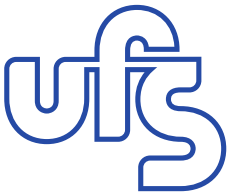


UNIVERSIDADE FEDERAL DE SERGIPE

CENTRO DE CIÊNCIAS EXATAS E TECNOLOGIA

DEPARTAMENTO DE COMPUTAÇÃO

PreOCR - Trabalho de Processamento de Imagens T01 2023.2

Professora: Beatriz Trinchão Andrade

Grupo 13 - Everton Santos de Andrade Júnior

São Cristóvão – Sergipe

2024

Sumário

1

Introdução e Resultados

Neste trabalho, desenvolvemos um programa capaz de processar imagens binárias no formato PBM ASCII (PGM tipo P1), contendo texto dos tipos e tamanhos de fontes, variando de 10 à 40. Conseguimos determinar o básico do trabalho que são o número de linhas e palavras no texto e os retangulos dessas palavras, gerando uma imagem de saída com esses retangulos. Além disso, nosso grupo implementou mais funcionalidades, como a detecção de colunas e blocos do texto, utilizando o conceito de distância alinhada. Includingo a *bouding box* (bbox) desses blocos e linhas que indicam as colunas.

Adicionalmente, para ilustrar nossos resultados de forma mais interativa, geramos uma série de imagens intermediárias que mostram o processo de detecção de blocos, colunas, palavras, e linhas destacando as regiões por um retangulo de diferentes cores. O formato dessas imagems é P3 (ascii RGB). Essas imagens foram usadas em sequencia de frames para gerar um video de saíoda usando o programa de linha de commando “ffmpeg“.

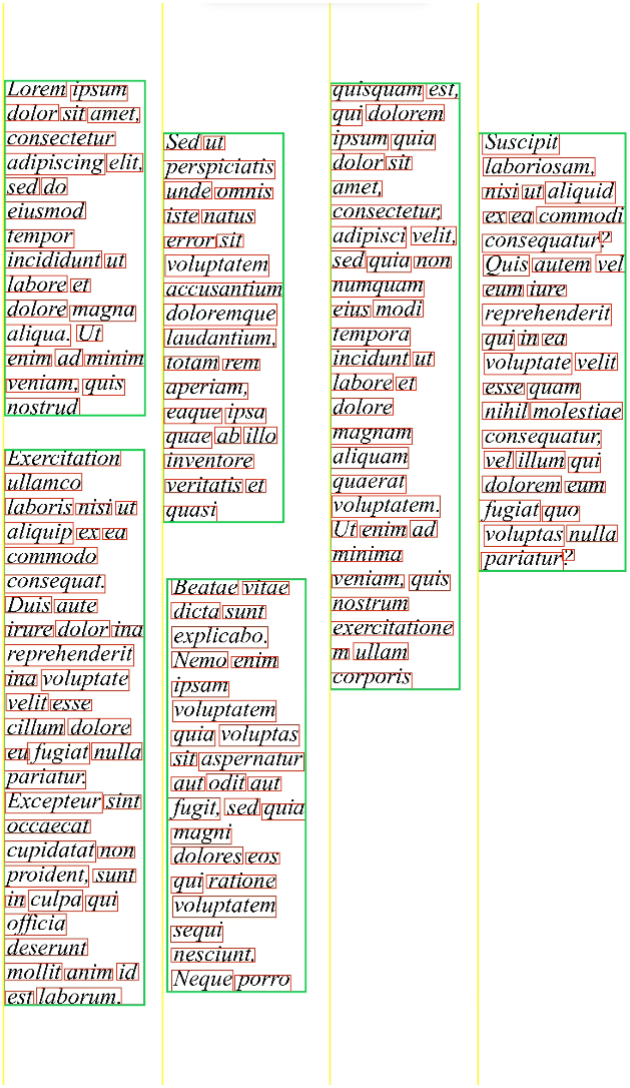
A nossa abordagem é dinamica, baseado na altura das palavras mudamos os parametros enquanto o programa roda. Desse modo é possivel identificar estruturas de texto em diferentes alinhamentos, como justificado, esquerda, centro e direita, além de lidar com diferentes tamanhos e estilos de fonte, incluindo Comic Sans, Impact, Cascadia Code, Arial e Times New Roman. Um exemplo de video gerado pode ser encontrado em <https://youtu.be/uA45GeodGss?si=ZOgsA2av7EKZeG69>.

Na Figura 3

1.1 Resultados

Inicialmente, o algoritmo lê a imagem de entrada e a pré-processa aplicando filtro da medianos (nome da função é no código `median_blur`) e invertendo as cores. Essas etapas visam retirar possíveis ruídos do tipo sal de pimenta. O tamanho do filtro mediano é 3 de altura e largulo

Figura 1 – Times New Roman em itálico, com 4 colunas, 6 blocos de texto.



Fonte: Autor.

já que foi confirmado que os ruídos não passam do tamanho de 1 pixel. A inversão da imagem é feita para aplicar algoritmos morfológicos, como a dilatação, para melhorar a conectividade dos componentes de texto e facilitar o processamento subsequente. Dependendo dos parâmetros especificados, o algoritmo pode aplicar operações adicionais como fechamento (closing) ou abertura (opening) para refinar ainda mais as regiões de texto. A ideia é utilizar um elemento estruturante, conhecido como SE (do inglês, *Structuring Element*), que connecte as letras; isso pode ser feito com um SE que seja horizontal. Foi testado com diferentes SEs, como fomarto circular, de cruz, vertical, mas o melhor é o horizontal. E faz sentido pois as letras vem logo à direta da anterior, então para conectalas, precisamos esticar na horizontal para "grudar"uma na

outra. Veja como é da uma "esticada"para horzinhotal ao aplicar um SE [000] [111] [000].

$$\begin{matrix} 0 & 0 & 0 \\ 1 & 1 & 1 \\ 0 & 0 & 0 \end{matrix}$$

Na Figura 3

Aplicação de operações de dilatação horizontal para melhorar a conectividade entre caracteres adjacentes. Após as operações morfológicas, indetificamos as palavras individuais através componentes conectados dentro da imagem. Cada palavra é contida em uma bbox, que é desenhada retângulos ao redor delas na imagem de saída. Componentes conectados é um conceito de grafos que pode ser aplicado numa imagem binarias, pixels brancos são nós. Existem tipos de connectividade. Se consideramos os vizinhos de um pixel como apenas esquerda, direita, cima e baixo, então temos 4-conectividade (GONZALEZ; WOODS, 2008, 2.5.2 Adjacency, Connectivity, Regions, and Boundaries). Se adicionarmos as diagonais, agora temos 8-conectividade. O Algortimo para encontrar esses componentes conectados se basea numa busca profunda, marcado os visitados, toda vez que na imagem for não zero fazemos a busca até não encontrar mais caminhos, considerando as digonais. Se não encontramos mais caminhos quer dizer que esse é um compoenente. Repetimos até encontrar todos atualizando o ponto mais a em esquerda inferior e o maiis à direita superior do componente para encontrar o bbox da palavra. O código ilustra esse processo.

Também fazemos o agrupamento dessas palavras em blocos com base em sua proximidade espacial, possibilitando uma análise melhor da organização do texto podemos dizendo que cada bloco é um paragrafo.

Opcionalmente, aplicação de operações de fechamento e abertura para refinar as regiões de texto, dependendo dos parâmetros definidos.

Há uma dilatação adicional da imagem, se necessário, com base na altura média dos componentes de texto já filtrados.

Ao longo do processo, o algoritmo incorpora várias otimizações e ajustes de parâmetros para se adaptar a diferentes tipos de imagens de entrada e layouts de texto. Por exemplo, parâmetros como tamanho do kernel e contagens de iteração são ajustados dinamicamente com base nas características da imagem de entrada, como tamanho do texto e níveis de ruído.

Filtragem dos componentes de texto para eliminar bbox de pontuação e manter apenas bboxes de palavras. Agrupamento de palavras em blocos com base em sua proximidade espacial. Desenho de caixas delimitadoras ao redor das palavras e blocos identificados na imagem original. Geração de um vídeo para visualizar interativamente as etapas do algoritmo. Opcionalmente, escrita de imagens separadas para cada palavra identificada, para uso posterior em tentativas de reconhecimento óptico de caracteres (OCR). Escrita da imagem final com caixas delimitadoras de palavras e blocos para análise adicional.

Além disso, o algoritmo fornece opções para gerar imagens intermediárias e vídeos para visualizar as etapas de processamento, auxiliando na depuração e compreensão do comportamento do algoritmo. Essas visualizações melhoram a transparência e a interpretabilidade da operação do algoritmo.

Em resumo, o algoritmo implementado demonstra uma abordagem abrangente para o reconhecimento de texto em imagens binárias, utilizando uma combinação de técnicas de processamento de imagem, regras heurísticas e parâmetros adaptativos. Ao refinar iterativamente regiões de texto e analisar seus relacionamentos espaciais, o algoritmo alcança uma detecção precisa de palavras, linhas, colunas e blocos, estabelecendo as bases para tarefas de reconhecimento de texto mais avançadas, como reconhecimento óptico de caracteres (OCR).

Ela é especialmente útil para preencher pequenos espaços ou quebrar conexões entre objetos. No Código 1 a função ‘understandable_dilate‘ implementa função implementa uma dilatação morfológica de forma compreensível, percorrendo cada pixel da imagem. A variável ‘kernel‘ representa o elemento estruturante, que define a forma e o tamanho da dilatação. A imagem é percorrida pixel a pixel, e para cada pixel, verifica-se se pelo menos um dos pixels na vizinhança definida pelo kernel é branco (valor 255). Se pelo menos um dos pixels na vizinhança é branco, o pixel central é definido como branco (255) na imagem resultante. Caso contrário, o pixel central é definido como preto (0) na imagem resultante. O parâmetro ‘iterations‘ indica quantas vezes a dilatação deve ser aplicada à imagem. Quanto maior o número de iterações, mais ampliado será o objeto na imagem.

Já no Código 2 ‘fast_dilate2‘:** função implementa uma dilatação morfológica mais rápida, aproveitando as operações de matriz do NumPy. A imagem é expandida com um preenchimento adequado para evitar problemas de borda durante a aplicação do kernel. O kernel é então aplicado à imagem usando a função ‘np.maximum‘, que calcula o máximo elemento a elemento entre a imagem e o subconjunto da imagem definido pelo kernel. Isso efetivamente realiza uma dilatação, onde o pixel central de uma vizinhança é definido como o valor máximo dessa vizinhança. Ambas as implementações alcançam o mesmo resultado de dilatação morfológica, mas a segunda função é mais eficiente computacionalmente devido ao uso de operações vetorizadas do NumPy. Isso resulta em uma execução mais rápida, especialmente em imagens grandes.

‘‘‘

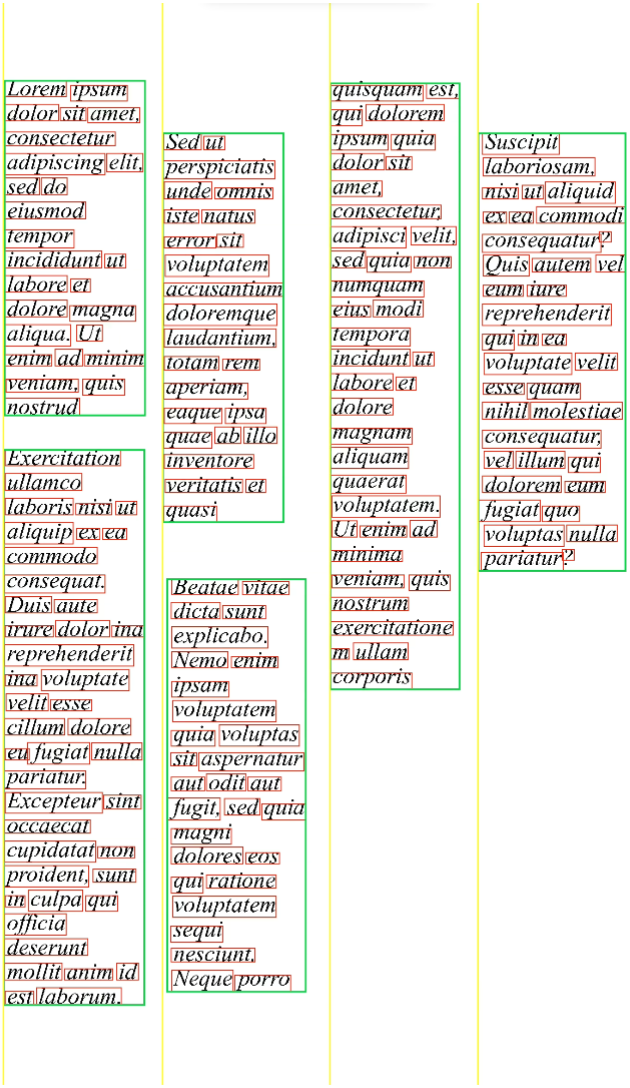
Código 1 – .

```
def understandable_dilate(image, kernel):
    result = np.zeros(image.shape)
    height = image.shape[0]
    width = image.shape[1]
    kernel_height = kernel.shape[1]
    kernel_width = kernel.shape[0]
    kernel_width_delta = kernel_width // 2
    kernel_height_delta = kernel_height // 2
    for y in range(height):
        for x in range(width):
            all_good = False
            for j in range(kernel_height):
                for i in range(kernel_width):
                    i_offset = i - kernel_width_delta
                    j_offset = j - kernel_height_delta
                    color = 0
                    if (
                        (x + i_offset) >= 0
                        and (x + i_offset) < width
                        and y + j_offset >= 0
                        and y + j_offset < height
                    ):
                        color = image[y + j_offset, x + i_offset]
                    kcolor = kernel[j, i]
                    if int(kcolor) * int(color):
                        all_good = True
                        break
                if all_good:
                    break
            if all_good:
                result[y, x] = 255
            else:
                result[y, x] = 0
    return result
```

Figura 2 – Cacadia Code em negrito com tamanho 10, 2 colunas e 5 blocos de texto. A entrada possue baixa resolução e ruídos.

Fonte: Autor.

Figura 3 – Times New Roman em itálico, com 4 colunas, 6 blocos de texto.



Fonte: Autor.

Figura 4 – Cacadia Code em negrito com tamanho 10, 2 colunas e 5 blocos de texto. A entrada possue baixa resolução e ruídos.

Fonte: Autor.

Código 2 – .

```
def fast_dilate2(image, kernel):
    global counter
    height, width = image.shape
    kernel_height, kernel_width = kernel.shape

    kernel_width_delta = kernel_width // 2
    kernel_height_delta = kernel_height // 2

    # We pad by the kernel delta, top, bottom, left and right
    padded_image = pad(
        image,
        kernel_height_delta,
        kernel_height_delta,
        kernel_width_delta,
        kernel_width_delta,
    )

    dilated = np.zeros(image.shape, dtype=np.uint8)
    for j in range(kernel_height):
        for i in range(kernel_width):
            if kernel[j, i] == 1:
                shifted_sub_image = padded_image[j : j + height, i :
                    i + width]
                dilated = np.maximum(
                    dilated, shifted_sub_image
                )

    return dilated
```

2

Estrutura do Projeto

3
Requisitos

```
python3 src/main.py assets/grupo_13_arial_esquerda_tamanho_16_colunas_2_blocos_4_
```

4

Métodos e Implementações

Seguindo as formulas de dilatação em (GONZALEZ; WOODS, 2008, capitulo 9)

5
Resultados

implementção <https://www.youtube.com/watch?v=uA45GeodGss> Descrição da implementação do programa. Destaque para soluções desenvolvidas para problemas específicos encontrados durante o desenvolvimento. Resultados:

Apresentação dos resultados obtidos. Inclusão de exemplos de imagens de entrada e saída. Discussão sobre a eficácia do programa e eventuais limitações.

6
Conclusão

Sumarização dos principais resultados e contribuições do trabalho. Reflexão sobre o aprendizado durante o desenvolvimento do programa. Sugestões para trabalhos futuros ou melhorias no programa.

Referências

GONZALEZ, R. C.; WOODS, R. E. *Digital Image Processing*. Upper Saddle River, NJ: Pearson Prentice Hall, 2008. Citado 2 vezes nas páginas 4 e 13.