YALMIC: Yet Another Logic Minimization Based Image Compressor

Behrouz Zolfaghari  
Engineering Department  
Islamic Azad University  
Garmsar Branch  
Garmsar, Iran  
zolfaghari@iust.ac.ir

Hamed Sheidaian  
Engineering Department  
Islamic Azad University  
Garmsar Branch  
Garmsar, Iran  
Sheidaian@masir.net

Saadat Pour Mozaffari  
Department of Computer Engineering and  
Information Technology  
AmirKabir University of Technology  
Tehran, Iran  
saadat@aut.ac.ir

Abstract—This paper proposes and evaluates a novel image compression approach and also a software tool named YALMIC that implements the approach. The main idea behind the proposed approach is considering segments of consecutive bytes in the image file as 8-variable minterms and using them to construct implicants. The implicats are simplified through discarding some variables. Storing the simplified forms of the implicants instead of the original bytes causes some bits to be discarded. YALMIC stores some extra bits in the compressed file in addition to the simplified implicant. These extra bits are necessary to determine the discarded variables and also to show the order of the disjointed minterms. The compression ratio of the tool is evaluated and compared to that of previous image compression tools and techniques through analytical modeling and experimental results.

Keywords—logic function minimization; image compression;

I. INTRODUCTION AND BASIC CONCEPTS

Many research works has focused on developing various data compression (especially image compression) techniques in recent years [1-4]. Among other approaches, many proposed techniques depend on logic function minimization [5, 7, 8, 9, and 11]. Some of these approaches are lossy [8, 10] and some of them directly depend on Huffman coding [5, 9]. Such techniques treat bit sequences as logic terms and try to simplify the logic expressions constructed of these terms. Simplification of logic expressions causes some terms and variables to be discarded from the expression and this can compress the input data by discarding the corresponding bits or bit sequences. For example consider the bit stream $S = 111011111011100$. This stream can be divided into four bit sequences each of which consists of four bits (called quartets). The first quartet (1110) represents a 4-variable minterm such as $xyzw$. Similarly, the next three quartets represent $xyzw$, $xy zw$, and $x zw$ respectively. Thus, we can model the whole bit stream by a disjunctive logic expression as follows:

$$S = xyzw + xyzw + xzw + xy zw$$  

Equation (1)

The above logic experiment be simplified to $S' = xy$ or equivalently $S' = 11$ using Karnaugh map or the Quine-McCluski method. If it was possible to regenerate $S$ from $S'$, we had come to a compressed form $S$. But due to the following two reasons, $S$ cannot be fully regenerated from its simplified form $S'$.

First: $S' = 11$ only characterizes a 2-variable minterm. But it contains no information concerning the positions of its constructing variables in the original 4-variable minterms. For example, it can represent $xy$, $yz$, or $zw$. If it is assumed to represent $xy$, then it will be considered as the simplified form of $E_1 = x y zw + x y zw + xzw + xzw$. But if it is considered as $yw$, it will be the simplified form of $E_2 = x y zw + x y zw + x y zw + xzw$. In fact, $S'$ cannot distinguish $E_1$ from $E_2$.

Second: $S'$ does not convey anything regarding the order of the 4-variable minterms (quartets) in the original image. For example, suppose that we know $S' = 11$. In this case, the following 4-variable minterms will be generated by $S': xy zw$, $xy zw$, $xyzw$ and $xyzw$ which are equivalent to the quartets 1100, 1101, 1110, and 1111 respectively. Now since the order of the minterms in the original image is unknown, we can regenerate $S$ as $111111011111011101111$ or $1111110111111101111$.

For the two reasons mentioned above, we have to add some extra bits to $S' = 11$ which make it possible to exactly regenerate $S$. YALMIC (Yet Another Logic Minimization Based Image Compressor) is a software tool that implements a novel lossless image compression algorithm based on logic minimization. In the algorithm used by this tool, segments of the image are first converted to hypothetical logic expressions and the expressions are simplified using standard methods. Then the simplified forms of the expressions are stored in the target compressed file along with some extra bits which allow the primary segment to be fully regenerated from its simplified form. YALMIC has been written in C#. Its screenshot is shown in figure 1. YALMIC computes the size of the compressed image and uses the following equation to calculate the compression ratio.

$$CR = \frac{U - C}{U} = 1 - \frac{C}{U}$$  

Equation (2)
In the above equation, \( U \) is the size of the original uncompressed file and \( C \) is that of the compressed file. The ratio \( \frac{C}{U} \) is referred to as the compression factor in this paper.

This paper is continued as follows. Section II discusses the proposed compression algorithm. Section III explains the decompression algorithm. Section IV evaluates the compression ratio of the algorithm using analytical modeling and experimental results. This section compares the compression ratio of the proposed algorithm to that of previous algorithms. Section V concludes the paper and suggests further works.

II. THE COMPRESSION ALGORITHM

The compression algorithm consists of two phases; the first phase is called the minimization phase and the second is referred to as the block construction phase.

In the minimization phase, the original image file is first divided into 8-bit sequences called octets. (Each pixel in a color bitmap image is assumed to be represented by three consecutive octets which indicate RGB values respectively. Our algorithm brings the octets indicating R values near each other, puts the G values next and places the B values at the end.) Each octet is then considered as a minterm of 8 variables which are called \( b_0 \), \( b_1 \), \( b_2 \), \( b_3 \), \( b_4 \), \( b_5 \), \( b_6 \) and \( b_7 \) with respect to their positions in the minterm. For example, the octet 10110110 is considered as \( b_0 \bar{b}_6 b_5 b_4 \bar{b}_3 b_2 b_1 \bar{b}_0 \). In the next step, a hypothetical ‘+’ operator is inserted between each pair of consecutive minterms. The ‘+’ operator represents the logic OR operation. In this step, the image is converted to a disjunctive logic expression represented as a sum of minterms. This sum is then scanned for sets of successive identical minterms or successive minterms which can construct implicants. Each of these sets is called a segment. Each segment should consist of 1, 2, 4 or 8 successive minterms. An 8-minterm segment is preferable to two 4-minterm segments and so on. The block construction phase stores an individual block in the compressed file for each segment detected in this phase. Each block contains a simplified form of the corresponding segment along with some extra bits. Segments containing identical minterms are called repetitive segments. The simplified form of such a segment is simply one of the identical minterms. For example, the sequence \( P=101101101011011010 \) is converted to the following expression.

\[
P = b_2 \bar{b}_6 b_4 b_3 b_2 b_1 \bar{b}_0 + b_7 \bar{b}_6 b_5 b_4 \bar{b}_3 b_2 b_1 \bar{b}_0
\]

Equation (3)

The simplified form of the above expression in the corresponding block will be 10110110.

If a segment can be simplified to an implicant, the implicant will be the simplified form of the segment. Such a segment is called an implicant segment. As an example, \( Q=10110110101101101010101010 \) is converted to the following experiment.

\[
Q = b_2 \bar{b}_6 b_4 b_3 b_2 b_1 \bar{b}_0 + b_7 \bar{b}_6 b_5 b_4 \bar{b}_3 b_2 b_1 b_0
\]

Equation (4)

As equation (2) shows, two variables \( b_0 \) and \( b_1 \) have been discarded and the implicant \( b_2 \bar{b}_6 b_4 b_3 b_2 b_1 b_0 \) has been formed. The simplified form of the above experiment will be 101101 which will be stored in the corresponding block.

In the block construction phase, simplified forms of sequences are stored in blocks. We refer to these blocks as compressed blocks. The structure of a compressed block is shown in figure 2.

The \( F0 \) field is a one-bit flag which distinguishes implicant segments from repetitive segments. This flag will be 0 for blocks corresponding to implicant segments (implicant blocks) and 1 for blocks corresponding to repetitive segments (repetitive blocks).

The \( F1 \) field shows the number of discarded variables for implicant segments. In other words, this field is equal to \( \log_2 n \) for an implicant segment of \( n \) minterms. \( n \) can be equal to 1, 2, 4 or 8 (\( n=1 \) represents a block consisting of a single minterm which cannot be part of any repetitive or implicant segment. Such blocks are called single blocks). Thus, the possible values for this field are 0, 1, 2 and 3. This field is 2 bits long. \( F1 \) is also equal to the logarithm of the number of identical minterms to the base 2 for repetitive segments.

\( F2 \) does not exist in repetitive blocks or single blocks in which \( F1=00 \). This field shows the combination of discarded variables in implicant blocks. If \( F1 \) is equal to 01, \( F2 \) should indicate one of 8 variables \( \{b_i \mid i \in [0,7] \} \). Thus, the length of this field will be equal to \( \lceil \log_2 8 \rceil = 3 \) bits.

If \( F1 \) is 10, \( F2 \) should determine one of \( \binom{8}{2} = 28 \) combinations which may have been discarded.
In this case, the length of $F_2$ will be equal to $\left\lceil \log_2 28 \right\rceil = 5$. If $F_1 = 11$, $F_2$ will be equal to $\left\lceil \log_2 \left( \frac{8}{3} \right) \right\rceil = 6$ bits.

The field $F_3$ indicates the order of the simplified minterms in the original file. This field does not exist in repetitive blocks or single blocks. If $F_1 = 01$, $F_3$ should determine one of $2! = 2$ possible orders. In this case, the length of this field will be equal to $\left\lceil \log_2 2 \right\rceil = 1$ bit. If $F_1 = 01$, $F_3$ should indicate one of $4! = 24$ possible orders for 4 minterms. In this case, the length of $F_3$ will be equal to $\left\lceil \log_2 24 \right\rceil = 5$ bits. If $F_1$ is 11, the length of $F_3$ will be equal to $\left\lceil \log_2 \left( \frac{8 \times 3}{2} \right) \right\rceil = 16$ bits.

The field $F_4$ in each block contains the simplified form of the corresponding segment. The length of this field will be equal to 8 bits for every repetitive block or every single block. This field will have 7 bits in 2-implicant blocks, 6 bits in 4-implicant blocks and 5 bits in 8-implicant blocks.

According to the above discussions, the compression algorithm will be as follows:

$$i=1$$

While Not EOF (OriginalFile) {

Allocate a new block

If minterm $i$ is identical to minterms $i+1$ to $i+7$ then

{ 

Set $F_0=1, F_1=11$

Consider no $F_2$ and no $F_3$

Put the minterm $i$ in $F_4$ in 8 bits

Store the constructed block in CompressedFile

Set $i=i+8$

}

Else If minterm $i$ is identical to minterms $i+1$ to $i+3$ then

{ 

Set $F_0=1, F_1=10$

Consider no $F_2$ and no $F_3$

Put the minterm $i$ in $F_4$ in 5 bits

Store the constructed block in CompressedFile

Set $i=i+4$

}

Else If minterm $i$ is equal to minterm $i+1$ then

{ 

Set $F_0=1, F_1=01$

Consider no $F_2$ and no $F_3$

Put the minterm $i$ in $F_4$ in 5 bits

Store the constructed block in CompressedFile

Set $i=i+2$

}

Else If minterm $i$ can be disjointed to minterms $i+1$ to $i+7$ then

{ 

Set $F_0=0, F_1=11$

Put the combination of the discarded variables in the array $C$

Find the ID of the combination using the CombinationToID algorithm and assign the ID to $F_2$

Put the minterms $i$ to $i+7$ in the array $nums$ with respect to their positions in OriginalFile

Find the ID of the permutation using the permutation algorithm and assign the ID to $F_3$ in 5 bits

Store the simplified form of the 8 minterms $i$ to $i+3$ to $F_4$ in 6 bits

Store the constructed block in CompressedFile

Set $i=i+8$

}

Else If minterm $i$ can be disjointed to minterms $i+1$ to $i+3$

{ 

Set $F_0=0, F_1=10$

Put the combination of the discarded variables in the array $C$

Find the ID of the combination using the CombinationToID algorithm and assign the ID to $F_2$ in 5 bits

Put the minterms $i$ to $i+3$ in the array $nums$ with respect to their positions in OriginalFile

Find the ID of the permutation using the permutation algorithm and assign the ID to $F_3$ in 5 bits

Assign the simplified form of the 4 minterms $i$ to $i+3$ to $F_4$ in 6 bits

Store the constructed block in CompressedFile

Set $i=i+4$

}

Else { 

Optionally Set $F_0=0, F_0=1$

Set $F_1=0$

Consider no $F_2$ and no $F_3$

Assign the minterm $i$ to $F_4$ in 8 bits

Store the constructed block in CompressedFile

Set $i=i+2$

}

The algorithm CombinationToID is a simple recursive algorithm that takes a combination of variables as the input and gives a numerical ID that uniquely represents the input combination. PermutationToID is also another simple algorithm that takes a permutation of numbers and gives a unique numerical ID for that permutation.

### III. The Decompression Algorithm

The decompression algorithm makes use of $F_0, F_1, F_2$ and $F_3$ fields of each block in the compressed file in order to convert the $F_4$ field to the corresponding segment and store the segment in the decompressed file. This algorithm works as follows:

$$i=1$$

While not EOF (CompressedFile) {

Read bit $i$ as $F_0$ and bits $i+1$ and $i+2$ as $F_2$

$i=i+3$

If $F_1=00$ then

{ 

Read the next 8 bits as $F_4$ and store it in DecompressedFile.

$i=i+8$

}

If $F_0=1$ and $F_1=01$ then

{ 

Read the next 8 bits as $F_4$ and store it twice in DecompressedFile.

}
If F0=1 and F1=10 then
\[
\{ \\
\text{Read the next 8 bits as } F4 \text{ and store it four times in } \text{DecompressedFile.} \\
\} \quad \text{i} = \text{i} + 8
\]

If F0=1 and F1=11 then
\[
\{ \\
\text{Read the next 8 bits as } F4 \text{ and store it eight times in } \text{DecompressedFile.} \\
\} \quad \text{i} = \text{i} + 8
\]

If F0=0 and F1=01 then
\[
\{ \\
\text{Read the next 3 bits as } F2, \text{ i} = \text{i} + 3 \\
\text{Read the next bit as } F3, \text{ i} = \text{i} + 1 \\
\text{Read the next 7 bits as } F4, \text{ i} = \text{i} + 7 \\
\text{Put once 0 and once 1 in } F2 \text{ and construct two octets.} \\
\text{If } F3=0 \text{ first store the octet containing 0 in } \text{DecompressedFile} \text{ and next} \\
\text{store the octet containing 1.} \\
\text{Otherwise, first store the octet containing 1 in } \text{DecompressedFile} \text{ and} \\
\text{next store the octet containing } 0. \\
\} \quad \text{i} = \text{i} + 8
\]

If F0=0 and F1=10 then
\[
\{ \\
\text{Read the next 5 bits as } F2, \text{ i} = \text{i} + 5 \\
\text{Read the next 5 bits as } F3, \text{ i} = \text{i} + 5 \\
\text{Read the next 6 bits as } F4, \text{ i} = \text{i} + 6 \\
\text{Pass } F2 \text{ as an integer number to the algorithm } \text{IDToCombination} \text{ and} \\
\text{determine the two discarded variables.} \\
\text{Put once 0 and once 1 for each of the two discarded variables in } F4 \text{ and} \\
\text{construct four octets.} \\
\text{Sort the four octets in ascending order in the array } C \text{ and pass } C \text{ with } F3 \\
\text{(as an integer number) to the } \text{IDToPermutation} \text{ algorithm to determine} \\
\text{the order of the octets.} \\
\text{Store the octets in } \text{DecompressedFile} \text{ in the determined order.} \\
\} \quad \text{i} = \text{i} + 8
\]

If F0=0 and F1=11 then
\[
\{ \\
\text{Read the next 6 bits as } F2, \text{ i} = \text{i} + 6 \\
\text{Read the next 16 bits as } F3, \text{ i} = \text{i} + 16 \\
\text{Read the next 5 bits as } F4, \text{ i} = \text{i} + 5 \\
\text{Pass } F2 \text{ as an integer number to the algorithm } \text{IDToCombination} \text{ and} \\
\text{determine the three discarded variables.} \\
\text{Put once 0 and once 1 for each of the three discarded variables in } F4 \text{ and} \\
\text{construct eight octets.} \\
\text{Sort the eight octets in ascending order in the array } C \text{ and pass } C \text{ with } F3 \\
\text{(as an integer number) to the } \text{IDToPermutation} \text{ algorithm to determine} \\
\text{the order of the octets.} \\
\text{Store the octets in } \text{DecompressedFile} \text{ in the determined order.} \\
\} \quad \text{i} = \text{i} + 8
\]

The algorithm \text{IDToCombination} \text{ is the reverse of} \text{CombinationToID}. \text{ It is a simple recursive algorithm that takes the ID of a combination of variables and constructs the combination.} \text{ IDToPermutation} \text{ is also the reverse of} \text{PermutationToID}. \text{ It takes the ID of a permutation and constructs the permutation.}

IV. PERFORMANCE EVALUATIONS

The probability that 8 consecutive minterms in the original file are the same can be obtained from the following equation.
\[
P_{8 - eq} = \left( \frac{1}{256} \right)^8 \quad \text{Equation(5)}
\]

In this case, the block length in the compressed file is equal to 11. Thus, the compression factor in this case can be calculated as follows.
\[
C_{8 - rep} = \frac{11}{8 \times 8} = \frac{11}{64} \quad \text{Equation(6)}
\]

\[
P_{4 - eq} \cdot C_{4 - eq} \cdot P_{2 - eq} \text{ and } C_{2 - eq} \text{ can be obtained from the following equations.}
\]

\[
P_{4 - eq} = \left( \frac{1}{256} \right)^3 \quad \text{Equation(7)}
\]

\[
C_{4 - eq} = \frac{11}{4 \times 8} = \frac{11}{32} \quad \text{Equation(8)}
\]

\[
P_{2 - eq} = \frac{1}{256} \quad \text{Equation(9)}
\]

\[
C_{2 - eq} = \frac{11}{2 \times 8} = \frac{11}{16} \quad \text{Equation(10)}
\]

If each minterm has \( n \) variables, the number of implicants constructed from \( 2^m \) minterms (considering the order of minterms) can be calculated from the following equation.
\[
N_{m, n}^a = \binom{n}{m} \cdot 2^{n-m} \cdot (2^n)! \quad \text{Equation(11)}
\]

In the above equation, \( \binom{n}{m} \) shows that \( m \) variables should be discarded. \( 2^{n-m} \) shows that the rest of variables can each be 0 or 1. \( (2^n)! \) demonstrates all possible orders of \( 2^m \) minterms. According to the above equation, the probability that a minterm in the original file can be disjointed to its 7 next minterms and construct an 8-implicant can be calculated as follows.
\[
P_{8 - imp} = \frac{8}{256} \cdot \frac{2}{3} \cdot \binom{3}{2} \quad \text{Equation(12)}
\]

The block length in the compressed file is equal to 30 bits in this case. Thus the compression factor will be calculated as follows.
\[
C_{8 - imp} = \frac{30}{8 \times 8} = \frac{30}{64} \quad \text{Equation(13)}
\]

\[
P_{4 - imp} \cdot C_{4 - imp} \cdot P_{2 - imp} \text{ and } C_{2 - imp} \text{ are obtained from the following equations.}
\]

\[
P_{4 - imp} = \left( \frac{8}{256} \right)^4 \quad \text{Equation(14)}
\]

\[
C_{4 - imp} = \frac{19}{4 \times 8} = \frac{19}{32} \quad \text{Equation(15)}
\]

\[
P_{2 - imp} = \left( \frac{8}{256} \right)^2 \quad \text{Equation(16)}
\]

\[
C_{2 - imp} = \frac{14}{2 \times 8} = \frac{14}{16} \quad \text{Equation(17)}
\]

Now let us calculate the number of ways to divide a set of 8 consecutive 8-variable minterms into segments of 1, 2, 4 and 8 minterms. To do this, we should first solve the following equation.
\[
m + 2n + 4p + 8q = 8 \quad \text{Equation(18)}
\]

\[
m, n, p, q \in [0, 8] \quad \text{Equation(18)}
\]
The compression factor of a 2-segment is calculated as follows.

\[
C_2 = \frac{256 \times \frac{11}{16} + 2048 \times \frac{14}{16}}{2048 + 256} \approx 0.85
\]

Equation(19)

We can calculate the compression factors of 4 and also 8-segments in a similar way.

\[
C_4 = \frac{256 \times \frac{11}{32} + 2048 \times \frac{19}{32}}{2048 + 256} \approx 0.60 \quad \text{Equation(20)}
\]

\[
C_8 = \frac{256 \times \frac{11}{64} + 2048 \times \frac{30}{64}}{2048 + 256} \approx 0.47 \quad \text{Equation(21)}
\]

The compression factor for a 1-segment is equal to 1.8 which is larger than unity. Now we can calculate the average compression factor for each of the 10 states in table 1. These factors will be equal to 0.6, 0.47, 0.66, 0.85, 0.93, 0.81, 0.89, 1.0, 1.10 and 1.19 respectively.

We can calculate the average compression factor of YALMIC algorithm by multiplying the compression factor of each state by its probability and adding the products to each other.

The sum of the mentioned products is \( C' = 0.8 \). Thus, the compression factor of YALMIC is equal to 0.8 and its compression ratio will be equal to \( CR' = 1 - 0.8 = 0.2 = 20\% \).

An important point to consider here is that the average compression ratio can be greater than 20% for normal images. The reason is that we have made no extra assumptions regarding dependencies among the pixels or bytes of the image while in real images, there is often a lot of dependency between the pixels and bytes. The experimental results verify this point. These results are shown in table II.

Table II consists of two parts. The upper part shows the results obtained from color images and the lower part shows the same results for gray scale images. Each row in this table contains the name, the original size, the compressed size and the compression ratio for one of the images which have been compressed using YALMIC. All the images are standard except for one of them which is named Dreamnight.

### Table I. Sets of answers and their probabilities

<table>
<thead>
<tr>
<th>Set of answers</th>
<th>Number of permutations</th>
<th>Probability</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 2 0</td>
<td>( \frac{2!}{2!} = 1 )</td>
<td>1 ( \frac{1}{66} )</td>
</tr>
<tr>
<td>0 0 0 1</td>
<td>( \frac{1!}{1!} = 1 )</td>
<td>1 ( \frac{1}{66} )</td>
</tr>
<tr>
<td>0 2 1 0</td>
<td>( \frac{3!}{2! \times 1!} = 3 )</td>
<td>3 ( \frac{3}{66} )</td>
</tr>
<tr>
<td>0 4 0 0</td>
<td>( \frac{4!}{4!} = 1 )</td>
<td>1 ( \frac{1}{66} )</td>
</tr>
<tr>
<td>2 3 0 0</td>
<td>( \frac{5!}{3! \times 2!} = 10 )</td>
<td>10 ( \frac{10}{66} )</td>
</tr>
<tr>
<td>2 1 1 0</td>
<td>( \frac{4!}{2! \times 1! \times 1!} = 12 )</td>
<td>12 ( \frac{12}{66} )</td>
</tr>
<tr>
<td>4 0 1 0</td>
<td>( \frac{5!}{4! \times 1!} = 5 )</td>
<td>5 ( \frac{5}{66} )</td>
</tr>
<tr>
<td>4 2 0 0</td>
<td>( \frac{6!}{4! \times 2!} = 15 )</td>
<td>15 ( \frac{15}{66} )</td>
</tr>
<tr>
<td>6 1 0 0</td>
<td>( \frac{7!}{6! \times 1!} = 7 )</td>
<td>7 ( \frac{7}{66} )</td>
</tr>
<tr>
<td>8 0 0 0</td>
<td>( \frac{8!}{8!} = 1 )</td>
<td>1 ( \frac{1}{66} )</td>
</tr>
</tbody>
</table>

### Table II. Experimental results

<table>
<thead>
<tr>
<th>Color Image</th>
<th>Res.</th>
<th>Original Size (bits)</th>
<th>Compressed Size (bits)</th>
<th>CR</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lena</td>
<td>512x512</td>
<td>6291456</td>
<td>4731162</td>
<td>25%</td>
</tr>
<tr>
<td>Baboon</td>
<td>500x480</td>
<td>5760000</td>
<td>5347825</td>
<td>7.2%</td>
</tr>
<tr>
<td>Pepper</td>
<td>512x512</td>
<td>6291456</td>
<td>4654743</td>
<td>26%</td>
</tr>
<tr>
<td>Goldhill</td>
<td>720x576</td>
<td>9953280</td>
<td>7355764</td>
<td>26%</td>
</tr>
<tr>
<td>Dreamnight</td>
<td>512x512</td>
<td>6291456</td>
<td>1157792</td>
<td>82%</td>
</tr>
<tr>
<td>Sailboat</td>
<td>512x512</td>
<td>6291456</td>
<td>5450349</td>
<td>13%</td>
</tr>
<tr>
<td>Splash</td>
<td>512x512</td>
<td>6291456</td>
<td>4223848</td>
<td>33%</td>
</tr>
</tbody>
</table>

Grayscale Images

<table>
<thead>
<tr>
<th>Color Image</th>
<th>Res.</th>
<th>Original Size (bits)</th>
<th>Compressed Size (bits)</th>
<th>CR</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lena</td>
<td>512x512</td>
<td>2097152</td>
<td>1533440</td>
<td>27%</td>
</tr>
<tr>
<td>Baboon</td>
<td>500x480</td>
<td>1920000</td>
<td>1821017</td>
<td>5.2%</td>
</tr>
<tr>
<td>Pepper</td>
<td>512x512</td>
<td>2097152</td>
<td>1514595</td>
<td>28%</td>
</tr>
<tr>
<td>Goldhill</td>
<td>720x576</td>
<td>9953280</td>
<td>2381777</td>
<td>28%</td>
</tr>
<tr>
<td>Dreamnight</td>
<td>512x512</td>
<td>6291456</td>
<td>1100866</td>
<td>48%</td>
</tr>
<tr>
<td>Sailboat</td>
<td>512x512</td>
<td>2097152</td>
<td>1632913</td>
<td>22%</td>
</tr>
<tr>
<td>Splash</td>
<td>512x512</td>
<td>2097152</td>
<td>1336786</td>
<td>36%</td>
</tr>
</tbody>
</table>

Average 29%

In the above equation, \( m, n, p, q \) represent the numbers of 1-segments, 2-segments, 4-segments and 8-segments respectively. The above equation has 10 sets of answers. There are a number of permutations for each set of answers. Table I shows all of the 10 sets of answers with the number of permutations corresponding to each of the sets. The table also shows the probability that the 8 consecutive minterms is divided in each way.

There are \( 2^8 = 256 \) possible 1, 2, 4 or 8-repetitive segments. The total number of possible 2-repetitive segments is equal to \( \left( \frac{8}{1} \right) \times \left( \frac{2}{1} \right)^2 = 2048 \). A 2-repetitive segment has compression factor of \( \frac{11}{66} \). Every 2-implicant segment has a compression factor of \( \frac{14}{66} \). Thus, the average compression factor of a 2-segment is calculated as follows.

\[
C_2 = \frac{256 \times \frac{11}{16} + 2048 \times \frac{14}{16}}{2048 + 256} \approx 0.85
\]

Equation(19)

The compression factor for a 1-segment is equal to \( \frac{19}{66} \approx 1.19 \) which is larger than unity. Now we can calculate the average compression factor for each of the 10 states in table 1. These factors will be equal to 0.6, 0.47, 0.66, 0.85, 0.93, 0.81, 0.89, 1.0, 1.10 and 1.19 respectively.

An important point to consider here is that the average compression ratio can be greater than 20% for normal images. The reason is that we have made no extra assumptions regarding dependencies among the pixels or bytes of the image while in real images, there is often a lot of dependency between the pixels and bytes. The experimental results verify this point. These results are shown in table II.

Figure 3. The Dreamnight Test Image
or applying the algorithm to text, audio and other kinds of
image into rectangular windows instead of linear segments
regenerate the original segments. Analytical modeling shows
not depend on Huffman coding. This algorithm divides the
data.

Dremnight is a simple painted image drawn by MS paint.
This image is shown in figure III. As shown in table II, the
compression ratio is larger than 25% for all images except
for Baboon. The average compression ratio is about 25%
which is quite comparable to the results reported in previous
works such as those reported by Augustine, et al. in [6], or
those obtained by Villarroya, et al. in [7].

Table III shows the results obtained by Augustine, et al.
[6].

As shown in table III, the average compression ratio of
their method (called Logic Coding) is almost 26.5.

Table IV shows the results reported in [7]. The latter
results have been obtained through applying three different
methods to CCITT fax images. These methods are called
CF-AE, CF-OBDD and SF-OBDD. As demonstrated by
table IV, the compression ratios of the used methods are
11.89, 14.68 and 18.4 respectively.

V. CONCLUSIONS AND FURTHER WORKS

This paper introduced a software tool for image
compression called YALMIC which is dependent on a novel
compression algorithm based on logic function
minimization. The proposed algorithm is lossless and does
not depend on Huffman coding. This algorithm divides the
image into segments of 1, 2, 4 or 8 octets of bits. The
segments are then modeled by disjunctive logic expressions.
YALMIC distinguishes two types of segments. The first type
consists of duplicated octets and the second consists of octets
which can construct prime implicants using the Quine-
McCluski logic minimization technique. The simplified
forms of logic expressions are stored in the compressed file
along with some extra bits which are required to fully
regenerate the original segments. Analytical modeling shows
that the average compression ratio of the algorithm will be
greater than 20%. Experimental results show that the average
compression ratio is almost 25%. This work can be
continued by enlarging the size of the segments, dividing the
image into rectangular windows instead of linear segments
or applying the algorithm to text, audio and other kinds of
data.

TABLE III. RESULTS REPORTED BY AUGUSTINE, ET AL. IN [6]

<table>
<thead>
<tr>
<th>Image</th>
<th>Method</th>
<th>Compression Ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>Girl</td>
<td>Logic coding</td>
<td>39.9</td>
</tr>
<tr>
<td></td>
<td>PVRG-JPEG</td>
<td>30.7-38.1</td>
</tr>
<tr>
<td>Mandrill</td>
<td>Logic coding</td>
<td>11.6</td>
</tr>
<tr>
<td></td>
<td>PVRG-JPEG</td>
<td>6.7-13.7</td>
</tr>
<tr>
<td>Boats</td>
<td>Logic coding</td>
<td>27.9</td>
</tr>
<tr>
<td></td>
<td>PVRG-JPEG</td>
<td>24.9-33.1</td>
</tr>
</tbody>
</table>

TABLE IV. RESULTS REPORTED BY VILLARROYA, ET AL. IN [7]

<table>
<thead>
<tr>
<th>Method</th>
<th>Average Compression Ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>CF-AE</td>
<td>11.89</td>
</tr>
<tr>
<td>CF-OBDD</td>
<td>14.68</td>
</tr>
<tr>
<td>SF-OBDD</td>
<td>18.4</td>
</tr>
</tbody>
</table>

REFERENCES

[1] Shoa A., Shirani S., Optimized Atom Position and Coefficient Coding for