MC920 - Trabalho 2

Yuri Luiz de Oliveira
188802

12 de Novembro de 2020

1 Introdução

O trabalho consiste em utilizar uma técnica de pontilhado com difusão de erro em imagens coloridas para distribuir a diferença entre os valores das bandas RBG de cada pixel de imagens coloridas.

1.1 Abordagens de difusão de erro

	$f(x,y)$	7/16
3/16	5/16	1/16

(a) Floyd e Steinberg

			$f(x,y)$		32/200	
12/200		26/200		30/200		16/200
	12/200		26/200		12/200	
5/200		12/200		12/200		5/200

(b) Stevenson e Arce

		$f(x,y)$	8/32	4/32
2/32	4/32	8/32	4/32	2/32

(c) Burkes

		$f(x,y)$	5/32	3/32
2/32	4/32	5/32	4/32	2/32
	2/32	3/32	2/32	

(d) Sierra

		$f(x,y)$	8/42	4/42
2/42	4/42	8/42	4/42	2/42
1/42	2/42	4/42	2/42	1/42

(e) Stucki

		$f(x,y)$	7/48	5/48
3/48	5/48	7/48	5/48	3/48
1/48	3/48	5/48	3/48	1/48

(f) Jarvis, Judice e Ninke

Figure 1: Imagem retirada da especificação do problema

1.2 Imagens

Todas as imagens abaixo foram utilizadas para testes, entretanto, somente a imagem "peppers" será utilizada neste relatório.

1.2.1 baboon.png

Figure 2: Imagem retirada de `https://www.ic.unicamp.br/~helio/imagens_coloridas/baboon.png`

1.2.2 monalisa.png

Figure 3: Imagem retirada de `https://www.ic.unicamp.br/~helio/imagens_coloridas/monalisa.png`

1.2.3 peppers.png

Figure 4: Imagem retirada de `https://www.ic.unicamp.br/~helio/imagens_coloridas/peppers.png`

1.2.4 watch.png

Figure 5: Imagem retirada de `https://www.ic.unicamp.br/~helio/imagens_coloridas/watch.png`

2 Ambiente de desenvolvimento e testes

A solução para o trabalho foi implementada e testada em Python 3.8.5, com auxílio das bibliotecas numpy, opencv, matplotlib e argparse. O sistema operacional utilizado foi Ubuntu 20.04.

2.1 Preparando o ambiente

Para instalar as bibliotecas necessárias basta entrar na pasta do trabalho e rodar o comando

```
$ pip3 install -r requirements.txt
```

3 Rodando a solução

A solução aceita argumentos de linha de comando e deve ser executada através do interpretador de python. O seguinte comando roda a solução para a imagem presente em *"images/peppers.png"*, com abordagem de difusão de erro *a* (Floyd e Stein) e salva a imagem em *"out/peppers_a.png"*. O identificador da abordagem de difusão de erros é a letra prefixada ao nome da abordagem na Figura 1.

```
$ python3 halftoning.py -i images/peppers.png -ed a -o out/peppers_a.png
```

Note que o diretório *"out"* precisa existir.

3.1 Outros argumentos de execução

Além dos argumentos utilizados no comando acima, existem opções que podem ser passadas para os comandos.

3.1.1 sweep-mode

A forma como a imagem será percorrida: da esquerda para direita sempre ou alternadamente. Este argumento é utilizado através do argument *-sm* ou *–sweep-mode* e aceita os valores *default* (Esqueda para direita) ou *alternate* (Alternada).

3.1.2 display-mode

O que será mostrado como resultado do programa. O argumento é usado através de *-dm* ou *–display-mode* com valores *images*, *hist* ou *off*, mostrando a imagem resultante e origem, o histograma da imagem resultante ou nada, respectivamente.

3.2 Comando para rodar todas as formas de difusão de erro de uma imagem específica

Para obter o resultado de todas as formas de difusão de erro de uma imagem específica, basta utilizar o script *all_error_distributions.sh*. Este script recebe a imagem para a qual deve ser rodada a solução e o diretório ao qual devem ser salvos os resultados. O comando seguinte executa o script para a imagem *"images/peppers.png"* e salva os resultados no diretório *"out"*. **Note que as imagens utilizadas tem que estar dentro do diretório *"images"* presente caminho atual e devem ter extensão *".png"***.

```
$ ./all_error_distributions.sh peppers out
```

3.3 Formato dos dados

As imagens processadas **devem ser RGB e estar em formato PNG**.

4 Detalhes e decisões de implementação

O algoritmo utilizado na solução foi extraído do slide (pág.) 132 do material de Realce presente na página da disciplina, acessível através do link `https://www.ic.unicamp.br/~helio/disciplinas/MC920/aula_realce.pdf`. Este algoritmo funciona para imagens em escalas de cinza, em que cada pixel é um inteiro sem sinal de 0 a 255. Foi necessário adaptar o algoritmo para imagens RGB de forma que a difusão de erro é calculada para cada banda da imagem separadamente.

4.1 Tratamento de borda

O tratamento de borda foi feito enquanto o erro é aplicado a cada pixel, verificando se os pixels em que será aplicado o erro está dentro dos limites da imagem. Optou-se por não adicionar bordar à imagem antes de executar o algoritmo, dado que foi necessário executar os passos do algoritmo dentro de loops de qualquer forma. **Não foi possível pensar em formas de implementações com vetorização**.

4.2 Abordagens de difusão de erro

Decidiu-se seguir com uma única implementação do algoritmo que recebe a difusão de erro através de uma matriz. A matriz recebida pelo parâmetro recebe zeros nas posições vazias e na posição do pixel atual. Desta forma é necessário percorrer as posições vazias da difusão de error, o que tende a deixar o código mais lento do que fazer uma implementação única para cada difusão de erro que utiliza somente os valores não nulos. Esta decisão foi tomada a fim de tornar o código mais fácil de modificar e dar manutenção, dado que para adicionar uma nova forma de difusão basta adicionar sua matriz e não há riscos de ter erros específicos à implementação de cada abordagem.

5 Resultados obtidos

Vamos analisar o resultados das imagens após aplicar a técnica de pontilhado com difusão de erro à imagem *peppers* (Figura 4) com as abordagens proposta na figura 1.

5.1 (a) Floyd e Steinberg

Figure 6: Imagem resultante após aplicar a abordagem Floyd e Steinberg percorrendo a imagem da esquerda para a direita.

Figure 7: Imagem resultante após aplicar a abordagem Floyd e Steinberg percorrendo a imagem alternadamente.

5.2 (b) Stevenson e Arce

Figure 8: Imagem resultante após aplicar a abordagem Stevenson e Arce percorrendo a imagem da esquerda para a direita.

Figure 9: Imagem resultante após aplicar a abordagem Stevenson e Arce percorrendo a imagem alternadamente.

5.3 (C) Burkes

Figure 10: Imagem resultante após aplicar a abordagem Burkes percorrendo a imagem da esquerda para a direita.

Figure 11: Imagem resultante após aplicar a abordagem Burkes percorrendo a imagem alternadamente.

5.4 (d) Sierra

Figure 12: Imagem resultante após aplicar a abordagem Sierra percorrendo a imagem da esquerda para a direita.

Figure 13: Imagem resultante após aplicar a abordagem Sierra percorrendo a imagem alternadamente.

5.5 (e) Stucki

Figure 14: Imagem resultante após aplicar a abordagem Stucki percorrendo a imagem da esquerda para a direita.

Figure 15: Imagem resultante após aplicar a abordagem Stucki percorrendo a imagem alternadamente.

5.6 (f) Jarvis, Judice e Ninke

Figure 16: Imagem resultante após aplicar a abordagem Jarvis, Judice e Ninke percorrendo a imagem da esquerda para a direita.

Figure 17: Imagem resultante após aplicar a abordagem Jarvis, Judice e Ninke percorrendo a imagem alternadamente.

5.7 Comparando as varreduras da esquerda para direita e alternada

Percebemos duas características marcantes na varredura alternada em relação à padrão (esquerda para direita):

- A imagem fica menos clara: isto é perceptível em todas as imagens resultantes cujos pixels foram percorridos alternadamente. Nas imagens há menor incidencia de pixels brancos (ou muito claros). Não tenho hipóteses do motivo disso.
- Há menos realce do "movimento" de varredura de imagem: este efeito fica bastante perceptível nas imagens com difusão de erro (b) Stevenson e Arce (Figuras 8 e 9).

6 Principais falhas durante o desenvolvimento

A primeira versão da solução tinha opção de mostrar o resultado em uma janela somente, não havia a possibilidade de salvar a imagem resultante em um arquivo. Ao implementar o salvamento em arquivo de saída, a imagem salva ficou escura, como visto na Figura 18. Isto aconteceu, pois os pixels da imagem de saída eram compostos de valores 0 ou 1 somente e a imagem salva foi interpretada como valores de 0 a 255. Por outro lado, quando a imagem era mostrada na janela, ela era interpretada como escala de 0 a 1 (sendo 0 considerado como o valor

mínimo da banda e 1 o valor máximo da banda). O problema foi resolvido retornando uma imagem normalizada com valores de 0 a 255. O resultado após a normalização foi uma imagem cujos pixel eram compostos de valores 0 ou 255 somente.

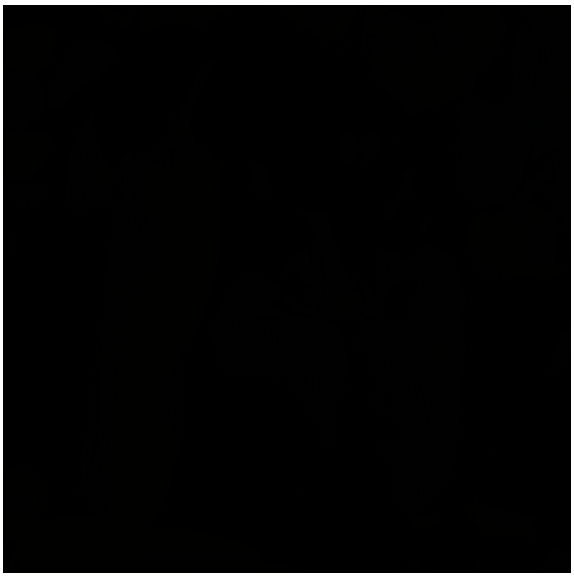

Figure 18: Imagem salva a um arquivo antes de efetuar normalização para valores de a 0 a 255.

Ao implementar a varredura alternada, o código que varria a imagem da esquerda para a direita já existia. E o código que percorre alternadamente foi implementado no mesmo trecho, fazendo apenas o chaveamento entre os valores mínimos e máximos da *range* utilizada para percorrer as colunas da imagem. Por exemplo: Uma imagem de dimensões 512x512 teria como mínimo e máximo da range que percorre as colunas da primeira linha os valores *0* como mínimo e *512* como máximo, como abaixo:

```
range(0, 512)
```

Por outro lado, as colunas da segunda linha seriam percorridas com a seguinte range:

```
# 511 para não incluir a posição 512 e -1 para incluir a posição 0
range(511, -1)
```

Assim em diante. Entretanto, a range acima não gera valor algum, ao contrário do esperado: valores decrescentes de 511 a 0. Como resultado, todas as linhas ímpares (considerando o índice das linhas de 0 a 511), não foram modificadas e ficaram com zero (valor padrão dos pixels da imagem resultante), como visto na Figura 19. Note que, esta imagem ficou notavelmente mais escura, dado que

uma cada duas linhas era completamente preta (como evidenciado na Figura 20). Para resolver o problema, utilizou-se uma variável *step* que possuía valor 1 quando as colunas deviam ser percorridas da esquerda para direita e valor -1 quando as colunas deveriam ser percorridas da direita para esquerda.

Figure 19: Imagem resultante do algoritmo com varredura alternada com linhas ímpares com valor zero.

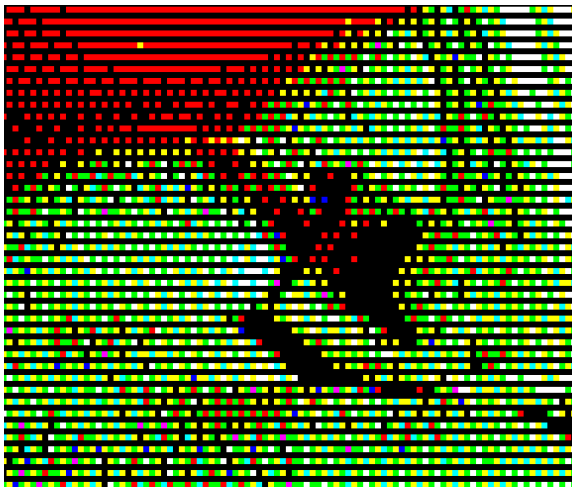

Figure 20: Imagem resultante do algoritmo com varredura alternada com linhas ímpares com valor zero com zoom.

7 Limitações

O algoritmo funciona somente para imagens RGB, não sendo possível utilizar em imagens monocromáticas. Outra limitação: por decisão de implementação, assume-se que a matriz que representa a abordagem de difusão de erro terá sempre a posição do pixel (x, y) na coluna do meio da primeira linha, impossibilitando, utilizar matrizes que representem abordagem que não possuam esta regra.

8 Conclusões

O algoritmo utilizado funciona bem para resolver como uma implementação de uma técnica de meio-tom por difusão de erro. Entretanto, a forma como foi implementado não é performática, pois não é vetorizada. Ainda, este algoritmo não é trivial de vetorizar, e provavelmente, precisaria ser escrito de outra forma para que fosse viável vetorizá-lo com bibliotecas como *numpy*.