Relatório Final: Técnicas de Processamento de Imagens em Python

Este relatório apresenta a implementação e análise de três técnicas fundamentais de processamento de imagens utilizando a linguagem Python e bibliotecas associadas como OpenCV, Scikit-image, NumPy e Matplotlib. Os tópicos abordados são: Filtros de Frequência, Segmentação Watershed e Crescimento de Regiões.

Relatório: Filtros de Frequência em Processamento de Imagens com Python

1. Introdução Teórica

A filtragem no domínio da frequência é uma técnica fundamental em processamento de imagens que permite a manipulação de características de uma imagem com base em suas componentes de frequência. Imagens, tipicamente representadas no domínio espacial (pixels), podem ser convertidas para o domínio da frequência utilizando transformadas como a Transformada Rápida de Fourier (FFT). No domínio da frequência, diferentes componentes correspondem a diferentes aspectos da imagem: baixas frequências geralmente representam as variações suaves e a estrutura geral, enquanto altas frequências capturam detalhes finos, bordas e ruídos.

A aplicação de filtros no domínio da frequência envolve multiplicar a transformada da imagem por uma função de filtro. Após a filtragem, a

transformada inversa de Fourier é aplicada para retornar a imagem ao domínio espacial, agora modificada.

Tipos Comuns de Filtros de Frequência:

- **Filtro Passa-Baixa (Low-Pass Filter - LPF):** Atenua ou remove as componentes de alta frequência, resultando em uma imagem mais suave (borrada). É útil para reduzir ruídos e remover detalhes excessivos. Um exemplo comum é o filtro passa-baixa Gaussiano, que aplica uma função Gaussiana como máscara no domínio da frequência.
- **Filtro Passa-Alta (High-Pass Filter - HPF):** Atenua ou remove as componentes de baixa frequência, realçando as bordas, detalhes finos e transições abruptas de intensidade. Isso pode levar a um aumento da nitidez da imagem. O complemento de um filtro passa-baixa pode funcionar como um filtro passa-alta.

O processo geral de filtragem no domínio da frequência é: 1. Transformar a imagem de entrada f(x,y) para o domínio da frequência, obtendo F(u,v) (e.g., usando DFT/FFT). 2. Multiplicar F(u,v) por uma função de filtro H(u,v) no domínio da frequência: G(u,v) = F(u,v) * H(u,v). 3. Aplicar a transformada inversa (e.g., IDFT/IFFT) em G(u,v) para obter a imagem filtrada g(x,y) no domínio espacial.

2. Implementação em Python

A seguir, apresentamos o código Python utilizado para aplicar filtros de frequência (passa-baixa e passa-alta Gaussianos) em imagens. Utilizamos as bibliotecas `OpenCV`, `NumPy`, `scikit-image` e `Matplotlib`.

```
import cv2
import numpy as np
from matplotlib import pyplot as plt
import os
import skimage.io
from skimage import img_as_ubyte, color, exposure

def aplicar_filtros_frequencia(caminho_imagem_entrada,
caminho_imagem_saida_base):
    """
    Aplica filtros de frequência (passa-baixa e passa-
alta Gaussiano) a uma imagem.

    Args:
        caminho_imagem_entrada (str): Caminho para a
imagem de entrada.
        caminho_imagem_saida_base (str): Caminho base
para salvar as imagens de saída (sem extensão).
    """
    print(f"[DEBUG] Iniciando aplicar_filtros_frequencia
para: {caminho_imagem_entrada}")
    diretorio_saida =
os.path.dirname(caminho_imagem_saida_base)
    if not os.path.exists(diretorio_saida):
        os.makedirs(diretorio_saida)
        print(f"[DEBUG] Diretório de saída criado:
{diretorio_saida}")

    img = skimage.io.imread(caminho_imagem_entrada)
    if img is None:
        print(f"[ERRO] Erro ao carregar a imagem:
{caminho_imagem_entrada}")
        return
    print(f"[DEBUG] Imagem carregada:
{caminho_imagem_entrada}, shape: {img.shape}")
```

```
    if len(img.shape) == 3:
        img_gray = color.rgb2gray(img)
    else:
        img_gray = img

    img_gray_ubyte = img_as_ubyte(img_gray)
    print(f"[DEBUG] Imagem convertida para escala de
cinza ubyte, shape: {img_gray_ubyte.shape}")

    # Transformada de Fourier
    dft = cv2.dft(np.float32(img_gray),
flags=cv2.DFT_COMPLEX_OUTPUT)
    dft_shift = np.fft.fftshift(dft)
    magnitude_spectrum_original = 20 *
np.log(cv2.magnitude(dft_shift[:, :, 0], dft_shift[:, :,
1]) + 1e-6) # Adicionado 1e-6 para evitar log(0)

    rows, cols = img_gray.shape
    crow, ccol = rows // 2, cols // 2

    # --- Filtro Passa-Baixa Gaussiano ---
    print("[DEBUG] Aplicando Filtro Passa-Baixa
Gaussiano")
    D0_low = 30
    mask_low = np.zeros((rows, cols, 2), np.float32)
    for r in range(rows):
        for c in range(cols):
            D = np.sqrt((r - crow)**2 + (c - ccol)**2)
            mask_low[r, c, 0] = np.exp(-(D**2) / (2 *
D0_low**2))
            mask_low[r, c, 1] = np.exp(-(D**2) / (2 *
D0_low**2))

    fshift_low = dft_shift * mask_low
    magnitude_spectrum_low = 20 *
```

```
np.log(cv2.magnitude(fshift_low[:, :, 0],
fshift_low[:, :, 1]) + 1e-6)
    f_ishift_low = np.fft.ifftshift(fshift_low)
    img_back_low = cv2.idft(f_ishift_low,
flags=cv2.DFT_SCALE | cv2.DFT_REAL_OUTPUT)
    img_back_low_norm = cv2.normalize(img_back_low,
None, 0, 255, cv2.NORM_MINMAX).astype(np.uint8)

    # --- Filtro Passa-Alta Gaussiano ---
    print("[DEBUG] Aplicando Filtro Passa-Alta
Gaussiano")
    D0_high = 30
    mask_high = np.zeros((rows, cols, 2), np.float32)
    for r in range(rows):
        for c in range(cols):
            D = np.sqrt((r - crow)**2 + (c - ccol)**2)
            mask_high[r, c, 0] = 1 - np.exp(-(D**2) / (2
* D0_high**2))
            mask_high[r, c, 1] = 1 - np.exp(-(D**2) / (2
* D0_high**2))

    fshift_high = dft_shift * mask_high
    magnitude_spectrum_high = 20 *
np.log(cv2.magnitude(fshift_high[:, :, 0],
fshift_high[:, :, 1]) + 1e-6)
    f_ishift_high = np.fft.ifftshift(fshift_high)
    img_back_high = cv2.idft(f_ishift_high,
flags=cv2.DFT_SCALE | cv2.DFT_REAL_OUTPUT)
    img_back_high_norm = cv2.normalize(img_back_high,
None, 0, 255, cv2.NORM_MINMAX).astype(np.uint8)

    skimage.io.imsave(f"{caminho_imagem_saida_base}
_original_gray.png", img_gray_ubyte)
    skimage.io.imsave(f"{caminho_imagem_saida_base}
_magnitude_spectrum_original.png",
cv2.normalize(magnitude_spectrum_original, None, 0, 255,
```

```
cv2.NORM_MINMAX).astype(np.uint8))
    skimage.io.imsave(f"{caminho_imagem_saida_base}
_passa_baixa_gaussiano.png", img_back_low_norm)
    skimage.io.imsave(f"{caminho_imagem_saida_base}
_magnitude_spectrum_low.png",
cv2.normalize(magnitude_spectrum_low, None, 0, 255,
cv2.NORM_MINMAX).astype(np.uint8))
    skimage.io.imsave(f"{caminho_imagem_saida_base}
_passa_alta_gaussiano.png", img_back_high_norm)
    skimage.io.imsave(f"{caminho_imagem_saida_base}
_magnitude_spectrum_high.png",
cv2.normalize(magnitude_spectrum_high, None, 0, 255,
cv2.NORM_MINMAX).astype(np.uint8))

    fig, axes = plt.subplots(2, 3, figsize=(15, 10))
    ax = axes.ravel()
    ax[0].imshow(img_gray_ubyte, cmap="gray")
    ax[0].set_title("Imagem Original Cinza")
    ax[1].imshow(magnitude_spectrum_original,
cmap="gray")
    ax[1].set_title("Espectro de Magnitude Original")
    ax[2].axis("off")
    ax[3].imshow(img_back_low_norm, cmap="gray")
    ax[3].set_title("Filtro Passa-Baixa Gaussiano
(D0=30)")
    ax[4].imshow(img_back_high_norm, cmap="gray")
    ax[4].set_title("Filtro Passa-Alta Gaussiano
(D0=30)")
    ax[5].axis("off")
    for a_plot in ax:
        if a_plot.images:
            a_plot.set_xticks([])
            a_plot.set_yticks([])
        else:
            a_plot.axis("off")
    fig.tight_layout()
```

```
    path_figura_comparacao =
f"{caminho_imagem_saida_base}
_comparacao_filtros_frequencia.png"
    plt.savefig(path_figura_comparacao)
    plt.close(fig)
    print(f"[DEBUG] Figura de comparação de Filtros de
Frequência salva em {path_figura_comparacao}")
```

3. Resultados e Discussão

O código foi aplicado a duas imagens de exemplo: `camera_original.png` e `text_original.png`. Para cada imagem, foram gerados os espectros de magnitude (antes e depois da filtragem) e as imagens resultantes da aplicação dos filtros passa-baixa e passa-alta Gaussianos.

Exemplo 1: `camera_original.png`

Imagem Original e Espectro:

Imagem Original Camera

Figura 1: Imagem "camera" original em tons de cinza.

Espectro Original Camera

Figura 2: Espectro de magnitude da imagem "camera" original.

Filtro Passa-Baixa Gaussiano (D0=30):

Camera Passa-Baixa

Figura 3: Imagem "camera" após filtro passa-baixa Gaussiano.

Espectro Passa-Baixa Camera

Figura 4: Espectro de magnitude após filtro passa-baixa.

Observa-se que o filtro passa-baixa suavizou a imagem, reduzindo os detalhes finos. O espectro de magnitude correspondente mostra a atenuação das altas frequências (regiões mais afastadas do centro).

Filtro Passa-Alta Gaussiano (D0=30):

Camera Passa-Alta

Figura 5: Imagem "camera" após filtro passa-alta Gaussiano.

Espectro Passa-Alta Camera

Figura 6: Espectro de magnitude após filtro passa-alta.

O filtro passa-alta realçou as bordas e detalhes da imagem. O espectro de magnitude mostra a atenuação das baixas frequências (região central).

Comparação Completa:

Comparação Filtros Camera

Figura 7: Comparação dos resultados para a imagem "camera".

Exemplo 2: `text_original.png`

Imagem Original e Espectro:

Imagem Original Text

Figura 8: Imagem "text" original em tons de cinza.

Espectro Original Text

Figura 9: Espectro de magnitude da imagem "text" original.

Filtro Passa-Baixa Gaussiano (D0=30):

Text Passa-Baixa

Figura 10: Imagem "text" após filtro passa-baixa Gaussiano.

Filtro Passa-Alta Gaussiano (D0=30):

Text Passa-Alta

Figura 11: Imagem "text" após filtro passa-alta Gaussiano.

Comparação Completa:

Figura 12: Comparação dos resultados para a imagem "text".

Para a imagem de texto, o filtro passa-baixa tende a borrar as letras, enquanto o filtro passa-alta realça suas bordas.

4. Conclusões

A filtragem no domínio da frequência demonstrou ser uma ferramenta eficaz para modificar seletivamente as características de uma imagem. Os filtros passa-baixa são úteis para suavização e remoção de ruído, ao custo de alguma perda de detalhes. Os filtros passa-alta são eficazes para realçar bordas e detalhes, o que pode ser útil em tarefas de pré-processamento para detecção de objetos ou análise de textura. A escolha do tipo de filtro e seus parâmetros (como a frequência de corte D0) depende da aplicação específica e das características desejadas na imagem resultante.

5. Referências

- Filtragem no domínio da frequência – Wikipédia, a enciclopédia livre. (Consultado em 2024).
- Gonzalez, R. C., & Woods, R. E. (2018). Digital Image Processing (4th ed.). Pearson.
- Scikit-image documentation. (Consultado em 2024).
- OpenCV documentation. (Consultado em 2024).

Relatório: Segmentação Watershed em Processamento de Imagens com Python

1. Introdução Teórica

A segmentação watershed é um algoritmo clássico e poderoso utilizado no processamento de imagens para separar objetos que estão em contato ou sobrepostos. A intuição por trás do algoritmo vem da geografia: imagine a imagem como uma paisagem topográfica, onde os níveis de intensidade dos pixels representam a altitude. O algoritmo "inunda" essa paisagem a partir de pontos mínimos locais (ou marcadores definidos pelo usuário). As linhas onde as "águas" provenientes de diferentes bacias de inundação se encontram são chamadas de linhas divisórias de águas (watershed lines), que definem as fronteiras dos segmentos.

Este método é particularmente eficaz quando se tem uma boa estimativa dos marcadores iniciais que indicam a presença de cada objeto. Frequentemente, a transformada de distância é utilizada como um passo de pré-processamento: calcula-se a distância de cada pixel do primeiro plano ao pixel de fundo mais próximo. Os máximos locais dessa transformada de distância (que correspondem aos "centros" dos objetos) são então usados como marcadores para o algoritmo watershed.

Existem duas abordagens principais para aplicar o watershed, dependendo de como os marcadores e a "paisagem" são definidos:

1. **Baseada na Transformada de Distância (comum com OpenCV):**
    - A imagem é geralmente binarizada (Otsu, por exemplo).
    - A transformada de distância é calculada sobre a imagem binária (ou sua negação).

    - Os picos da transformada de distância (ou um limiar sobre ela) são usados para definir os marcadores do primeiro plano (`sure_fg`).
    - Uma região de fundo (`sure_bg`) é identificada (e.g., dilatando a imagem de abertura).
    - A região desconhecida (unknown) é a diferença entre `sure_bg` e `sure_fg`.
    - Os componentes conectados dos marcadores `sure_fg` são rotulados.
    - O algoritmo watershed do OpenCV é aplicado à imagem original (colorida ou tons de cinza) usando esses marcadores.

2. **Baseada no Gradiente (comum com Scikit-image):

    - A magnitude do gradiente da imagem (e.g., usando o filtro de Sobel) é calculada. As bordas dos objetos terão altos valores de gradiente, formando as "cristas" da paisagem topográfica.
    - Marcadores são definidos, por exemplo, selecionando regiões com intensidades muito baixas e muito altas, ou usando `peak_local_max` na imagem de distância (negativa).
    - A função watershed do Scikit-image é aplicada à imagem de gradiente usando os marcadores.

2. Implementação em Python

A seguir, o código Python para aplicar a segmentação watershed usando as bibliotecas `OpenCV`, `scikit-image`, `NumPy`, `SciPy` e `Matplotlib`.

```
import cv2
import numpy as np
from skimage.segmentation import watershed
from skimage.filters import sobel
from scipy import ndimage as ndi
from matplotlib import pyplot as plt
import os
import skimage.io
from skimage import img_as_ubyte, color

def
aplicar_segmentacao_watershed(caminho_imagem_entrada,
caminho_imagem_saida_base, usar_distancia=True):
    """
    Aplica a segmentação watershed a uma imagem.

    Args:
        caminho_imagem_entrada (str): Caminho para a
imagem de entrada.
        caminho_imagem_saida_base (str): Caminho base
para salvar as imagens de saída (sem extensão).
        usar_distancia (bool): Se True, usa a
transformada de distância para encontrar marcadores
(OpenCV style).
                               Se False, usa gradiente e
mínimos regionais (Scikit-image style).
    """
    print(f"[DEBUG] Iniciando
aplicar_segmentacao_watershed para:
{caminho_imagem_entrada}")
    diretorio_saida =
os.path.dirname(caminho_imagem_saida_base)
    if not os.path.exists(diretorio_saida):
        os.makedirs(diretorio_saida)
```

```
    img_color_original =
skimage.io.imread(caminho_imagem_entrada)
    if img_color_original is None:
        return

    img_para_watershed_cv = img_color_original.copy()

    if len(img_color_original.shape) == 3:
        img_gray = color.rgb2gray(img_color_original)
    else:
        img_gray = img_color_original

    img_ubyte = img_as_ubyte(img_gray)

    if usar_distancia:
        print("[DEBUG] Usando método de transformada de
distância (OpenCV)")
        _, thresh = cv2.threshold(img_ubyte, 0, 255,
cv2.THRESH_BINARY_INV + cv2.THRESH_OTSU)
        kernel = np.ones((3,3),np.uint8)
        opening_img =
cv2.morphologyEx(thresh,cv2.MORPH_OPEN,kernel,
iterations = 2)
        sure_bg =
cv2.dilate(opening_img,kernel,iterations=3)
        dist_transform =
cv2.distanceTransform(opening_img,cv2.DIST_L2,5)
        _, sure_fg = cv2.threshold(dist_transform,
0.7*dist_transform.max(),255,0)
        sure_fg = np.uint8(sure_fg)
        unknown = cv2.subtract(sure_bg,sure_fg)
        _, markers_cv = cv2.connectedComponents(sure_fg)
        markers_cv = markers_cv + 1
        markers_cv[unknown==255] = 0

        if len(img_para_watershed_cv.shape) == 2:
```

```
            img_para_watershed_cv =
color.gray2rgb(img_as_ubyte(img_para_watershed_cv))

        cv2.watershed(img_para_watershed_cv, markers_cv)
        img_resultado_watershed =
img_para_watershed_cv.copy()
        img_resultado_watershed[markers_cv == -1] =
[255,0,0]

        etapa_intermediaria_titulo = "Transformada de
Distância"
        etapa_intermediaria_img_plot = dist_transform
        marcadores_plot =
cv2.normalize(markers_cv.astype(np.float32), None, 0,
255, cv2.NORM_MINMAX).astype(np.uint8)
        marcadores_titulo = "Marcadores (OpenCV)"
    else:
        print("[DEBUG] Usando método de gradiente
(Scikit-image)")
        gradient = sobel(img_ubyte)
        markers_sk_bool = np.zeros_like(img_ubyte,
dtype=bool)
        markers_sk_bool[img_ubyte <
np.percentile(img_ubyte, 20)] = True
        markers_sk_bool[img_ubyte >
np.percentile(img_ubyte, 80)] = True
        markers_sk_labeled, _ =
ndi.label(markers_sk_bool)
        labels_ws = watershed(gradient,
markers_sk_labeled, mask=img_ubyte > 0)
        img_resultado_watershed =
color.label2rgb(labels_ws, image=img_color_original,
bg_label=0, kind="overlay")
        etapa_intermediaria_titulo = "Magnitude do
Gradiente (Sobel)"
        etapa_intermediaria_img_plot = gradient
```

```
        marcadores_plot =
color.label2rgb(markers_sk_labeled,
image=img_color_original, bg_label=0)
        marcadores_titulo = "Marcadores (Scikit-image)"

    skimage.io.imsave(f"{caminho_imagem_saida_base}
_original.png", img_as_ubyte(img_color_original))
    etapa_intermediaria_img_plot_save =
cv2.normalize(etapa_intermediaria_img_plot, None, 0,
255, cv2.NORM_MINMAX).astype(np.uint8) if
etapa_intermediaria_img_plot.dtype in [np.float32,
np.float64] else
img_as_ubyte(etapa_intermediaria_img_plot)
    skimage.io.imsave(f"{caminho_imagem_saida_base}
_etapa_intermediaria.png",
etapa_intermediaria_img_plot_save)
    skimage.io.imsave(f"{caminho_imagem_saida_base}
_marcadores.png", img_as_ubyte(marcadores_plot))
    skimage.io.imsave(f"{caminho_imagem_saida_base}
_segmentada_watershed.png",
img_as_ubyte(img_resultado_watershed))

    fig, axes = plt.subplots(nrows=1, ncols=4,
figsize=(16, 5), sharex=True, sharey=True)
    ax = axes.ravel()
    ax[0].imshow(img_color_original, cmap=plt.cm.gray if
len(img_color_original.shape)==2 else None)
    ax[0].set_title("Imagem Original")
    ax[1].imshow(etapa_intermediaria_img_plot_save,
cmap="gray")
    ax[1].set_title(etapa_intermediaria_titulo)
    ax[2].imshow(marcadores_plot,
cmap=plt.cm.nipy_spectral if (usar_distancia and
marcadores_plot.ndim == 2) else None)
    ax[2].set_title(marcadores_titulo)
    ax[3].imshow(img_resultado_watershed)
```

```
    ax[3].set_title("Segmentação Watershed")
    for a_plot in ax: a_plot.axis("off")
    fig.tight_layout()
    path_figura_comparacao =
f"{caminho_imagem_saida_base}
_comparacao_watershed_{"dist" if usar_distancia else
"grad"}.png"
    plt.savefig(path_figura_comparacao)
    plt.close(fig)
```

3. Resultados e Discussão

O algoritmo watershed foi aplicado a duas imagens de exemplo: `coins_original.png` e `chelsea_original.png`. Para a imagem `chelsea`, ambos os métodos (baseado em distância e baseado em gradiente) foram testados.

Exemplo 1: `coins_original.png` (Método da Transformada de Distância)

Comparação Visual:

Comparação Watershed Coins

Figura 1: Resultados da segmentação watershed para a imagem "coins" usando o método da transformada de distância. Da esquerda para a direita: Imagem Original, Transformada de Distância, Marcadores (OpenCV) e Segmentação Watershed.

Para a imagem `coins_original.png`, o método baseado na transformada de distância conseguiu separar a maioria das moedas sobrepostas. A transformada de distância (segunda imagem) mostra claramente os centros das moedas como picos de intensidade. Os marcadores derivados desses picos (terceira imagem) guiam o algoritmo watershed para produzir a segmentação final (quarta imagem), onde as bordas entre as moedas são delineadas em vermelho.

Imagens Individuais:

- Original:

  Original Coins

- Etapa Intermediária (Distância):

  Distância Coins

- Marcadores:

  Marcadores Coins

- Segmentada:

  Segmentada Coins

Exemplo 2: `chelsea_original.png`

Método da Transformada de Distância

Comparação Visual:

Comparação Watershed Chelsea Dist

Figura 2: Resultados da segmentação watershed para a imagem "chelsea" usando o método da transformada de distância.

Na imagem `chelsea_original.png`, que é mais complexa, o método da transformada de distância ainda tenta segmentar regiões. No entanto, a natureza da imagem (texturas, falta de objetos claramente definidos como as moedas) torna a segmentação menos intuitiva. As regiões segmentadas podem corresponder a áreas de cor ou textura similar.

Imagens Individuais (Método Distância):

- Original:

  Original Chelsea

- Etapa Intermediária (Distância):

  Distância Chelsea

- Marcadores:

  Marcadores Chelsea

- Segmentada:

  Segmentada Chelsea

Método do Gradiente (Scikit-image)

Comparação Visual:

Comparação Watershed Chelsea Grad

Figura 3: Resultados da segmentação watershed para a imagem "chelsea" usando o método do gradiente.

O método baseado no gradiente, aplicado à imagem `chelsea_original.png`, utiliza a magnitude do gradiente (segunda imagem) como a topografia a ser inundada. Os marcadores (terceira imagem), definidos por limiares de intensidade, guiam a segmentação. O resultado (quarta imagem) mostra uma segmentação baseada nas bordas e variações de intensidade, que pode ser útil para identificar regiões com diferentes texturas ou características.

Imagens Individuais (Método Gradiente):

- Original:

  Original Chelsea Grad

- Etapa Intermediária (Gradiente):

  Gradiente Chelsea

- Marcadores:

  Marcadores Chelsea Grad

- Segmentada:

  Segmentada Chelsea Grad

4. Conclusões

A segmentação watershed é uma técnica versátil para separar objetos em imagens. Sua eficácia depende crucialmente da qualidade dos marcadores e da "paisagem" topográfica utilizada. O método baseado na transformada de distância é muito eficaz para objetos bem definidos com interiores relativamente homogêneos, como as moedas. O método baseado no gradiente pode ser mais adequado para imagens complexas com texturas, onde as bordas e variações de intensidade são mais informativas para a segmentação. A escolha da abordagem e o ajuste dos parâmetros (como a definição dos marcadores) são essenciais para obter resultados de segmentação significativos e precisos.

5. Referências

- Watershed segmentation - scikit-image documentation. (Consultado em 2024).
- OpenCV-Python Tutorials - Image Segmentation with Watershed Algorithm. (Consultado em 2024).
- Gonzalez, R. C., & Woods, R. E. (2018). Digital Image Processing (4th ed.). Pearson.

Relatório: Segmentação por Crescimento de Regiões em Python

1. Introdução Teórica

A segmentação por crescimento de regiões é uma técnica de processamento de imagens que agrupa pixels ou sub-regiões em regiões maiores com base em critérios de similaridade pré-definidos. O processo começa com um conjunto de "pixels semente". A partir dessas sementes, as regiões crescem iterativamente, anexando pixels vizinhos que satisfazem uma condição de homogeneidade, como similaridade de intensidade, cor, textura ou outra característica.

Os principais componentes do algoritmo de crescimento de regiões são:

1. **Seleção de Sementes:** A escolha dos pixels iniciais (sementes) é crucial. As sementes podem ser selecionadas manualmente pelo usuário, automaticamente com base em alguma heurística (e.g., pixels com intensidade extrema) ou de forma interativa.
2. **Critério de Similaridade (ou Homogeneidade):** Define a condição que um pixel vizinho deve satisfazer para ser adicionado à região em crescimento. Um critério comum é a diferença de intensidade entre o pixel candidato e o valor da semente original (ou a média da região já crescida) ser menor que um limiar especificado.
3. **Conectividade:** Define quais pixels são considerados vizinhos (e.g., 4-conectividade ou 8-conectividade).
4. **Condição de Parada:** O processo de crescimento para uma região específica para quando não há mais pixels vizinhos que satisfaçam o critério de similaridade.

Vantagens: * Geralmente produz regiões conectadas e com bordas bem definidas. * Os conceitos são relativamente simples de entender e implementar. * Pode ser adaptado para diferentes tipos de imagens e critérios de similaridade.

Limitações: * A seleção de sementes pode ser desafiadora e impacta significativamente o resultado. * Sensível ao ruído, que pode levar a um crescimento incorreto ou à formação de "buracos" nas regiões. * O critério de similaridade (especialmente o limiar) precisa ser cuidadosamente escolhido; um limiar inadequado pode levar a subsegmentação ou supersegmentação. * Pode ter dificuldade em segmentar regiões com gradientes suaves de intensidade ou texturas complexas se o critério de similaridade for muito simples.

2. Implementação em Python

A seguir, o código Python utilizado para implementar o algoritmo de crescimento de regiões. Utilizamos as bibliotecas `OpenCV`, `NumPy`, `scikit-image` e `Matplotlib`.

```
import cv2
import numpy as np
from matplotlib import pyplot as plt
import os
import skimage.io
from skimage import img_as_ubyte, color

class Stack(): # Classe auxiliar para a pilha usada no
algoritmo
    def __init__(self):
        self.item = []
        self.obj = [] # Não utilizado na implementação
atual, mas mantido da referência

    def push(self, item, obj):
        self.item.append(item)
        self.obj.append(obj)

    def pop(self):
        if not self.isEmpty():
            return self.item.pop(), self.obj.pop()

    def isEmpty(self):
        return len(self.item) == 0

def aplicar_crescimento_regioes(caminho_imagem_entrada,
caminho_imagem_saida_base, semente_coords,
limiar_similaridade=10):
    """
    Aplica o algoritmo de crescimento de regiões a uma
imagem a partir de uma semente.
    """
    diretorio_saida =
os.path.dirname(caminho_imagem_saida_base)
    if not os.path.exists(diretorio_saida):
```

```
        os.makedirs(diretorio_saida)

    img_original_color =
skimage.io.imread(caminho_imagem_entrada)
    if img_original_color is None: return

    if len(img_original_color.shape) == 3:
        img_gray = color.rgb2gray(img_original_color)
    else:
        img_gray = img_original_color

    img_gray_ubyte = img_as_ubyte(img_gray)
    altura, largura = img_gray_ubyte.shape
    img_segmentada = np.zeros_like(img_gray_ubyte)
    visitados = np.zeros_like(img_gray_ubyte,
dtype=bool)
    pilha = Stack()

    semente_linha, semente_coluna = semente_coords
    if not (0 <= semente_linha < altura and 0 <=
semente_coluna < largura):
        print(f"[ERRO] Coordenadas da semente
{semente_coords} fora dos limites.")
        return

    valor_semente = img_gray_ubyte[semente_linha,
semente_coluna]
    pilha.push((semente_linha, semente_coluna),
valor_semente)
    visitados[semente_linha, semente_coluna] = True
    img_segmentada[semente_linha, semente_coluna] = 255

    while not pilha.isEmpty():
        (x, y), _ = pilha.pop()
        for dx in [-1, 0, 1]:
            for dy in [-1, 0, 1]:
```

```
                if dx == 0 and dy == 0: continue
                nx, ny = x + dx, y + dy
                if 0 <= nx < altura and 0 <= ny <
largura and not visitados[nx, ny]:
                    visitados[nx, ny] = True
                    valor_vizinho = img_gray_ubyte[nx,
ny]
                    if abs(int(valor_vizinho) -
int(valor_semente)) <= limiar_similaridade:
                        img_segmentada[nx, ny] = 255
                        pilha.push((nx, ny),
valor_vizinho)

    img_original_com_semente = img_original_color.copy()
    if len(img_original_com_semente.shape) == 2:
        img_original_com_semente =
color.gray2rgb(img_as_ubyte(img_original_com_semente))
    cv2.circle(img_original_com_semente,
(semente_coluna, semente_linha), radius=5,
color=(255,0,0), thickness=-1)

    skimage.io.imsave(f"{caminho_imagem_saida_base}
_original.png", img_as_ubyte(img_original_color))
    skimage.io.imsave(f"{caminho_imagem_saida_base}
_original_com_semente.png",
img_as_ubyte(img_original_com_semente))
    skimage.io.imsave(f"{caminho_imagem_saida_base}
_segmentada.png", img_segmentada)

    fig, axes = plt.subplots(1, 3, figsize=(15, 5))
    ax = axes.ravel()
    ax[0].imshow(img_original_color, cmap=\'gray\' if
len(img_original_color.shape)==2 else None)
    ax[0].set_title("Imagem Original")
    ax[0].axis("off")
    ax[1].imshow(img_original_com_semente)
```

```
    ax[1].set_title(f"Semente em ({semente_linha},
{semente_coluna})")
    ax[1].axis("off")
    ax[2].imshow(img_segmentada, cmap="gray")
    ax[2].set_title(f"Região Crescida (Limiar:
{limiar_similaridade})")
    ax[2].axis("off")
    fig.tight_layout()
    plt.savefig(f"{caminho_imagem_saida_base}
_comparacao_crescimento.png")
    plt.close(fig)
```

3. Resultados e Discussão

O algoritmo de crescimento de regiões foi aplicado a três imagens de exemplo: `astronaut_original.png`, `horse_original.png` e `text_original.png`. Para cada imagem, foram testadas duas sementes diferentes com um limiar de similaridade de intensidade de 20.

Exemplo 1: `astronaut_original.png`

Semente 1 (Rosto): (200, 200), Limiar: 20

Comparação Astronauta Semente 1

Figura 1: Crescimento de região na imagem "astronauta" com semente no rosto.

A semente no rosto do astronauta resultou no crescimento de uma região que cobre parte do rosto e do capacete, onde as intensidades dos pixels são similares ao pixel semente dentro do limiar estabelecido.

Semente 2 (Capacete): (100, 100), Limiar: 20

Comparação Astronauta Semente 2

Figura 2: Crescimento de região na imagem "astronauta" com semente no capacete.

Com a semente no capacete, a região crescida se concentra principalmente na área clara do capacete.

Exemplo 2: `horse_original.png`

Semente 1 (Corpo do Cavalo): (150, 200), Limiar: 20

Comparação Cavalo Semente 1

Figura 3: Crescimento de região na imagem "cavalo" com semente no corpo do cavalo.

A semente no corpo escuro do cavalo resultou na segmentação de grande parte do cavalo, pois essa área possui intensidade relativamente uniforme e distinta do fundo mais claro.

Semente 2 (Fundo): (50, 50), Limiar: 20

Comparação Cavalo Semente 2

Figura 4: Crescimento de região na imagem "cavalo" com semente no fundo.

Uma semente no fundo claro da imagem do cavalo segmentou uma grande porção do fundo, excluindo o cavalo mais escuro.

Exemplo 3: `text_original.png`

Semente 1 (Letra): (80, 100), Limiar: 20

Comparação Texto Semente 1

Figura 5: Crescimento de região na imagem "texto" com semente em uma letra.

Com a semente em uma letra, o algoritmo conseguiu segmentar partes das letras que possuem intensidade similar à semente, diferenciando-as do fundo.

Semente 2 (Fundo): (20, 20), Limiar: 20

Comparação Texto Semente 2

Figura 6: Crescimento de região na imagem "texto" com semente no fundo.

Uma semente no fundo da imagem de texto resultou na segmentação da maior parte do fundo, excluindo as letras mais escuras.

4. Conclusões

O algoritmo de crescimento de regiões demonstrou ser capaz de segmentar áreas de interesse com base na similaridade de intensidade a partir de um pixel semente. Os resultados são altamente dependentes da escolha da semente e do valor do limiar de similaridade. Sementes bem posicionadas em regiões homogêneas e um limiar adequado podem levar a segmentações precisas. No entanto, a seleção manual de sementes pode ser tediosa para múltiplas regiões, e a determinação de um limiar ótimo pode exigir experimentação. Para imagens mais complexas ou com variações graduais de intensidade, critérios de similaridade mais sofisticados ou estratégias de seleção de sementes adaptativas podem ser necessários.

5. Referências

- Gohil, M. (2023). Region Growing: An Inclusive Overview from Concept to Code in Image Segmentation. Medium. (Consultado em 2024).
- Gonzalez, R. C., & Woods, R. E. (2018). Digital Image Processing (4th ed.). Pearson.
- Scikit-image documentation. (Consultado em 2024).