Assignment E.318-C

Castro, Jonatán .... jonatan.dan@hotmail.com
Chimenti, Tomás .... tach.365@gmail.com
Minsburg, Álex .... alexminsburg@gmail.com
Urquiza, Elías .... eurquiza@fi.uba.ar
Venegas, David .... venegasr.david@gmail.com

2° Semester 2019
10/09/19

1 Summary

A program was made in the C programming language to provide simple staganographic functionality. Using the command line it is able to hide and unhide ciphertext using LSB techniques in a given bitmap image. Also, a bench-marking program is included to compare two same-size bitmap images using MSE, PSNR and SSIM criteria.

2 Introduction

The main objective was to implement a program to perform the following tasks:

- Hide text in a bitmap image based on the least significant bit (LSB).
- Extract hidden text from an steganographic image.
- Given an original BMP and a stego-BMP, output the values of "Mean Square Error" (MSE), "Pick Signal-to-Noise Ratio" (PSNR) and "Structural Similarity" (SSIM).

It was decided to write said algorithms down in the C programming language in order to take advantage of its built-in capacity to perform bitwise operations and make use of an available library, that we've worked with in the past, that allowed us to manage images with a BMP image format.

3 Procedure

In order to implement the required functions, we decided to build the three of them in the same program, allowing the users to pick the functionality they desire at a given time through a single command-line argument. This can be found in the "main.c" source code.

3.1 Hiding Text

To hide text, the program calls the *start_hideInformation*(); function that is written in "hideinformation.c". This function reads the bitmap in which the information will be stored and generates another bitmap of the same size in which the information is copied. Once this is done, the message is hidden in this newly formed bitmap. We should point out that, ideally, the software works with 32bpp bitmap images. However, if an image of lesser quality is given, the program will convert it to 32bpp.

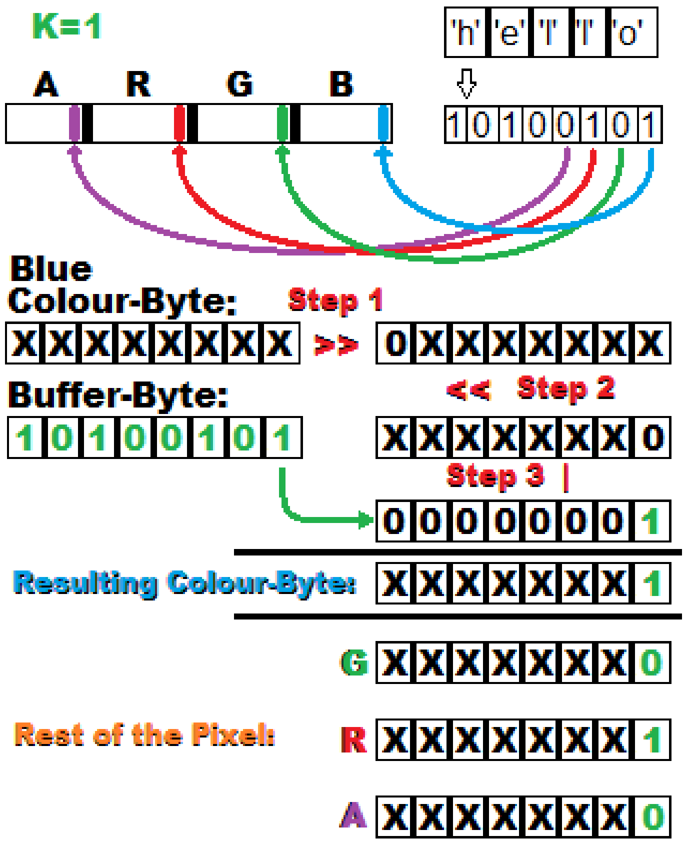

Figure 1: Process for hiding a given character in a pixel for K=1. The bottom half shows the bitwise operations performed on a given colour-byte for storing one bit.

In order to hide the text, we store the bits that make up a single character of the message in a buffer the size of a single byte whose information is later on stored in a bit array as can be seen in Fig. 2 through the use of the "*separate_bits*()" function. By keeping track of which character, and which bit of the char buffer, we're working on, we proceed to place them in 'k' LSBs of the bytes that conform a pixel (Alpha, Red, Green and Blue). To perform the required bit replacements, we first considered using masks. However, due to the fact that we have a 'k' that is determined by the user, we chose to use the "binary left" and "right" *switching* operations. For example, in the byte that corresponds with the "Blue" parameter, we may choose to use $k = 2$ and so we perform a binary right switch for two bits and then a binary left switch for two bits again, essentially removing any information that was on the two least significant bits. This way, we can now store through an *OR* operation a byte that contains only the LSB of the char and another byte that contains only the second to least significant bit (for example: 00000001 and 00000010 from a buffer that stores 11011011). This procedure is repeated on each byte and on each pixel until the entire message is hidden in the new image. An example for k=1 can be seen in Figure 1.

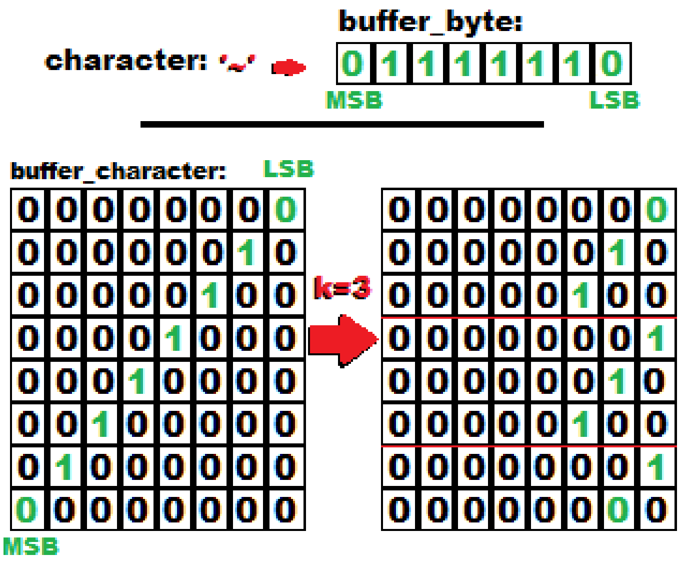

Figure 2: Representation of "*separate_bits*()" for k=3

To be more precise, the "*buffer_character*" array is rearranged to suit the value of 'k' provided. A problem to consider while doing this, however, is that one may run out of bits in the char buffer while storing information in a byte. So we made it so the program also keeps track on where in the pixel it is currently replacing info, so it may jump to the next character and start with a new buffer in the same bit where it left off. An example of said problem can be seen in Figure 3.

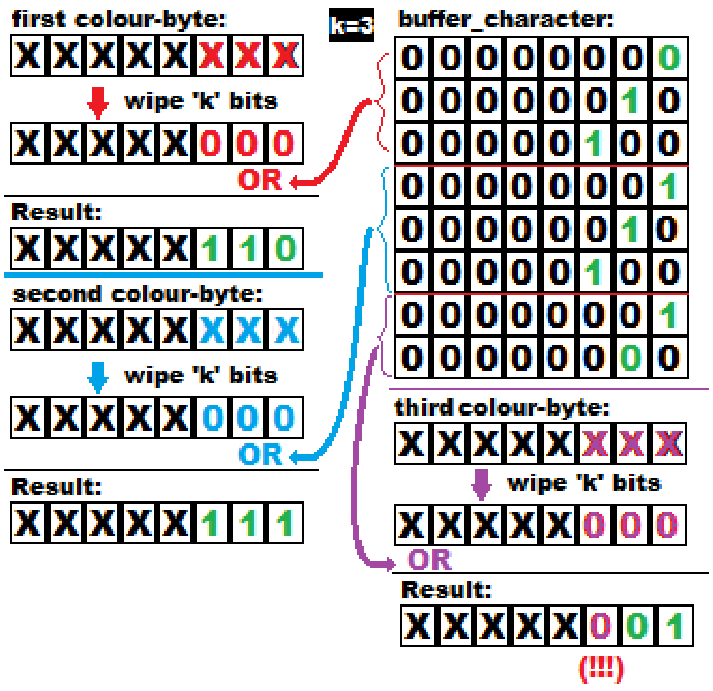

Figure 3: If an odd value is given as key, the char buffer could run out before utilising all the available positions of the last colour used.

For the case in which there's still some bits available to use in the current colour, despite the current character having been fully inscribed, we proceed to switch to the next character and repeat the procedure using the "*separate_bits*()" function, which is designed to rearrange the buffer in such a way that the first bits to insert are correctly positioned. In the example of Fig. 3, the first line of the buffer would have its information bit in the third LSB. This way, we're able to hide any text into a bitmap image given that 'k' is not longer than '8'.

3.2 Extracting Text

To extract a hidden text from an image, the program first asks the user for the value of 'k', this is a design choice since it is expected that the users agree on a "private key" beforehand for sending and receiving information. Afterwards, the program skips the header and reads the first pixel, starting from the lowest colour byte that conforms it.

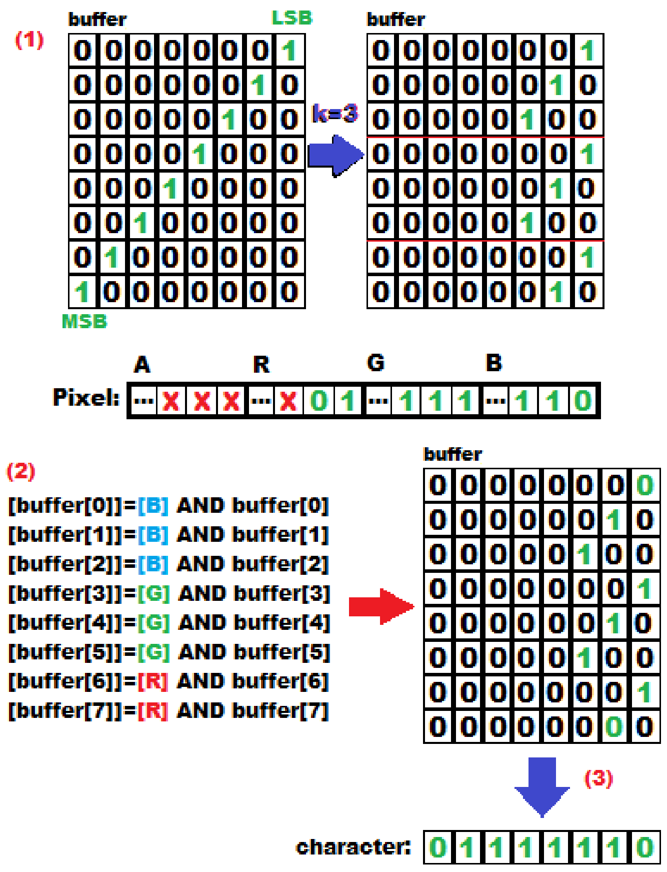
(1)
buffer
LSB
MSB
k=3
buffer
A
R
G
B
Pixel:
(2)
[buffer[0]]=[B] AND buffer[0]
[buffer[1]]=[B] AND buffer[1]
[buffer[2]]=[B] AND buffer[2]
[buffer[3]]=[G] AND buffer[3]
[buffer[4]]=[G] AND buffer[4]
[buffer[5]]=[G] AND buffer[5]
[buffer[6]]=[R] AND buffer[6]
[buffer[7]]=[R] AND buffer[7]
buffer
(3)
character: 0 1 1 1 1 1 1 0

Figure 4: Extracting process. (1): Sets buffer up; (2): Loads data on buffer; (3): Buffer data is stored in a character byte.

Following the steps shown in figure 4 we're able to extract information from a pixel and turn it into a character. This method is based on the type of buffer we utilised for hiding information. It is designed in such a way that the buffer is re-arranged in the correct way to extract information. For example, when the following character - represented with "don't care" crosses - is gonna be loaded, the new buffer's first line will be "0|0|0|0|0|1|0|0" and the following ones will again be arranged as is needed for k=3.

3.3 Comparing Images

For this part of our program, the user can compare two BMPs: one original image and another one being a stego-image of the original. In order to compare both images, the "Mean Squared Error" (MSE)[1] is measured along with the "Peak Signal-to-noise Ratio" (PSNR)[2] and the "Structural Similarity" (SSIM)[3] between them.

3.4 Determining size of text

In order to be able to extract information on how long the hidden text is, we decided to store that information in the first eight bytes during the hiding process. For this information to be encrypted, once the plaintext's length has been measured, we multiply it for the size of the file and divide it by 32. Once that's done, a constant is added to generate noise. Afterwards, the generated number is divided by 27 multiple times until the result is smaller than 27 and we take the remainder of each division. We add 97 to these remainders - in order to give them an ascii value- and these are stored in a char vector, which is then stored in the image. To extract this information, we repeat this process in reverse.

4 Benchmarks and Conclusions

Through the implementation of these algorithms and by bench-marking several images in 30 different configurations we've concluded that the program designed works between acceptable standards for a maximum value of 'K'. Therefore, it is recommended that the users utilise values no bigger than the one mentioned above.

It is shown that the proposed method of hiding and extracting information is a viable mean of private communication, which can be potentially improved by hiding ciphertext instead of plaintext. This is so that, in case of an attacker figuring out the steganographic techniques utilised in the images sent between two or more people, even if the correct key is given, the result will appear to be meaningless. We conclude that our program is able to be used in this way.

References

[1] *Mean Squared Error - Wikipedia*. https://en.wikipedia.org/wiki/Mean_squared_error.

[2] *Peak Signal-to-noise Ratio - Wikipedia*. https://en.wikipedia.org/wiki/Peak_signal-to-noise_ratio.

[3] *Structural Similarity - Wikipedia*. https://en.wikipedia.org/wiki/Structural_similarity.