CSCI 4810 Assignment 3

Encoding and Decoding Secret Messages Using Steganography

Ellemieke Van Kints, ejv88036@uga.edu
Hamid Arabnia, hra@uga.edu

November 18, 2022

1 Introduction

This report begins with an overview of my steganography algorithm, which allows users to encode and decode secret messages in images. The `encode` operation, in particular, works by manipulating the least significant bits of each pixel value in an image. The code to perform least significant bit insertion and extraction was provided in a *Medium* article written by Devang Jain [1]. After the algorithm overview, I perform tests on the `encode` and `decode` operations using various secret messages. The number of modified least significant bits remains constant in these tests for demonstration purposes. Finally, I experiment with how the number of least significant bits modified during the `encode` operation impacts the resulting image. The `cat.png` image is used by default for all tests throughout this report, seen in *Figure* 1 below. This image has a resolution of 3024 x 4032 pixels.

Figure 1: The `cat.png` image, which is used by default for all tests throughout this report.

2 Program Overview

My algorithm begins by asking the user whether they wish to `encode` or `decode` an image. If `encode` is chosen, the user is asked to provide the image path, message to encode, and the number of least significant bits to modify, indicated by the variable n. First, the `encode` function maps the source image to a 2D array containing the pixel values for each band at each location. Then, a delimiter is added to the end of the secret message, so that when the image is decoded, the program knows when to stop. The updated message is then converted into its binary form. Lastly, steganography is performed by replacing

the n least significant bits in each pixel with the bits of the secret message, until the complete message is hidden.

If `decode` is chosen, the user is only asked to provide the image path. The same procedure as before is repeated by mapping the source image to a 2D array of pixel values. Then, the n least significant bits are extracted from each of the pixels in the image array and stored in group of 8 bits. This iteratively constructs the binary representation of each character in the secret message. Once the end of the image array is reached, the 8-bit binary values are converted into their corresponding ASCII characters. Finally, the program checks if the delimiter was found or not. If no delimiter is found, that means there was no hidden message in the image.

It should be noted that the program sets n to 3 by default at the beginning of the program, and n is updated according to user input during the `encode` operation. This is to ensure n is set if the `decode` operation is called before the `encode` operation.

3 Encoding Images

To demonstrate that my `encode` operation works correctly, I will first encode the short message "This is a picture of my cat named Margo." into the `cat.png` image. The number of least significant bits modified per pixel will be set to 3. The output of this test is shown in *Figure* 2 below.

Figure 2: The resulting image, encoded with the message "This is a picture of my cat named Margo."

I will now encode the first 1,002 words from the *Cat* Wikipedia entry [2] into

the image. The same number of least significant bits modified per pixel will be used. The output of this test is shown in *Figure* 3 below.

Figure 3: The resulting image, encoded with the first 1,002 words from the *Cat* Wikipedia entry.

I will now encode the entire *Bee Movie* script into the image. The same number of least significant bits modified per pixel will be used. The output of this test is shown in *Figure* 4 below. If we look closely towards the top of the image, the encoded parts become visible to the human eye. This is due to the length of the secret message. The longer the message, the more pixels must be modified.

4 Decoding Images

To demonstrate that my `decode` operation works correctly, I will first decode the image containing the message "This is my cat named Margo." This image is named `cat-enc.png`. The terminal output from this test is provided in *Figure* 5 below.

I will now decode the image containing the first 1,002 words from the *Cat* Wikipedia entry. This image is named `cat-enc-2.png`. The terminal output from this test is provided in *Figure* 6 below. It should be noted that the whole decoded text is not included in *Figure* 6, as it was too much to fit on one screen. Only a snippet is provided.

I will now decode the original `cat.png` image, which does not contain an encoded message. This test is done to demonstrate that my program is able to

Figure 4: The resulting image, encoded with the entire *Bee Movie* script.

```
Command Options
 - Encode: e
 - Decode: d
 - Quit: q

Choose a command: d

Enter the source image path: cat-enc.png

Image opened successfully!

Decoding...

Hidden Message: This is a picture of my cat named Margo.
```

Figure 5: The terminal output after decoding the `cat-enc.png` image containing the message "This is a picture of my cat named Margo."

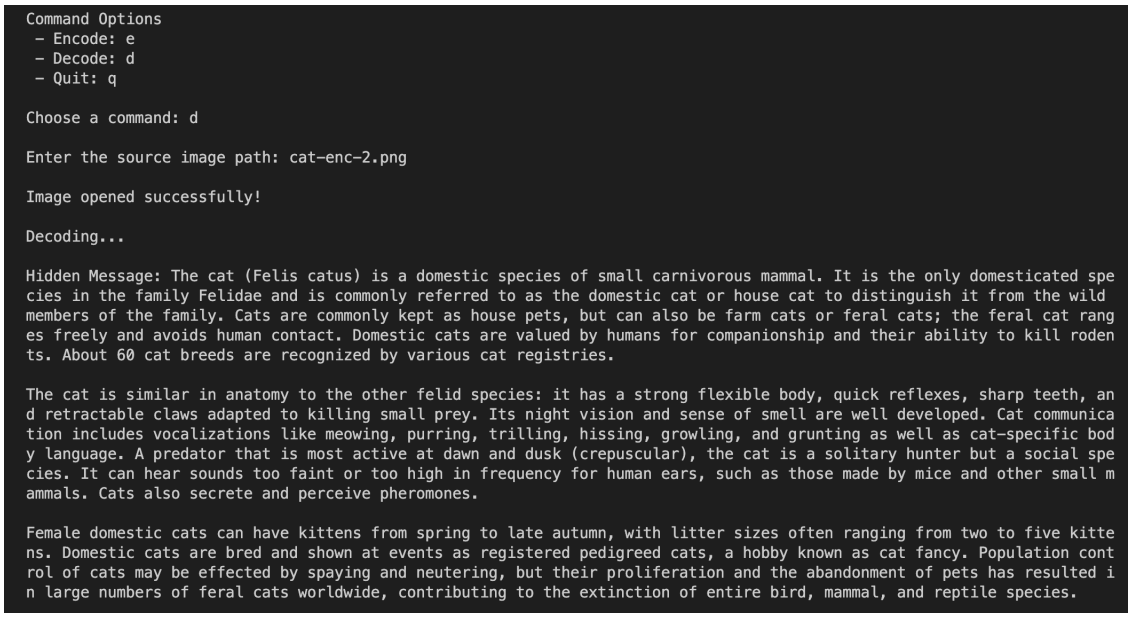

Figure 6: The terminal output after decoding the `cat-enc-2.png` image containing the first 1,002 words from the *Cat* Wikipedia entry.

detect when there is *not* an encoded message in an image. The terminal output from this test is provided in *Figure* 7 below.

5 Experiments with Least Significant Bits

I will now experiment with the number of least significant bits modified during the `encode` operation, referred to by the variable n. This value was set to 3, by default, in previous tests. However, in this section, I will run tests with various n values and discuss how it impacts the resulting image. For consistency, the entire *Bee Movie* will be the encoded message for all tests in this section.

For the first test, n will be set to 1. The output can be seen in *Figure* 8 below. Here, the encoded parts of the image become much more visible. This is because we are only modifying 1 bit per pixel, so more pixels must be modified in order to hide our entire message.

Next, n will be set to 6. The output can be seen in *Figure* 9 below. Here, the encoded parts of the image become less visible. This is because we are modifying 5 bits per pixel, so less pixels must be modified overall in order to hide our message. However, each modified pixel has 6 out of 8 bits changed. So, if we look closely, we will notice that the encoded parts of the image look drastically different than before. A close-up of the resulting image is provided in *Figure* 10.

```
Command Options
 - Encode: e
 - Decode: d
 - Quit: q

Choose a command: d

Enter the source image path: cat.png

Image opened successfully!

Decoding...

No Hidden Message Found
```

Figure 7: The terminal output after decoding the original `cat.png` image, which does not contain an encoded message.

Figure 8: The resulting image, encoded with the entire *Bee Movie* script, with $n = 1$.

Figure 9: The resulting image, encoded with the entire *Bee Movie* script, with $n = 6$.

Figure 10: A close-up of the resulting image, encoded with the entire *Bee Movie* script, with $n = 6$.

References

[1] D. Jain, "LSB image steganography using Python," Medium, 08-Apr-2021. [Online]. Available: https://medium.com/swlh/lsb-image-steganography-using-python-2bbbee2c69a2. [Accessed: 16-Nov-2022].

[2] "Cat," Wikipedia, 16-Nov-2022. [Online]. Available: https://en.wikipedia.org/wiki/Cat. [Accessed: 16-Nov-2022].