Modified Golomb Code for Integer Representation

Nelson Raja Joseph[1] Jaganathan P[2] Domnic Sandanam[3]
[1]Department of Computer Science [2,3]Department of Computer Applications
[1]Bharathiyar University, Coimbatore, India [2]PSNA College of Engineering & Technology Dindigul, India [3]National Institute of Technology, Tiruchirappalli India

Abstract— In this computer age, all the computer applications handle data in the form of text, numbers, symbols and combination of all of them. The primary objective of data compression is to reduce the size of data while data needs to be stored and transmitted in the digital devices. Hence, the data compression plays a vital role in the areas of data storage and data transmission. Golomb code, which is a variable-length integer code, has been used for text compression, image compression, video compression and audio compression. The drawback of Golomb code is that it requires more bits to represent large integers if the divisor is small. Alternatively, Golomb code needs more bits to represent small integers if the divisor is large. This paper proposes Modified Golomb Code based on Golomb Code, Extended Golomb Code to represent small as well as large integers compactly for the chosen divisor. In this work, as an application of Modified Golomb Code, Modified Golomb Code is used with Burrows-Wheeler transform for text compression. The performances of Golomb Code and Modified Golomb Code are evaluated on Calcary corpus dataset. The experimental results show that the proposed code provides better compression rate than Golomb code on an average. The performance of the proposed code is also compared with Extended Golomb Codes (EGC). The comparison results show that the proposed code achieves significant improvement for the binary files of Calgary corpus comparing to EGC.

Key words: Variable Length Code, Golomb Code, Modified Golomb Code, Burrows-Wheeler Compression

I. INTRODUCTION

The main aim of data compression is to store data with the minimum number of bits in storage devices and transmit in low band width communication networks. Data-compression methods can be generally classified into two types i.e lossy and lossless. In lossless compression, data can be compressed and decompressed as exactly identical with the source data without any loss of data. Lossless compression technique is used, in which the decompressed data must be identical to the source data such as financial data, executable programs, text documents, and source code. Lossless data compression is used in many applications such as zip tools and wireless sensor networks. Lossy data compression has a certain loss of information and decompressed data is not 100% identical to the source data. Lossy data compression technique is used to compress video, audio and images. Various codes have been applied for data compression [1].

In contrast with the fixed-length codes, statistical coding methods achieve compression by assigning short-length codes to the more frequent occurring symbols and long-length codes to rarely occurring symbols of the source file which needs to be compressed. The statistical methods require the probabilities of the input symbols to generate variable-length codes. Huffman coding [2] and Shannon-Fano [3] methods are examples for statistical methods which use symbol tables while decoding the compressed data. There are other coding methods such as Elias Gamma codes, Elias Delta code, Golomb code, Fibonacci codes [4] and Extended Golomb Code (EGC) [5], which do not require the probability values of the input data to produce variable-length codes and these methods are called as variable-length integer coding methods or variable-length integer codes. Since variable-length integer codes do not require symbol table and probability values, these are more preferable in the applications which require fast encoding and storage.

In this paper, we propose a new code, Modified Golomb Code (MGC), to produce variable-length codes by representing non-negative integers. Alternatively, MGC can encode or represent non-negative integers very compactly. Golomb Code (GC) [6] has been used in several applications such as lossless image codecs, audio codecs and search engines [7 - 8]. But, the disadvantage of GC is that it requires more bits to represent large integers if the divisor (d) is small. Alternatively, GC needs more bits to represent small integers if the divisor is large. Hence, GC could not be the best choice for the applications which have the distribution of small and large integers. To overcome the drawback of GC, we propose Modified Golomb Code based on GC.

II. GOLOMB CODE (GC)

Golomb Code was proposed by Solomon Golomb in 1966 for lossless data compression. In GC, the compact representation of non-negative integers n_i depends on the selection of the divisor d. In the first step of GC, the given number n (>0) is first divided by a divisor d. The quotient (q) and the remainder (r) of the given n are then used to generate codes. The formula given in Equation (1) is used to calculate the quotient (q) and the remainder (r) for the given n.

$$\left.\begin{aligned} q &= \frac{n-1}{d} \\ r &= n - qd - 1 \end{aligned}\right\} \qquad (1)$$

GC contains two parts. The first part is the quotient value of $(q+1)$ which is coded in unary code (i.e q zeros followed by single one or q ones followed by single zero) [1] and the second part is binary code of remainder (r). For example, when divisor d = 3, it produces three remainders, 0, 1, 2, and are coded as 0, 10 and 11 respectively (See Table 1). Table 2 shows the GC for divisors d =2, 3 and 4. The bit lengths of GC (d=2, 3 and 4) to represent the integers in the range 0 - 255 are calculated and are given in Table 4. It is observed from Table 4 that GC ($d=2$) offers compact representation for small range (1-5) and provides poor representation for middle (32-63) and large (64-255) range of integers. Also, for other divisors (d = 4, 8), GC does not give better representation for middle and large range of integers. In order to improve the integer representation of GC, a new method based on GC is proposed in this paper.

Remainders	**Binary codes**		
	d=2	***d=3***	***d=4***

0	0	0	00
1	1	10	01
2	-	11	10
3	-	-	11

Table 1: Codes for remainders divisor d = 2, 3 and 4

Integer *n*	**GC**		
	d=2	***d=3***	***d=4***
1	0 | 0	0 | 0	0 | 00
2	10 | 0	0 | 10	0 | 01
3	10 | 1	0 | 11	0 | 10
4	110 | 0	10 | 0	0 | 11
5	110 | 1	10 | 10	10 | 00
6	1110 | 0	10 | 11	10 | 01
7	1110 | 1	110 | 0	10 | 10
8	11110 | 0	110 | 10	10 | 11
9	11110 | 1	110 | 11	110 | 00
10	111110 | 0	1110 | 0	110 | 01

Table 2: GC for the integers 1 to 10

III. MODIFIED GOLOMB CODE (MGC)

In this section, a new variable-length integer code, Modified Golomb Code (MGC), is proposed to represent non-negative integers compactly. The proposed MGC is designed based on GC and EGC. In GC, the given number n (>0) is first divided by a divisor d to obtain the quotient (q) and the remainder (r). Then, the q and r of given n are used to generate codes. But, the number of bits required by unary in GC is more for large range of integers. Hence, GC has the drawback of requiring long-bit length to represent middle, large range of integers. In EGC, the given integer n (>0) is divided by a divisor d recursively until the last quotient becomes zero. The remainders (r_i) obtained in each division and the number of divisions (C) are used to generate codes. The drawback of EGC is that the divisions are made successively until the last quotient becomes zero whether the successive division gives better representation (less bits) or not. Hence, in MGC, if the number of bits needed to represent current quotient is less than the number bits required after the division (i.e bits requirements to represent next quotient, remainder and count), then the division will be stopped. Due to this condition, MGC can overcome the drawback of GC and to achieve better representation for large integers than EGC.

In MGC, the given integer n is divided by a divisor d ($2^m \leq d < 2^{m+1}$) successively until either the condition q_c becomes zero or

$$(q_c + C + C \times \log_2 d) < (q_{c+1} + 1 + (C+1) + (C+1) \times \log_2 d).$$

Alternatively, In MGC, successive division will be stopped when $q_c = 0$ or $(q_c < (q_{c+1} + 1 + \log_2 d)$. Here, q_c is the quotient obtained in C^{th} division. In MGC, all the remainders of n obtained by the divisor d are preserved r_i (i=1, 2...C). MGC has three parts to represent an given integer n: the quotient (q_c) , count (C) and remainders (r_i). The quotient (q_c) and the count (C) are encoded using binary code and unary code (described in section 2), respectively. The remainders r_i are coded using binary code. The format of MGC is given as: Binary Code (q_c) | Unary Code (C) | Binary Code ($r_c\, r_{c-1} \ldots r_1$).

A. Algorithm for MGC Integer Encoding

1) The non-negative integer n is divided by the divisor d ($2^m \leq d < 2^{m+1}$) repeatedly C times until any one of the following conditions is satisfied.
   - $q_c < (q_{c+1} + 1 + \log_2 d)$
   - $q_c = 0$
2) Count the number of divisions made as *C* and preserve the remainders produced in each division as $r_1, r_2 \ldots r_c$.
3) Encode the last quotient (q_c) obtained in step-1 and the count (*C)* obtained in step-2 using $\log_2(m+1)$ bits and unary code, respectively. The remainder r_i is coded in $\log_2\left(d - (2^m - 1)\right)$ bits when $q_c = 0$ && $C \geq 2$, in $\log_2(d-1)$ bits when $q_c = 0$ && C =1 and in $\log_2 d$ bits for all other cases. Then, the MGC for *n* is generated by combining the codes for q_c, *C* and r_i in the coding format given below:
   Binary Code (q_c) | Unary Code (C) | Binary Code (r_C , $r_{C-1} \ldots r_1$)
   Repeat steps 1- 3 for all the integers to be coded.

It is shown in the Table 3 that the possible last quotients and remainders for the divisors *d=3&4* if the proposed method is applied to represent integers from 1 to 255. It is observed from the Table 3 that the number of possible last quotients for d=3 is two (0,1), for d=4 it is three (0,1,2). Also, the last remainder is only 2 when $q_c=0$ and C ≥ 2 for d=3 and for d=4, it is only 3. These are the unique pattern occurred due the condition given in the algorithm. The same trend happens for other devisors also. According to this, the remainders, the last quotient and the last remainder are coded as given in the encoding algorithm.

n	***d=3***			***d=4***		
	q_c	r_c	***c***	q_c	r_c	***c***
1	0	1	1	0	1	1
2	0	2	1	0	2	1
3	1	0	1	0	3	1
4	1	1	1	1	0	1
5	1	2	1	1	1	1
10	1	1	2	2	2	1
15	1	2	2	0	3	2
25	0	2	2	1	2	2
50	1	2	3	0	3	3
100	1	0	4	1	2	3
200	0	2	5	0	3	4
255	1	0	5	0	3	4

Table 3: The last quotients and remainders of MGC for the integers 1 to 255

n	**MGC**	
	d=3	***d=4***
1	0|1|0	0| 1 | 0
2	0|1|1	0| 1| 10
3	1|1|0	0 | 1| 11

4	1|1|10	10 | 1 | 00
5	1|1|11	10 |1 | 01
6	0|01|0	10 | 1 | 10
7	0|01|10	10 | 1 | 11
8	0|01|11	11 | 1 | 00
9	1|01|00	11 |1 | 01
10	1|01|010	11 | 1 | 10

Table 4: MGC for the integers 1 to 10

Illustration: n = 50, d = 3

$50/3 \Longrightarrow q_1 = 16, \quad r_1 = 2 \quad C_1 = 1$

$16/3 \Longrightarrow q_2 = 5, \quad r_2 = 1 \quad C_2 = 2$

$5/3 \Longrightarrow q_3 = 1, \quad r_3 = 2 \quad C_3 = 3$

Here, Since $q_3 < q_3/3 + 1 +$ Code (remainder of $q_3/3$), dividing further is stopped.

MGC(50) = Binary Code (q_3) | Unary Code (C) | Code (r_3, r_2, r_1)

= Binary Code (1) | Unary Code (3) | Code (2, 1, 2)

= 1| 001| 111011 Use Algorithm. (for d =3, remainders 0 (0), 1(10), 2(11))

Table 4 shows the MGC for integers 1 to 10 for *d* = 3 and 4

B. Algorithm for MGC Integer Decoding

The following steps are used to decode the compressed data.

1) Read $\log_2 m$ bits and decode the bits into respective last quotient and assign into q_c.
2) Read the *C* bits until bit '1' is encountered, which is used to read the *C* number of remainders.
3) Then, read $\log_2\left(d-(2^m-1)\right)$ bits if q_c =0 and $C \geq 2$ (else) $\log_2(d-1)$ bits if q_c =0 and $C = 1$ (else) $\log_2 d$ bits for all other cases and decode the first remainder. Then, read $((C-1) \times \log_2 d)$ number of bits further to decode $(C-1)$ remainders.

Repeat steps 1- 3 for all the integers to be decoded.

Decode: 1| 001| 111011 ; $d = 3$, $C = 3$

$q_3 = 1$; $C = 3$; Codes: 11,10, 11 denote the remainders $r_3 = 2$, $r_2 = 1$, $r_1 = 2$, respectively.

$q_2 = q_3 \times d + r_3$ ($C = 3$) (the value of $q_3 = 1$ & $r_3 = 2$)

$q_2 = 1 \times 3 + 2 \quad \therefore \; q_2 = 5$

$q_1 = q_2 \times d + r_2$ ($C = 2$) (the value of $q_3 = 5$ & $r_2 = 1$)

$q_1 = 5 \times 3 + 1 \quad \therefore \; q_1 = 16$

$n = q_1 \times d + r_1$ ($C = 1$) (the value of $q_1 = 16$ & $r_1 = 2$)

$n = 16 \times 3 + 2 = 50$.

In general, the given integer *n* is decoded using eq.(2) .

$$n = \sum_{i=c}^{1} (q_i \times d + r_i) \qquad (2)$$

IV. BIT-LENGTH COMPARISON

The bit lengths of MGC, GC and EGC for divisor (*d*=3 and 4) have been calculated and are given in Table 5. It is observed from Table 5 that GC gives compact representation for small range of integers (*i.e* 1 - 10) and gives poor representation for other range of integers.

n	***d=3***		***d=4***		
	MGC	**GC**	**EGC**	**MGC**	**GC**	**EGC**
1	3	3	2	3	3	2
2	3	3	2	4	3	3
3	3	4	4	4	3	3
4	4	4	4	5	3	5
5	4	4	5	5	4	5
10	6	5	6	5	5	6
15	6	7	7	5	6	6
25	8	10	8	8	8	8
50	10	19	10	8	15	9
100	11	35	12	11	27	11
200	13	69	13	11	52	12
255	13	87	14	11	66	12

Table 5: Bit length of comparison of MGC, GC and EGC

MGC offers significantly better representation for small to large range of integers. For small values, MGC is one bit longer than GC. But, GC requires more bits than MGC for mid-range values and large values. It is also observed from Table 5 that MGC achieves better representation significantly than EGC for large integers.

V. EXPERIMENTAL RESULTS AND DISCUSSION

Variable length integer codes (VLC) have been used to compress text data [11], medical data [12] and remote sensing data [13]. In this section, as an application of MGC, MGC is used as the final stage coder of BWT compressor for text data compression as shown in Figure 1. BWT compressor has four stages as shown in Figure 1. In first stage of BWT compressor, BWT computes the permutation of the given input. Then, move-to-front (MTF) coder encodes the output of first stage of BWT. After this, the output of MTF will be encoded by run-length encoding(RLE). In the final stage, the output of RLE will be encoded by the VLC coders. In the experiment, Calgary corpus dataset [9] is used to test the performance of MGC. The calgary corpus dataset contains both text files (bib, book1, book2, news, paper1, paper2, paper3, paper6, progc, progl, progp, trans and binary files (geo, obj1, obj2, pic). Compression rate given in equation (3) is used as a metric for performance evaluation. The compression results of MGC are compared with the results of GC and EGC as given in Table 6.

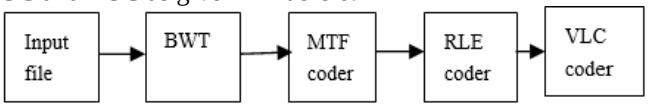


Fig. 1: Stages of Burrows-Wheeler Compressor

$$\text{Compression rate} = \frac{\text{Size of the Compressed file}}{\text{Number of symbols in the input file}} \qquad (3)$$

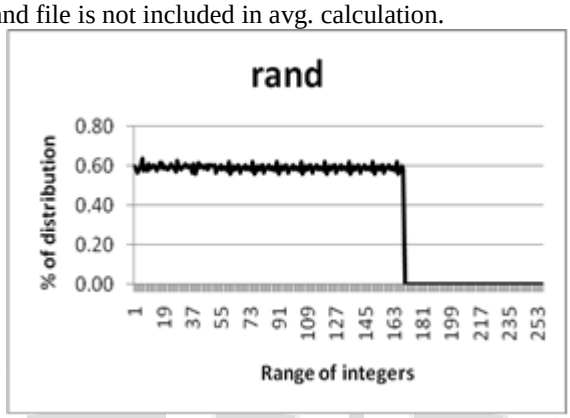
It is observed from Table 6 that MGC achieves low compression rate on an average than GC. GC provides better compression rate for text files (bib, book1, book2, news, paper1-paper6) of calgary corpus and gives poor compression rate when *d* is increased. But, it achieves better results for binary files when *d* is large. When MGC is compared with GC, MGC gives better results for both text and binary files than GC (*d=8,16*) with large divisor. For small divisor (*d=4*), GC performs better than MGC for some of the text files and gives poor performance for binary files. However, when a

large divisor is selected, GC may achieve better result for binary files, but it cannot achieve better results for text files compared to MGC. The reason is that GC needs more bits for small integers when *d* is large and require more bits for large integer when *d* is small. Since the output of BWT compressor contains more small range of integers and less middle range of integers for text files; and contains all range of integers (small, middle and large integers) for binary files including rand file, GC with small divisor can perform well at some extent for text files and might not perform well for binary files when compared to MGC. The distributions of the integers for sample files (bib, geo, rand) are shown in the Fig.2. It is observed from Fig.2 that bib file shows the significant distribution of small range of integers and other files (rand, geo) show the significant distribution of small, middle and large range of integers. It is concluded that GC could not perform well for the files which contain significant distribution of small, middle and large range of integers. Both MGC and EGC could perform well for both text files and binary files. Since MGC can obtain better representation of middle and large range of integers as shown in Table 5 than EGC, MGC could perform well than EGC when the collections contain more middle and large range integers. It is observed from Table 6 that MGC obtains better compression performance than EGC for geo, obj1, pic and rand files. But, it is inferior to EGC for text files. Finally, it is concluded that the performances of the codes depend on the distributions of integers. The proposed code could perform well for the collections which contain significant distributions of middle and large range integers comparing to GC and EGC.

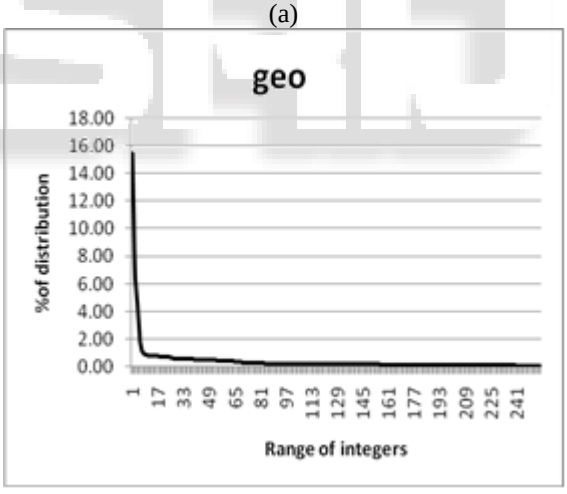
Corpus	**GC**			**MGC**			**EGC**
	d=4	**d=8**	**d=16**	**d=3**	**d=4**	**d=2**	**d=3**
bib	2.362	2.576	2.999	2.354	2.533	2.219	2.216
book1	3.130	3.600	4.301	3.256	3.586	3.077	3.074
book2	2.751	3.097	3.659	2.775	3.032	2.573	2.588
geo	11.120	7.350	5.873	5.411	5.348	6.152	5.563
news	3.268	3.412	3.889	3.120	3.340	2.997	2.972
obj1	7.726	5.529	4.978	4.279	4.287	4.628	4.294
obj2	4.416	3.605	3.512	2.898	2.993	2.878	2.785
paper1	2.985	3.266	3.807	2.951	3.188	2.752	2.767
paper2	2.881	3.246	3.839	2.932	3.197	2.745	2.755
paper3	3.160	3.530	4.158	3.220	3.493	3.042	3.045
paper5	3.092	3.231	4.525	3.646	3.888	3.491	3.466
paper6	3.111	3.385	3.934	3.048	3.289	2.828	2.850
pic	0.942	0.861	0.911	**0.827**	**0.862**	0.884	0.834
pogc	3.092	3.231	3.681	2.928	3.134	2.763	2.757
progl	2.145	2.352	2.744	2.100	2.266	1.901	1.940
progp	2.222	2.379	2.740	2.101	2.248	1.873	1.919
trans	1.983	2.127	2.453	1.881	2.007	1.669	1.723
rand	23.586	14.045	9.789	9.940	**9.586**	12.069	10.490
Comp. Rate *	**3.552**	**3.340**	**3.670**	**2.925**	**3.099**	**2.851**	**2.797**

Table 6: Compression performance (bits per symbol) of GC and MGC

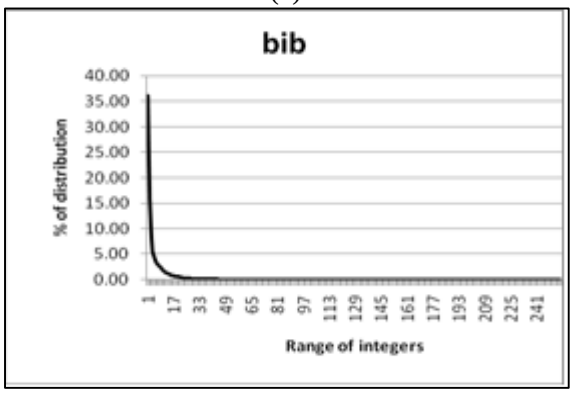
* rand file is not included in avg. calculation.



(a)



(b)



(c)

Fig. 2: The distribution of integers in (a) rand, (b) geo (c) bib

VI. CONCLUSION

In this paper, a new variable-length integer code is proposed. It has been designed based on GC and EGC. GC provides better representation for small range of integers. But, the proposed MGC offers competitive and compact representation for small, mid-range and large integers when compared to GC and EGC. The overall performance of MGC is better than GC. The performance of MGC for text compression on calgary corpus dataset is tested and the experimental results show that MGC performs better than GC and EGC when the collections which contain significant distributions of middle and large range integers.

REFERENCES

[1] D. Salomon. Variable-length Codes for Data Compression. Springer-Verlag, London, pp. 69-100, 2007.
[2] Huffman D.A, 1952: "A method for the construction of minimum-redundancy codes", Proceedings of the Institute of Radio Engineers, Cambridge Vol.40, pp.1098-1101.
[3] [Shannon C.E, 1948: A Mathematical Theory of Communication, Bell System Technical Journal, Vol.27, pp.379-423, 623-656.
[4] S. David, "Data Compression Book", 2nd ed. New York: Springer-Verlag, 2004, pp. 41–11.
[5] K. Somasundaram and S. Domnic, "Extended golomb code for integer representation", IEEE Transactions on Multimedia, vol. 9, no. 2, pp239–246, 2007.
[6] Golomb S.W, 1966: "Run-length encodings", IEEE Transactions on Information Theory, Vol.12 (3), pp.399-401.
[7] S. Buttcher, C. L. A. Clarke, and G. V. Cormack. Information Retrieval: Implementing and Evaluating Search Engines. MIT Press, Cambridge MA, 2010.
[8] Witten, Ian Moffat, Alistair Bell, Timothy. "Managing Gigabytes: Compressing and Indexing Documents and Images." Second Edition. Morgan Kaufmann Publishers, San Francisco CA. 1999.
[9] Burrows M, Wheeler D, 1994: "A block sorting lossless data compression algorithm", Technical Report 124, Digital Equipment Corporation.
[10] Witten I.H, Bell T, 1990: "The Calgary / Canterbury Text Compression Corpus"
[11] Peter Fenwick, "Burrows–Wheeler compression with variable length integer codes," Softw—Pract. Exper., vol. 32, no. 13, pp. 1307–1316, Nov. 2002.
[12] SM Basha, BC Jinaga , " A Novel Optimized Golomb-Rice Technique for the reconstruction in Lossless Compression of Digital Images", ISRN Signal Processing, Vol.2013, 2013.
[13] Jing-Jing Zheng et.al, Fast algorithm for remote sensing image progressive compression", IEEE IGARSS, Honolulu, HI, 2010.

