Compression Using Huffman Coding on Digital Image for LSB Steganography

Dionesius A Perkasa – 13516043[1]
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jl. Ganesha 10 Bandung 40132, Indonesia
[1]13516043@std.stei.itb.ac.id

Abstract—**LSB steganography is one common method in digital steganography. Its implementation is simple but it has several limitations. One limitation is that the size of the cover image is much larger than the file concealed within it. To solve this limitation, the author attempts to compress the concealed file in the form of an image using Huffman coding method in order to reduce its size and therefore reduce the size required to hide that image.**

Keywords—**Data compression, digital image, Huffman coding, LSB method, steganography.**

I. INTRODUCTION

Steganography is the practice of hiding an information or file within another file. One common method used in digital steganography is LSB steganography. LSB method simply replaces the last bit of every byte of an image with the bits of the hidden information or file.

For a byte of concealed data, it needs 8 bytes of data to cover it. This means, for a standard 8-bit grayscale image, it requires 8 times the size of that image to conceal it. The cover-to-hidden data ratio is 8:1. This ratio is amplified if the hidden image is colored. For an RGB image, with R, G, and B channel, it needs another image with $8 \times 3 = 24$ times its size to be hidden. The ratio becomes 24:1.

This becomes a problem if the size of the file which is to be hidden is large. It needs an even larger cover file to be concealed within. One solution for this problem is to reduce the size or compress the hidden file so that it doesn't need a very large space to be concealed.

To reduce the size of a file, a data compression algorithm can be used. A common and simple algorithm for data compression is Huffman coding. It was first introduced in 1952 by David A. Huffman. It uses statistical information in a file or data in order to reduce the average length of representing bits or binary codes which correspond to all symbols in the file, therefore reducing the file size itself.

A data compression does not change the amount of vital information in a data. It just changes the way the data is represented, often by reducing the length of the representing binary codes. It stores the data in a more efficient way.

In this paper, a series of attempts to compress data are made. The data used are in the form of 8-bit colored and grayscale bitmap image files with various sizes. In total, there are 6 images, 3 colored images and 3 grayscale images. An average of compression ratio will be calculated for colored and grayscale images and these information will be used in calculating theoretical cover-to-hidden data ratio.

II. DATA COMPRESSION

Compression is the process of reducing the data quantity used to represent a file without excessively reducing the quality of the original data. It also reduces the number of bits required to store and transmit digital media [1]. There are some techniques in acquiring the purpose, one of them is to reduce redundant information within the file. Another one is to simply throw away the less important parts of the data and keep the important ones.

A. Digital Data Representation

Digital data consist of a sequence of symbols from a set of finite alphabet. For a data compression to still contain meaningful information, there is a standard representation for the original data that codes each symbol using the same number of bits. For a text file, for example, every symbol is represented by an ASCII code, which is a one byte long binary code that corresponds to every symbol in a standard keyboard.

A data compression is successful when the compressed data can be represented by shorter codes on average than the original data. So for a compression to be meaningful, there must be a standard representation canonized for the data compression.

B. Types of Data Compression

There are two main classifications of data compression based on the information retention.

Lossy compression means that some information is lost during the compression process. Lossy compression is based on the fact that there are some limitations on what human sensory ability can and cannot perceive. Thus some little information in the file whose presence human can't really tell can be removed. It is commonly used in the compression of media files such as MPEG and MP3 compression.

Lossless compression means that no information is lost during the compression process. When the data is decompressed, the result matches perfectly bit-by-bit with the original data. The compressed data uses less space than the original data, but no information is removed, thus making it more efficient. An example of program that uses lossless compression is the popular compressing program WinZIP. One example of lossless data compression is Huffman coding.

C. Data Compression Techniques

Simple repetition is simply replacing series of successive symbols which occur in a sequence with another token or flag. For example, 319607000000000000000000000000000000 can be replaced with 319607z30. The symbol 'z' is the flag for zero.

Application of simple repetition includes zero length suppression (such as the example above), silence in audio files, images with bitmap format, and whitespaces (blanks, new line symbols, or tabs) in text files.

Run-length encoding or ***RLE*** is the method to rewrite the data as pairs of values (v,n) with v is the value (for example, in the case of an image, the color value) and n is the number of successive occurrence. As an example, see Fig. 1.

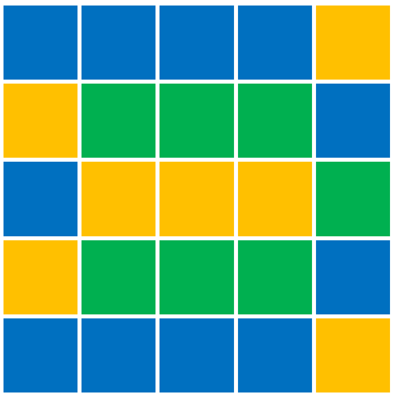


Fig. 1. Simple three colored 5x5 grid image

First, a symbol is assigned to each of the colors, B for blue, Y for yellow, and G for green. The image is stored with the symbol representation, i.e. BBBBYYGGGBBYYYGYGGGBBBBBY. The image size is 25 characters long.

Now RLE is applied on the symbol representation using the value pairs (v_i,n_i). For example, the first reoccurring color is 4 blocks of blue. For these 4 blocks of blue, the pair is (B,4). Doing this for all the data, the result is (B,4), (Y,2), (G,3), (B,2), (Y,3), (G,1), (Y,1), (G,3), (B,5), (G,1). To store this compressed data, the representation is B4Y2G3B2Y3G1Y1G3B5G1, which is just 80% of the original size.

The disadvantage of RLE is that if the image or data is too irregular or has too much noise, the compression might yield a data with larger size than the original. RLE is used as a complementary method in JPEG compression.

Huffman coding is an algorithm for lossless data compression. The concepts and algorithm for Huffman coding will be discussed further in another section.

III. HUFFMAN CODING

Huffman Coding, first introduced by David A. Huffman in 1952, is an algorithm for lossless data compression. The concept is to assign some codes with variable length to input characters. The length of the code is based on the frequency of its corresponding characters. Character with the most occurrence gets the shortest code and the one with the least occurrence gets the longest code.

In order to achieve an effective and non-ambiguous set of code for a particular set of input characters, there are some restrictions in the coding. The following basic restrictions will be imposed on an ensemble code:

(a) There will not be 2 messages consisting of identical arrangements of coding digits.

(b) The message codes will be constructed in a way such that there is no need of any additional indication to specify where a message code begins and ends once the starting point of a sequence of messages is known.

As Huffman [2] stated, restriction (b) necessitates that no message be coded in such a way that its code appears, digit for digit, as the first part of any message code of greater length. It is to state that no code should be a prefix code of any other code.

A. Prefix Codes

To understand prefix codes, look at this little example below. Let there be four input characters A, B, C, D, and their corresponding codes 0, 1, 00, and 01 respectively. This way of coding leads to ambiguity since character A's code is a prefix of the codes assigned to C and D. For example, if the result of compression is 00100100, the decompressed original data might be CBADC, AABCAD, ADADC, or some other possibilities.

Now let's see another way of code assigning. Let's assign codes 00, 01, 10, 11 to input characters A, B, C, and D. If we now get the same string of compression result as above (00100100), we can be sure that the decompressed original data is ACBA. There's no ambiguity in this coding.

B. Generating a Huffman Tree

An efficient way to assign codes to a set of input characters is by using a Huffman tree. A Huffman tree is a binary tree for determining what code should be assigned to which character. The algorithm for generating a Huffman tree for a text file, referencing from [3], is as follows.

1. Count the occurrence frequency of every symbol in the text.
2. Take two symbols with the least occurrences (e.g. P and Q which, for example, have 1/7 probability each) and treat them both as parent nodes.
3. Make parent nodes from those two nodes so that there is a new symbol PQ with 1/7 + 1/7 = 2/7 probability.
4. Take the next two symbols, including the new symbol, with the least occurrences. Do step 3 so that another new symbol with its probability is acquired.
5. Repeat step 4 until there is one parent node which represents every symbol and has the probability of occurrence 1.
6. Label every node in a way that the left branches are labeled 0 and the right branches are labeled 1.
7. The label on every leaf corresponds to the symbol in which the leaf represents.

With this algorithm, the least frequent symbols will correspond to the relatively longer codes and the most frequent symbols will correspond to the relatively shorter ones. This also ensures that no code is a prefix of any other code, eliminating any ambiguity. See Fig. 2 for an example.

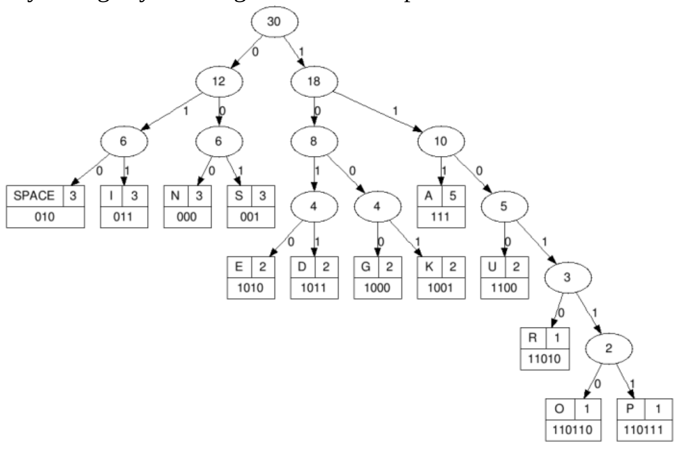


Fig. 2. Huffman tree for the string that constructs the author's name, "DIONESIUS AGUNG ANDIKA PERKASA" and the corresponding code for each symbol.
(Source: http://huffman.ooz.ie/?text=DIONESIUS%20AGUNG%20ANDIKA%20PERKASA)

B. Huffman Coding for Digital Images

Compressing digital images using Huffman coding is similar with using Huffman coding for compressing text files. One major difference is that in a digital image, there are some bytes in the beginning of the file which serve as the file header. This file header contains information about the file itself. It describes how bits are used to encode information in a digital storage. This file header cannot be altered, thus the compression process skips this section.

The next section is the actual file itself. This section is variable-sized. It also can be modified. However, if a special modification is performed, such as changing the binary representation, a new file header that informs how to read the file is needed. Therefore, a new file format might be needed for storing this new compressed file.

An illustration in Fig. 3 describes how a grayscale image can be compressed using Huffman coding.

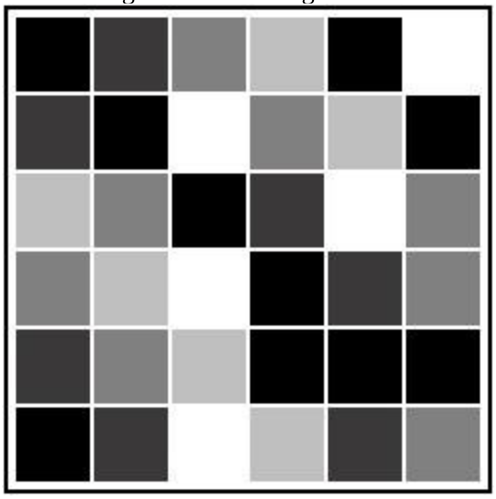


Fig. 3. Grayscale 6x6 grid image

In Fig. 3, a 1x1 square represents a pixel. The value of a pixel in grayscale images is represented in integer from 0 to 255 in decimal or 00000000 to 11111111 in binary (8-bit). Value 0 refers to black and value 255 refers to white. In the image there are 5 values of gray. Let's say, from darkest to lightest, they are 0, 51, 102, 153, and 255 (00000000, 00110011, 01100110, 10011001, and 11111111 in binary).

Suppose that the image in Fig. 3 is in 8 bit-per-pixel (8bpp) bitmap format which has at least 21 bits of header file. Then the file size is $(36 \times 8) + 21 = 309$ bits or 39 bytes. The binary code of image file is like this.

```
<21 bits of file header> ...

00000000 00110011 01100110 10011001
00000000 11111111 00110011 00000000
11111111 01100110 10011001 00000000
10011001 01100110 00000000 00110011
11111111 01100110 01100110 10011001
11111111 00000000 00110011 01100110
00110011 01100110 10011001 00000000
00000000 00000000 00000000 00110011
11111111 10011001 00110011 01100110
```

A compression will be performed onto the image. Similar with Huffman coding for text files, a Huffman tree is generated to make Huffman codes for the corresponding pixel values.

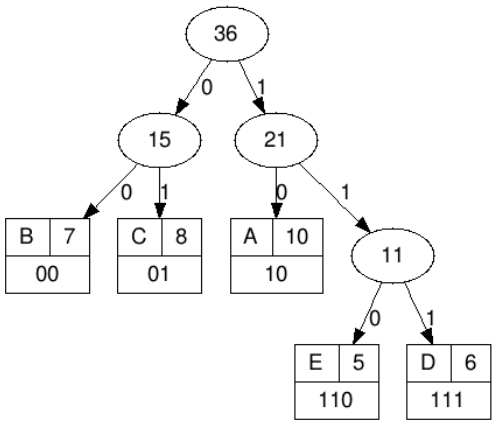


Fig. 4. Huffman tree for image in Fig. 3. A refers to value 0, B to 51, C to 102, D to 153, and E to 255.
(Source: http://huffman.ooz.ie/?text=ABCDAEBAECDADCABECCDEABCBCDAAAABEDBC)

As seen in Fig. 4, the code length for every color is decreased from 8 bits per pixel to on average 2.4 bits per pixel. Now replacing the bytes with Huffman codes will yield this.

```
10 00 01 111 10 110 00 10 110 01 111
10 111 01 10 00 110 01 01 111 110 10
00 01 00 01 111 10 10 10 10 00 110
111 00 01
```

Suppose the header, despite contains different information, increases to 30 bits, the size of the compressed file is

$$\big((5+6)\times 3\big) + \big((7+8+10)\times 2\big) + 30 = 219 \text{ bits}$$

or 27 bytes. This compression has a compression ratio of 1.4:1.

IV. Steganography

The word *steganography* comes from two Greek words, *steganos* meaning "concealed" or "protected" and *graphein* meaning "writing". As defined by D. Kundur and K. Ahsan [4], steganography is the process of discreetly hiding data in a given host carrier with the intent to enhance value or concealedly exchange information.

The host carrier is any message of some sort which contains redundancy or irrelevancy. For example, digital image files are often used to embed secret information; the limitation of human vision in noticing subtle differences between hues allows data hiding of this type of media.

A. Steganography on Digital Images

According to A. A. J. Altaay, S. bin Sahib, and M. Zamani [5], steganography has 3 important measurements: capacity, imperceptability, and robustness.

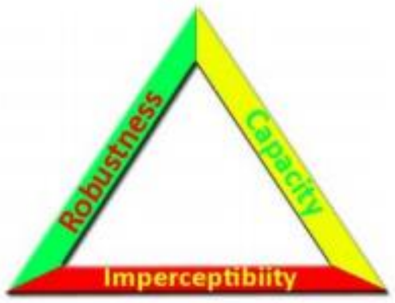


Fig. 5. Measurement triangle of steganography [5] (Source: [5])

Capacity is the maximum size of secret information can be embedded in a file. As explained in [5],

Capacity either can be defined as an absolute value in term of number of bits for particular cover or as a relative number regarding necessary bits to save final stego file (Altaay, 2012).

Capacity value depends on embedding function and cover properties. For example, for an 8-bit grayscale image as a cover in LSB steganography technique with 1 bit per pixel embedding, the capacity would be equal to or less than 1/8 or 12.5% of the image size since the header, the first 7 bits of every byte is not embeddable.

A stego image should not have important perceptual artifact, hence ***imperceptability***. The higher fidelity of the stego image, the more imperceptable it is. This is to mention that the stego image and the original image must not be distinguishable.

One important method for measuring imperceptability is called Peak Signal to Noise Ratio (PSNR). PSNR is a metric to evaluate the ratio between possible peak signal and effect of noise caused by manipulation to fidelity of its representation. This method is formulated as follows.

$$\mathrm{PSNR} = 10 \times \log_{10}\left(\frac{\mathrm{MAX}_1^2}{\mathrm{MSE}}\right)$$

$$\mathrm{MSE} = \frac{1}{mn}\sum_{i=0}^{m-1}\sum_{j=0}^{n-1}[I(i,j) - K(i,j)]^2$$

Robustness is property of harness of eliminating secret information from stego image. In other words, it is the resistance level of the stego image when being intentionally distorted by another party. Robustness metrics of steganographic algorithms have distortion classifications such as geometric transformations or additive noise [5].

V. LSB STEGANOGRAPHY

Least significant bit steganography or LSB steganography one of the main techniques in spatial domain image steganography [6]. It simply makes use of the fact that the level of precision in image formats (e.g. the RGB levels) is far greater than what the average human eye can perceive. This means, if the color value of an image is altered slightly, the altered image will be indistinguishable from the original image without the aid of a computer.

The standard 8-bit image LSB steganography with 1 bit-per-byte embedding, which means 1 bit of secret information is embedded in every byte of the image, requires eight bytes of pixels to store 1 byte of secret information. That is why, in the previous chapter, the theoretical maximum size of the secret information can be hidden is only 1/8 or 12.5% the size of the image.

7	6	5	4	3	2	1	0

Secret information (size: 1 byte)

7	6	5	4	3	2	1	0
7	6	5	4	3	2	1	1
7	6	5	4	3	2	1	2
7	6	5	4	3	2	1	3
7	6	5	4	3	2	1	4
7	6	5	4	3	2	1	5
7	6	5	4	3	2	1	6
7	6	5	4	3	2	1	7

Embedded image bytes (size: 8 byte)

Secret-information-to-image size ratio: 1/8 = 12.5%

Fig. 6. Illustration of LSB steganography and how the secret information bits are stored within the image.

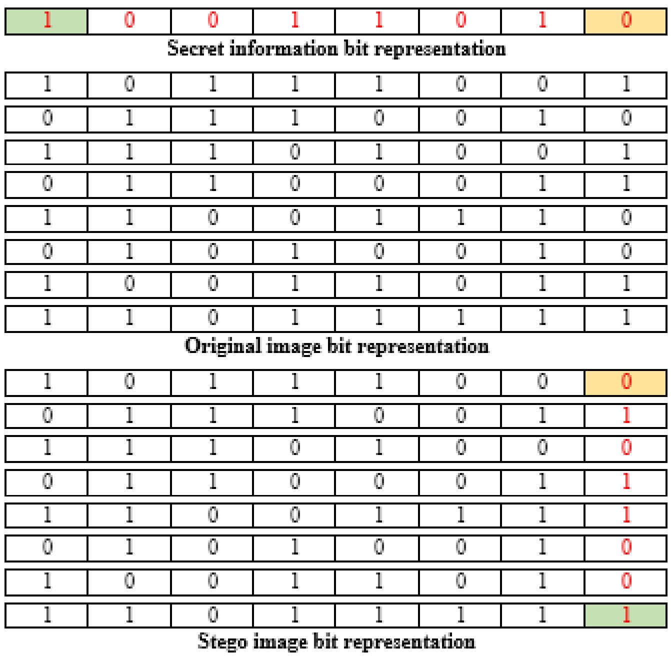
As illustrated in Fig. 6, bits of the secret information are spread into 8 different bytes of the image.

A. Embedding Secret Information

As LSB steganography replaces only the last bit of every pixels, the method of embedding secret information bits into the cover image is quite simple. The following is the steps to embedding the bits.

1. If the information bit value is 1 and the pixel value modulo 2 is 0, increase the pixel value by 1.
2. If the information bit value is 0 and the pixel value modulo 2 is 1, decrease the pixel value by 1.
3. If the information bit value is equal to the pixel value modulo 2, no alteration needed and jump to the next information bit and next pixel.

See the example below for more explanation.



1	0	0	1	1	0	1	0

Secret information bit representation

1	0	1	1	1	0	0	1
0	1	1	1	0	0	1	0
1	1	1	0	1	0	0	1
0	1	1	0	0	0	1	1
1	1	0	0	1	1	1	0
0	1	0	1	0	0	1	0
1	0	0	1	1	0	1	1
1	1	0	1	1	1	1	1

Original image bit representation

1	0	1	1	1	0	0	0
0	1	1	1	0	0	1	1
1	1	1	0	1	0	0	0
0	1	1	0	0	0	1	1
1	1	0	0	1	1	1	1
0	1	0	1	0	0	1	0
1	0	0	1	1	0	1	0
1	1	0	1	1	1	1	1

Stego image bit representation

Fig. 7. Binary representation of secret information, original image, and stego image.

Note that the secret information in Fig. 7 is stored backwards, the last bit is stored first.

In Fig. 7, assume that the image used is grayscale image, the gray values of the original image are as follows.

- 1st pixel : $10111001_2 = 185_{10}$
- 2nd pixel : $01110010_2 = 114_{10}$
- 3rd pixel : $11101001_2 = 233_{10}$
- 4th pixel : $01100011_2 = 99_{10}$
- 5th pixel : $11001110_2 = 206_{10}$
- 6th pixel : $01010010_2 = 82_{10}$
- 7th pixel : $10011011_2 = 155_{10}$
- 8th pixel : $11011111_2 = 223_{10}$

Now, doing the steps mentioned above will yield the stego image as in Fig. 7.

1. 1st pixel value is 185 and the secret information bit value is 0. Because 185 mod 2 = 1, decrease the pixel value by 1.
2. 2nd pixel value is 114 and the secret information bit value is 1. Because 114 mod 2 = 0, increase the pixel value by 1.
3. 3rd pixel value is 233 and the secret information bit value is 0. Because 233 mod 2 = 1, decrease the pixel value by 1.
4. 4th pixel value is 99 and the secret information bit value is 1. Because 99 mod 2 = 1, skip and jump to the next step.
5. 5th pixel value is 206 and the secret information bit value is 1. Because 206 mod 2 = 0, increase the pixel value by 1.
6. 3rd pixel value is 82 and the secret information bit value is 0. Because 82 mod 2 = 0, skip and jump to the next step.
7. 7th pixel value is 155 and the secret information bit value is 0. Because 155 mod 2 = 1, decrease the pixel value by 1.
8. 8th pixel value is 223 and the secret information bit value is 1. Because 223 mod 2 = 1, leave it as it is.

As elaborated before, a difference of 1 or 2 levels in the color value of an image doesn't change the way human eyes see them. They are still perceived as pretty much the same color by a person with an average vision. For an 8-bit grayscale image, gray value 103 and 104 are pretty much the same gray. With this method, the stego image will be altered at most by 1 level for each pixel value from the original image, hence no perceivable difference between the two images.

B. *Retrieving Secret Information from a Stego Image*

Retrieving secret information from a stego image is just the matter of reversing the embedding process. The steps to extracting the secret information embedded in an image is as follow.

1. Calculate the difference for every pixel value between the original and the stego image. The difference for every pixel must be -1, 0, or 1.
2. If the difference is -1, the information bit value is 0.
3. If the difference is 1, the information bit value is 1.
4. If the difference is 0, the information bit value is equal to the byte's LSB of the stego image.

The steps above can be expressed mathematically as below.

$$I_n = \begin{cases} 0, & B_{S_n} - B_{O_n} = -1 \\ 1, & B_{S_n} - B_{O_n} = 1 \\ \mathrm{LSB}\left(B_{S_n}\right), & B_{S_n} - B_{O_n} = 0 \end{cases}$$

$$I = I_1 I_2 I_3 \ldots I_n$$

I_n is the pixel value difference between nth byte of the stego image (B_{S_n}) and original image (B_{O_n}) and I is the secret information.

Take the previous example for a quick calculation (note that I_8 is on the first pixel, I_7 is on the second pixel, and so on).

1. $10111000_2 - 10111001_2 = -1 \longrightarrow I_8 = 0$
2. $01110011_2 - 01110010_2 = 1 \longrightarrow I_7 = 1$
3. $11101000_2 - 11101001_2 = -1 \longrightarrow I_6 = 0$
4. $01100011_2 - 01100011_2 = 0 \longrightarrow I_5 = 1$
5. $11001111_2 - 11001110_2 = 1 \longrightarrow I_4 = 1$
6. $01010010_2 - 01010010_2 = 0 \longrightarrow I_3 = 0$
7. $10011010_2 - 10011011_2 = -1 \longrightarrow I_2 = 0$
8. $11011111_2 - 11011111_2 = 0 \longrightarrow I_1 = 1$

Now that I_1 to I_8 is acquired, the final step is to arrange I_1 to I_8 so that the secret information is constructed.

$$\begin{aligned} I &= I_1 I_2 I_3 \ldots I_n \\ I &= I_1 I_2 I_3 I_4 I_5 I_6 I_7 I_8 \\ I &= 10011010 \end{aligned}$$

V. Methodology

An experiment was conducted to find the average compression ratio of Huffman coding for bitmap image compressing. The compressed image will then be hidden into a cover image (the *carrier*) by LSB steganography.

In theory, after the image compression, the image size will be smaller, thus an information (image) whose original size is larger than 12.5% the carrier size (the standard maximum information size for 8-bit LSB steganography) can be embedded.

The test images and Huffman coding result can be viewed in Fig. 8 and Table 1 below.

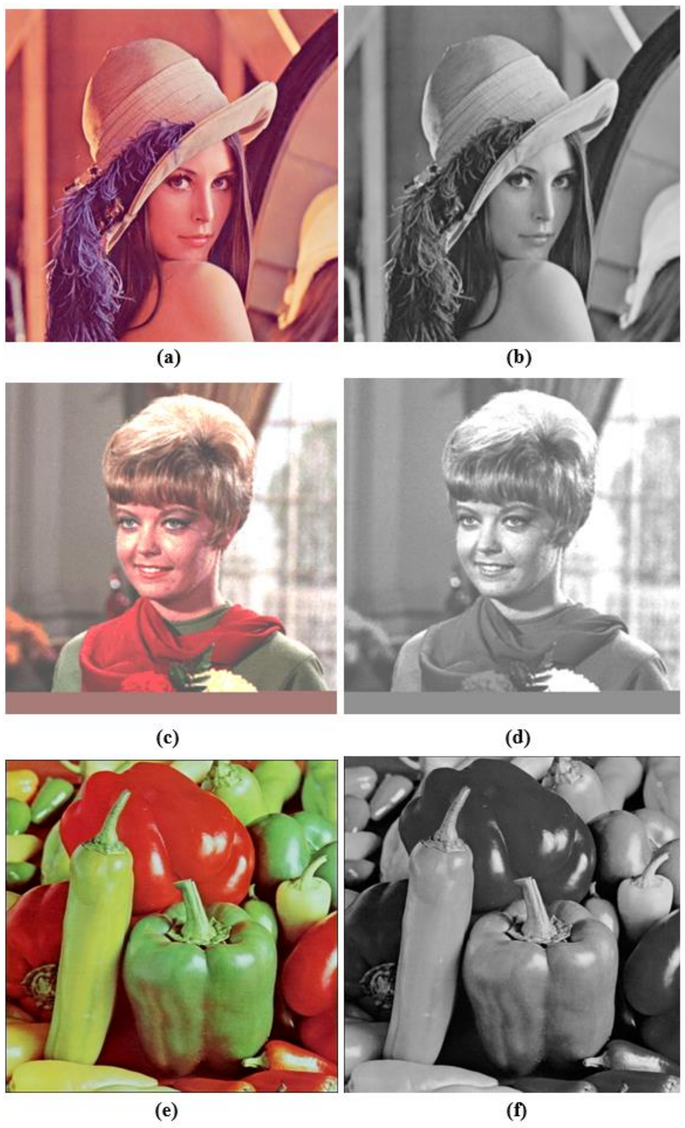


Fig. 8. Test images: **(a)** lena-color.bmp, **(b)** lena-grayscale.bmp, **(c)** girl-color.bmp, **(d)** girl-grayscale.bmp, **(e)** peppers-color.bmp, and **(f)** peppers-grayscale.bmp.
(Source: http://informatika.stei.itb.ac.id/~rinaldi.munir/Koleksi/Citra%20Uji/CitraUji.htm)

TABLE 1
HUFFMAN CODING RESULT FOR 6 IMAGES

Name	Type & Format	Original Size	Cmp’d Size	Cmp’n Ratio
lena-color.bmp	Colored bitmap image	786.5 kB	766.0 kB	1.03:1
girl-color.bmp		66.6 kB	65.3 kB	1.02:1
peppers-color.bmp		786.5 kB	756.9 kB	1.04:1
lena-gray.bmp	Grayscale bitmap image	66.6 kB	54.1 kB	1.23:1
girl-gray.bmp		66.6 kB	63.3 kB	1.05:1
peppers-gray.bmp		263.2 kB	251.2 kB	1.05:1

Based on the test result, the average compression ratio for colored bitmap image is

$$\mathrm{CR_C} = \frac{1.03 + 1.02 + 1.04}{3} = 1.03$$

and the average compression ratio for grayscale bitmap image is

$$\mathrm{CR_G} = \frac{1.23 + 1.05 + 1.05}{3} = 1.11$$

VI. THEORETICAL CALCULATIONS

Based on the conducted experiment, the average compression ratio for color bitmap image is 1.03 and for grayscale bitmap image is 1.11.

If the concealed file (secret information) and the cover image are grayscale images, with Huffman coding, the maximum information size is equivalent to $\mathrm{I_{max}} = 1.11 \times 12.5\% = 13.80\%$ the size of the cover image, which means the cover-to-hidden ratio is decreased from 8:1 to 7.21:1.

VII. CONCLUSION AND FURTHER WORKS

Based on the test result, it can be inferred that Huffman coding is more effective and gives a bigger compression ratio if there are less color value variations within the image. The average compression ratio for colored bitmap image is 1.03:1 and for grayscale bitmap image is 1.11:1.

With Huffman encoding, the maximum information size that can be embedded into a cover image using LSB Steganography can be increased from 12.5% to 13.88%.

For further efforts, the author hopes to improve his huffman coding algorithm source code as his code at the time of this paper being released doesn’t perform very well at compressing bitmap images, but performs really well at compressing text files.

Another works in the future includes researching the actual possible maximum information after doing Huffman compression and comparing the result with the theoretical results calculated in this paper.

VIII. ACKNOWLEDGMENT

The author is highly grateful to God Almighty for His amazing grace and blessings so that the author can finish this paper. Also to Dr. Ir. Rinaldi Munir, M.T. as the lecturer of IF2120 Discrete Mathematics K01 for his dedication in lecturing his students for this semester. The author also wishes to express his gratitude to his parents, brother, and friends for their support

in finishing this paper.

REFERENCES

[1] M. Sharma, "Compression Using Huffman Coding," *International Journal of Computer Science and Network Security,* vol. 10, no. 5, 2010.

[2] D. A. Huffman, "A method for the Construction of Minimum-Redundancy Codes," *Proceedings of the IRE,* September 1952.

[3] R. Munir, Matematika Diskrit, Bandung: Departemen Teknik Informatika Institut Teknologi Bandung, 2003.

[4] D. Kundur and K. Ahsan, "Practical Internet Steganography: Data Hiding in IP," Texas Wksp. Security of Information Systems, College Station, 2003.

[5] A. A. J. Altaay, S. bin Shahib and M. Zamani, "An Introduction to Image Steganography Techniques," in *2012 International Conference on Advanced Computer Science Applications and Technologies*, 2012.

[6] B. S. Champakamala, K. Padmini and D. K. Radhika, "Least Significant Bit algorithm for image steganography," *International Journal of Advanced Computer Technology,* vol. 3, no. 4, 2015.

PERNYATAAN

Dengan ini saya menyatakan bahwa makalah yang saya tulis ini adalah tulisan saya sendiri, bukan saduran, atau terjemahan dari makalah orang lain, dan bukan plagiasi.

Bandung, 3 Desember 2017

[illegible]

Dionesius Agung Andika Perkasa
13516043