Steganography scheme based on DCT coefficients

Author: Eric Vidal *Supervisor:* Aurélien Coillet

***Abstract*—Steganography is the art and science of covert communication. This paper presents a steganography method based on exchanging DCT coefficients by secret message bits. Its embedding position depends on a hash function to reinforce the security procedure. Moreover, the Static Huffman code compression algorithm is used to codify and shorten the secret message embedded.**

We studied the cases of hiding a message string of characters and hiding a one-layer image. The cover image used was larger than the secret one, and we also applied the procedure for an RGB cover image. In both cases, the cover image and the resulting stego image are highly similar based on their structural similarity index (SSIM), which is a standard indicator for comparing similarity and quality between two images. It was a special success that the RGB image achieved a 0.999989 SSIM between the cover image and the stego image, although it was the most complex procedure.

Furthermore, once the message had been extracted from the stego image, we reconstructed the image. The SSIM comparing the reconstructed image with the stego image was equal to 1 in all cases, as expected.

***Index Terms*—Steganography, block DCT, static Huffman code, hash function**

I. Introduction

STEGANOGRAPHY is the practice of concealing data within an innocuous-looking cover object. In today's digital world, the old ways of hiding secret messages have been replaced by more versatile and practical covers, such as digital documents, images, videos, and audio files. This paper is focused on stego images which consist of cover images that embed a secret message. The goal is to prevent the human eye and even steganalysis tools from detecting if an image has been modified to embed a secret in it.

Steganography is applied in a large range of applications, e.g., enhancing the robustness of ID photos that contain personal information, as well as, copyright control materials, among many other applications. One of the most interesting applications is Medical Imaging Systems, to preserve patients' privacy. When a test image is digitized, as it could be an X-ray or MRI test, private information such as the name, address, physicians, and illnesses, can indeed be contained in these medical results. A cutting-edge steganography method for this purpose was published by Xin Liao et al. in [1] .

Since there are plenty of steganography image processing methods, they can be classified. An accurate description and classification has been done by Abbas Cheddad et al. in a state-of-the-art review of image steganography [2]. To sum up, steganography methods can be classified into three main groups:

E. Vidal and A. Coillet are with the Faculty of Physics, University of Burgundy, Dijon, France.

- *Spatial domain*: The secret data is encoded directly in the least significant beat (LSB) of the cover image pixels. It provides a high capacity for insertion and low perception. However, it is vulnerable to being lost due to compression, and what's more, most current steganalysis methods can detect and extract the secret image from LSB-embedded images. It means that non-authorized users could eavesdrop on private information.
- *Frequency domain*: This technique emerged as a consequence of the information theory development and new mathematical tools available, such as the Discrete Fourier Transform (DFT), Discrete Cosine Transform (DCT), and Discrete Wavelet Transform (DWT). Hence, the mathematical transformations enable us to hide secret information in the frequency domain image. Schemes linked with these mathematical tools make the stego image more resistant to attacks compared to LSB methods.
- *Adaptive methods*: Despite frequency domain techniques meant a huge improvement in terms of security compared to LSB, these can be enhanced. The new attribute that comes into play is studying the statistical global features of the image before applying the LSB/DCT coefficients.

The main purpose of this paper is to develop a code that can perform a basic DCT process, similar to a scheme proposed by Mandal, J.K. in [3]. Firstly, DCT blocks of the cover image are computed, and then the secret image is codified using the Huffman tree algorithm, and embedded in the off-diagonal elements using a hash function.

This scheme could be improved by undertaking quantization steps of the DCT blocks. A suitable way could be the F5 method [[4]]. Apart from that, one could define cost functions for deciding which are the most suitable blocks to modify as explained by C. Wang and J. Ni in [5]. Note that this paper targets to succeed in its educational purpose, mostly focused on the fundamental aspects of frequency domain steganography, far from intending to improve the current methods that are being investigated.

II. Proposed scheme

A. Discrete cosine transform (DCT)

The DCT consists of a finite sequence of data elements that conform to a sum of cosine functions oscillating at different frequencies. One of the most important properties is decorrelation and DCT importance lies in image formats. The DCT can be defined in 8 different analytical ways, we will use the most common one known as the DCT Type II. In `scipy.fftpack.dct` Python function is defined as,

$$y_k = 2 \sum_{n=0}^{N-1} x_n \cos\left(\frac{\pi}{2N}(2n+1)\ k\right) \tag{1}$$

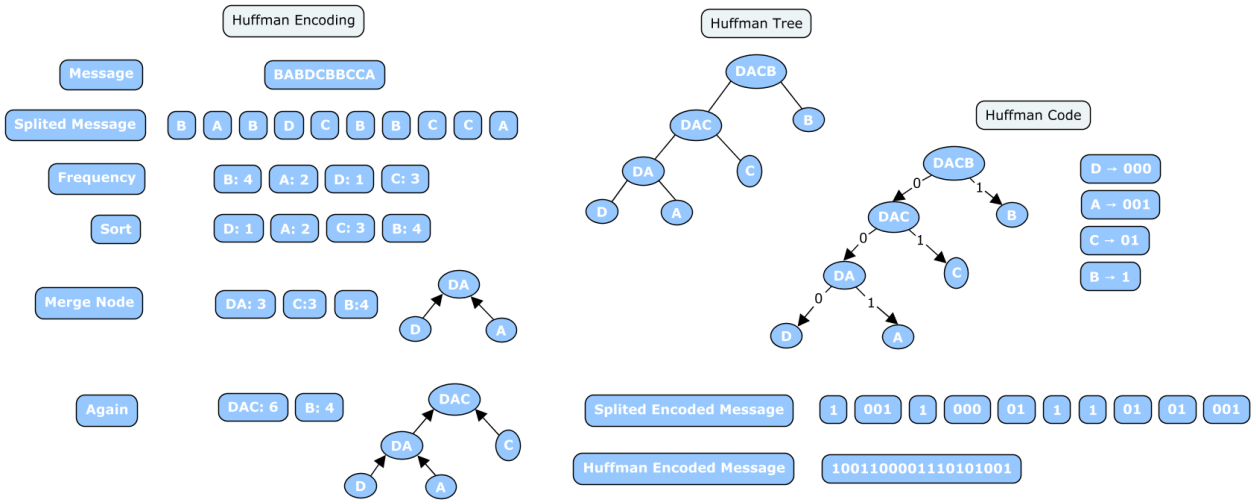

Fig. 1. This is an exemplifying diagram that represents, step by step, how to encode a message using the static Huffman Encoding algorithm.

with x_n being an element of the finite sequence of N data points, and $k = 0, 1, 2, \ldots, N-1$. For normalizing it is multiplied by a scaling factor,

$$\alpha_k = \begin{cases} \sqrt{\frac{1}{4N}} & \text{if } k = 0 \\ \sqrt{\frac{1}{2N}} & \text{otherwise} \end{cases}$$

The inverse DCT (IDCT) for the DCT Type II orthonormalized is exactly defined by the DCT Type III orthonormalized,

$$y_k = \frac{x_0}{\sqrt{N}} + \sqrt{\frac{2}{N}} \sum_{n=1}^{N-1} x_n \cos\left(\frac{\pi}{2N}(2k+1)\, n\right) \qquad (2)$$

with x_n being an element of the finite sequence of N data points, and $k = 0, 1, 2, \ldots, N-1$, as before.

From these definitions, we can extract interesting properties. For example, the first coefficient of the DCT is kind of the average value of the sample, and it receives the name of DC coefficient, the other ones are known as AC coefficients. It is known that images are 2D arrays so they have to be applied to a 2D DCT. Due to DCT linear properties, we can compute the 1D DCT along one axis and then along the other one. Hence, diagonal elements have more impact on the non-transformed image. This is the reason why the secret message is embedded only at the non-diagonal coefficients.

DCT scheme is applied by dividing the cover image into blocks of dimensions $n \times n$ to apply the 2D DCT in each block. A large number of methods use 8×8 blocks, because of their efficiency, as shown in [1] and [4].

Implementation of the DCT

1) Divide image in blocks.
2) Compute DCT of each block.
3) Embed the secret image.
4) Compute the IDCT.
5) Put every block together to build the stego image.

B. Static Huffman code

The effectiveness and quality of stego image not only relies on the method applied, but the length of the message embedded also plays an important role. Consequently, the shorter the message gets the better will be the stego image in comparison with the cover image. Apart from that, more space will be available to embed more information, i.e., a bigger image or more characters of a string message in the cases that have been considered. For the sake of the process, the static Huffman algorithm is used to encode the secret message.

Huffman tree algorithm

1) Calculate the frequency of each character.
2) Sort nodes in increasing order of frequency.
3) Make each node as a leaf tree.
4) Create an empty node *. Assign the minimum frequency to the right child of * and the second minimum to the left child. The frequency of the new node * is the sum of both children's frequencies and the new node (*) symbol becomes the sum of the two symbols.
5) Remove the two minimum frequencies that were used for creating * and put now * in the queue with the rest of the leaf.
6) Repeat steps 4 and 5 till there is just one remaining node which is the treetop.
7) For each none-leaf node, assign 0 to the right and 1 to the left.

All steps described are illustrated in Fig. 1. Once the Huffman tree is built, the corresponding Huffman Code (i. e. string of 0s and 1s) to each symbol can be obtained. This process starts from the top tree and goes down through the tree until reaches the leaf associated with the desired symbol. Every time it turns right means that a 0 has to be added to the new value and every time it turns left a 1 has to be added until the symbol leaf is reached. It is arbitrary to assign 0 to the right and 1 to the left or vice-versa, as soon as the decoding is done in the same fancy. Finally, one gets the Huffman codes for every symbol.

Decoding Huffman code:

1) Read the first value of the encoded message.
2) Go to the top of the tree.
3) If the read value is 1 go to the right node, if it is 0 go to the left node.
4) Repeat steps 2 and 3 and change to the following value of the encoded message. When a leaf is reached save its symbol and restart from the top again.

The static Huffman code method explanation and coding are mainly based on [6], [7], [8] and [9].

C. Algorithm scheme

Embedding part:

Inputs: Cover image $(N_{row} \times N_{col})$ and secret message $(N_2 \times N_2)$.
Outputs: Stego image $(N_{row} \times N_{col})$ and the decode dictionary.
Method: The scheme followed in the embedding part is described in Fig. 2.

1) If the side length of the cover image does not fit with the chosen square length of the blocks, either the cover image must be re-scaled or cut, or the square length reconsidered. Another possibility for remaining with the parameters chosen would be taking the cover image until the squares fit, saving its untouched borders, and eventually adding them to the square stego image with the secret message embedded.
2) Divide the cover image into blocks of the fixed square length and compute the 2D DCT on each block.
3) Get the Huffman code of the secret message (i. e. the string of 0s and 1s to be embedded) and the Huffman tree using the Huffman code algorithm.
4) Introduce every bit of the message in the cover image using the hash function. The hash function returns the bit position of the pixel where the corresponding secret message bit has to be embedded. We have defined it as computing the Python function `mod`, which returns the division remainder between the sum of the cover image pixel row and column and 3, i.e., `(Row+Column) % 3`.
5) Once the message is embedded, compute the IDCT of each block and join them to build the square stego image. Using the borders previously saved construct the final stego image.
6) The final outputs are the stego image and the Huffman tree.

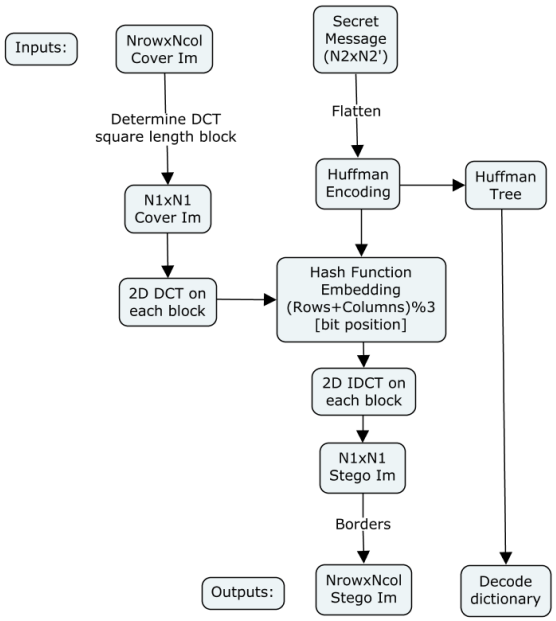

Fig. 2. Diagram flow of how to embed a secret message and create the stego image via DCT coefficients.

Extracting part:

Inputs: Stego image $(N_{row} \times N_{col})$ and decode dictionary.
Outputs: Reconstructed image $(N_{row} \times N_{col})$ and secret message.
Method: The scheme followed in the extracting part is described in Fig. 3.

1) If the side length of the cover image does not fit with the accorded square length of the blocks, it means that the message is embedded until the last square that fits in the stego image. One should save its untouched borders if one wants to reconstruct the image.
2) Divide the stego image into blocks and compute the 2D DCT on each block.
3) Extract the secret message using the hash function, which is the same as explained before.
4) Decode the secret message using the decode dictionary.
5) Once the message is extracted, compute the IDCT of each block.
6) The final image is assembled with the saved borders, so finally the reconstructed image and the secret message are decoded.

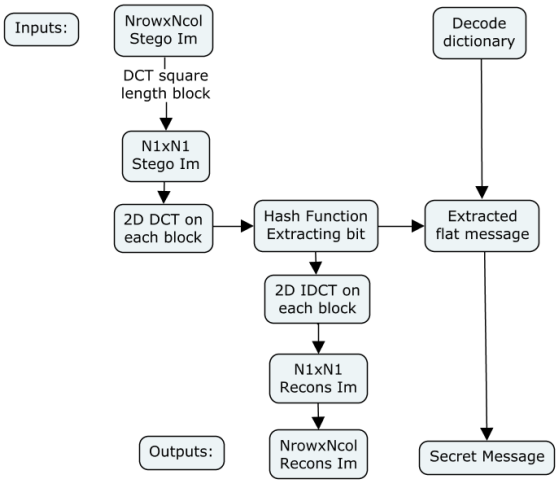

Fig. 3. Diagram flow of how to extract and decode a secret message and reconstruct the image via DCT coefficients.

III. Results

For the method described in the paper is not necessary to use any specific kind of message. Indeed, any format that can be described as a list of bits would be keen to be embedded into a cover image. For simplicity reasons, the images used are extracted from the Python module `skimage.data`.

To prove that the scheme proposed before works properly, 5 different cases have been performed. Apart from that, the effectiveness of the method has been determined by comparing the SSIM of the corresponding images. Before starting the different cases, note that the SSIM between the stego and the recovered image is a sanity check result. Hence, both images should be the same, so SSIM is expected to be 1.

A. String of characters

The first case we will consider is the secret message as a string of characters, specifically the text,

> Steganography among other rare disciplines is honored to be described as both an art and Science field.

which is the one embedded in the Fig. 4.

Secret text without Huffman encoding: The secret text is passed to binary by means of the character's ASCII values. Then the message is embedded into the cover image, thus creating the stego image. Finally, the message is extracted from the stego image and passed from binary to ASCII again, recovering exactly the same text, and reconstructed image as displayed in Fig. 4. The SSIM between the stego image and the cover image is 0.999989, which means that is practically the same image, completely indistinguishable for the human eye.

Secret text with Huffman encoding: The secret text is encoded to binary using the static Huffman encoding algorithm. Then the message is embedded into the cover image, thus creating the stego image. Finally, the message is extracted from the stego image and passed from Huffman code to binary, and then to ASCII again, recovering exactly the same text, and reconstructed image as displayed in Fig. 4, we use the same figure as they cannot be distinguished from the ones in Huffman encoding.

The SSIM between the stego image and the cover image is 0.999994, which is closer to 1 than the one without Huffman encoding. This is because in the past case, it represented embedding 824 bits, and the Huffman code for this message is 417, roughly half of it.

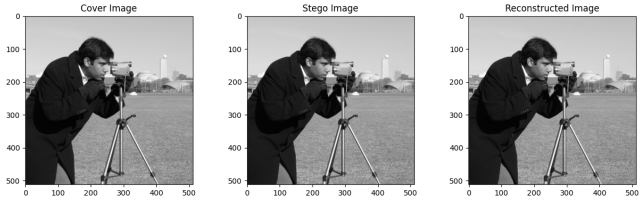

Fig. 4. Results of the cover image, stego image, and reconstructed image for embedding a string message.

B. 1-layer image

Once, we have already hidden a string of characters, the challenge is hiding a 1 layer image, which involves significantly a higher amount of bits. For this purpose, we will use another 1-layer image, which is the Luminescence (Lum) of an RGB image, and this same RGB image.

Secret image without Huffman encoding: The secret image is passed to binary with `numpy` function `unpackbits`. Then the message is embedded into the cover image, thus creating the stego image. Finally, the message is extracted from the stego image and passed from binary values to integer values again, using the `numpy` function `packbits`. The SSIM between the stego and the cover image is 0.996. The SSIM between the secret image and the extracted secret image is 0.9991, practically the same image despite a normalization factor, Fig. 5.

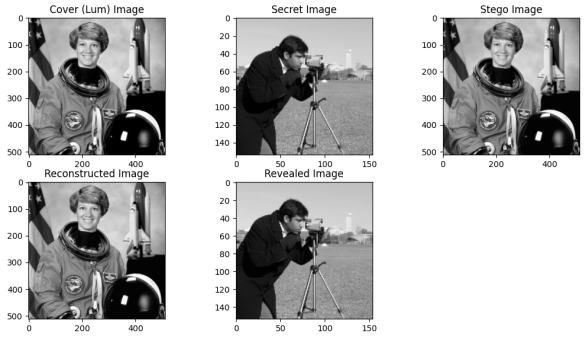

Fig. 5. Results of the cover image, secret image, and stego image revealed a secret image and reconstructed image for embedding a secret image.

Secret image with Huffman encoding: The secret image is passed to binary using the static Huffman encoding algorithm. Then the message is embedded into the cover image, thus creating the stego image. Finally, the message is extracted from the stego image and passed from Huffman code to binary, and then to integer. The 1D array of integers is reshaped to recover the hidden image. The SSIM between the stego and the cover image is 0.997, which is a little bit better than the one without Huffman encoding as instead of hiding 189728 bits, it was 170759 due to the Huffman encoding. The SSIM between the secret image and the extracted secret image is 0.9991, practically the same image despite a normalization factor. As it has been done in the string case the image with and without Huffman encoding is the same because these cannot be distinguished, so refer to Fig. 5 to see the images obtained.

Secret image with Huffman encoding in an RGB image: The secret image is passed to binary using the static Huffman encoding algorithm. Then the message is divided into three parts, each part embedded into each of the three, R, G, and B, layers, thus creating the stego image. Finally, the secret message is extracted from the stego, joined as a whole secret message, and passed from binary to integer using the Huffman decoding algorithm. The 1D array of integers is reshaped to recover the hidden image. The SSIM between the stego

image and the cover image for each layer is 0.997 as before. However, as we have three channels, we have to compute the SSIM between the two RGB images as a whole, the three-layer stego image, and the three-layer cover image, which is 0.999989, which is completely outstanding so they cannot be distinguished as we can see in Fig. 6. The Huffman encoding supposed that, instead of hiding 524288 bits, it was 470009 bits to be hidden. In terms of the SSIM between the secret image and the extracted secret image is 0.9992, practically the same image despite a normalization factor. In this case, we did not perform without Huffman encoding because we wanted to reduce the number of bits to be hidden and make it the most realistic process.

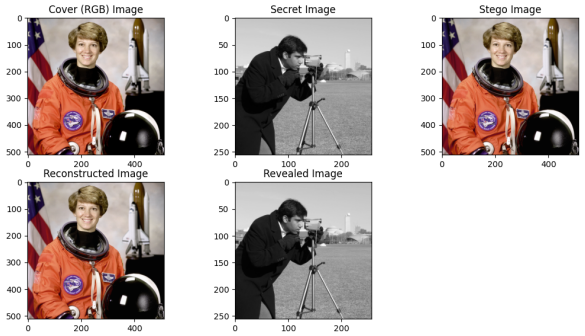

Fig. 6. Results of the cover image which now is RGB, secret image, stego image revealed secret image and reconstructed image for embedding a secret image.

IV. Conclusion

On the one hand, his paper presents a steganography method on the frequency domain based on DCT coefficients. It could be enhanced by introducing the study of global image parameters, for deciding which are the best image locations to embed the bits, as well as improve the hash function. However, this was out of the scope of the project, and it has been presented in a didactically way to understand all the method phases. The code has been organized in the same fancy and it can be found on https://github.com/Eric-Vidal-hub/Steganography.

On another hand, the development of this paper confirms the efficiency of the static Huffman encoding algorithm in reducing the size of the original secret message. However, it is worth noting that it works better for short bitstrings and it can be optimized for larger files or images as much as using the same procedure of `.zip` files construction. Finally, one should highlight the high performance shown by the method with an SSIM near 1 between cover images and stego images for all the cases studied.

References

[1] Xin Liao, Jiaojiao Yin, Sujing Guo, Xiong Li, Arun Kumar Sangaiah,*Medical JPEG image steganography based on preserving inter-block dependencies*. Computers & Electrical Engineering, Volume 67,2018, Pages 320-329. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0045790617302756

[2] Abbas Cheddad, Joan Condell, Kevin Curran, Paul Mc Kevitt,*Digital image steganography: Survey and analysis of current methods*, Signal Processing, Volume 90, Issue 3, 2010, Pages 727-752. https://www.sciencedirect.com/science/article/pii/S0165168409003648

[3] Mandal, J.K. (2020).*Discrete Cosine Transformation-Based Reversible Encoding.* In Reversible Steganography and Authentication via Transform Encoding. Studies in Computational Intelligence, vol 901. Springer, Singapore.[Online]. Available: https://link.springer.com/chapter/10.1007/978-981-15-4397-5_5

[4] Moskowitz, Ira S., *Information Hiding 4th International Workshop, IH 2001,* Pittsburgh, PA, USA, April 25-27, 2001. Proceedings. Ed. Ira S. Moskowitz. 1st ed. 2001. Berlin, Heidelberg: Springer Berlin Heidelberg, 2001. Pages 289-301.

[5] C. Wang and J. Ni, *An efficient JPEG steganographic scheme based on the block entropy of DCT coefficients,* 2012 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP), 2012, pp. 1785-1788, doi: 10.1109/ICASSP.2012.6288246.

[6] Arnab Dey, 2021, *Github Huffman-Coding-Python*, last checked: 06/12/2023, Available: https://github.com/arnab132/Huffman-Coding-Python

[7] *Huffman Coding*, last checked: 06/12/2023. Available: https://www.programiz.com/dsa/huffman-coding

[8] Yağmur Çiğdem Aktaş, 11/08/2021, *Huffman Encoding & Python Implementation*, last checked: 06/12/2023. Available: https://towardsdatascience.com/huffman-encoding-python-implementation-8448c3654328

[9] Yağmur Çiğdem Aktaş, 11/08/2021, *Huffman Encoding & Python Implementation*, last checked: 06/12/2023. Available: https://github.com/YCAyca/Data-Structures-and-Algorithms-with-Python/tree/main/Huffman_Encoding