Relatório Trabalho 2

MC920 – Introdução ao Processamento de Imagem Digital

Julianny Favinha Donda RA 156059

1. Solução

Para os testes, foram usadas imagens no formato PNG no modelo de cores RGB, onde cada canal de cor possui valor em $\{0, ..., 255\}$. Dessa forma, cada decimal ocupa 8 bits.

1.1. Codificação

O programa `code.py` codifica uma mensagem no plano de bits informado.

```
/* Entrada: imagem f, inteiro plane (0, 1 ou 2) e mensagem a ser escondida. Saída: imagem g
com a mensagem codificada, imagem p do plano de bits plane, imagem p7 do plano 7. */

CODE(f, plane, message):
	B ← Concatene os valores binários de cada caractere de message e termine com \0.
	linha, coluna ←  0
	cor ← R

	Para cada bit b em B faça
		Se chegou ao final de uma linha então
			linha ← linha + 1 e coluna ← 0

		Transforme o valor f[linha][coluna][cor] em binário.
		R ← Substitua plane-ésimo bit menos significativo pelo bit b.
		G[linha][coluna][cor] ← Transforme R em decimal.

		Se passou por todas as cores de uma coluna então
			coluna = coluna + 1

		cor ← próximo valor de {R, G, B}

	Para cada pixel g[linha][coluna] faça
		Para cada cor c em {RED, GREEN, BLUE} faça
			plane[linha][coluna][cor] ← plane-ésimo bit menos signific * 255.
			plane7[linha][coluna][cor] ← bit mais significativo * 255.

	Devolva g, plane, plane7
```

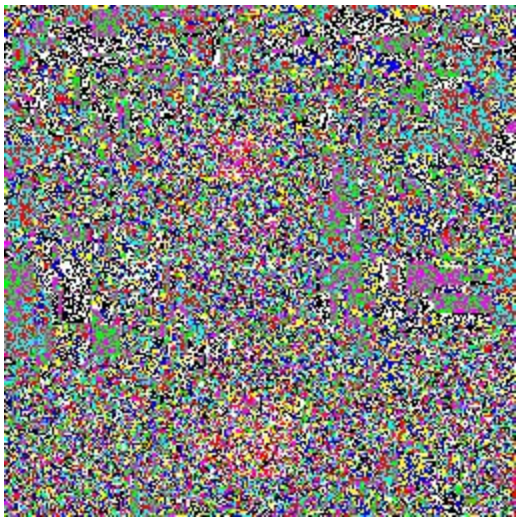
Os caracteres da mensagem estão codificados em ASCII, tendo seu valor decimal associado.

O algoritmo também verifica se a mensagem cabe na imagem, ou seja, o processo de esteganografia acontece somente se a mensagem puder ser colocada integralmente na imagem.

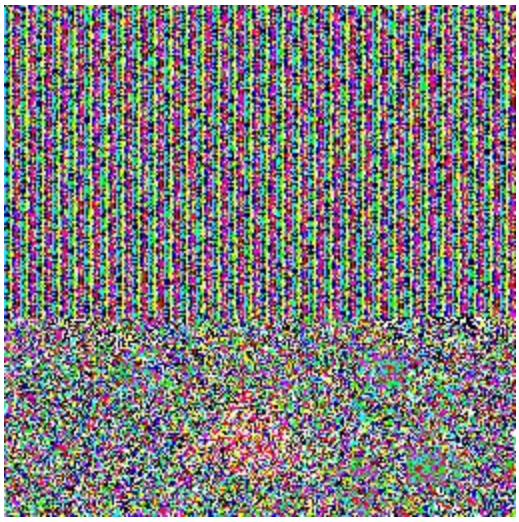
O algoritmo é linear em relação ao tamanho do texto a ser codificado na imagem mas quadrático em relação às dimensões da imagem, portanto executa em tempo quadrático.

Os valores binários que o algoritmo determina para cada caractere possuem sempre 8 bits. Além disso, para fins de decodificação, foi colocado o caractere de parada `\0 = 00000000` logo após o final da mensagem. Dessa forma, a decodificação é mais eficiente e o arquivo texto onde estará a mensagem decodificada ficará sem caracteres desnecessários, ou seja, ficará apenas com caracteres que pertencem à mensagem.

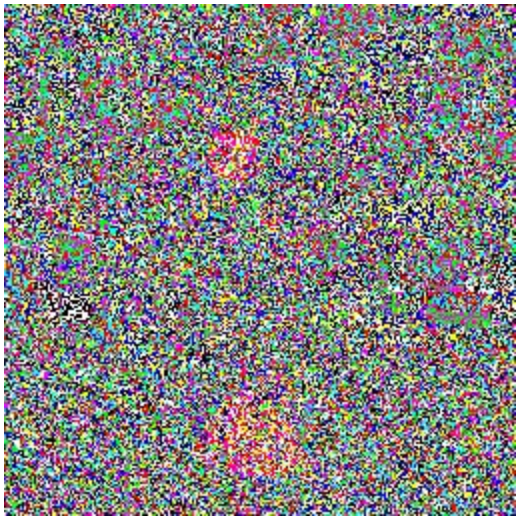
Figura 1 - Imagem original (`monalisa.png`) à esquerda e imagem à direita (`monalisa_output.png`) gerada após a execução de `code.py`, com a mensagem codificada no plano de bits 1. A mudança na imagem resultante é imperceptível a olho nu.



(a) Imagem do plano de bits 0.



(b) Imagem do plano de bits 1.



(c) Imagem do plano de bits 2.

(d) Imagem do plano de bits 7.

Figura 2 - Imagens dos planos de bits indicados. Nota-se que a imagem do plano de bits 1 possui uma fatia com um padrão que difere do restante dos planos vizinhos. Isso indica que é possível que esse plano tenha sido alterado.

Pela imagem do plano de bits 7 podemos perceber que quanto mais significativo o plano de bit, mais ele se parecerá com a imagem original. É por esse fato que o processo de esteganografia é feito nos bits menos significativos; dessa forma, a alteração feita é transparente para as pessoas que não sabem dessa informação.

Para fins de visualização, os bits de cada plano de cor das imagens anteriores foram multiplicados por 255. Se estivessem com seus valores normais, as imagens de saída seriam muito escuras, o que comprometeria a visualização.

1.2. Decodificação

O programa `decode.py` decodifica a mensagem no plano de bits informado pelo usuário.

```
/* Entrada: imagem g, inteiro plane indicando o plano de bits que será modificado, sendo 0, 1 ou 2. Saída: arquivo texto contendo a mensagem que foi decodificada. */

DECODE(g, plane):
	Para cada pixel g[linha][coluna] faça
		Para cada cor c em {RED, GREEN, BLUE} faça
			Transforma o valor g[linha][coluna][c] em binário.
			Concatena o plane-ésimo bit menos significativo em B.

			Se tamanho de B é igual a 8 então
				file ← Transforme o binário B no seu decimal.
				Limpe B.

			Se B é \0 então
				Break.

	Devolva file.
```

Os bits são sempre lidos de 8 em 8, pois, como feito na codificação, a cada 8 bits temos um caractere a ser lido.

Note que o algoritmo é quadrático em relação às dimensões da imagem de entrada e finaliza assim que encontrar o caractere de parada na mensagem.

2. Execução dos códigos

É necessário que as bibliotecas **numpy** e **scikit-image** estejam instaladas. No terminal, execute

```
$ python3 code.py [nome da imagem].png plano_bits [nome do arquivo da mensagem].txt
```

Onde `plano_bits` é um inteiro 0, 1 ou 2 que indica em qual plano de bit menos significativo a mensagem será escrita.

O programa `code.py` gera os seguintes arquivos:

- Imagem `[nome da imagem]_output.png`, que é a imagem onde ocorreu o processo de esteganografia;
- Imagem `[nome da imagem]_output_bit_plane[plano_bits].png`, que é a imagem do plano de bits de valor `plano_bits`;

- Imagem `[nome da imagem]_output_bit_plane7.png`, que é a imagem do plano de bits de valor 7, ou seja, o plano de bits mais significativo.

```
$ python3 decode.py [nome da imagem]_output.png plano_bits
```

Onde `plano_bits` é um inteiro 0, 1 ou 2 que indica em qual plano de bit menos significativo a mensagem foi escrita.

O programa `decode.py` gera os seguintes arquivos:

- Arquivo texto `[nome da imagem]_output_text.txt`, que contém a mensagem decodificada.

Além disso, ambos os programas exibem na saída padrão o tempo decorrido de execução, em segundos.

3. Limitações

3.1. Tempo de execução dos códigos

Como explicitado anteriormente, foi usado uma quantidade significativa de comandos de repetição para percorrer cada pixel da imagem, tanto no caso do programa `code.py` como no programa `decode.py`.

Para traçar um comparativo, foram feitos os testes a seguir. Os arquivos de entrada podem ser encontrados junto aos programas.

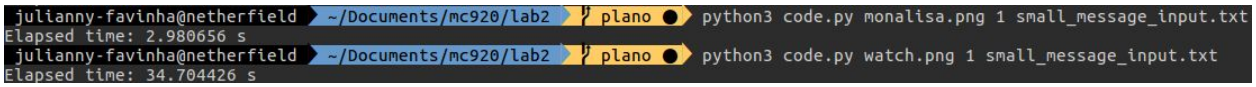


Figura 3 - Captura de tela da execução do programa `code.py`, usando imagens de tamanhos diferentes.

Nota-se que houve um aumento significativo no tempo de execução para imagens com dimensões maiores. Isso se deve ao fato do algoritmo implementado em `code.py` ser quadrático em relação às dimensões da imagem.

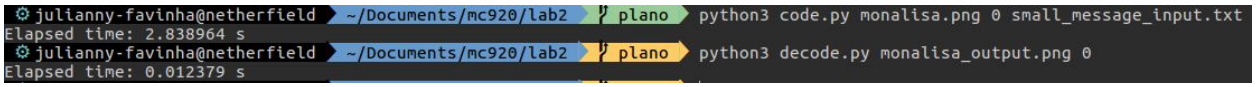


Figura 4 - Captura de tela da execução do programas `code.py` e `decode.py`, para imagem pequena e arquivo de texto pequeno.

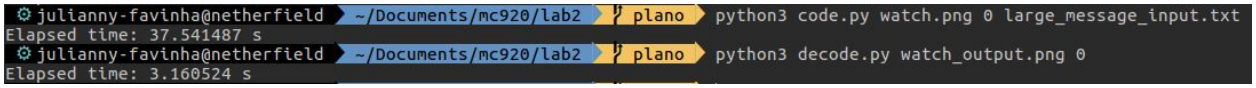


Figura 5 - Captura de tela da execução do programas `code.py` e `decode.py`, para imagem grande e arquivo de texto grande.

Mais especificamente, a imagem `monalisa.png` possui dimensões 256 x 256, e a imagem `watch.png` possui dimensões 1024 x 768.

Nota-se que, ao executarmos o programa `decode.py` para as duas imagens, o tempo de execução aumenta significativamente, já que o algoritmo elaborado é quadrático em relação às dimensões da imagem de entrada.

Portanto, pensando em imagens que sejam muito grandes, o algoritmo pode demorar um tempo considerável para gerar uma resposta. Uma saída seria transformar os comandos de repetição em operações vetoriais, ou utilizar bibliotecas que fornecem funções que manipulam bits em imagens, dessa forma pode-se ganhar em tempo de performance.

Figura 6 - Captura de tela da execução do programas `code.py` e `decode.py`, para a imagem `watch.png` e arquivos de texto pequeno e grande.

Ao testar os programas para uma mesma imagem, porém com arquivos texto de tamanhos diferentes, também vemos um aumento significativo no tempo no momento da decodificação. Apesar de ser quadrático, o algoritmo finaliza assim que encontra o caractere de parada `\0`, dessa forma há uma certa eficiência, sendo o pior caso quando o tamanho da mensagem é igual ao tamanho da imagem.

3.2. Execução para outros formatos de imagem

Ao executar a codificação para uma imagem JPG, como por exemplo a imagem `starrynight.jpg`, foi possível constatar que os algoritmos não funcionam para esse tipo de formato, apesar do valor do pixel para cada banda ser um `uint8`, assim como em imagens de formato PNG.