Less Redundant Codes for Variable Size Dictionaries

Zhen Yao* Nasir Rajpoot†

Abstract

In this paper, we report work on a family of variable-length codes with less redundancy than the *de facto* flat code used in most of the variable size dynamic dictionary based compression methods. The experiments show that by replacing the flat code with these codes in the original LZW compression algorithm, nearly 3%–7% redundancy can be removed, whereas up to 72% improvement can be obtained for a running sequence from the artificial Canterbury corpus.

1 Introduction

Most of the lossless data compression methods used in practice are based on maintaining a dynamic *dictionary* of strings that are also called *phrases*. The input to a dictionary based compression method is *parsed* into substrings that are replaced by pointers corresponding to these phrases in the dictionary also known as the *codewords* in order to achieve compression. Many variants of the original dictionary compression schemes – the LZ77 method [11], the LZ78 method [12], and its LZW variant [10] – or *textual substitution* [9] methods have been proposed to date. Almost all of these variants suggest improved ways of dictionary construction, dictionary maintenance, or parsing [4, 5, 7, 6] – for a detailed coverage of these variants, please refer to [8].

Based on the variability of the size of dictionary, dictionary based compression methods can be divided into two categories; methods which have a dictionary of fixed size, and methods whose dictionary can grow or shrink as the input is processed. Many LZ77 variants fall into former of the above categories, whereas most of the dictionary methods based on the LZ78 compression scheme belong to the latter category. In this paper, we study the improvement of variable size dictionary compression methods from a different perspective. The following question is addressed: *Given a variable size dictionary based compression algorithm, is it possible to improve the compression performance by introducing a more clever way of encoding the codewords of this algorithm?* Consider the LZW compression scheme which uses a variable-length code to encode the codewords where length l of the code varies with size of the dictionary D as given by $l = \lceil \log_2 N \rceil$, where $N = |D|$ denotes the dictionary size or the number of phrases in the dictionary. This type of code, also termed as the *flat* code, can be *wasteful* since the optimal code length $l^\star$ for encoding a codeword

*Department of Computer Science, University of Warwick, Coventry CV4 7AL, UK; email: csvhe@dcs.warwick.ac.uk

†Department of Computer Science, University of Warwick, Coventry CV4 7AL, UK; telephone: +44 (24) 7657-3795; email: nasir@dcs.warwick.ac.uk

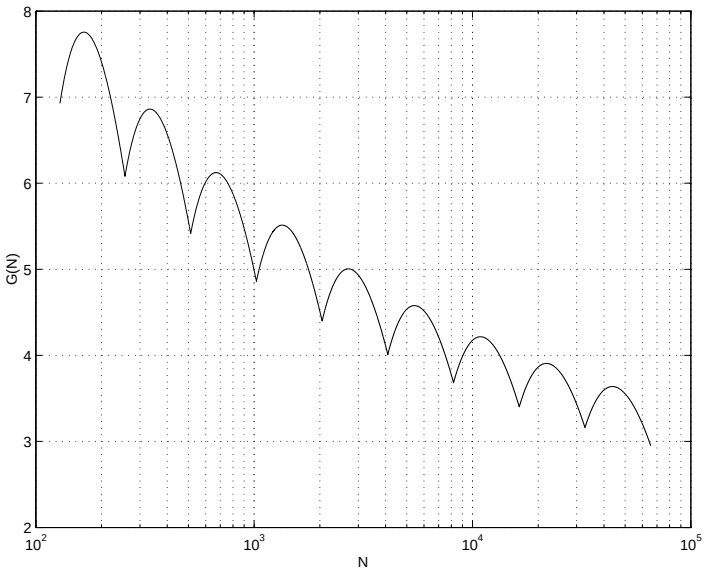

Figure 1: Percentage relative redundancy versus dictionary size

in this scenario is always given by $l^\star \leq \lceil \log_2 N \rceil$. The *wastage* of bits (or the redundancy) W in case of the flat code can be expressed in terms of dictionary size N by

$$W(N) = \sum_{i=2}^{N} (\lceil \log_2 i \rceil - \log_2 i). \tag{1}$$

Figure 1 shows a graph of the *relative redundancy* $G(N) = W(N)/F(N)$, where $F(N)$ is the total number of bits when using the flat code, plotted against N. It illustrates that an improvement of 2% to 8% for a variable size dictionary based compression algorithm (whose dictionary phrases all have almost equal frequency of occurrence) is possible provided a better alternative to the flat code is employed to encode the codewords[1].

In this study, we report on a family of less-redundant codes in an LZW setting: **(1)** We place in this family of less-redundant codes one already published and two new types of less-redundant codes termed as the phase-in-binary (PB) code [1], the depth-span (DS) code, and the balanced code. **(2)** We show that the DS code is optimal in the sense of code-length contrast, whereas the balanced code is optimal in the sense of average codeword length. **(3)** Our experimental results show that the relative redundancy of flat codes can be brought down to nearly zero by using these new codes.

2 Family of Less-Redundant Codes (LRC)

As described earlier, reasonable improvements can be made on top of the ordinary flat code without any significant additional computational complexity. The type of code that achieves this improvement can be termed as a less-redundant code. In other words,

[1]It is also clear from the graph that improvement on the flat code performance $F(N) = \sum_{i=2}^{N} \lceil \log_2 i \rceil$ generally decreases as the dictionary grows larger.

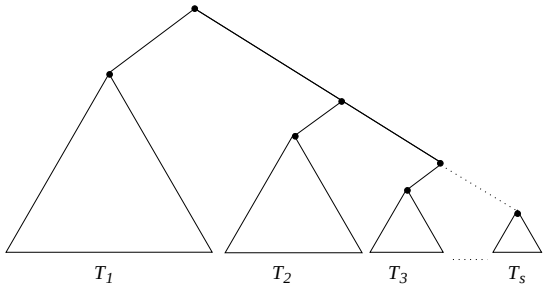

Figure 2: A phase-in-binary tree

Definition 1 *A set of binary codes C is a set of less-redundant codes (LRC), with each of its members corresponding to a unique codeword, if it satisfies the following two conditions:*

1. $\forall c \in C$: $l(c) \leq \lceil \log_2 |D| \rceil$, *where* $l(c)$ *is the length of the code* c.
2. $\log_2 |D| \notin \mathcal{N}$, *the set of positive integers,* $\longrightarrow$ $\exists c \in C : \; l(c) < \lceil \log_2 |D| \rceil$.

Each of the three codes described below is associated with a binary tree whose structure explains construction of the code itself. By following the convention that every left branch is labelled 0 and every right branch 1, the traversal of a path from the root node to a leaf node represents the binary code for the codeword corresponding to that particular leaf node.

2.1 The Phase-in-Binary Code

This code was first reported in [1] as an alternative to the flat code for enhancing the performance of the LZW compression algorithm.

Definition 2 *Given a positive integer N expressed as $N = \sum_{m=1}^{s} 2^{a_m}$, where $a_1 > a_2 > ... > a_s \geq 0$ and $s \geq 1$, a* **phase-in-binary tree** *T with N leaf nodes is a rooted binary tree that consists of s complete binary trees (represented by a solid triangle in this paper) $T_1, T_2, ..., T_s$ of heights $a_1, a_2, ...a_s$ respectively.*

Given a phase-in-binary tree T with N leaf nodes, similar to the one shown in Figure 2, the *phase-in-binary code* of a codeword C_n, is defined by the path which traverses from the root of T to the nth leaf node in T.

An interesting feature of the phase-in-binary tree is that all of its subtrees are complete binary trees. From an implementation viewpoint, simple binary shift operations can be used to construct the phase-in-binary tree in an efficient way.

2.2 The Depth-Span Code

A depth-span (DS) code differs from the PB code only in the way the right half of the binary tree is constructed. The depth-span tree tends to take the leftmost leaf node at a depth less than the maximum depth to the lowest level before completing the subtree at its level. It can be defined as follows:

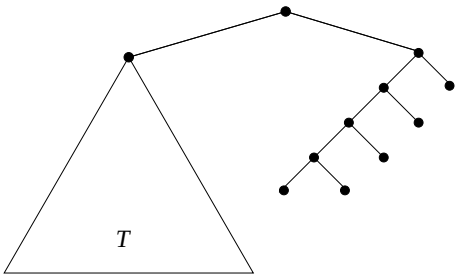

Figure 3: A Depth-Span Tree

Definition 3 *For a positive integer N, a* **depth-span tree** *with N leaf nodes is a rooted binary tree whose root has as its left child a complete binary tree and as its right child another binary tree with each its leaf node extending to the maximum depth before allowing its sibling nodes on the right to grow.*

A typical depth-span tree is shown in Figure 3. The code length remains the same or decreases when moving from left to right, as is the case with the PB code. This assures us that length of the codeword associated with the rightmost leaf node remains minimum. Therefore, some gain can be expected if phrases corresponding to the recent codewords are repeated in the input. In other words, the *code length contrast*, which is the range of the depth levels of the leaf nodes, remains maximum.

Lemma 1 *The DS code is optimal with respect to the code-length contrast.*

Proof: From the definition of the DS code, the branches with minimum depth are kept while giving priority to the completion of binary trees to their left. This ensures that the difference between the length of the longest and the shortest DS codes remains maximum.

2.3 The Balanced Code

Definition 4 *Given a positive integer N, a* **balanced tree** *with N leaf nodes is a rooted binary tree which is either a complete binary tree with a depth of $\log_2 N = \lceil \log_2 N \rceil$, or is complete up to a depth of $n = \lfloor \log_2 N \rfloor$ and adds new leaf nodes to itself at depth $n+1$ in a left-to-right manner.*

It is clear from the above definition that it is a depth-balanced tree, i.e. a tree whose subtrees differ in depth by no more than one and the subtrees are also depth-balanced. If the balanced tree is not complete, or in other words if $n = \lceil \log_2 N \rceil$ is not an integer, then it has $2^{n+1} - N$ leaf nodes at depth n and rest of the leaf nodes are at depth $n+1$ starting from the leftmost node and right onwards as illustrated in Figure 4. A *balanced code* of a codeword C_i is defined by the path traversed from the root of the balanced tree to the ith leaf node. Due to its being balanced, the code-length contrast is either 0 or 1. We note here that a similar code is also mentioned in [2] with the name of *phasing in binary code*.

Lemma 2 *The balanced code is optimal with respect to the average codeword length.*

Proof: It is clear from the above definition of a balanced code that the average length of a codeword using balanced codes is $O(\log_2 N)$ where N denotes the current dictionary size.

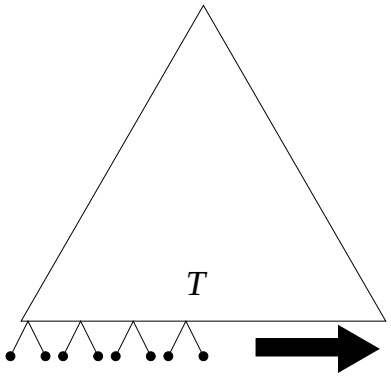

Figure 4: Development of the Balanced Code

Both the PB and the DS codes exploit the fact that the codewords corresponding to phrases at the high end of dictionary may occur more frequently than those corresponding to the phrases added not so recently to the dictionary. As opposed to the PB and the DS codes, the balanced code remains largely unaffected by the probability distribution of codewords. Assuming that all the codewords have an equal probability of occurence, and no specific range of codewords is more likely to occur than the others, the average codeword length is minimal when a balanced code is used since it gives almost equal weight to any of the codewords by assigning to them almost same number of bits.

3 Effects of Codeword Distribution

As mentioned above in the previous section, both the PB and the DS codes try to keep some codes as short as possible (the DS code being optimal in the sense of code-length contrast, as mentioned above) with the hope that a few codewords would *hit* these shorter codes in turn yielding improvement in terms of compression performance. They are particularly good to compress data from sources with large amount of redundancy, such as a running sequence of one particular symbol.

In this section, we look at the distribution of codewords belonging to different sections of the dictionary and show that the compression performance can further be improved by a simple shifting of the distribution during the compression process.

3.1 Block Occurrences

Given the usually large dictionary size, it is not practical to compute the frequency of occurence of each of the codewords in the dictionary. The codeword distribution can, however, be approximated by a histogram of the consecutive blocks of codewords in the dictionary. We divide the dictionary D into b disjoint blocks of consecutive codewords $\{D_1, D_2, ..., D_b\}$ with each block consisting of B codewords, where $B = |D|/b$. A block occurrence histogram can show which part of the dictionary is more often being used than others. Figure 5 shows block occurence histograms for an LZW dictionary when compressing files from the large Canterbury corpus with the dictionary divided into 30 blocks. It is interesting to note that the histograms for two English text files *bible.txt* and *world192.txt* are almost similar with the histogram count of the last block of the dictionary being particular

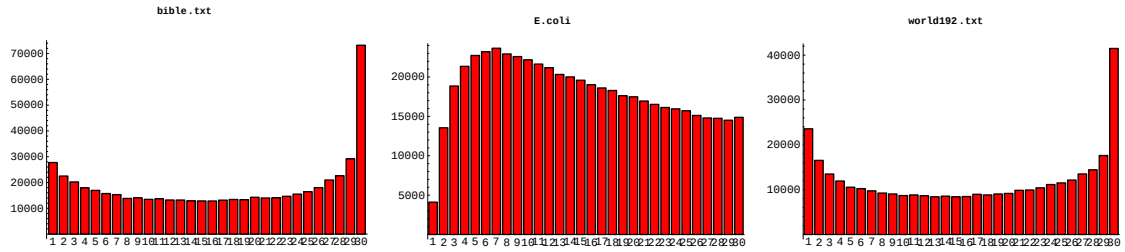

Figure 5: Block occurrences in LZW dictionary

higher than that of the intermediate blocks. This can be regarded as a typical English text file characteristic and is in conformance with the evidence that if a word occurs once in an English text, then there is a high probability that it will occur again soon [3].

3.2 Dynamic Block Shifting

If the distribution of codewords for *E.coli* can be altered in such a way that the block of codewords occurring most frequently is shifted to the end of the dictionary, small improvement can be expected when using the PB or DS codes. This simple procedure, termed as the *dynamic block shifting* (DBS), dynamically updates the block occurrence histogram in an efficient way. Instead of actually moving the whole block of codewords from their original places in the dictionary, it suffices to use simple arithmetic operations to compute the post-shifting codeword values.

4 Experimental Results

In this section, we present the experimental results of five variable size dictionary based compression algorithms: the original LZW algorithm using the standard flat code as well as the LZW algorithm using the PB code, the DS code, the balanced code, and the PB code with dynamic block shifting. The maximum number of phrases in each of the coders' dictionary was allowed to be 2^{20} in order to avoid unnecessary loss of information due to flushing of the dictionary.

Three data sets were used from the Canterbury corpora: (1) the ordinary corpus, containing various kinds of source data; (2) the large corpus, which includes three files of size $>$ 1**MB**, an DNA sequence from *E-coli* bacteria, and two text files, *bible.txt* and *world192.txt*; and (3) the artificial corpus, containing files with little or no repetition (e.g. *random.txt*), and a file with large amounts of repetition (*aaa.txt*).

The experimental results for all these data sets using the codes mentioned earlier are shown in Tables 1–3. Results from both ordinary and large Canterbury corpora show that the LRC achieve an improvement of 2.5% to 6.5% which is very close to the relative redundancy estimates discussed earlier. In our opinion, the most interesting results of our experiments are those obtained by applying the new codes on the Canterbury artificial corpus (Table 3). Here, the DS code achieves an improvement of 72.5% for a running sequence *aaa.txt*, whereas the PB code achieves an improvement of over 18% on another file with repititions *alphabet.txt*. The improvement for *random.txt* achieved by the PB code with dynamic block shifting is just below 3% suggesting that the DBS does a good job by

File	Original	Balanced	D.S	P.B	P.B-DBS	LZW
alice29.txt	152089	**60450**	62155	61091	61090	62244
asyoulik.txt	125179	**53550**	54927	54006	53992	54987
cp.html	24603	**10946**	11263	11036	11032	11314
fields.c	11150	4731	4951	4735	**4727**	4961
grammar.lsp	3721	1710	1796	1710	**1708**	1810
kennedy.xls	1029744	314668	320229	309831	**305212**	320787
lcet10.txt	426754	**158245**	163023	159193	159190	163170
plrabn12.txt	481861	**193292**	198359	195744	195165	198464
ptt5	513216	60362	61599	**59747**	59749	62212
sum	38240	19282	19932	18805	**18803**	20099
xargs.1	4227	**2244**	2333	2280	2277	2336
Total	2810784	879480	900567	878178	**872945**	902384

Table 1: The Canterbury corpus

File	Original	Balanced	D.S	P.B	P.B-DBS	LZW
bible.txt	4047392	1208875	1240534	1206281	**1206279**	1241874
E.coli	4638690	**1203801**	1230017	1219761	1216023	1230082
world192.txt	2473400	772208	793712	772006	**771989**	795209
Total	11159482	**3184883**	3264263	3198048	3194291	3267166

Table 2: The Canterbury large corpus

moving the most frequent block towards the end of dictionary.

5 Conclusions

In this paper, we presented a family of codes that are less redundant than the flat codes used by most of the variable size dynamic dictionary based compression methods. It was shown that the DS and the balanced codes are optimal with respect to the code-length contrast and the average code length respectively. The experimental results verify simplistic theoretical estimates that for dictionaries with uniform codeword distribution, the improvement can be 2%–8%. However, large gains can be obtained for dictionaries which have non-uniform codeword distribution.

References

[1] T. Acharya and J.F. Já Já. Enhancing Lempel-Ziv codes using an on-line variable-length binary encoding. In *Proceedings IEEE Data Compression Conference (DCC'96)*, March 1996.

[2] T.G. Bell, J.G. Cleary, and I.H. Witten. *Text Compression*. Prentice-Hall, Englewood, Cliffs., NJ, 1990.

[3] R. DeMori and R. Kuhn. A cache-based natural language model for speech recognition. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 12(6):570–583.

[4] D.T. Hoang, P.M. Long, and J.S. Vitter. Dictionary selection using partial matching. *Information Sciences*, 119(1-2):57–72, 1999.

File	Original	Balanced	D.S	P.B	P.B-DBS	LZW
aaa.txt	100000	471	**145**	328	329	527
alphabet.txt	100000	2778	2927	**2488**	**2488**	3050
random.txt	100000	91258	92372	92135	**89681**	92374

Table 3: The Canterbury artificial corpus

[5] R.N. Horspool. The effect of non-greedy parsing in Ziv-Lempel compression methods. In *Proceedings IEEE Data Compression Conference (DCC'95)*, 1995.

[6] Y. Matias, N.M. Rajpoot, and S.C. Sahinalp. The effect of flexible parsing for dynamic dictionary based data compression. In *Proceedings IEEE Data Compression Conference (DCC'99)*, pages 238–246, March 1999.

[7] V.S. Miller and M.N. Wegman. Variations on a theme by Lempel and Ziv. *Combinatorial Algorithms on Words*, pages 131–140, 1985.

[8] N.M. Rajpoot and S.C. Sahinalp. Dictionary based data compression: A combinatorial perspective. In Khalid Sayood, editor, *Handbook of Data Compression*, 2001. to appear.

[9] J.A. Storer. *Data Compression: Methods and Theory.* Computer Science Press, Rockville, Maryland, 1988.

[10] T.A. Welch. A technique for high-performance data compression. *IEEE Computer*, pages 8–19, January 1984.

[11] J. Ziv and A. Lempel. A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory*, IT-23(3):337–343, May 1977.

[12] J. Ziv and A. Lempel. Compression of individual sequences via variable-rate coding. *IEEE Transactions on Information Theory*, IT-24(5):530–536, Sep 1978.