Trabalho 3

Processamento Digital de Imagem

Ieremies Vieira da Fonseca Romero

Introdução

Agora nós temos uma imagem representada por zeros e uns onde o texto é demarcado pelos pixels com valor de 1. Filtramos qualquer componente conexa com menos de 370 pixels.

```
def save(img):
    # turn back to gray scale
    img[img == 0] = 255
    img[img == 1] = 0
    cv2.imwrite('out.png', img)

save(img)
```

O programa

Neste trabalho, utilizamos as bibliotecas `numpy` 1.24.1 e OpenCV 4.7.0.72 utilizado via `cv2`.

```
import cv2
import numpy as np
```

Para leitura de imagens, utilizaremos a função do `cv2` e modificaremos os pixels brancos em 0 e os preto em 1 para realizar os métodos de morfologia.

```
import cv2
import numpy as np

img = cv2.imread('bitmap.pbm',
                 cv2.IMREAD_UNCHANGED)
img[img == 0] = 1   # all black pixels to 1
img[img == 255] = 0 # all whites pixels to 0
```

Ressaltamos apenas o uso de `cv2.IMREAD_UNCHANGED` para que a função de leitura não altera-se os valores.

Para escrita, o método irá variar baseado em qual estágio do processo estamos. Utilizaremos a nossa função `save` para desfazer a conversão descrita acima e salvar novamente como uma imagem binária.

```
def save(file, img):
    aux = img.copy()
    aux[aux == 0] = 255
    aux[aux == 1] = 0
    cv2.imwrite(file, aux)
```

Já ao final, depois de transformar-mos a imagem original em *RGB* utilizando a função `cv2.cvtColor`, então usaremos o `cv2.imwrite`.

Dilatações e erosões

Como descrito no enúnciado, primeiramente realizaremos a dilatação e posterior erosão, o que é chamado de operação de **fechamento**. Realizaremos uma vez utilizando um elemento estruturante de dimensões $(1, 100)$ e outra vez com um elemento de dimensões $(200, 1)$. Assim, um foca na horizontal enquanto outro na vertical.

```
kernel1=np.ones((1, 100), np.uint8)
img1=cv2.dilate(img, kernel1, iterations=1)
img1=cv2.erode(img1, kernel1, iterations=1)

kernel2=np.ones((200, 1), np.uint8)
img2=cv2.dilate(img, kernel2, iterations=1)
img2=cv2.erode(img2, kernel2, iterations=1)
```

Por fim, juntaremos ambas imagens geradas por meio de um operador lógico de *and* e utilizamos a função `cv2.morphologyEx` com parâmetro `cv2.MORPH_CLOSE` e um operador morfológico de dimensão $(1, 30)$ para realizar o fechamento.

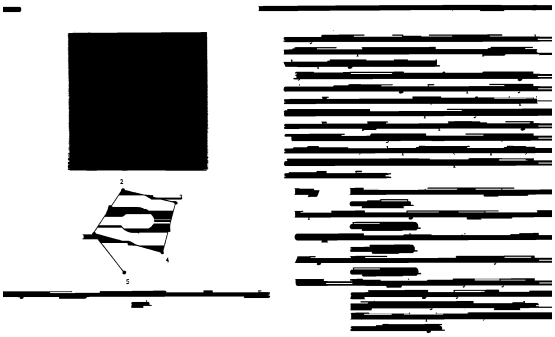

Figura 1: Resultado do primeiro fechamento, *kernel* $(1, 100)$.

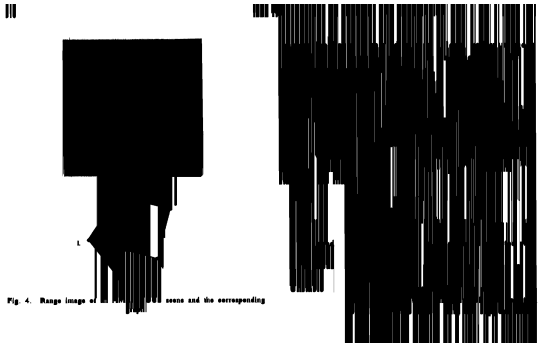

Figura 2: Resultado do segundo fechamento, *kernel* $(200, 1)$.

```
img = cv2.bitwise_and(img1, img2)

kernel = np.ones((1, 30), np.uint8)
img = cv2.morphologyEx(img, cv2.MORPH_CLOSE,
↪   kernel)
```

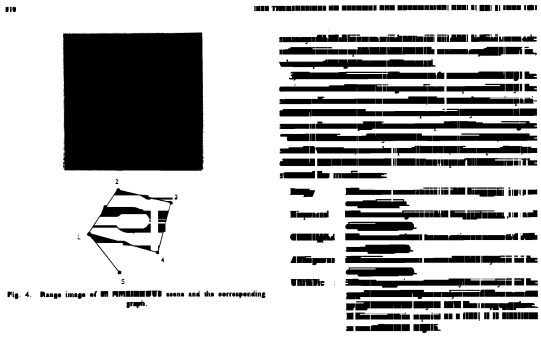

Figura 3: Resultado da intersecção entre as imagens produzidas nos passos anteriores.

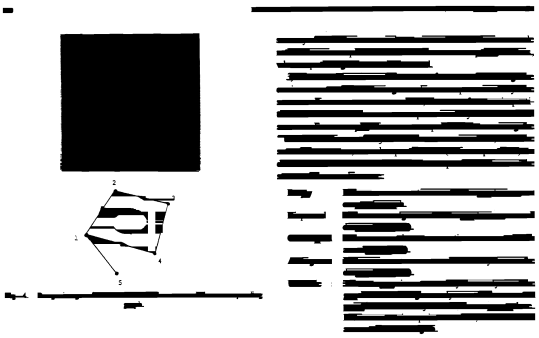

Figura 4: Resultado da operação de fechamento, *kernel* $(1, 30)$.

Assim, ao final, possuímos componentes conexas que muito se assemelham à região que a linha de texto ocupava, com exceção das imagens. Precisamos, por tanto, analisar mais a fundo as propriedades destas regiões para determinar quais são textos e quais não o são.

Componentes conexas

O primeiro passo é detectar as componentes conexas que aparecem na image. Para tal, utilizamores a função `cv2.connectedComponentsWithStats`. Esta recebe a imagem sob a qual aplicaremos o algoritmo de detecção de componentes conexas, bem como o tipo de vizinhança (utilizaremos 8). Demais parâmetros são utilizados para indicar onde esta função irá colocar os resultados, mas neste caso, iremos recebe-los no retorno desta e os mantemos como `None`.

```
nlabels, _, stats, _ =
    cv2.connectedComponentsWithStats(
        img, None, None, None, 8, cv2.CV_32S
    )
```

Assim, possuímos uma lista de dados sobre as componentes (lista `stats`) e a quantidade destas (inteiro `nlabels`). A variável `stats` possui `nlabels` elementos com as informações de cada componente conexo. Estas incluem: as coordenadas do canto superior esquerdo do retângulo ao redor da componente e as dimensões de tal retângulo, bem como outras informações que não nos serão úteis nesse momento.

Com tais informações, podemos remover janelas da imagem original que correspondem a tais componentes conexas e, a partir destas janelas, analisar certas estatísticas que nos permitirão determinar se o conteúdo delas é ou não texto.

```
# read original image
img = cv2.imread("bitmap.pbm",
                 cv2.IMREAD_UNCHANGED)
text = [] # data about the windows
for i in range(nlabels):
    x, y, w, h = stats[i][:4]
    crop = img[y : y + h, x : x + w]
    text.append([x, y, w, h, b])
    text[-1].append(percentage(crop))
    text[-1].append(v_transitions(crop))
    text[-1].append(h_transitions(crop))
```

A primeira estatística interessante é o percentual de pixels pretos em cada janela. Calculamo-no utilizando a nossa função `percentage`.

```
def percentage(img) -> float:
    black = np.count_nonzero(img == 0)
    total = img.shape[0] * img.shape[1]
    return round(black / total, 2)
```

Observamos que a maioria das janelas possuíam

Outro dado interessante é a quantidade de transições verticais ou horizontais de pixels brancos para pretos em relação ao número total de pixels pretos. Podemos encontrar tal dado com as nossas funções `h_transitions` e `v_transitions`. Por questões de espaço, aqui só reproduziremos uma delas, mas a outra é similar.

```
def h_transitions(img) -> int:
    transitions = 0
    for i in range(img.shape[0] - 1):
        for j in range(img.shape[1]):
            if (img[i][j] == 0 and
                img[i + 1][j] == 255):
                transitions += 1
    black = np.count_nonzero(img == 0)
    if black == 0:
        return 0
    return round(transitions / black, 3)
```

Com tais informações em mãos, determinamos que as componente que realmente representam texto são aquelas que possuem percentual de pixels pretos entre 9% e 40% e frequência de transições de pixels brancos para pretos entre 9% e 40%. Por fim, adicionamos uma retrição na altura da componente conexa em especial para eliminar a componente que envolve a imagem inteira.

```
text = [
    comp
    for comp in text
    if (
        20 < comp[3] < 50
        and 0.09 < comp[5] < 0.4
        and 0.09 < comp[6] < 0.4
        and 0.09 < comp[7] < 0.4
    )
]
```

Por fim, nossa variável `text` possui as informações sobre cada *bounding box* de cada linha. Podemos desenha-las numa imagem *RGB* com a cor verde para ilustrar.

```
img = cv2.cvtColor(img, cv2.COLOR_GRAY2RGB)
for line in text:
    x, y, w, h = line[:4]
    img = cv2.rectangle(img, # imagem
```

```
                    (x, y), # toplef
                    (x + w, y + h),
                    (0, 255, 0), # color
                    2) # thickness
```

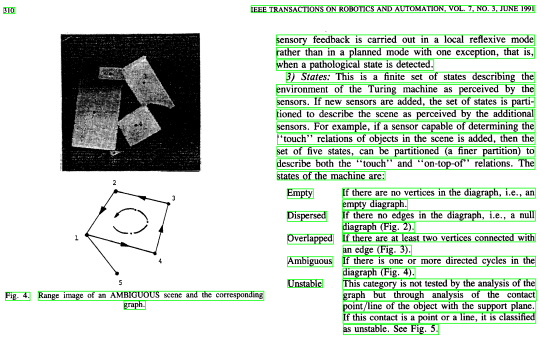

Figura 5: Linhas detectadas cercadas por suas *bounding boxes* em verde.

Palavras

Para detectar então cada palavra individualmente, podemos aplicar técnicas similares, mas agora apenas em janelas da imagem as quais sabemos ser texto. Através de experimentação e se baseando na teoria, buscamos realizar a dilatação vertical e horizontal (elementos estruturantes $(1, 10)$ e $(10, 1)$) das imagens e posterior fechamento (elemento estruturante $(1, 3)$).

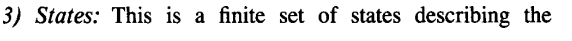

Figura 6: Linha de texto inicial.

Figura 7: Linha de texto dilatada com *kernels* $(1, 10)$ e $(10, 1)$.

Assim, cada palavra se tornou um conexo "borrão", o que nos permite utilizar a mesma função de componentes conexas para determinas suas posições.

Figura 8: Linha de texto após fechamento com *kernel* $(1, 3)$.

```
def detect_words(img):
    kernel1 = np.ones((1, 10), np.uint8)
    img = cv2.dilate(img, kernel1,
                     iterations=1)

    kernel2 = np.ones((10, 1), np.uint8)
    img = cv2.dilate(img, kernel2,
                     iterations=1)

    kernel3 = np.ones((1, 3), np.uint8)
    img = cv2.morphologyEx(img,
                           cv2.MORPH_CLOSE,
                           kernel3)

    return (
        cv2.connectedComponentsWithStats
            (img,None,None,None,8,cv2.CV_32S)
    )
```

Por fim, podemos usar as *bounding boxes* azuis para marcar as palavras e repetirmos o processo para cada linha. Perceba que, as vezes, nosso algoritmo interpreta sequências como `(Fig.` como uma palavra.

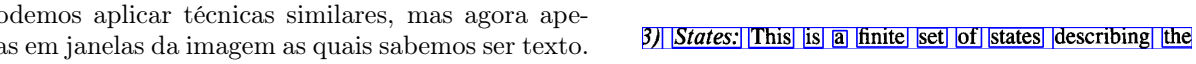

Figura 9: Linha com as palavras marcadas em *bouding boxes* azuis.

Conclusão

Neste trabalho, fomos capazes de aplicar os operadores morfológicos e, com isso, detectar 35 linhas de texto e 241 palavras.

Como trabalhos futuros, ressaltamos a possibilidade de utilizar técnicas de clusterização para determinar as condições que caracterizam texto, como fizemos manualmente aqui. Assim seríamos capazes de processar textos mais diversos. Além disso, baseado nas propriedades de componentes conexos de cada caractere, como a quantidade de buracos, pode ser feita a detecção de cada um deles e transcrição para texto.

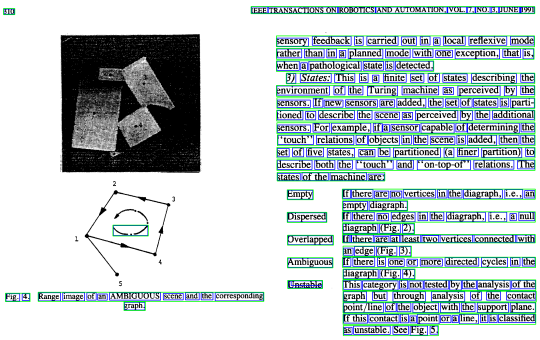

Figura 10: Imagem final. Linhas estão marcadas por caixas verdes e palavras por caixas azuis.