F28HS Coursework 1

steg.c for encoding or decoding a message into a .ppm file

Andrea Pavan
username: ap103
Reg. Number: H00256513

Steg.c: Introduction & How to use

steg.c is a program used for encoding a message into an image or decoding a message from an image. When calling the program a char is needed (‘e’ for encoding / ‘d’ for decoding) and the name of a .ppm file.
e.g.

./steg e in.ppm > out.ppm

This is an example of a steg call for encoding, the output gets redirected to ‘out.ppm’, which is the image with the message encoded in it. The message to encode and the key value used to encrypt to message are requested when running the program.

If the call if for decoding, the program only asks for a key value, used to decode the message from the image given.
e.g.

./steg d image.ppm

This is an example of steg call for decoding, the file ‘in.ppm’ has a message encoded in it.

steg.c was created only for .ppm images, that have the following format:
e.g.

P3
#comment 1
. . .
#comment n
width height
max
r1 g1 b1
r2 g2 b2
r3 g3 b3
. . .

Steg.c: Data structures used

Two data structures are used:

struct PPMPixel, consisting of:

i. unsigned int red: positive integer holding the RGB value for red.
ii. unsigned int green: positive integer holding the RGB value for green.
iii. unsigned int blue: positive integer holding the RGB value for blue.
iv. int isVisited: this int is used as a boolean, its function is to tell the encoding and decoding algorithm that this pixel has already been used to write or read the bit(s) of a message. This parameter is fundamental, as it prevents the program from writing on the same pixel more than once or reading the same bit more than once.

struct PPM, consisting of:

i. char * PPMFormat : char pointer that references the format of the image (P3, P6, …).
ii. char * PPMComment : char pointer that references the comment(s) at the start of the image.
iii. int width: holding the width of the file.
iv. int height: holding the height of the file.
v. int maxValue: holds the maximum value of a RGB value
vi. struct PPMPixel * pixelArray: this is a pointer that references the array of PPMPixel forming the .ppm file.

Steg.c: Algortithms used

```
/* this is the function that converts a positive integer value to a binary number
 * e.g. toBinary(157) = 10011101
 */
int toBinary (int n);
```

To find a binary value from an integer number n, I use a char array and MAX_RGB_VALUE which is equal to 256.

Consider i = MAX_RGB_VALUE / 2 (= 128).

Check if (n-i) is bigger or equal than 0:
if True: bit in the 8th position of he binary value is set to 1, and remove i from n.
if False: bit in the 8th position of he binary value is set to 0.

get i and divide it by 2.

Loop until (i == 0).

e. g.
157 - 128 is bigger or equal than 0, thus bit in the 8th position of binary number will be 1, then remove 128 from 157 (= 29) and divide 128 by 2 (= 64).
If we keep repeting this process until we get to i = 0, we end up with 10011101.

/* this is the function that converts a binary number to a positive integer number
 * e.g. toBinary(10011101) = 157
 */
int toDecimal (int n);

To find the decimal number from a binary value n, I use powTwo which holds 2 to the power of i (which is the loop number) and a variable decimal that will contain the decimal value of the binary number.

(n / 10 * 10 == n) this line checks if bit in the 1st position on the binary is either a 1 or a 0.
e.g.

Consider n = 1011 0101.
We do n/10 (=1011 010) and than we multiply the result by 10 (=1011 0100), in this way, the digit in the least significant position will always be equal to 0. Comparing (n / 10 * 10 == n), we can check if this bit is either 1 or 0.
If this bit is equal to 1 than we get decimal and we add powTwo.
If instead, it's equal to 0, we keep decimal as it is.
We then, multiply powTwo by 2 and we divide n by 10.

If we keep repeting this loop until i is equal to 8 (=8 bit =1 byte), we will eventually have the decimal number from the given binary value n.

struct PPM * encode(struct PPM * im, char * message, unsigned int mSize, unsigned int secret)

This is the function that encodes the message in the .ppm image. First, it gets the string message and converts every char into binary, then it puts them into binMessage.

The second part of the function, gets the full binary numbers in binMessage and puts them bit by bit into binMessageArray. To do so, it uses a mask, which is equal to 10000000 at the start.
For every loop, the mask is compared with the binary number, if the number is bigger or equal than the mask, then 1 is stored in the array and the mask is removed from the binary number, soon after, the mask is divided by 10. This loop is repeated 8 times (8 bit for each binary number).
e.g.

Let n = 1001 1101 and mask = 1000 0000
(n > mask), so first binary digit is 1.
then, we do n - mask = 1 1101.
(n < mask), so second binary digit is 0.
n remains 1 1101.

We keep repeating this and we get an array with values: 1 0 0 1 1 1 0 1.

To store each bit in the image, I first use srand() to set the seed. Then, I use rand() sequentially to get all the pseudo-random numbers, to make sure that the random value points to an existent pixel, we module it with the size of the image. Then, I use the same principle as in Hash Tables. Every time I get a new index, I check that the corresponding pixel has not already been written. I check this using the isVisited variable in struct PPMPixel. If it's not been written yet, I store 3 bits, in the red, green and blue values as the least significant bit. Then, I set the pixel to visited. I keep repeating this until all bit from the message are stored.

char * decode(struct PPM * im,unsigned int secret)

To decode the message from an image, I use a very similar principle to the one used to encode. First, using the same srand() and rand() functions, I get the pixel and the 3 bits which are stored in the corresponding RGB values. If the seed used to encode is the same used to decode, we will retrieve all the same bit(s).

To get the least significant bit from an RGB value I simply do a pointwise addition with 1. This is always going to be equal to the 1st bit of the RGB value.

After that, I reverse engineer the array of 1s and 0s to get a 1 byte binary number. I use an algorithm very similar to the one explained in the encode section. At the end, I use the function toDecimal() to get the ASCII value of the char that was encoded and I return the array of char forming the message.

Steg.c: Design Choices and Other Comments

When creating the decode algorithm, I had to face the problem of knowing how long the encoded message would be. To solve this I have decided to encode a special character '*' at the end of the message to encode. In the same way, the decode algorithm, stops decoding when the character '*' is found.

Also, to read a string, I had to read each char separately, as the function scanf() stops reading the line that is entered when finding a whitespace.

I'm not taking into account .ppm files that do not have any comments, if this happens, I exit the program and I output an error for a bad file format, explaining that the '#' token is missing.

If a .ppm file has multiple random whitespaces or random new lines, my program will detect them and ignore them, when redirecting the output, it will give a better format to the new .ppm file.

To print some text in the command line I had to use fprintf with stderr, as otherwise the output would be redirected to out.ppm.