

Universidade Estadual de Campinas

Instituto de Computação

Introdução ao Processamento Digital de Imagem (MC920A)

Professor: Hélio Pedrini

Relatório – Trabalho 1

Gabriel Augusto Bertolino Gomes

RA: 248287

03/04/2024

1. Introdução

Esteganografia é a arte de esconder informações, tornando-as ocultas. É uma prática antiga que permitia transmissão de dados de forma seguras em período de guerras e conflitos. Para o contexto de processamento de imagens, a mensagem é escondida dentro da imagem, de modo que passe despercebido a existência de alguma modificação em seu conteúdo.

Com isso em mente, este relatório tem por objetivo discutir e apresentar resultados da implementação da técnica de esteganografia em imagens.

2. Materiais e Métodos

Os materiais usados para processar e esconder tais mensagens foram seguem a seguir:

- Python 3.11.6
 - Bibliotecas
 - opencv-python (cv2) 4.9.0.80
 - numpy 1.26.1
- Visual Studio Code
- Imagens usadas: imagem_entrada.png (baboon.png) e imagem_entrada_grande.png (Link: https://ar.pinterest.com/pin/232639136986687586/?amp_client_id=CLIENT_ID%28%29&mweb_unauth_id=)

Para além desses itens, abaixo encontra-se a estrutura de pasta como estão organizados o código e os arquivos anexos que são itens primordiais para o funcionamento do programa.

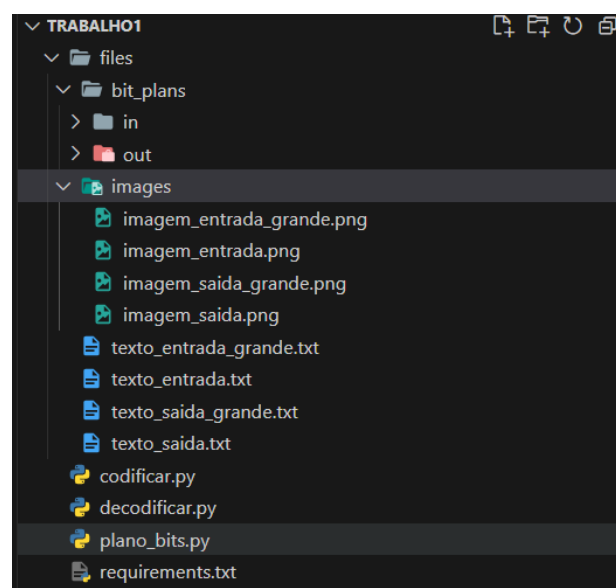


Figura 1 – Estrutura de pasta com os códigos fontes e anexos

Caso seja necessário a instalação de todas as dependências do programa, basta executar no terminal o seguinte comando:

- `pip install -r requirements.txt`

Isso será responsável por instalar todas as bibliotecas que são dependências do programa que realiza a esteganografia.

3. Modo de Uso

Após a instalação de todas as dependências deve-se executar os códigos: `codificar.py`; e `decodificar.py`; nesta ordem.

Primeiramente, para execução do `codificar.py`, é preciso acessar o terminal e digitar o comando:

- `python codificar.py <path_imagem_entrada.png> <path_texto_entrada.txt> <plano_bits> <path_imagem_saida.png>`
 - **<path_imagem_entrada.png>**: caminho para a imagem de entrada no formato png;
 - **<path_texto_entrada.txt>**: caminho para o arquivo .txt com a mensagem a ser escondido;
 - **<plano_bits>**: plano de bits em que o texto será escondido. Os valores válidos aqui vão de 0 a 7, haja visto que essas são as faixas de bits utilizadas na imagem.
 - **<path_imagem_saida.png>**: caminho onde a imagem de saída será salva no formato png;

Por fim, para execução do `decodificar.py`, novamente acessa-se o terminal e digita-se o seguinte comando:

- `python decodificar.py <path_imagem_saida.png> <plano_bits> <path_texto_saida.txt>`
 - **<path_imagem_saida.png>**: caminho para a imagem de saída no formato png;
 - **<plano_bits>**: plano de bits em que o texto será escondido. Os valores válidos aqui vão de 0 a 7, haja visto que essas são as faixas de bits utilizadas na imagem. Aqui para que a decodificação funcione, é preciso passar o mesmo plano de bits que foi passado ao codificar.
 - **<path_texto_saida.txt>**: caminho para o arquivo .txt com a mensagem a ser decodificada a partir da imagem;

4. Resultados e Discussões

Primeiramente, deve-se escolher os parâmetros iniciais de entrada, para então conseguir discutir os resultados obtidos com a execução dos programas de codificação e decodificação. Com isso, tem-se na figura 2, a imagem de entrada escolhida para realizar o processo de esteganografia, juntamente com o plano de bits 012, nesta ordem, e uma mensagem que está no arquivo texto_entrada.txt cujo tamanho é de 281 KB.

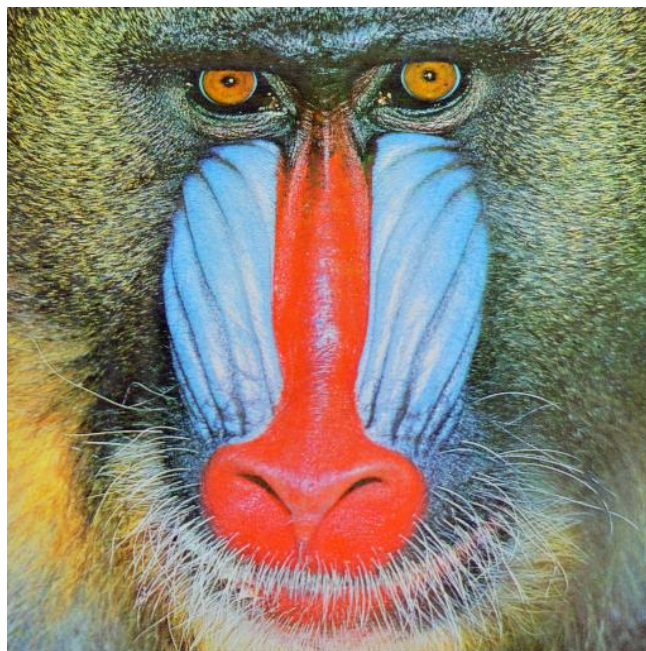


Figura 2 – Imagem de entrada: baboon.png

A imagem de entrada, trata-se de um RGB com 3 canais de cores, e possui as seguintes dimensões, 512 de altura por 512 de comprimento e tamanho de 623 KB.

4.1. Esteganografia da Imagem

Para realizar tal tarefa, é necessário inicialmente transformar o arquivo de texto de entrada em um array numpy. Isso é feito de modo que, todos os bits de todos os caracteres são mapeados e adicionados em único objeto. Tudo isso de forma vetorizada, visando obter desempenho. Como forma de marcação, para auxiliar na posterior decodificação, ao final do texto, é adicionada três @, que irão servir como um endpoint, indicando que todo o texto já foi todo lido.

Feito a transformação necessária, agora, basta olhar qual o plano de bits passado como entrada, para então gerar a imagem modificada, que irá se tornar a imagem de saída, construída de forma iterativa. Para que isso ocorra, faz-se necessário o cálculo

de qual é o tamanho de uma faixa de bit exclusivamente. Para isso, utilizaremos da seguinte fórmula:

$$numero_{bits} = h . w . n_c$$

Em que h é a altura, w é o comprimento e n_c é o número de canais de cores.

Dito isso, cada plano de bits possuirá um array a ser escondido de tamanho máximo $numero_{bits}$. E como a entrada permite passar mais de um plano de bits, é possível deslocar o array de $numero_{bits}$ em $numero_{bits}$, para assim, cada plano ser responsável de salvar uma parcela de bits do texto. Esta lógica pode ser vista no trecho de código a seguir:

```
82 # Acomoda os bits decodificados do texto de acordo com o plano de bits passado
83 for i in range(len(bit_plans)):
84     bit_plan = int(bit_plans[i])
85
86     image = encode_image(height, width, image, bin_to_hide[i * max_bits:(i + 1) * max_bits], bit_plan)
```

Figura 3 – Código com a lógica para percorrer o array de bits e escondê-los

Aqui o plano de bits é passado como parâmetro, logo, é arbitrário em ordem e tamanho. Desde que os valores estejam entre os valores válidos (isto é, entre 0 e 7). O tamanho é variável, é permitido construir um plano de bits de tamanho 2, como por exemplo, “02”, e um de tamanho 5, por exemplo, “71236”. Vale ressaltar, ainda, que a ordem de escrita do plano importa. Uma vez que, um plano de bits “012” é diferente de “210”, mesmo que as bandas de bits sejam as mesmas. Isso acontece, pois, o algoritmo acomoda os bits na ordem em que chegam conforme a entrada. Dessa forma, é possível gerar um padrão de esteganografia diferente e dinâmico, sem a necessidade de alterar o código fonte para possíveis testes e necessidades, como por exemplo, aumentar o número de bandas de bits devido ao tamanho da mensagem.

Por fim, como visto na figura 3, é capaz de observar a função **encode_image** que é responsável por definitivamente esconder os bits dentro da imagem. Ela abstrai o seguinte racional. Inicialmente, é feito o reshape da imagem, para deixá-la flat, ou seja, passa de matriz para um vetor linear. Posteriormente a isso, realiza-se a limpeza dos bits que serão utilizados para esteganografia usando do operador lógico **and**, o que tem por finalidade zerar todos os bits naquela faixa. Com a chegada dos novos bits que trarão o conteúdo da mensagem, tal ação faz-se necessária pois não é viável que essas informações contenham ruídos, ou valores espúrios, haja visto, que isso faria com que o conteúdo fosse corrompido e não pudesse ser descriptografado, no processo de decodificação. Finalizada a limpeza, os bits correspondentes da mensagem são alinhados com os bits da imagem na faixa de bits escolhida, passado como parâmetro no plano de bits. Para realizar a escrita de tais dados é utilizado o operador lógico **or**, que, por construção, será feito um **or** entre zero e o bit correspondente a ser escondido. Logo, fica claro que, a partir deste momento, o bit que está dentro da imagem passa a ser o bit da mensagem. Finalmente, é feito o reshape da imagem para devolvê-la da forma que ela chegou.

Isso pode ser visto na figura 4.

```

42 def encode_image(height, width, image, bits_to_hide, bit_plan):
43     try:
44         mod_image_flat = image.reshape(-1)
45
46         max_bits_to_hide = len(bits_to_hide)
47
48         mod_image_flat[:max_bits_to_hide] = np.bitwise_and(mod_image_flat[:max_bits_to_hide], select_bit_to_clean(bit_plan))
49
50         mod_image_flat[:max_bits_to_hide] = np.bitwise_or(mod_image_flat[:max_bits_to_hide], bits_to_hide << bit_plan)
51
52         return mod_image_flat.reshape(height, width, 3)
53     except Exception as error:
54         print(f'Ocorreu um erro ao encriptar a mensagem no plano de bit: {bit_plan}')
55         return image

```

Figura 4 – Implementação da função encode_image

Ao final, basta salvar a nova imagem processada e com a mensagem escondida. Com todos esses passos, o algoritmo devolve a seguinte imagem de saída, que, como pode-se ver na figura 5, a mudança é quase que imperceptível mesmo com um texto que possui um terço de seu tamanho.

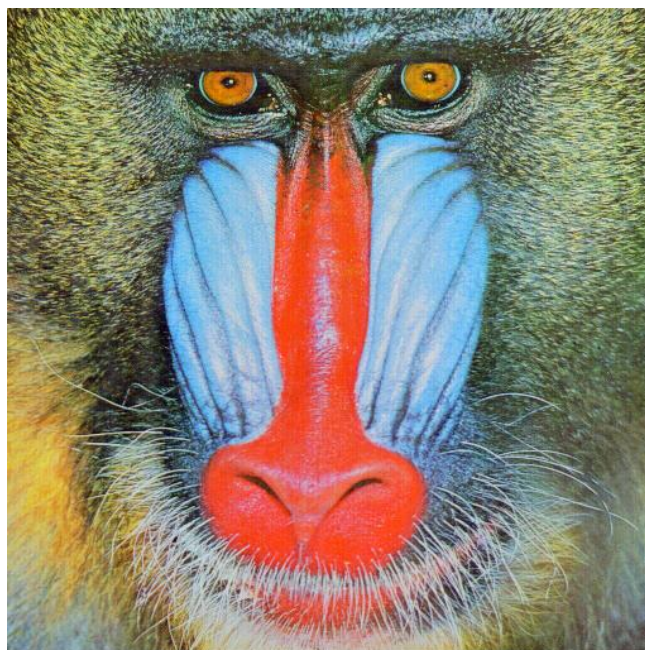


Figura 5 – Imagem de saída do algoritmo

4.2. Decodificação da Mensagem na Imagem

Inicialmente, fica claro que a ideia do algoritmo é, por meio de uma imagem, decodificar a mensagem que está presente dentro dela. Sabe-se que tal texto tem seus bits “escondidos” no meio dos canais de bits da figura.

Com isso em mente, é preciso elencar as seguintes entradas, são elas: o endereço da imagem com o texto escondido, o plano de bits em que ele foi escondido e o local onde será salvo o arquivo de saída. É importante salientar que, o mesmo plano de bits passado para a codificação, deve ser o mesmo plano de bits passado para decodificação. Aqui o plano de bits funciona como uma chave, que permite dar sentido aos dados de saída, já que, somente a partir dele, o algoritmo é capaz de devolver a mensagem corretamente. Sem passá-lo de forma idêntica, não há garantia disso.

Ao obter as entradas, consegue-se assim iniciar toda a lógica de decodificação.

Primeiramente, cria-se um vetor de bits que será responsável por armazenar os bits das faixas de bits passadas por parâmetro no plano de bits. Cada valor dentro desse plano, tem por objetivo obter os bits correspondentes dentro da imagem. Para realização dessa tarefa, faz-se o uso, novamente, do operador and. Tal operação, combinada com um bit shift à direita, é responsável por transformar a entrada completa da imagem que são valores de 0 a 255, em 0's e 1's, um vetor binário. A última coisa a ser realizada aqui, é converter a matriz, como está formatada a imagem, em um array de uma dimensão.

Após realizar esse procedimento por todas os planos de bits, na ordem em que foram entregues, tem-se um vetor binário. Esse vetor binário, deve, agora, ser convertido para texto. Mas antes de tudo, é preciso convertê-lo para bytes.

Essa conversão é simples, basta pegar pacotes de 8 bits em 8 bits dentro do vetor e ir transformando tais valores de binário para decimal. Todo esse processo é abstraído na função chamada packbits, cuja entrada é justamente o array gerado anteriormente.

Feito isso, obtém-se, por fim, o array de bytes correspondente ao texto. Portanto agora, é necessário decodificá-lo segundo um encoding, que nesse caso é o padrão utf-8. Como dentro do array de bytes podem ocorrer alguns valores que não existem em tal codificação, para evitar que aconteçam erros nesse processo, é essencial que os erros sejam ignorados. Haja visto que, se há algum erro na decodificação do texto, esse erro não faz parte do texto escondido. Portanto é prudente, e necessário, que tais caracteres sejam ignorados.

Finalmente, possui-se a string que contém a mensagem de saída. Contudo, ela ainda não está pronta. Ainda resta uma operação pendente, que se trata de obter a faixa de dados que interessa como informação válida. Para isso acontecer, é preciso lembrar-se do artifício posicionado no fim da mensagem no programa de codificação, que é a marcação ao final do texto com três @. Quebra-se a string nesse ponto, e resgata a primeira parte, garantindo, assim, o texto decodificado. Ele será escrito para dentro de um arquivo de saída.

Tudo isso descrito anteriormente, pode ser visto na figura 6, que mostra a parte principal do programa de decodificação.

```
79     for bit in bit_plans:
80         bits_array = np.append(bits_array, unpackbits_to_array(image, int(bit)))
81
82     # Convert bits to bytes
83     bytes_array = np.packbits(bits_array)
84
85     # Convert bytes to string
86     string_from_bits = bytes_array.tobytes().decode(errors="ignore")
87
88     string_from_bits = string_from_bits.split("@@@")[0]
89
```

Figura 6 – Core do programa decodificacao.py

4.3. Análise dos Planos de Bits

Após o processo de decodificação, gera-se algumas imagens, a partir das imagens de entrada e saída da codificação, que são os chamados planos de bits 0, 1, 2 e 7 divididos pelos canais de cor: azul, verde e vermelho.

Para gerar tais arquivos, é possível utilizar o programa `plano_bits.py` que recebe como entrada o caminho para a imagem de entrada, o caminho para a imagem de saída, a pasta onde serão salvos os planos de bits da entrada e a pasta onde serão salvos os planos de bits da saída. Segue o formato de entrada para o programa.

```
➤ python    plano_bits.py    <imagem_entrada.png>    <imagem_saida.png>  
    <pasta_plano_bits_entrada> <pasta_plano_bits_saida>
```

Eles são usados como forma de identificar o efeito da esteganografia dentro da imagem e se é possível identificá-lo mesmo nas bandas mais altas dos bits.

Inicialmente, observa-se a primeira faixa de bit 0, a partir da tabela 1, percebem-se três imagens, cada uma correspondente ao plano de bit 0 das cores azul, verde e vermelho nessa ordem, obtidas todas a partir da figura 5. É perceptível nas imagens que os três canais de cores possuem um padrão similar, como se todos os pixels estivessem organizados em linhas verticais espaçadas. O que deixa claro que algo ocorreu dentro dessa imagem e suas informações foram alterados.

Sabe-se que, tal padrão de organização dos bits, deve-se, justamente, aos bits do texto posicionados na imagem.

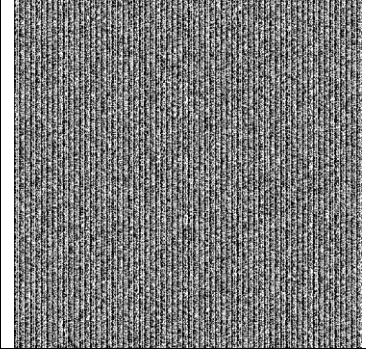
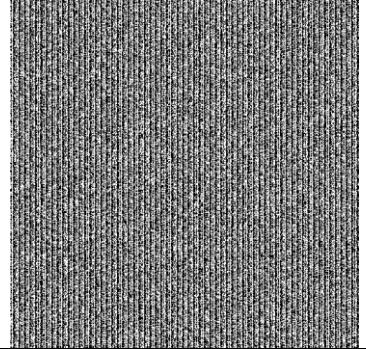
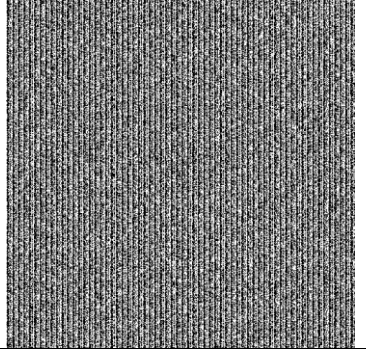
		
Azul	Verde	Vermelho

Tabela 1 – Plano de bits 0 na imagem de saída

Para o plano de bit 1, a mesma organização de bits é notada. Várias seções pretas que cortam a imagem na vertical, com um espaçamento parecido. Novamente, isso é devido à contribuição dos bits da mensagem codificada que ocupam todos os planos de bits 1 dos três canais de cores.

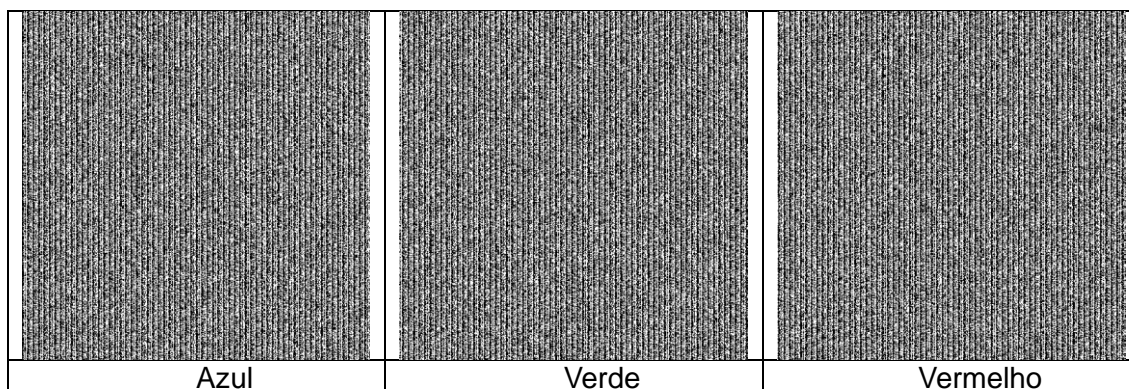


Tabela 2 – Plano de bits 1 na imagem de saída

Enquanto no plano de bits 2, visto na tabela 3, fica claro que ocorre uma mudança em relação aos planos 0 e 1. Tanto no azul, verde e vermelho, a parte superior da imagem segue o mesmo padrão dos planos anteriores. Porém, na parte inferior, esse padrão para, e observa-se uma organização diferente, um chiado sem forma definida, como anteriormente. Isso deve-se ao término do conteúdo da mensagem. Ou seja, todos os bits dela já foram acomodados e não é necessário utilizar as últimas posições do plano.

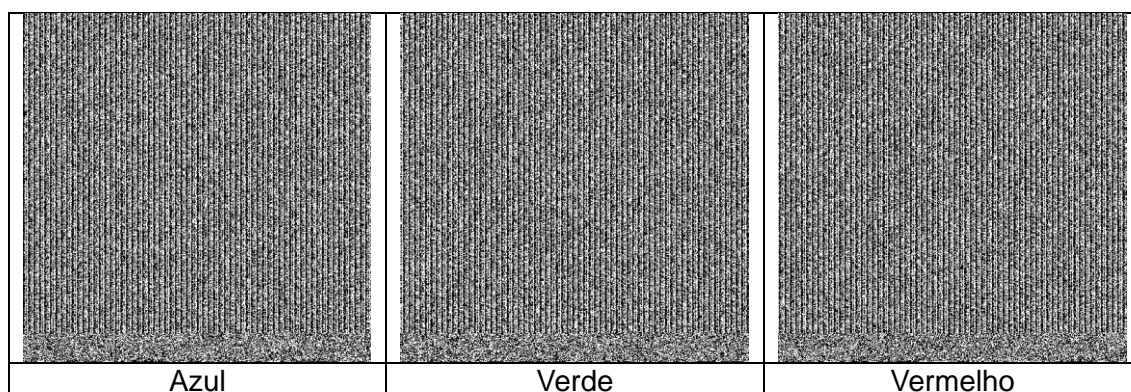


Tabela 3 – Plano de bits 2 na imagem de saída

Por fim, no último plano de bits a ser visto aqui, tem-se o plano de bit 7. Nele a ideia é tentar identificar se a mudança incorre em alguma interferência no bit mais significativo. Contudo, nota-se que isso não acontece. Não se encontra nenhum padrão como visto na tabela 1, 2 e 3. Logo, mesmo fazendo alterações consideráveis dentro do conteúdo da imagem, não alteramos os bits que dão forma a ela. O que é o objetivo, transmitir algum dado de forma imperceptível a olho nu.

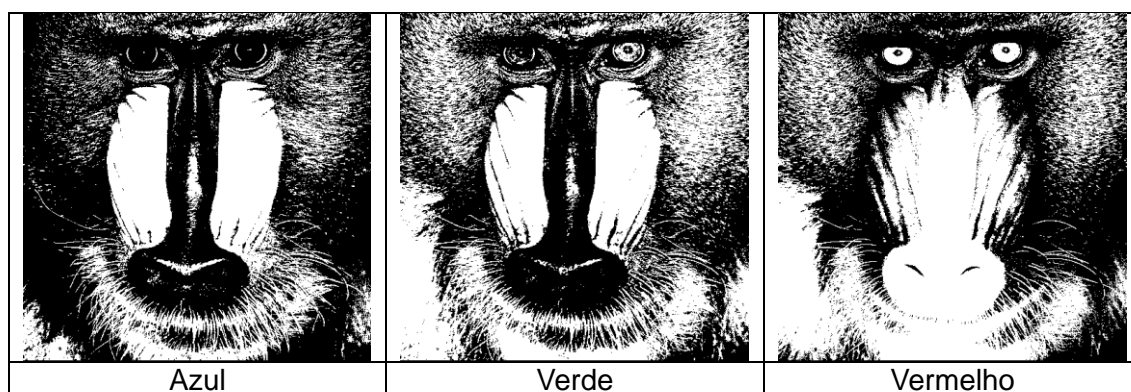


Tabela 4 – Plano de bits 7 na imagem de saída

4.4. Estressando o algoritmo

Para finalizar os testes, resta estressar o algoritmo com uma entrada maior, para entender como ele se comporta e em quanto tempo ele consegue devolver a saída.

De início, observa-se quais foram os resultados para a imagem do baboon.png, que neste caso é chamado de imagem_entrada.png, usado nos testes anteriores.

Para uma imagem de 623 KB de armazenamento e um texto de entrada de 281 KB o algoritmo leva 150ms para codificar e 70ms para decodificar em média.

Agora, fazendo uso da figura 7, cujo tamanho em bytes é 287 KB com uso de compactação. Uma vez que após a codificação a imagem passa a possuir 4087 KB de tamanho. E um texto de entrada grande, que possui 2660 KB de armazenamento, correspondente a um livro de 1000 páginas copiado e colado uma vez, dessa forma, tem-se um livro de quase 2000 páginas.



Figura 7 – Imagem de entrada grande para teste

Devido ao tamanho da imagem, é necessário usar mais de três bandas de bits, com isso, para acomodar a mensagem toda na imagem, é passado o plano de bits **0123**. Obtendo como saída a figura 8 modificada e com a mensagem em seu conteúdo.

Para gerar tal saída, o algoritmo leva 545ms em média para codificar e 360ms em média para realizar a decodificação. A partir desse resultado e dessa constatação, é notável que o desempenho do algoritmo é razoável, uma vez que, mesmo com uma entrada maior com mais bits e usando de 4 bandas de bits, o gasto de tempo cresceu de forma comportada.



Figura 8 – Imagem de saída grande para teste

Como foi utilizado até o plano de bit 3, já fica claro na imagem da figura 8 que o conteúdo foi alterado, haja visto que a contribuição dos bits da mensagem já pode ser vista em diversos pontos da imagem. Isso não acontece na figura 5, que usa somente os planos **012**.

4.5. Limitações e Desafios

Com relação às limitações, caso o texto possua caracteres especiais, caracteres que não estejam na codificação do utf-8, o algoritmo acaba não conseguindo atingir seu objetivo. Uma vez que, ele cai em um tratamento de erro, o que incorre em não converter o texto para bits e dessa forma, nada acontece com a imagem. Isso é informado ao usuário, que verifica quais os planos de bits tiveram sua escrita concluída, sem a ocorrência de erros.

Quanto aos desafios, aplicar algumas técnicas, que eram até comuns para outros temas, porém, um pouco diferente quando trazido ao contexto de imagens. Como por exemplo, executar operações lógicas em uma imagem. Conseguir visualizar os resultados. Ou mesmo, como entender o que cada saída significa e como tratá-la de maneiras diferentes, para determinar se algo está correto ou não.

5. Conclusão

Dessa forma, com a implementação do codificador e do decodificador, todo o processo de esteganografia foi concluído. Da entrada da imagem e do texto, até a saída da imagem com a mensagem e, após decodificação, a mensagem original novamente.

De acordo com os testes realizados, pode-se concluir, ainda, que alterar os bits menos significativos, 0, 1 e 2 é o suficiente e o ideal para que a mensagem não seja percebida ao visualizar a imagem. Já que, quando a quarta banda foi utilizada (plano de bit 3) já foi perceptível a adulteração em seu conteúdo.

Bem como, o algoritmo construído, fazendo uso da vetorização em grande parte do código, trouxe vários benefícios, no que diz respeito ao desempenho. O programa se comportou bem para entradas pequenas e manteve um bom comportamento quando testado com entradas maiores de texto e imagem.

Em suma, o algoritmo cumpre com a premissa da esteganografia, fazendo com que, quando utilizados nos bits menos significativos, o conteúdo da aparentemente não foi alterado. Só é possível notar tal mudança quando se faz a análise separada dos planos de bits. Para além disso, o programa fornece uma interface satisfatória e de simples uso para o usuário. Mesmo com a limitação de entrada, ele aceita a maior parte das mensagens e devolve os resultados de forma rápida e eficiente.