NEW TECHNIQUE FOR DATA COMPRESSION

Eyas El-Qawasmeh and Ahmed Kattan
Computer Science Dept.
Jordan University of Science and Technology

ABSTRACT. This paper suggests a new algorithm for data compression that reverses the usage of K-map. On the compressor side, the stream of the data is transferred into 1s and 0s. Then it is divided into blocks where each block consists of 16 bits. Each block is used as an input to 4-variables K-map. Quine-McClusky approach is applied to this input in order to find the minimized expression. The minimized expressions of the input data stream are stored in a file. Later, the Huffman coding is applied to this file. The obtained Huffman code is used to convert the original file into a compressed one. On the decompression side, the Huffman tree is used to retrieve the original file. The experimentation results of the proposed algorithm showed that the saving ratio on average is around 50%. In addition, the worst case was investigated and a remedy to it was suggested. The proposed algorithm can be used for various file formats including images and videos.

KEY WORDS. Compression, Decompression, Karnaugh map, Quine-McClusky.

1. INTRODUCTION

Data compression aims to remove redundant data in order to reduce the size of a data file. For example, an ASCII file is compressed into a new file, which contains the same information, but with smaller size. The compression of a file into half of its original size implies that the storage area is doubled. [Mandal, 2000]. The same idea applies to the transmission of a message over the internet, which implies the reduction of bandwidth.

Data compression has a wide range of applications in today's world applications especially in data storage and data transmission [Ziviani, et al., 2000]. These applications include file compression techniques which were used in many archiving systems such as ARC [ARC, 1986], and PKARAC [PKARAC, 1987]. Currently, the communications through the networks resulted in a large amount of data transferred daily. This transferred data especially the multimedia data slows the networks. By using efficient compression techniques for the transferred data, the time of the transfer will be reduced and thus the cost of the communications will decrease [Adler, et al., 2001] [Wu, et al., 2001]. Compression will be one of the key factors in the future expansion of the Web. It will improve the transmission of multimedia files such as video and other large files.

This paper suggests a new technique for data compression that reverses the use of K-map. On the compressor side, the stream of data that we would like to do compression on it is transferred into 1s and 0s. Then it is divided into blocks where each block consists of 16 bits. Each block is used as an input to 4-variables K-map. Quine-McClusky is applied to this input in order to find the minimized expression. The minimized expressions for all data are stored in a file. Later, the Huffman coding is applied to this file, and the obtained Huffman code is used to convert the original file into a compressed one. On the decompression side, the Huffman tree is used to retrieve the original file. The experimentation results of this algorithm showed that the saving ratio on average is around 50%. In addition, the worst case was investigated and a remedy to it was suggested. The proposed algorithm can be used for binary and ASCII files including images and videos.

The organization of this paper will be as follows: Section 2 presents a review of the K-map. Section 3 is the suggested compression method. Section 4 is the decompression process. Section 5 is worst case. Section 6 is a remedy improvement. Section 7 is the performance results, and finally section 8 is the conclusions.

2. K-MAP REVIEW

K-map is a technique for presenting Boolean functions. It comprises a box for every variable (represented by a column) in the truth table. All the inputs in the map are arranged in a way that keeps Gray code true for every two adjacent cells [Ercegovac, et al., 1999]. The maximimal rectangular groups that cover the 1s give a minimum implementation. The proposed compression technique uses 4-inputs K-map as can be seen in Figure 1.

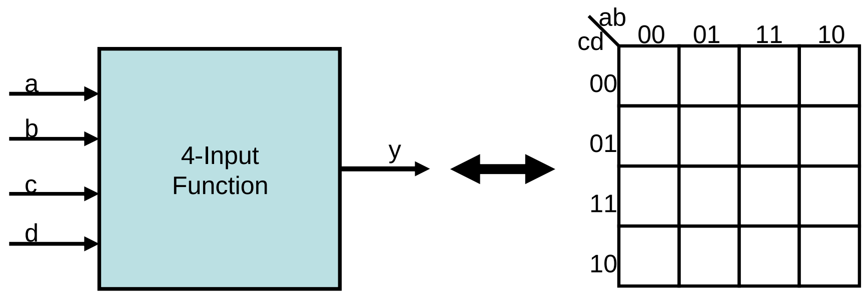


Figure 1: K-map for input functions

The proposed algorithm uses a K-map with 4-variables. Thus, the number of different combinations will be 4^2. This value is equal to 256 different possibilities for K-map, which has 4 inputs. These 256 different combinations will be evaluated to only 82 distinct expressions. The minimized expressions might contain 1, 2, 3 or 4 variables. For example, the number of different expressions that contain exactly one variable is 10 expressions out of 82 expressions. For the input variables (A, B, C, D), the set S_1 that contain exactly one variable will be S_1= {A, A`, B, B`, C, C`, D, D`, 0, 1}. The distribution of these 82 expressions and the number of variables in it is as follows:

No. of terms	1-variable	2-variables	3-variables	4-variables
	10	24	32	16

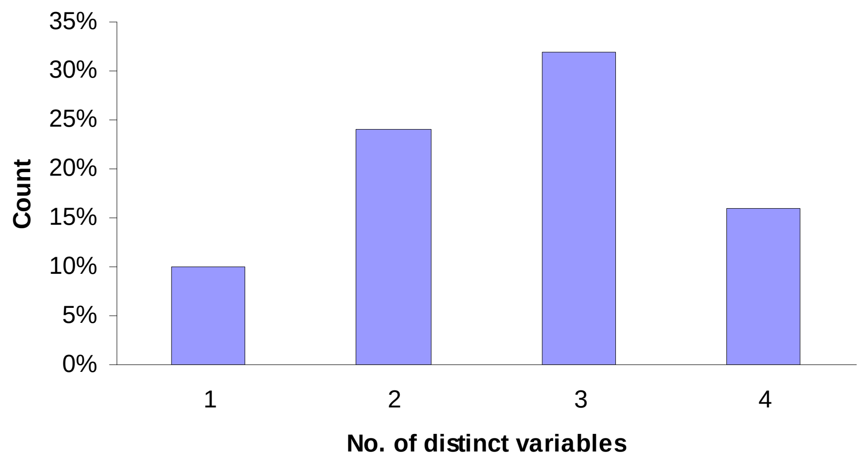


Figure 2: The distribution of possible minimized terms using 4 inputs K-map

Following is Table (1) which contains all the possibilities of the minimized terms with the corresponding distinct variables in the minimized term.

1-varaible	2- variables			3- variables				4- variables	
A	AB	A`D	CD	ABC	AB`D	A`CD	B`C`D	ABCD	A`BC`D
B	AB`	A`D`	CD`	ABC`	AB`D`	A`CD`	B`C`D`	ABCD`	A`BC`D`
C	A`B	BC	C`D	AB`C	A`BD	A`C`D		ABC`D	A`B`CD
D	A`B`	BC`	C`D`	AB`C`	A`BD`	A`C`D`		ABC`D`	A`B`CD`
A`	AC	B`C		A`BC	A`B`D	BCD		AB`CD	A`B`C`D
B`	AC`	B`C`		A`BC`	A`B`D`	BCD`		AB`CD`	A`B`C`D`
C`	A`C	BD		A`B`C	ACD	BC`D		AB`C`D	
D`	A`C`	BD`		A`B`C`	ACD`	BC`D`		AB`C`D`	
0	AD	B`D		ABD	AC`D	B`CD		A`BCD	
1	AD`	B`D`		ABD`	AC`D`	B`CD`		A`BCD`	

Table (1): All possible combinations for 4 input K-map.

These are the only possible terms that can be generated from the K-map. Note that a set of terms will constitute an expression. The proposed compression algorithm creates a frequency lookup table of 82 distinct elements that are listed in Table (1). Each element in this frequency lookup table will be associated with a frequency counter that will be initialized to zero when compression process starts. Later, any detected term updates its associated frequency counter table in the frequency lookup table.

3. PROPOSED COMPRESSION TECHNIQUE

The proposed method reads any input file, which consists of 0s and 1s, then starts processing every 16-bits as one block. Each 16 bits will be used to fill in a 4-input K-map and then find the minimal expression for this block. Each term of the 82 entries in the frequency lookup table has an initial counter frequency value equal to zero which will be incremented by 1 for the corresponding term in the expression. Once the frequency lookup table is computed for the whole file, we will use Huffman coding to find a binary representation for each one of the 82 terms [Chowdhury, et al., 2002]. The generated Huffman code is used to convert the original file into a compressed file. The compression procedure is depicted in Figure 3.

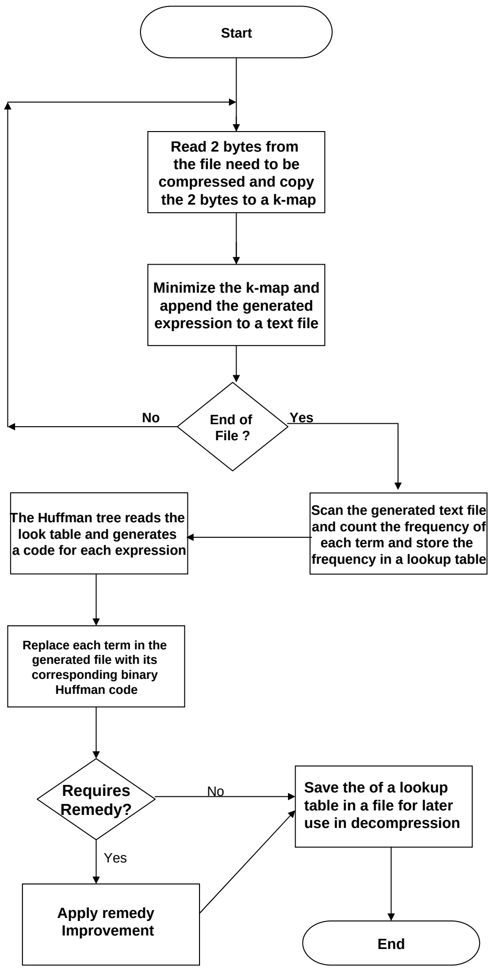

Start
Read 2 bytes from the file need to be compressed and copy the 2 bytes to a k-map
Minimize the k-map and append the generated expression to a text file
End of File ?
No
Yes
Scan the generated text file and count the frequency of each term and store the frequency in a lookup table
The Huffman tree reads the look table and generates a code for each expression
Replace each term in the generated file with its corresponding binary Huffman code
Requires Remedy?
No
Yes
Save the of a lookup table in a file for later use in decompression
Apply remedy Improvement
End

Figure 3: A flowchart of the proposed compression algorithm

The algorithm, which describes each step in the compression process, is listed below

Algorithm Compression (Input: File)
Begin

Step 0:
Create an empty text file (temporary file) that will be used to store the generated expressions.

Step 1:
Covert the ASCII code of the file, which needs to be compressed into, blocks of 0s and 1s. Each block must contain 16 bits. A block is the basic unit that we will work on. However, if

the last set of bits is less than 16 bits, then the last set of bits will be treated alone. This treatment involves writing a flag in the file if the number of bits in the file modules 16 is equal to zero or not.

```
Step 2:
While (there are more data blocks) do the following
    For each 16 bits in the file call procedure Convert_Binary_into_Miniterms( ).
    /* This procedure will be explained in the following section. The output of this
    procedure will be an expression that represents the 16-bits. The generated expression
    will be appended (stored) to the temporary text file, which we created in the first
    step of this algorithm. */
End While

Step 3:
Call procedure Find_Frequency(file)
// Procedure Find-Frequency generate a lookup table consists of 82 distinct elements with
// the corresponding frequency for each term.

Step 4:
Apply Huffman code to the generated frequencies. Here we will find the Huffman tree and
then find the associated code for each term in our 82-table elements.
// It is not necessary to have all the 82 elements especially in small files. However, for very
large files, it is expected to have all of them. The output of this step is the Huffman tree.

Step 5:
Read the original file and replace each 16 bit with the corresponding code from Huffman
tree.
```

End of Compression

The compression algorithm calls procedure ***Convert_Binary_into_Miniterms()***. The input of this procedure will be a block of 16-bits. Procedure ***Convert_Binary_into_Miniterms()*** uses Quine-McClusky algorithm to convert the 16-bits into a simplest reduced expression. The output of it will be the minimal expression of the 16-bits. Quine-McClusky algorithm has the advantage of being easily implemented with a computer or programmable calculator. The details of this algorithm can be found in many texts [Ercegovac et al., 1999].

Following is procedure ***Convert_Binary_into_Miniterms()***

```
Procedure Convert_Binary_into_Miniterms()
Begin
    Take the 16-bits and apply Quine-McClusky algorithm in order to get a reduced expression.
    The generated expression will be sent back to algorithm compression so that it will be
    stored in the temporary text file generated in step 1 of compression algorithm.
End
```

We should note that the previous procedure inputs 16-bits that are converted to minterms. For example, if we have a block such as : 1001110101001111, then the minterms of it will be as follows: $f(a,b,c,d)= \sum m(0,1,2,5,6,7,8,9,10,14)$. These minterms will be used to fill in the K-map and further processing on it will occur so that the reduced simplest expression is found.

In addition, algorithm ***Compression*** (File) called procedure ***Find_Frequeny (File)*** without providing any details about it. The purpose of procedure ***Find_Frequency(File)*** is to establish a frequency lookup table consists of 82 distinct terms. Each term will contain one of the terms that were listed in table 1 and its associated frequency in the temporary generated text file. This procedure scans the temporary generated text file. Figure 4 shows an example of the format of the frequency lookup table.

a\`b\`c\`d\`	a\`b\`c\`	a \`b c	▪ ▪ ▪ ▪ ▪	a\`b\`c\`d\`	a\`b\`c\`	a \`b c
10	12	7	▪ ▪ ▪ ▪ ▪	5	8	17
1	2	3	…………	80	81	82

Figure 4: An example of lookup table

Note that it is possible for small files to get less than 82 different terms. In this case, it is possible that the number of distinct terms in the lookup table will be less and this will reduce the size of the compressed file and hence improve the compression ratio.

Following is the details of the procedure ***Find_Frequency(Input: text file)***

```
Procedure Find_Frequency (Input: text file)
Begin
        // Inputs : A temporary text file
        // Output: A two dimensional array contains the frequencies called frequency lookup table.

        Step 0:
        Declare a two dimensional array with 2 rows and 82 columns. The first row will contain all
        the 82 distinct values of table 1. All the numeric values in the second row will represent a
        counter frequency for each term where each value is initialized to zero at the beginning.

        Step 1:
        Scan the temporary generated file and for each detected expression increase the counter
        frequency by 1.

End
```

The complexity of proposed algorithm is equal to O(cn) where c is a very small constant.

The initial performance results of the compression algorithm are as follows:

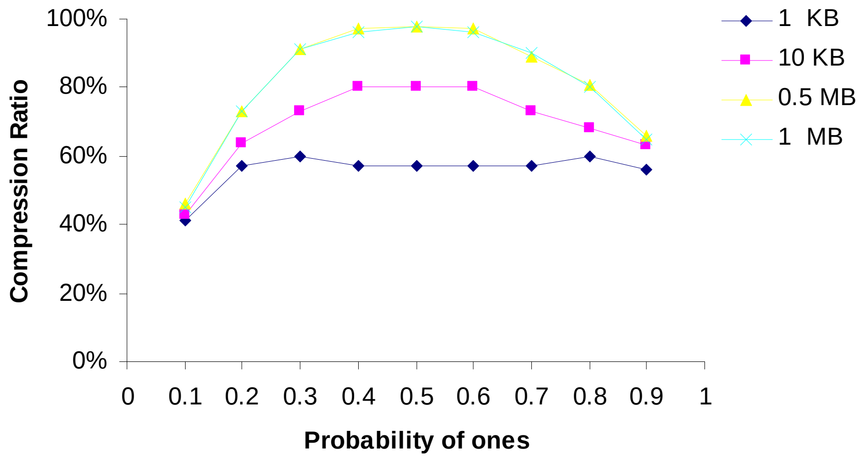


Figure 5: Initial compression ratio

Note that the x-axis in Figure 5 represents the probability of ones. For example, a probability of 0.4 on the x-axis means that the number of ones in 32-bits is approximately 0.4 * 32= 12 bits. We should report here that we generate each bit of the 32-bits individually rather than generating the whole number. The reason for this is to get more uniform and fair random number. The random number generator, which is used, is listed in [Press, et al., 1995].

Note that Figure 5 showed the worst case. The worst case occurs always with probability of ones equal to 0.5. We should point out that the probability of this worst case is a very rare case and it occurs in less than 5% of the total cases. The compression ratio for large files such as 1 MB and 0.5 MB is almost the same. We have noticed that the compression ratio for large files is almost fixed. The reason for this is that we get almost the same Huffman code for the 82 entries in the frequency lookup table.

4. DECOMPRESSION PROCESS

The decompression process requires that the receiver must have the Huffman tree or the Huffman table, which contains the associated terms and the corresponding code for it. This seems to be an extra overhead. However, the Huffman tree contains only 82 nodes at most. In case we use a Huffman table, it contains only 82 elements at most. Our proposed compression algorithm prefers to use Huffman tree. The reason for this is that the leaf nodes in the Huffman tree inform us of the end of the coding for a certain term. This enable us to get ride from the + sign between different terms in any single expression, and this is the reason for selecting it. For example, the term **AB** corresponds to code 010 in the Huffman tree and so on.

It is necessary to have a Huffman tree for the decompression of the received file. Procedure ***Decode*** takes care of the decompression process. This procedure reads the compressed file and process the file in order to replace it with a chain of 0s and 1s by the corresponding symbol blocks using Huffman tree. As a result, the received file will consists of a set of terms where each term is separated from the other by a comma. Each term of the file will be expanded to 1s and 0s and translated into a table that consists only of 0s, 1s, and –s (See Figure 6). For example, the expression A`B `CD will be replaced with 0011 where A' is the complement and C is the variable. In case we get an expression like AC then we will replace it with 1-1-. Note that the missing variable in the expression will be replaced by –. An overall example, which shows a sample text of the compressed file and its corresponding match, is presented in Figure 6.

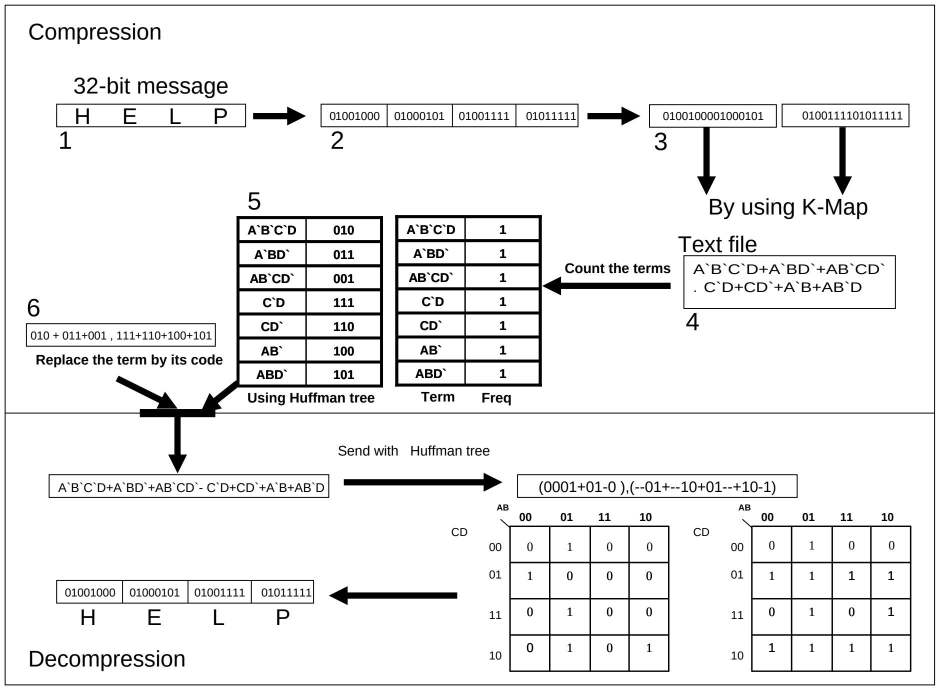


Figure 6: The complete cycle of compression and decompression for a 32-bit message contain the text "HELP"

Following is detailed expression of procedure ***Decode (inputs: Compressed file, Huffman tree)***

Procedure Decode (inputs: Compressed file, Huffman tree)

Begin

```
 Step 0:
Create an empty file

 Step 1:
 While (there are more bits in the compressed file) do
 {
        Select one bit
        Walk from the root of the Huffman tree one edge that match the read bit
         If the visited node is a leaf node then stop and append the corresponding term that
         belongs to Huffman code of the path from the root to the leaf node.
 }

 Step 3:
// Read the file and generate a new file which replace each Huffman code with its associated
// expression using the Huffman tree.
While (there are more expressions in the received file) do
{
      //Scan the expression character by character
      If (the first character is A) then
                  insert 1 into answer
      Else if the first character is A` then
                  insert 0 into answer
      If (the first character is not A or A`) then
                  insert – into answer
        If (the second character is B) then
                insert 1 into answer
        Else if the second character is B` then
                insert 0 into answer
        If (the second character is not B or B`) then
                insert – into answer

        If (the third character is C) then
                insert 1 into answer
        Else if the third character is C` then
                insert 0 into answer
        If (the third character is not C or C`) then
                insert – into answer
        If (the fourth character is D) then
                insert 1 into answer  into answer
        Else if the fourth character is D` then
                insert 0 into answer
        If (the fourth character is not D or D`) then
                insert – into answer

    end while
```

End Procedure *Decode*

Following are four examples that clarify how this procedure works.

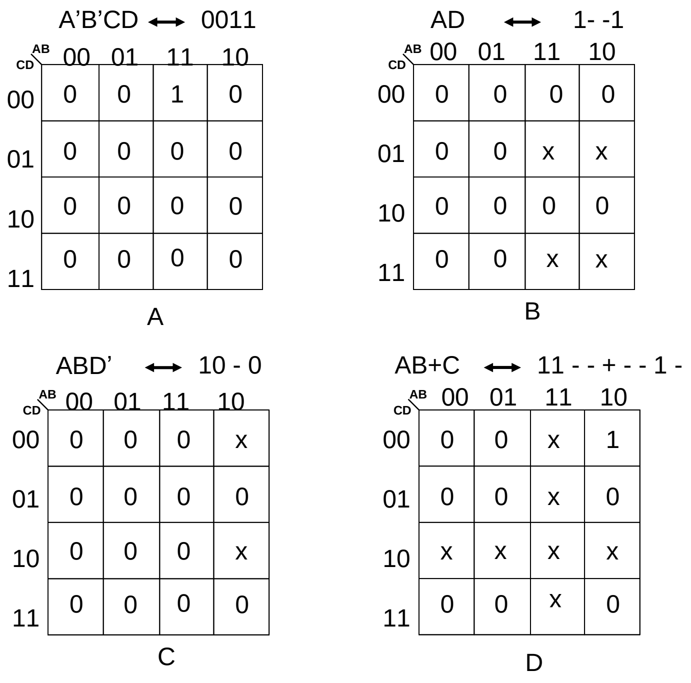


Figure 7: Examples generated from procedure ***decode***

In Figure 7, each x denotes either 0 or 1. Therefore, we will replace each x by the value 0, then 1. For example, Figure 7 part B, corresponds to the result of the decompression of 1- - 1. In this part, we use the K-map, which corresponds to it and replaces each – by 0 then by 1. Now we will get 4 boxes for this. The minterms is the original message

Note that if the expression contains a bar like A` then we compare it with 0 rather than one.

An interesting case arise when we have an expression like A - - B + - - C` . In this case we convert it one more time into 1- - 1+ - - 0 –. Note that the expression 1- -1 generates 4 terms while the expression 1- -1+--0- generate 8 terms.

The complexity of the decompression procedure is a polynomial and it is equal to O(cn) where c is a very small constant.

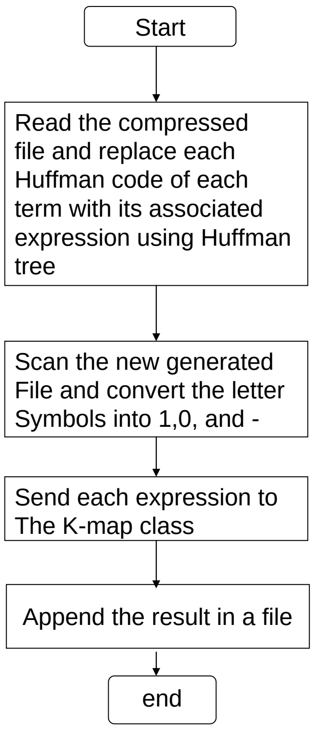
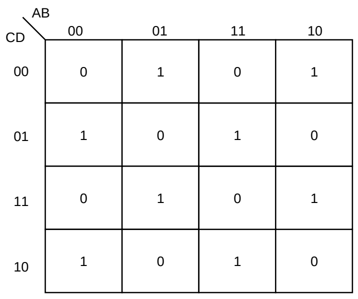

Start
Read the compressed file and replace each Huffman code of each term with its associated expression using Huffman tree
Scan the new generated File and convert the letter Symbols into 1,0, and -
Send each expression to The K-map class
Append the result in a file
end

Figure 8: A flowchart of the decompression algorithm

5. WORST CASE

There is possibility that we reach a block of 16-bits that make the worst case. As we saw in Figure 5, the worst case compressed to 98%. This means that it has a saving ratio of 2%. This case will occur when we have a block such as 1001011010010110. This block corresponds to the following K-map



CD \ AB	00	01	11	10
00	0	1	0	1
01	1	0	1	0
11	0	1	0	1
10	1	0	1	0

Figure 9 : An example of the worst case

The previous block, which represented in Figure 9 generates the following expression with 8 terms as follows:
A`BC`D` +AB`C`D`+A`B`C`D+ABCD`+A`BC`C`+AB`CD+A`B`CD`+ABCD`. Let us assume that each term in the previous expression takes two bits. As a result, it will take (8x2) = 16 bits. This means that there will be no saving with the compression for this particular block. However, in reality, some terms might take more than 2 bits or less.

To reduce the effect of this worst case, we suggest two solutions. The first one is to do compression one more time on the compressed file. We have investigated this case, and we did a compression for the second time. The overall saving was around 10%. The results of this improvement are depicted in Figure 10.

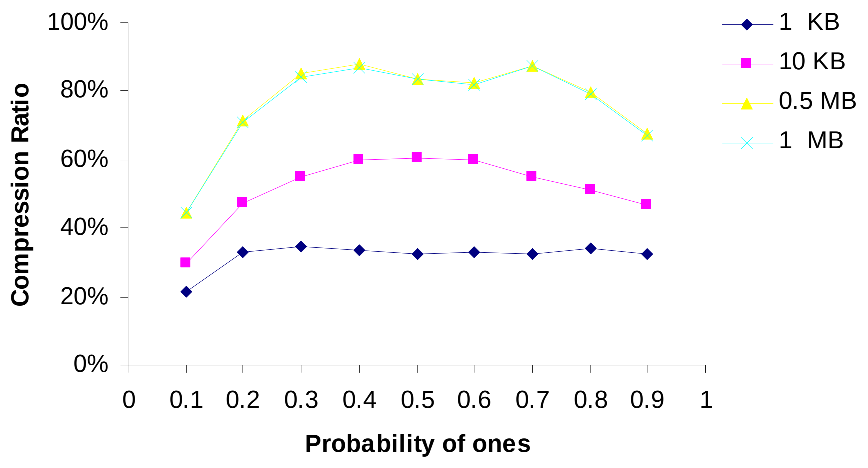
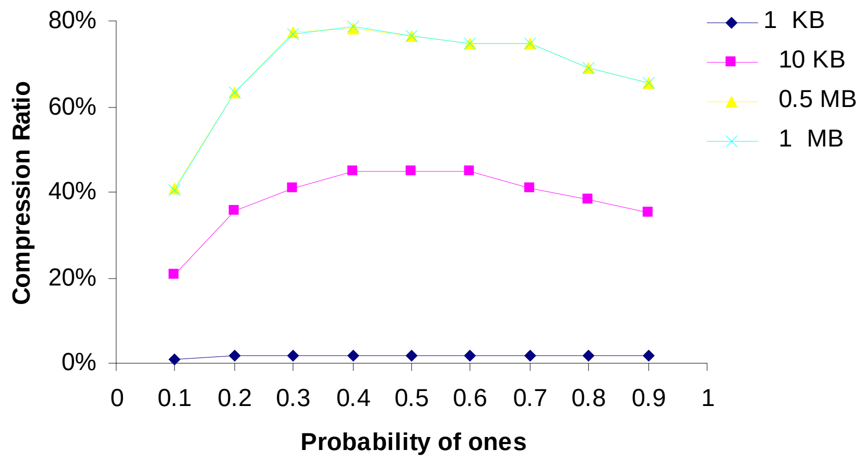


Figure 10: The first remedy which do compression for the file two times

In addition, we investigated the possibility of doing 3 levels of compression. In this case, we will have three Huffman trees each one corresponds to a certain level. The results of this experiment are depreciated in Figure 11.



The second solution is to process the worst case alone. In other words, we check the compressed file for the blocks that occupy 16-bits or more. These blocks are called problem blocks. In this case, a remedy will be used when a problem block occurs. Note that the value of the problem block is not necessary equal to 16 bits. We can change it to make it equal to 15 bits or even less. The following remedy works on problem blocks that are 16 bits or higher.

6. REMEDY TO THE WORST CASE

We have noticed in Figure 9 the occurrence of the worst case at probability of 0.5. The total number of cases that this case occurs can be computed, but it is less than 5% of all different forms of files. There are two different remedies. The first one is to re-do compression. The results of this are drawn in Figure 10. The second one is suggested following. It consists of the following steps:

1- Check our compressed file for any expression, which has a problem block of 16 bits or more. We keep track of them and the frequency of their occurrences in general.

2- Expand the frequency lookup table so that it will contain some of the problem blocks with high frequency. The expanded frequency lookup table entries were 120 entries where 82 of them are reserved as in Table 1. We were able to reduce the compression ratio by a significant ratio, which is about 15% on average. The next section presents the results of this remedy.

7. PERFORMANCE RESULTS

An experiment was conducted to see the compression ratio for different probabilities. We generate a sequence of bits where each bit is generated individually with a certain probability. Please note that probability 0.5 represents the worst case.

Figure 11 shows the compression ratio when the frequency lookup table was expanded to 120 entries.

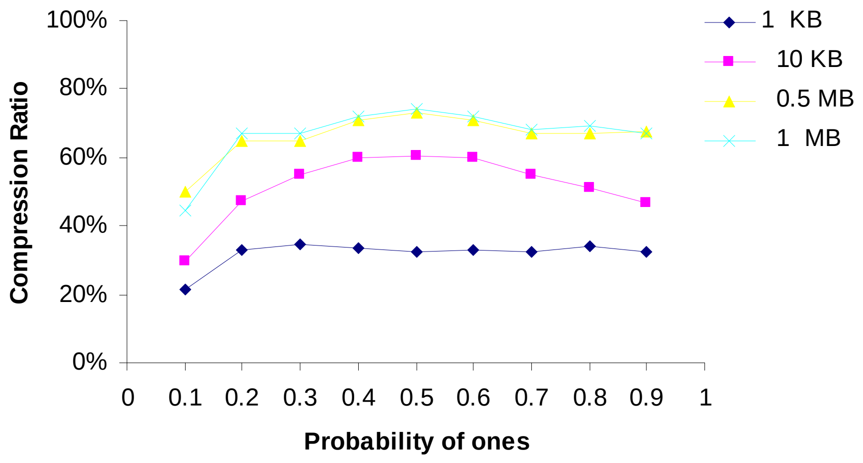


Figure 11: Compression ratio using a remedy with 16 bits problem blocks and a frequency table of 120 entries

It is expected that if we consider the bad block as 15-bits or 14 bits then we will improve the compression ratio.

8. CONCLUSION

This paper suggests a new scheme that reverses the use of K-map that it can be used for compression. Results showed that the suggested algorithm reduce the size of the files/messages.

When some remedies are used, the compression ratio is improved. The performance results showed that the compression ratio is around 50% on average. The suggested algorithm can be improved further.

REFERENCES

1) Adler, M. and Mitzenmacher, M. “Towards Compressing Web Graphs,” In Proc. of the IEEE Data Compression Conference, Utah, USA, pp. 203-212, Mar. 2001.

2) ARC File Archive Utility. 1986. Version 5.1. System Enhancement Associates. Wayne, N. J.

3) Brisaboa, N., Iglesias, E., Navarro, G., and Paramá, J., "An Efficient Compression Code for Text databases," In Proc. Of 25th European Conference on IR Research, ECIR 2003, Pisa, Italy, pp. 468-481, Apr., 2003.

4) Chowdhury, R., Kaykobad, M. and King, I., "An Efficient Decoding Technique for {Huffman} Codes," Information Processing Letters, Vol. 81, No. 6, pp. 305-308, 2002.

5) Mandal, J., “An Approach towards Development of Efficient Data Compression Algorithms and Correction Techniques,” Ph.D. Thesis, Jadavpur University, India, 2000.

6) Ercegovac, M.D., Lang, T., Moreno, J., “Introduction to Digital Systems,” John Wiley and Sons Inc., 1999.

7) PKARC FAST! File Archival Utility. 1987. Version 3.5. PKWARE, Inc. Glendale, WI.

8) Press, W., Teukolsky, S., Vetterling, W., and Flannery, B. Numerical Recipes in C. Second edition. Reprinted with correction, Cambridge University Press, 1995.

9) Wu, D., Hou, Y., Zhu, W., Zhang, Y., and Peha, J., "Streaming Video over the Internet: Approaches and Directions," IEEE Transactions on Circuits and Systems for Video Technology, Vol. 11, No. 3, pp. 282-300, Mar. 2001.

10) Ziviani, N., Moura, E., Navarro, G., and Baeza-Yates, R. “Compression: A Key for Next-Generation Text Retrieval Systems,” IEEE Computer, Vol. 33, No. 11, pp. 37- 44, Nov. 2000.

Filename:	Conference
Directory:	C:\EYAS\PAPERS\Under_Writing\Compression
Template:	C:\Documents and Settings\Eyas\Application Data\Microsoft\Templates\Normal.dot
Title:	A NEW COMPRESSION ALGORITHM FOR
Subject:	
Author:	Eyas El-Qawasmeh
Keywords:	
Comments:	
Creation Date:	10/7/2004 7:30 PM
Change Number:	3
Last Saved On:	10/7/2004 7:31 PM
Last Saved By:	
Total Editing Time:	2 Minutes
Last Printed On:	10/25/2004 8:46 PM
As of Last Complete Printing	
Number of Pages:	13
Number of Words:	3,420 (approx.)
Number of Characters:	19,499 (approx.)