SEQUENCE DYNAMIC CODE FOR STREAM OF TWO BYTE DATA (16-BIT) COMPRESSION[+]

Mohammed Mustafa Siddeq[*]

Abstract

This paper is concerned with 16-bit stream data compression by using sequence dynamic code, with the minimum code of 4-bit and maximum of 19-bit for 65536 probabilities, and this means compressing 16-bit data to 4-bit depending on file size. In this algorithm Start Code is used to compress many strings to length of 4-Bit, if the probability of strings is 15-string or less. And the Dynamic Code is used when the probability of strings more than 15-strings, the maximum length reached by the dynamic code is 19-Bit; also this paper describes the header file size for this algorithm. The computer tests show the results on two types of text The results are compared with Huffman and Shannon Fanon coding by using compression performance.

المستخلص

هذا البحث يتعامل مع كبس البيانات ذات الحجم (16-bit) بأستخدام الرمز الديناميكي المتسلسل اذ تقوم الخوارزمية بتقليص الحجم ما بين (4-bit) و (19-bit) أذا كان أحتمالية البيانات 65536 وهذا يعتمد على حجم الفايل المراد ضغط .في هذا البحث يوضح أستخدام Start Code الذي يقلص حجم الى (4-Bit) أذا كانت أحتمالية البيانات أكثر من 15 أحتمال أو اقل.وفي حال أحتمالية البيانات أكثر من 15 أحتمال في هذه الحالة يجب أستخدام Dynamic Code الذي يصل حجمه الى (19-Bit)، والمخطط الأنسيابي يوضح كيفية عمل الخوارزمية. ويوضح هذا البحث كيفية تقليص المعلومات الرئيسية حول الفايل (Header file) . وطبقت هذه الخوارزمية على نوعين من العبارات ذات الحجم مختلف بأستخدام الكومبيوتر وأيضاً قورنت هذة الخوارزمية مع Huffman coding و Shannon fanon coding بأستخدام معادلات تحسب كفاءة ضغط البيانات للخوارزميات المستخدمة في هذا البحث وتبين أن الطريقة المستخدمة أفضل من الطريقتين المذكورتين في تقليص حجم .

Introduction

Data compression is achieved by reduction but this also makes the data less reliable, more prone to errors. Making data more reliable, on other hand, is one by adding check bits and parity bits. A process that increases the size of code, thereby increasing recurrence. Reliability is thus the opposite, and it is interesting to note that the latter is relatively recent field, whereas the former existed even before the advent of computers [1,9]. The Braille code of 1820, and Morse coding of 1838 use simple compression. It therefore seems that reducing

[+] **Received on13/1/2005 , Accepted on 29/6/2005**
[*] **Assistant Lecturer/Technical College / Kirkuk**

recurrence comes naturally to anyone who works on codes [8,10]. Today these methods are mostly of historical interest, since they are generally inefficient and cannot compete with the modern compression methods developed during the past 20 years [10,7].

The Braille was invented in 1820 is still in common use today, after being modified several times. This code consists of group of dots (matrix 3×2) embossed on thick paper [6, 8]. Each of 6 dots in a group may be flat or rose, so the information content of a group is equivalent to 6-bits, resulting in 64 possible groups. Another old code worth mentioning is Baudot code, this was a 5-bit code developed by J. M. E. Baudot in1880 for telegraph communication. It became popular and by 1950, was designated as international telegraph code.

It was used in 1^{st} and 2^{nd} generation computers [2, 8]. This code is not reliable because no parity bit is used for the dictionary code can be compressed using the concept of front compression. This is based on the observation that adjacent words in such a list tend to share some of their initial characters [1]. A word can be compressed by dropping the *n* character which is shared with its predecessor in the list and replacing with *n*[3]. The MacWrite word processor uses a special 4-bit code to code the most common 15 characters *"_12asdfghjklzxc"* plus an escape code. Any of these 15 characters are encoded by 4-bits. Any other characters are encoded as the escape code followed by the 8-bit ASCII, code of the characters, a total of 12-bits. Each paragraphs is coded separately and, if these results in expansion, the paragraph are stored as plain ASCII one bit is added to each paragraph to indicate whether or not it uses compression [7,8,9].

The sequence dynamic code is very interactive for data compression . In this paper the algorithm is tested with two different sizes of text and the results show the performance of this algorithm (See section 3). The compression algorithm depends on three points: (1) Constant start code (2) Generate dynamic code (3) Minimize header file size (information about compressed file).

Compression Algorithm

The first step in this algorithm is to generate *start code* which is used to recognize the codes and to reduce the header file size. The second part of this algorithm is to generate dynamic code for 16-bit data, starting from 1-bit and finishing at 15-bit (without the start code).

Start Code Representation

The start code constant for all typs of files and the size for start code (4-bit) are shown in Table 1.

Table 1. The start code and dynamic code

String Level "String size 16-bit"	Start Code	Dynamic Code	Number of Bits "Compressed String"
0	1000	-	4-Bit
1	1001	X	5-Bit
2	1010	XX	6-Bit
3	1011	XXX	7-Bit
4	1100	XXXX	8-Bit
5	1101	XXXXX	9-Bit
6	1110	XXXXXX	10-Bit
7	1111	XXXXXXX	11-Bit
8	0000	XXXXXXXX	12-Bit
9	0001	XXXXXXXXX	13-Bit
10	0010	XXXXXXXXXX	14-Bit
11	0011	XXXXXXXXXXX	15-Bit
12	0100	XXXXXXXXXXXX	16-Bit
13	0101	XXXXXXXXXXXXX	17-Bit
14	0110	XXXXXXXXXXXXXX	18-Bit
15	0111	XXXXXXXXXXXXXXX	19-Bit
	Total = 65535 Byte		

In the above table the *"Level"* is representing the higher priority string to give it a minimum bit (4-Bit). The dynamic code depends on the *"Start Code"* section, "X" in *"Dynamic code"* section represents (0, 1) sequentially, for example assume the code "1001X" it is meaning "10010" and "10011" the "X" its one bit, because the start code "1001" at level one. Assume two strings size each one of them 16-bit "a_1a_2" to compress these two strings by using above table yields "10010" and "10011", the 1st code "10010" for "a_1" and 2nd code 10011 for "a_2". The total compressed size for the string "a_1a_2" is 10-bit. For decompression the algorithm reads first the start code (4-bit) to recognize the level then reads the number of bits according to string level. Also we can use the start code for compression without using dynamic code if the number of the strings is 16-string after computing the recurrences for it.

Assume the following example: - "ABCDEFFFFF"

Compression algorithm is applied on the above text (size 10-Byte) and the size for the text after compression is 2-Byte, as shown in Table 2.

Table 2. Compression algorithm

Level	Strings 16-Bit	Start code 4-Bit	Dynamic code	Recurrence of strings	Compressed size
0	FF	1000	NON	2	4-Bit
1	AB	1001	NON	1	4-Bit
2	CD	1010	NON	1	4-Bit
3	EF	1011	NON	1	4-Bit
		Compressed Size = 2-Byte			

Generation Of Dynamic Code

Dynamic code is generated when the number of strings in a file exceeds 16 levels, the dynamic code generated sequentially depends on start code sequence (See Table 1.). Consider the following example:-

"01122334567890abcdefghijkmnopqrstuvwxyzz"

The text size of 40-byte, the dynamic code is applied to this text and the results are shown on Table 3.

Table 3. Dynamic code application

Level	Strings 16-Bit	Start code 4-Bit	Dynamic code	Recurrence of strings	Compressed size
0	01	1000	**-**	1	4-Bit
1	12	1001	**0**	1	5-Bit
2	23	1001	**1**	1	5-Bit
3	34	1010	**00**	1	6-Bit
4	56	1010	**01**	1	6-Bit
5	78	1010	**10**	1	6-Bit
6	90	1010	**11**	1	6-Bit
7	ab	1011	**000**	1	7-Bit
8	cd	1011	**001**	1	7-Bit
9	ef	1011	**010**	1	7-Bit
10	gh	1011	**011**	1	7-Bit
11	ij	1011	**100**	1	7-Bit
12	km	1011	**101**	1	7-Bit
13	no	1011	**110**	1	7-Bit
14	pq	1011	**111**	1	7-Bit
15	rs	1100	**0000**	1	8-Bit
16	tu	1100	**0001**	1	8-Bit
17	vw	1100	**0010**	1	8-Bit
18	xy	1100	**0011**	1	8-Bit
19	zz	1100	**0100**	1	8-Bit
		Total = 16.75 Byte			

The size of the text after compression is 16.75 byte, the minimum bits are 4-bit and maximum bits are 8-bit. This means the strings of (16-bits) are compressed to 4-bit, 5-bit, 6-bit, 7-bit and 8-bit sequentially. The level zero has 4-bit (start code) without dynamic code it is constant for all types of files, to obtain maximum compression rate. We compute the performance of compression for above example using the equation [1,10]:-

$$Compression\ Rate = \frac{Size\ after\ compression}{Size\ before\ compression} \quad (1)$$

$$Compression\ Performance = [100(1 - Compression\ Rate)]\% \quad (2)$$

The compression performance is 58% for the above example, this means the algorithm saves 58% from the text. Figure -1 shows the compression and decompression for the sequence dynamic code algorithm for text *"$b_1b_2b_3b_4b_5b_6$"*.

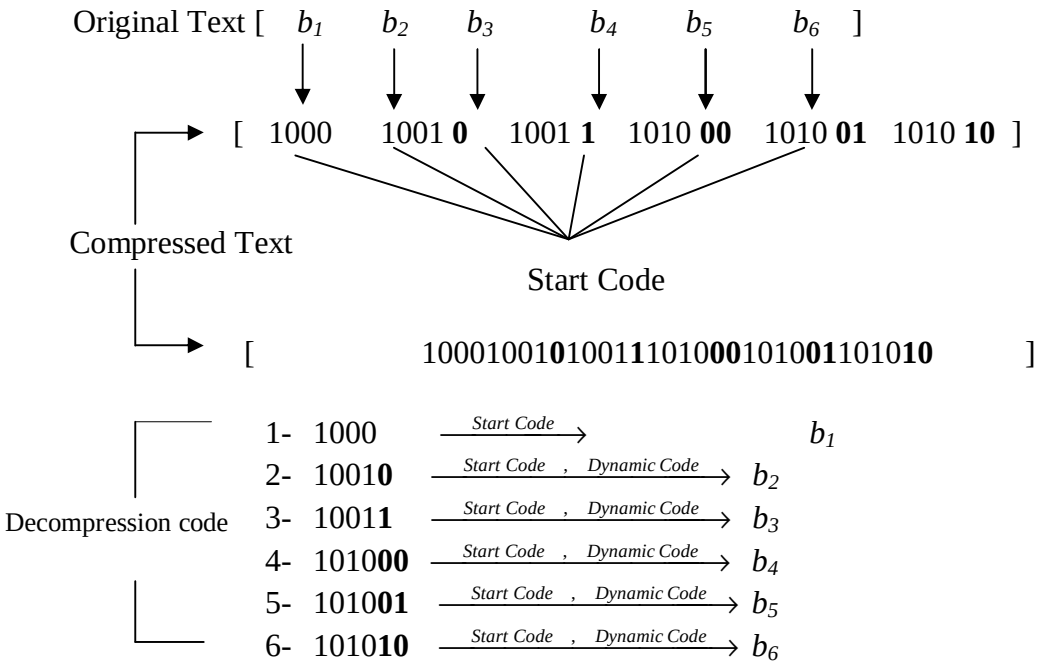


Figure -1. Illustrates Compression and Decompression
For compression algorithm

The decompression for the algorithm first reads the 4-bit (start code) then reads the bits according their level from the start code, at last the data are retrieved from the header file (See Section 4). The Compression algorithm can be illustrated in the following steps:-

Compression Algorithm

- Open file for compute the recurrences:-

```
  Begin
        Integer String[65535];
          fopen original file(f1);

 While (not end of file) do
 {
    B1 = fgetc(f1); B2=fgetc(f1);
    Found=Search for bytes(B1,B2,String);
    IF (Found) THEN
      String[Found]++;
    ELSE
      {  Pointer++;
         String[Found].Flag=1;
      }
  }
 END;
```

-Sort Array "String" according to maximum:-

```
 Begin
    For I=1 to 65536 Do
      For J=I+1 to 65535 Do
```

```
    {
        IF (String[I] <= String[J]) THEN
          SWAP (I,J,String);
     }
  END;
```

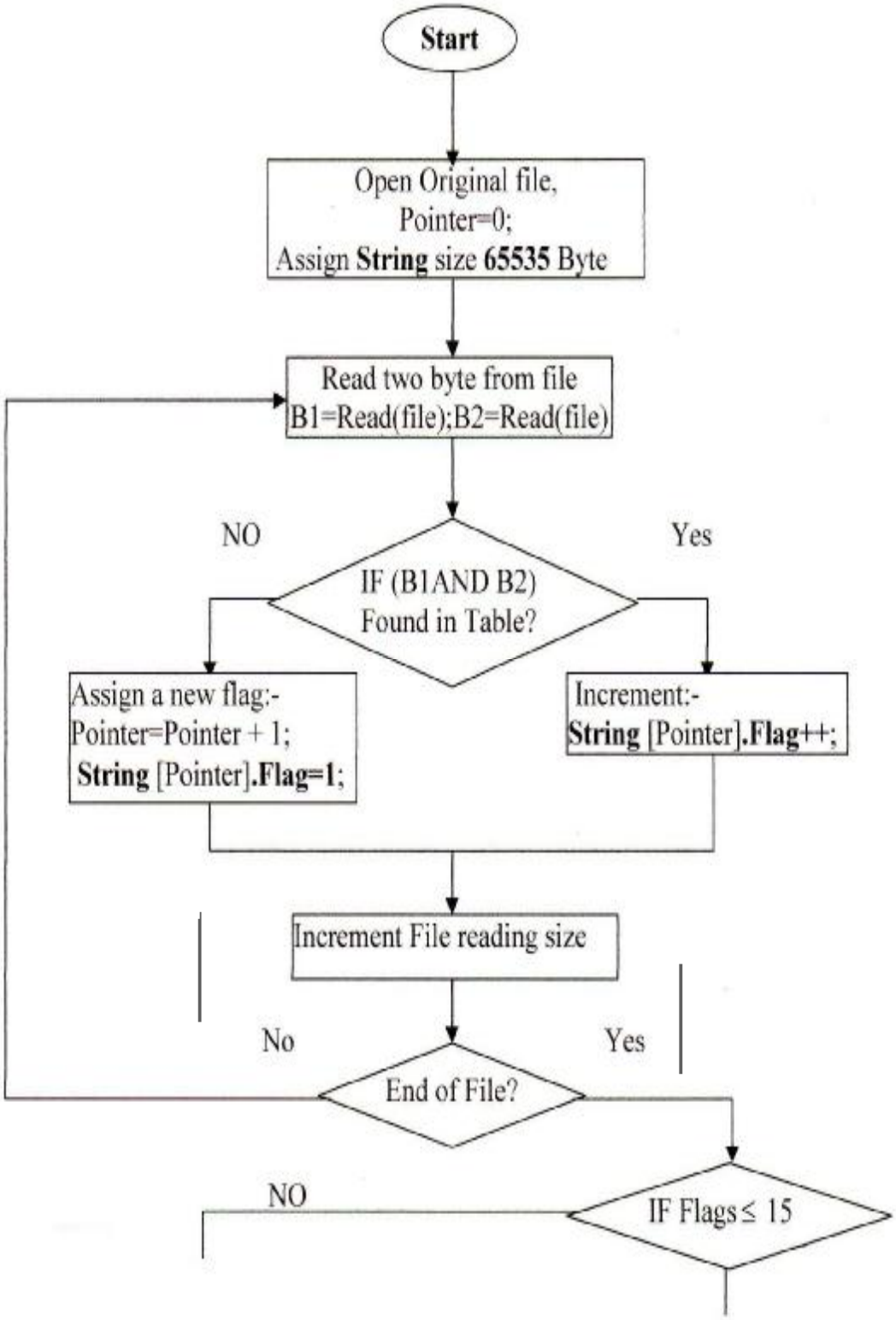
- Assign Code for "string" by using "Table":-

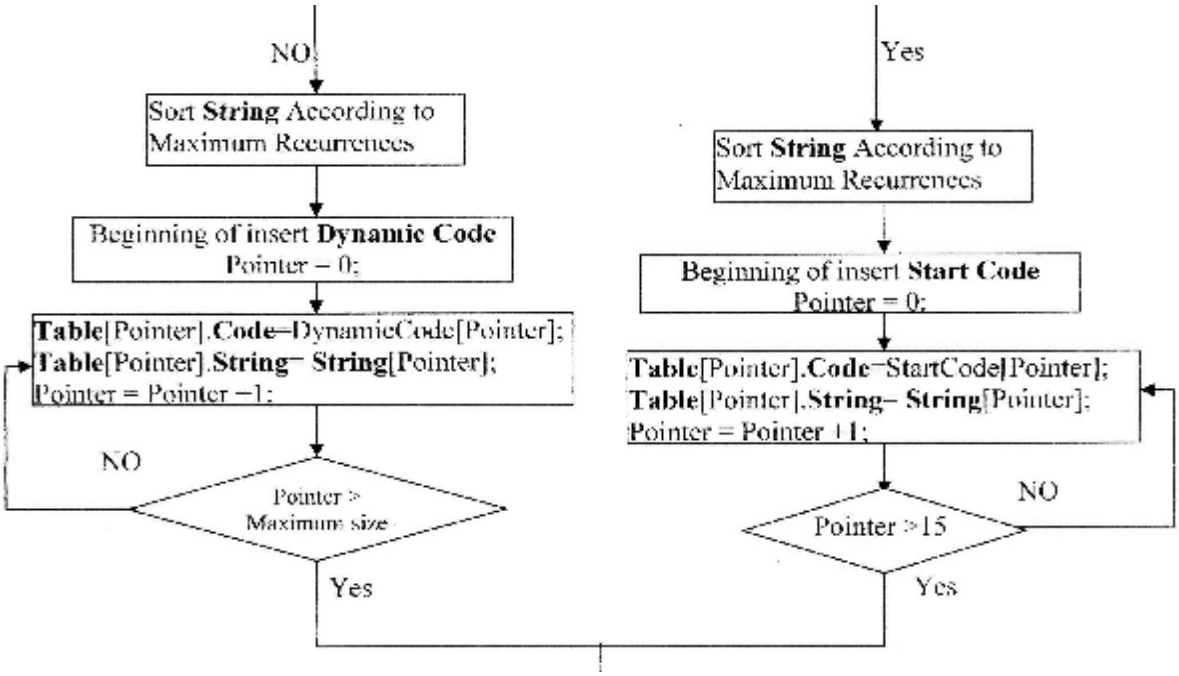
```
  Begin
    Integer StratCode[15];
    Integer Dynamic Code[65536];
      IF (Pointer <= 15) THEN
      {
         For I=1 to Pointer Do
          Table[I].Code=StartCode[I];
          Table[I].String=String[I];
      }
      ELSE
       {  For I= 1 to Pointer Do
          Table[I].Code=Dynamic Code[I];
          Table[I].String=String[I];
       }
END;
```

-Store Code in a file (Compression file) according to original file:-

```
 BEGIN
   Create a new Compression file(f1);
  fopen original file (f2);
  While (not end of file (f2)) do
   {
    B1=fgetc(f2); B1=fgetc(f2);
    Found=Search for byte(B1,B2,Table);
    Put to File(Table[Found],Code,f1);
   }
END;
```

In the above algorithm the function "*Found=Search for bytes(B1,B2,String)*" is used for searching for two bytes (B1,B2) in array "*String*" and return the index otherwise return "-1". The function "*SWAP (I,J,String)*" is used to exchange the content of location (i, j) in array "*String*", at last the function "*Put to File(Table[Found],Code,f1)*" stores the code in "*Bits*" at a file(See Section 4). The flowchart for compression algorithm is shown in Figure – 2.


Start
Open Original file,
Pointer=0;
Assign String size 65535 Byte
Read two byte from file
B1=Read(file);B2=Read(file)
NO
Yes
IF (B1AND B2)
Found in Table?
Assign a new flag:-
Pointer=Pointer + 1;
String [Pointer].Flag=1;
Increment:-
String [Pointer].Flag++;
Increment File reading size
No
Yes
End of File?
NO
IF Flags ≤ 15



(Continue)

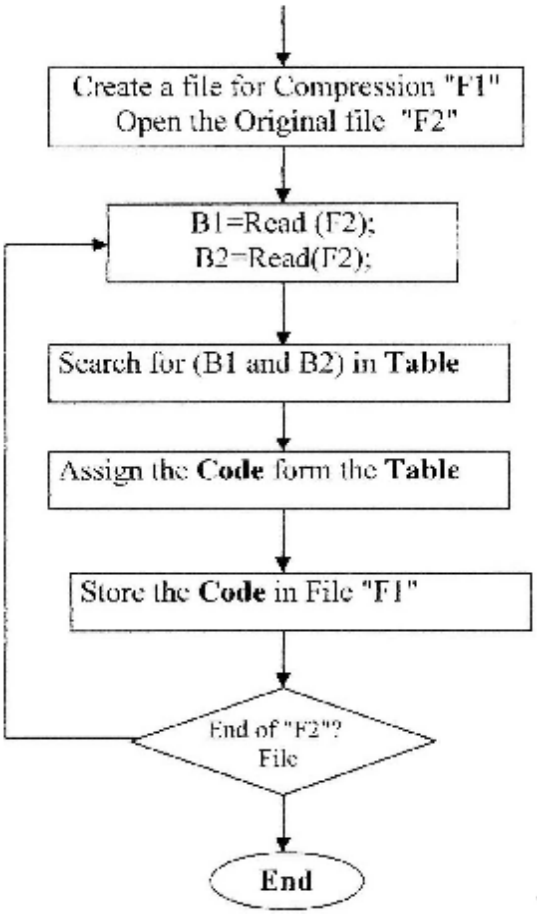


Figure – 2 Compression Flowchart

Computer Test

The algorithm is applied to two different sizes of text, also Huffman and Shannon Fanon methods are applied to these texts to compare with our algorithm by using compression performance.

Results

Assume the first text:- **"HGGFFFEEEEDDDDDCCCCCCBBBBBBBAA"**

The text size is 30-Byte, the results are shown on Table 4.

Table 4. Result for first text.

Level	Strings 16-Bit	Start code 4-Bit	Dynamic code	Recurrence of strings	Compressed size
0	BB	1000	NON	3	4-Bit
1	CC	1001	NON	2	5-Bit
2	EE	1001	NON	2	5-Bit
3	DD	1010	NON	2	6-Bit
4	HG	1010	NON	1	6-Bit
5	GF	1010	NON	1	6-Bit
6	FF	1010	NON	1	6-Bit
7	DC	1011	NON	1	7-Bit
8	CD	1011	NON	1	7-Bit
9	AA	1011	NON	1	7-Bit
		Total Size = 5 Byte			

After computing the repeated strings the number of strings on the above table is 10 strings, therefore *start code* is used as a compression code without needing dynamic code. The compression performance (1) is shown in Table 5.

Table 5. Compression performance for first text.

Text size before compression	text size after compression	Compression performance
30 Byte	5 Byte	83%

Assume the second text:-
"My_Name_is_Mohammed_Mustafa_Siddeq,I am_ _29_years_old,_and_I am_ a teacher_in_Technical_Collage_/_Software_Engineering".

The text size is 115-Byte, Table 6 shows the result for the second text with need for dynamic code.

Table 6. Result for second text.

Level	Strings 16-Bit	Start code 4-Bit	Dynamic code	Recurrence of strings	Compressed size

0	am	1000	**-**	3	4-Bit
1	e_	1001	**0**	2	5-Bit
2	d_	1001	**1**	2	5-Bit
3	My	1010	**00**	1	6-Bit
4	is	1010	**01**	1	6-Bit
5	_M	1010	**10**	1	6-Bit
6	oh	1010	**11**	1	6-Bit
7	me	1011	**000**	1	7-Bit
8	_N	1011	**001**	1	7-Bit
9	Mu	1011	**010**	1	7-Bit
10	st	1011	**011**	1	7-Bit
11	af	1011	**100**	1	7-Bit
12	a_	1011	**101**	1	7-Bit
13	si	1011	**110**	1	7-Bit
14	dd	1011	**111**	1	7-Bit
15	eq	1100	**0000**	1	8-Bit
16	,I	1100	**0001**	1	8-Bit
17	_I	1100	**0010**	1	8-Bit
18	n_	1100	**0011**	1	8-Bit
19	29	1100	**0100**	1	8-Bit
20	_y	1100	**0101**	1	8-Bit
21	ea	1100	**0110**	1	8-Bit
22	rs	1100	**0111**	1	8-Bit
23	_o	1100	**1000**	1	8-Bit
24	ld	1100	**1001**	1	8-Bit
25	,_	1100	**1010**	1	8-Bit
26	an	1100	**1011**	1	8-Bit
27	Ia	1100	**1100**	1	8-Bit
28	m_	1100	**1101**	1	8-Bit
29	te	1100	**1110**	1	8-Bit
30	ac	1100	**1111**	1	8-Bit
31	he	1100	**00000**	1	9-Bit
32	r_	1110	**00001**	1	9-Bit
33	in	1110	**00010**	1	9-Bit
34	_T	1110	**00011**	1	9-Bit
35	ec	1110	**00100**	1	9-Bit
36	hn	1110	**00101**	1	9-Bit
37	ic	1110	**00110**	1	9-Bit
38	al	1110	**00111**	1	9-Bit
39	_C	1110	**01000**	1	9-Bit
40	ol	1110	**01001**	1	9-Bit
41	la	1110	**01010**	1	9-Bit
42	ge	1110	**01011**	1	9-Bit
43	_/	1110	**01100**	1	9-Bit
44	_S	1110	**01101**	1	9-Bit
45	of	1110	**01110**	1	9-Bit
46	tw	1110	**01111**	1	9-Bit
47	ar	1110	**10000**	1	9-Bit
48	En	1110	**10001**	1	9-Bit
49	gi	1110	**10010**	1	9-Bit
50	ne	1110	**10011**	1	9-Bit
51	er	1110	**10100**	1	9-Bit
52	in	1110	**10101**	1	9-Bit
53	g	1110	**10110**	1	9-Bit
			Total Size =53.6 Byte		

The compressed size is 53.6-Byte, the minimum bits are 4-bit and the maximum bits are 9-bit. Performance of the compression is shown on Table 7.

Table 7. Compression performance for second text.

Text size before compression	text size after compression	Compression performance
115 Byte	53.6 Byte	54%

Comparison methods

Two types of traditional algorithms (Huffman and Shannon fanon) are used . These two algorithms are applied to same texts for comparison with our algorithm. Table 8 shows the coding for the two methods applied to first text [10,4,5] (See Section 3.1).

Table 8. Huffman coding and Shannon coding

character	Count number of characters	Huffman coding	Shannon coding
B	7	01	11
C	6	00	10
D	5	111	011
E	4	101	010
F	3	100	001
G	2	1101	0001
A	2	11001	00001
H	1	11000	00000

The compression size and the performance for the two methods are compared with our algorithm and shown in Table 9.

Table 9. First text compared with sequence dynamic code**.**

Methods	Huffman coding	Shannon coding	Sequence dynamic coding
Size	10.6 Byte	10.6 Byte	5 Byte
Performance	64%	64%	83%

Also are applied to two methods the second text (See Section 3.1) and the results are shown to Table 10.

Table 10. Huffman coding and Shannon coding

Character	Count number of characters	Huffman coding	Shannon coding
_	15	000	01
a	12	0010	10
e	11	0011	11
i	7	0100	001
n	7	0101	010
m	6	0110	011
s	5	0111	0001

d	5	10000	0010
o	4	10001	0011
r	4	10010	00001
l	4	10011	00010
c	3	10100	00011
M	3	10101	000001
h	3	10110	000010
t	3	10111	000011
y	2	11000	0000001
f	2	11001	0000010
I	2	11010	0000011
u	1	11011	00000001
q	1	11100	00000010
2	1	11101	00000011
9	1	111100	000000001
T	1	111110	000000010
C	1	11111100	000000011
w	1	11111101	0000000001
E	1	11111110	0000000010

The compression performance and size by using the two methods are compared with our algorithm and shown in Table 11.

Table 11. Second text compared with sequence dynamic code.

Methods	Huffman coding	Shannon coding	Sequence dynamic coding
Size	57.8 Byte	51.8 Byte	53.6 Byte
Performance	49.7%	54.9%	53.3%

Header Size for Compression Algorithm

Sequence dynamic code algorithm needs to store the information about the original characters and dynamic codes. The size for header file is very important to reduce the size for a file, when the file has much type of 16-bits this leads to increase the file size. One way to reduce the header size is to make comparison between the characters, as shown in following text:-

"**FFFFBBBAab**"

The header for above text contains the following information:-

Character	ASCII	Comparison by ASCII
F	70	70
B	66	4
A	65	5
a	97	-27
b	98	-28

The above information means that the character "F" has maximum recurrence in the text and the ASCII for it is "70", then the algorithm takes the difference of the ASCII between "F" and the next character "B" the difference between them is "4", after that the difference is taken between "B" and the next character "A" (according to above information). The difference between them is "5" and so on. These numbers are reduced to less than 8-bit. In the above information the first number "70" is called *BASE*, because the other character depends on it. The other numbers are stored in form less that 8-bit, for example the numbers (4,5,-27,-28) from the above information can be stored as "01000110010011011111100" the first 8-bit is base and others represented as variable size bit. The simple header for the above text is shown in Table 12.

Table 12. Simple header

Character	ASCII	Required Bit
F	70	8-Bit
B	4	4-Bit
A	5	4-Bit
a	-27	5-Bit
b	-28	5-Bit

Table 13. Final Header

ASCII	Required Bit
70	8-Bit
4	4-Bit
5	4-Bit
-27	5-Bit
-28	5-Bit
size to store 26-bit	size to store 26-bit

The "Required Bit" in the header means the required bits needed to store the information about the text. When the ASCII value in the above table is positive; the number of bits needed must be "EVEN", otherwise when the ASCII value is negative the number of bits is "ODD". The header does not need to store the original characters (8-bit), just it needs to store the "ASCII" and "Required bits", the final header is shown in Table 13.

The *Dynamic code* and *Start code* are not needed to store on header, because they are sequentially coded. The header needs to store the strings (16-Bit) sequentially according to "ASCII". For decompression the algorithm reads the strings (16-Bit) by using "ASCII" then recognizes the dynamic code according to the level of that string (16-Bit) resident on it. The size of the above header (Table 13.) is 52-Bit. Figure-2 shows how the header is stored in disk

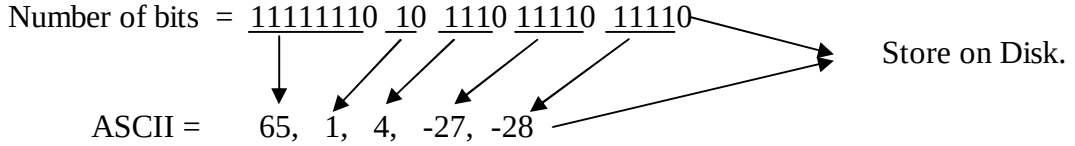


Figure -2. Stored header on dis

The header information about the first text (See Section 3.1) is shown in Table 14 and it is size for it is 50-Bit (6.2 Byte).

Table 14. Header information about first text.

ASCII	Required of bits	
65	8-Bit	← B
1	2-Bit	← C
1	2-Bit	← D
1	2-Bit	← E
1	2-Bit	← F
1	2-Bit	← G
-6	3-Bit	← A
7	4-Bit	← H
Total size =25-Bit	Total size 25-Bit	

Table 15. Header information about second text.

ASCII	required of bits	
95	8-Bit	← _
2	2-Bit	← a
4	4-Bit	← e
4	4-Bit	← i
5	4-Bit	← n
-1	3-Bit	← m
6	4-Bit	← s
15	4-Bit	← d
11	4-Bit	← o
27	6-Bit	← h
12	4-Bit	← t
5	4-Bit	← y
-19	5-Bit	← f
6	4-Bit	← I
9	4-Bit	← u
-4	3-Bit	← q
-63	7-Bit	← 2
7	4-Bit	← 9
27	6-Bit	← T

3	2-Bit	← r
-6	3-Bit	← l
-9	5-Bit	← c
-22	5-Bit	← M

-17	5-Bit	← C
20	6-Bit	← W
-18	5-Bit	← E
Total size 115-Bit	Total size 115-Bit	

The Huffman and Shannon coding header needs to store the characters and their codes [4,5], the header size for the first text and the second text (See Section 3.1) for the two methods are compared with sequence dynamic code and the resulte are shown in Table 16.

Table 16. Header size for two methods compared with our algorithm

Methods	**Huffman coding**	**Shannon coding**	**Sequence dynamic coding**
Header size First text	11.3 Byte	11.3 Byte	6.2 Byte
Methods	**Huffman coding**	**Shannon coding**	**Sequence dynamic coding**
Header size Second text	42.6 Byte	42.5 Byte	28.7 Byte

Conclusion

The advantags of this algorithm are given in the following steps:-

1- compresses 16-Bit data to less than 16-Bit coding by using sequential start code and sequential dynamic code, as shwon in test results and comparison with Huffman and Shannon coding.
2- This algorithm does not need to store original characters and does not need to store the sequence dynamic code it , just stores the difference between the ASCII for characters after computing the recurrences for it.
3- This approach leads to store minimum bits and the header has dynamic size in this algorithm.

The disadvantage of this algorithm

1- For some files the number of 16-bits may not have recurrence, this leads to the dynamic code length reaching 19-bit
2- For long size files; the header may reached to 65535-Byte (64-KByte). Also when the difference between ASCII for characters is taken, some characters are still 8-bit for positive numbers and 9-bit for negative numbers, this affects the header size.

References

[1] ZIV, and . Lempel, I. A. "Compression of Individual Sequence Via Variable Rate Coding", *IEEE _Transaction on Information Theory* IT-24(5):530-536, April,1978.

[2] ZIV, and J. A. Lempel, "Universal Algorithm for Sequential Data Compression", *IEEE Transaction on Information Theory* IT-23(3): 337-343, November,1978.

[3] Kontoyainnis I. and Meyn S. P., "Large Deviation Asymptotic And the Spectral Theory of Multiplicatively Regular Markov Processes.", *Electronic Journal of Probability*,No.10, paper 3 pp. 61-72, February, 2005.

[4] Kontoyainnis, I. "Pattern Matching And Lossy Data Compression on Random Field", *IEEE Trans. on Inform. Theory*, 49, pp. 1047-105 1. April, 2003.

[5] Castelli V.. Robinson J. and J.J. Turek, "Multiresolution Lossless/Lossy Compression And Storage of Data for Efficient Processing Thereof", *U.S. Patent* No. 6,141,445. Octobar, 2000.

[6] Robinson P., and Singer, ll. "Another Spelling Correction Prooralil",*IEEE Transaction on communication of* the ACM 24(5): 297-298, 1981

[7] Kontoyainnis, I. "Pointwise Redundancy in Lossy Data Compression and Universal Lossy Data Compression", *IEEE* Trans. on Inform. Theory. 46. pp. 136-152, January, 2000.

.[8] Young, D. M, "*Mac Write File Format*", Wheels for the Mind 1:34,1985.

[9] Harrison M. and Kontoyiannis I., "Maximum Likelihood Estimation For Lossy Data Compression", (Invited paper). 40th *Allerton Conference on Communication, Control and Computing, Allenton, IL*, October, 2002.

[10] Dembo A. and Kontoyainnis, 1. "Source Coding Large Deviations And Approximate Pattern Matching", Invited paper in *IEEE Trans. Theory Special Issue On Shannon theory*, 48. pp. 1590 - 1615, June, 2002.