860	1,255,536
c6.gb	1,434,080	1,409,500	1,387,704	1,364,666
c7.gb	8,774,061	8,473,740	8,245,063	8,032,090
c8.gb	4,339,441	4,179,656	4,029,293	3,879,077

Table 8: Compression results of LZW with different values of N

5.1 Compression Results

The cascading algorithms of LZW with PDC, 16Huff and IDC are called LZWPDC, LZW16Huff and LZWIDC respectively. In the above paragraph, it is said that when $N = 17$ or 18, the compression results of LZW are the best. However, N is set to be 16 in the cascading model of LZW. If N is greater than 16, the number of nodes in the Huffman tree would be so large that the compression and decompression processes are very slow. Moreover, larger number of dictionary entries means fewer repetitions of the entries, so the Huffman coding scheme may not take any advantage when encoding the entries. If N is less than 16, the dictionary is not large enough to hold sufficient phrases for matching. Therefore, $N = 16$ is used as a default setting for the different cascading algorithms. The compression results of these algorithms are shown in Table 9. It is observed that LZWPDC achieves higher compression ratios than the other two algorithms, and LZWIDC performs the worst.

Name	LZWPDC	LZW16Huff	LZWIDC
c1.gb	128,814(1.83)	144,361 (1.64)	144,766 (1.63)
c2.gb	210,252(1.52)	231,041 (1.38)	232,434 (1.37)
c3.gb	413,937(2.04)	446,751 (1.89)	448,342 (1.88)
c4.gb	426,879(1.74)	456,626 (1.63)	458,689 (1.62)
c5.gb	1,223,893(1.89)	1,263,797 (1.83)	1,268,638 (1.83)
c6.gb	1,316,973(1.88)	1,362,091 (1.82)	1,367,751 (1.81)
c7.gb	7,758,557 (2.01)	7,749,346(2.02)	7,773,862 (2.01)
c8.gb	3,902,765(2.09)	3,938,609 (2.07)	3,953,481 (2.06)

Table 9: Compression results of LZWPDC, LZW16Huff and LZWIDC

When an input text is being read, the dictionary in LZW grows gradually. It is found that the output entries are within some repeating intervals, and the repetitions of the intervals are quite high. It would be better if we can encode the intervals and the numbers within the intervals separately. Only PDC can achieve this goal. It encodes the first 8 bits by a tree and the remaining 8 bits by another trees. The arrangement can capture the property of high repetition of the intervals (the first 8 bits). None of the LZW16Huff and LZWIDC can capture this property.

Most entries in the dictionary consist of several 16-bit characters, and the repetition of consecutive dictionary entries are expected to be low. In other words, applying IDC to the entries of LZW dictionary has no benefit because it is hard to cut the entries into partitions without duplication. The weakness of LZWIDC is similar to the case of the cascading algorithm LZSSIDC.

To sum up, by taking the compression ratio into consideration, PDC is better for both the cascading models of LZSS and LZW.

6 Conclusions

Among different character-based Huffman coding schemes, IDC seems to be the best. It outperforms 16Huff and PDC by around 5% in compression ratio. The average compression ratio achieved by IDC is 1.7. However, when considering the cascading models of LZSS and LZW, PDC is better. In the cascading model of LZSS, LZSSPDC outperforms the LZSS16Huff and LZSSIDC by around 1%. It can achieve an average compression ratio of about 2. In the cascading model of LZW, LZWPDC outperforms LZW16Huff and LZWIDC by around 5%. It can achieve an average compression ratio about 1.9. In conclusion, it is more effective to choose PDC for the two cascading models. From the experimental results, the best cascading algorithm is LZSSPDC. It outperforms a famous UNIX cascading compressor GZIP by 5% on average.

References

[Bel86] T.C. Bell, Better OPM/L text compression, *IEEE Transactions on Communications*, 34(12) (1986), 1176–1182.

[CLR90] T.H. Cormen, C.E. Leiserson, and R.L. Rivest, *Introduction to Algorithms*, MIT Press, 1990.

[CWY99] K.S. Cheng, K.F. Wong, and Gilbert H. Young, A study on Word-based and Integral-bit Compression Chinese Text Compression Algorithms, *Journal of American Society of Information Science* 50(3) (1999), 218–228.

[CY97] K.S. Cheng and Gilbert H. Young, Chinese Text Compression: A Survey, In *Proceedings of the Seventeenth International Conference on Computer Processing of Oriental Languages*, pages 162–167, Hong Kong, April 1997.

[HC92] R.N. Horspool and G.V. Cormack, 'Constructing Word-Based Text Compression Algorithms, In *Proceedings of Data Compression Conference 1992*, pages 62–71, Snowbird, Utah, March 1992.

[HT90] T.H. Huang and L.Y. Tseng, A Predictive Coding Method for Chinese Text File Compression, *Journal of Computers* 2(3) (1990), 18–23.

[Huf52] D.A. Huffman, A Method for the Construction of Minimum Redundancy Codes, *Proceedings of the Institute of Radio Engineers* 40(10) (1952), 1098–1101.

[Mar92] N. Mark, *The Data Compression Book*, M & T Books, 1992.

[Wel84] T.A. Welch, A Technique for High-performance Data Compression, *IEEE Computer* **17**(6) (1984), 8–19.

[YC99] G.H. Young and K.S. Cheng, Indicator Dependent Huffman Coding Scheme for Chinese Text Compression, *Computer Processing of Oriental Languages*, (1999) to appear.

[ZL77] J. Ziv and A. Lempel, A Universal Algorithm for sequential Data Compression, *IEEE Transactions on Information Theory* **IT-23**(3) (1977), 337–343.

[ZL78] J. Ziv and A. Lempel, Compression of Individual Sequences via Variable-rate Coding, *IEEE Transactions on Information Theory* **IT-24**(5) (1978), 530–536.