Retoque fotográfico mediante reconstrucciones geométricas heurísticas

Ariel Nowik, Joaquin Mestanza, Rocio Parra, Martina Máspero, Marcelo Regueira
22.05 - Análisis de Señales y Sistemas Digitales - Grupo 1

ITBA: Instituto tecnológico de Buenos Aires
Ciudad de Buenos Aires, Argentina

***Resumen*—En este trabajo se estudiaron diversos métodos de retoque de imagenes para eliminar elementos no deseados presentes en diversas fuentes. Finalmente se procedió a realizar una implementación en función de las tecnicas analizadas seguida de un análisis de sus ventajas y desventajas.**

I. Introducción

El problema elemental a resolver consiste en la eliminación de un objeto no deseado en una imagen. Naturalmente no es posible .ªdivinar"lo que se encuentre por detrás, ya que requiere información adicional, la cual en principio no es accesible, solo se dispone de la imagen. Por lo tanto la idea es, de algun modo asimilar la zona de la imagen a reemplazar con el resto de la misma. En lo que continua de este trabajo describiremos con un mayor detalle diversos métodos para llevar a cabo este proceso.

I-A. Descripción general del algoritmo

El algoritmo trabajo en varias etapas. En cada iteración del algoritmo se ejecuta cada paso una vez.

- Cálculo del borde
- Selección del punto del borde a eliminar
- Selección de rectángulo correspondiente para el reemplazo

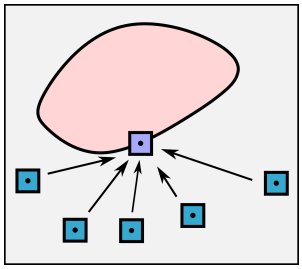

Figura 1. Descripción gráfica de una iteración del algoritmo

A medida que avanza el algoritmo se va disminuyendo el tamaño de la región a eliminar ya que se la asimila a los pixeles que estan por fuera. Cuando dicha región ya no existe se termina de correr el programa.

I-B. Descripción de la zona a eliminar

Es muy importante que el usuario que necesite retocar la imagen ïndique"que región sea necesaria eliminar. En nuestro algoritmo decidimos que se ingrese como entrada una imagen de dos colores "blancoz "negro"donde la zona negra sea aquella que se necesite borrar. En el desarrollo de el método muchas veces es necesario trabajr con el borde de esta región; se estudió entonces como utilizar la librería opencv que es muy conocida en el ambito de procesamiento de imagenes, la cual ofreció métodos optimizados para la detección de bordes de tanto regiones conexas, como regiones "multiplemente conexas"

I-C. Cálculo de gradientes de imagen

Para el funcionamiento del algoritmo fue necesario el cálculo de los gradientes de la imagen. Para ello se cálculo primero la imagen en una escala de grises y luego se aplicó el operador Sobel, el cual combinó tanto derivada como suavización gaussiana.

I-D. Determinación de prioridades

En cada iteración se necesito calcular de todos los pixeles del borde aquellos con una mayor prioridad para ser asimilados. En los papers citados se muestran fundamentos por los cuales es más apropiado poderar por un lado el gradiente de la imagen y por el otro la confianza La formula que determina

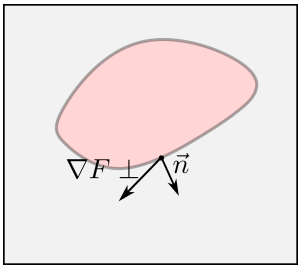

Figura 2. Gradiente y normal

la prioridad esta dada por

$$P_i = C_i \nabla F_i \perp \vec{n}_i \tag{1}$$

Donde $\nabla F_i \perp$ es la perpendicular al gradiente de la imagen en el pixel i. Es el punto donde ocurre el cambio más suave en la intensidad del color de la imagen. El algoritmo realiza el producto escalar entre $\nabla F_i \perp \vec{n}_i$ lo cual consigue la componente normal del la perpendicular al gradiente, que nos

indica que tan suave cambia la intensidad de la imagen en la dirección normal al borde de la región

I-E. Determinación del rectángulo a reemplazar

Para mejorar la eficiencia del algoritmo se decidió elegir rectángulos eficientes al rededor del pixel para asimilar el pixel de la imagen a la textura. Se comparó entre todos los candidatos utilizando la norma 2 comparando el cuadrado de la región a reemplazar y el candidato. A continuación se muestra la fórmula utilizada para el cálculo de de norma 2

$$\sum (R_{1i} - R_{2i})^2 + (G_{1i} - G_{2i})^2 + (B_{1i} - B_{2i})^2 \quad (2)$$

Se utilizó una distribución normal para elegir al azar los cantidatos

I-F. Parámetros, ventajas y desventajas

Finalmente con el algoritmo implementado se probó variar diversos parárametros, entre ellos:

- Tamaño de los cuadrados
- Region para buscar cuadrados
- Cantidad de muestros

Variando dichos parámetros en generar el ”tradeoff”fue eficiencia vs una mejor calidad en el procesamiento de la imagen

II. Resultados

A continuación se mostrará una selección con los resultados del algoritmo

II-A. Ejemplo 1

Se procedió a probar el algoritmo con una moneda en un fondo de textura prácticamente unifirme

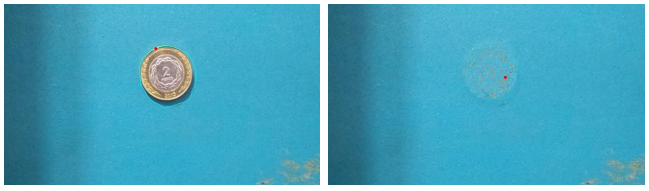

Figura 3. Resultados - ejemplo 1

Si bien se notaron algunas impurezas dado que fue la primera versión del algoritmo, los resultados fueron satisfactorios

II-B. Ejemplo 2

Se probó el algoritmo esta vez con un fondo gris y unas piedritas de decoración

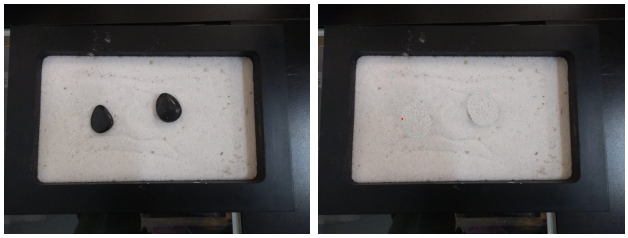

Figura 4. Resultados - ejemplo 2

Se obeservó que si bien se logró cubrir correctamente las areas a eliminar, la transcisión de colores no fue totalmente continua. Esto induce a que en un futuro se desarrolle algun procedimiento adicional que suavice dicha transición

II-C. Ejemplo 3

El tercer ejemplo fue corrido luego de convertir la distribución de la elección de cuadrados de lineal a gaussiana

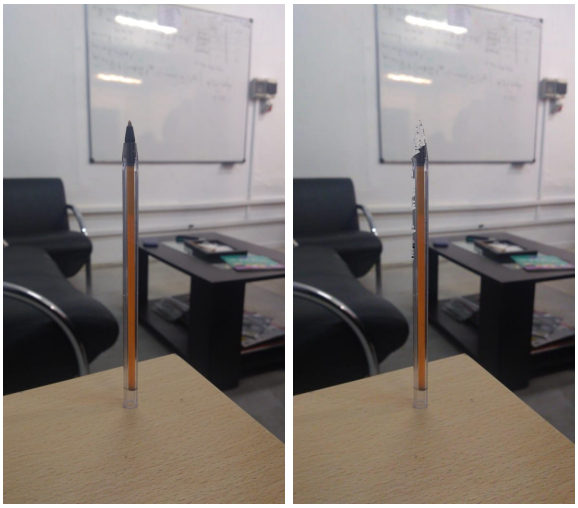

Figura 5. Resultados - ejemplo 3

Se corrieron solo 280 iteraciónes del algoritmo pero se observaron resultados satisfactorios, el algoritmo tendió a absorber la lapicera.

III. Código

```
import cv2
import numpy as np
from matplotlib import pyplot as plt
import matplotlib.image as mpimg
import imutils
from numpy import random
import scipy.misc
from PIL import Image
from numpy import sqrt
import time

# imagen a procesar
img = cv2.imread('output3/imagen.jpeg', 3)
# mascara con area a remover.
# la zona negra (0,0,0) es la que se remueve,
    la blanca se deja como esta (255,255,255)
mask = cv2.imread("output3/mask.jpeg")
# imagen pasada a escala de grises se guarda
    en esta variable
grey_scale = np.zeros(img.shape, dtype=np.
    uint8) #uint8


#lado de los cuadrados que utilizaremos para
    rellenar la imagen
square_size = 5
square = []
```

```
# guardamos en un arreglo las coordenadas que
    describen al cuadrado
for i in range(square_size):
   for j in range(square_size):
      square.append(
         [
            i - square_size//2,
            j - square_size//2
          ]
      )

# tamanio del cuadrado de busqueda para el
    parche que reemplaza la posicion a
    rellenear
search_square_size = 1000

# cuantas veces buscamos al azar por un parche
search_times = 100


def procesar(imagen, mask):
   iteraciones = 1000

   lower = np.array([0, 0, 0])
   upper = np.array([15, 15, 15])
   # re-mapeamos a 0-1 la mascara. 1 es para
       la zona retocada, 0 para la que no
   shapeMask = cv2.inRange(mask, lower, upper)

   c = shapeMask[:, :] == 0 # maxima confianza
        en la zona que no se retoca

   for iteracion in range(iteraciones):
      # primero detectamos el borde de la
          mascara

      lower = np.array([0, 0, 0])
      upper = np.array([15, 15, 15])
      shapeMask = cv2.inRange(mask, lower,
          upper)

      ## conseguimos un arreglo con todos los
          contornos

      cnts = cv2.findContours(shapeMask.copy()
          , cv2.RETR_EXTERNAL, cv2.
          CHAIN_APPROX_NONE)
      cnts = imutils.grab_contours(cnts)
      # cada contorno cerrado forma un arreglo

      # luego tenemos que calcular la funcion
          de costos
      best_benefit = 0
      best_benefit_point = None

      # conseguimos la escala de grises
      grey_scale = cv2.cvtColor(img, cv2.
          COLOR_BGR2GRAY)

      # conseguimos el gradiente en x e y de
          la escala de grises, la funcion
          sobel no solo hace gradiente
      # sino que suaviza
      sobel_x = cv2.Sobel(grey_scale, cv2.
          CV_64F, 1, 0, ksize=5)
      sobel_y = cv2.Sobel(grey_scale, cv2.
          CV_64F, 0, 1, ksize=5)
      sobel_x, sobel_y = -sobel_y,sobel_x

      # por cada contorno cerrado
      for contorno in range(len(cnts)):

         ## necesitamos generar las normales
             de cada punto del contorno
         border_normal = []

         n = len(cnts[contorno])

         for i in range(n):
            #print(cnts[0][i])

            dx = cnts[contorno][i][0][0] -
                cnts[contorno][(i-1) % n
                ][0][0]
            dy = cnts[contorno][i][0][1] -
                cnts[contorno][(i-1) % n
                ][0][1]

            border_normal.append((dy, -dx))
            # esta formula nos da la normal.
                no le damos importancia a la
                orientacion

         index = 0

         for border_point in cnts[contorno]:
            x, y = border_point[0]

            # consigo la confianza del punto
                del contorno actual
            confidence = 0

            for dx, dy in square:
               if shapeMask[y + dy, x + dx] ==
                    0: # si fuera de la region
                    a retocar
                  confidence += c[y + dy, x +
                      dx]

            confidence /= len(square)

            # consigo la componente normal del
                gradiente
            nx, ny = border_normal[index]

            # consigo el gradiente mas grande
                de la region

            max_grad = 0
            max_grad_value = 0, 0

            for dx, dy in square:
               # solo sumamos si esta fuera de
                    la zona a retocar
               if shapeMask[y + dy, x + dx] ==
                    0:

                  dx = np.sum(sobel_x[y][x])/3
                       # promediamos los tres
                      componentes del
                      gradiente
                  dy = np.sum(sobel_y[y][x])/3

                  p = dx ** 2 + dy ** 2
```

```
            if p > max_grad: # buscamos
                el mayor gradiente en
                norma
               max_grad = p
               max_grad_value = dx, dy

      # producto escalar del gradiente
          con la normal acorde a la
          formula

      d = max_grad_value[0] * nx +
          max_grad_value[1] * ny

      # el beneficio es la confianza por
           el factor d

      benefit = abs(d * confidence)

      # buscamos maximizar el beneficio
      if benefit > best_benefit:
         best_benefit = benefit
         best_benefit_point = x, y

if not best_benefit_point:
   print("No hay mas borde. Fin")
   break

# ahora vamos a calcular el parche que
    minimize la distancia

px, py = best_benefit_point

best_patch = px, py # default
patch_distance = np.Infinity

for i in range(search_times):
   # x = random.randint(px -
       search_square_size//2, px +
       search_square_size//2)
   # y = random.randint(py -
       search_square_size//2, py +
       search_square_size//2)
   x = int(random.normal(px,
       search_square_size//2**5,1))
   y = int(random.normal(py,
       search_square_size//2**5,1))

   if shapeMask[y, x] == 255:
      continue # no es de interes ya que
           esta en la region blanca

   #patch = imagen[y - square_size//2:y
       + square_size//2, x - square_size
       //2:x + square_size//2]
   #original = imagen[py - square_size
       //2:py + square_size//2, px -
       square_size//2:px + square_size
       //2]
   #total_sum = np.array([0])
   total_sum = 0
   # decidi usar fors porque se me
       estaban copiando los arreglos y
       en definitiva como son
   # todas operaciones elemento a
       elemento no son optimizables

   for yi in range(-square_size//2,
       square_size//2):
      for xi in range(-square_size//2,
          square_size//2):
         sum = 0
         for cmp in range(3):
            patch = int(imagen[y + yi][x
                + xi][cmp])
            original = int(imagen[py +
                yi][px + xi][cmp])

            sum += (patch - original)**2
         sum = sqrt(sum)
         #np.append(total_sum,sum**2)
         total_sum += sum**2
   #total_sum = total_sum.sum()
   #print(np.square(patch-original))

   if total_sum < patch_distance:
      patch_distance = sum
      best_patch = x, y

bx, by = best_patch # best_patch_x,
    best_patch_y

imagen[py - square_size//2: py +
    square_size//2, px - square_size//2:
     px + square_size//2] = \
   imagen[by - square_size//2: by +
       square_size//2, bx - square_size
       //2: bx + square_size//2]

## copiamos la confianza del parche
    elegido a la la confianza del lugar
    donde copiamos el parche
c[py - square_size // 2: py +
    square_size // 2, px - square_size
    // 2: px + square_size // 2] = \
   c[by - square_size // 2: by +
       square_size // 2, bx -
       square_size // 2: bx +
       square_size // 2]*0.99

## marcamos la zona reemplazada como
    blanca
mask[py - square_size // 2: py +
    square_size // 2, px - square_size
    // 2: px + square_size // 2] = \
   [255, 255, 255]

im2 = np.copy(imagen)

if iteracion % 20 == 0:
   print("Iteracion ", iteracion)
   #for cnt in cnts:
   # cv2.drawContours(im2, [np.array(cnt
       )], 0, (255, 255, 0), 1)

   #cv2.drawContours(im2, [np.array([
       best_benefit_point])], 0, (0, 0,
       255), 5)
   im = Image.fromarray(cv2.cvtColor(im2
       , cv2.COLOR_BGR2RGB))
   im.save("output3/imagen" + str(
       iteracion) + ".jpeg")

#plt.imshow(cv2.cvtColor(im2, cv2.
    COLOR_BGR2RGB))
```

```
      #plt.savefig("output/imagen" + str(
          iteracion) + ".jpeg", dpi=1000)
      #scipy.misc.toimage(im2).save("output/
          imagen" + str(iteracion) + ".jpeg")


      #plt.imshow(cv2.cvtColor(mask, cv2.
          COLOR_BGR2RGB))
      #plt.savefig("output_mask/image/n" + str
          (iteracion) + ".jpeg", dpi=1000)
      #plt.show()



   #plt.imshow(cv2.cvtColor(imagen, cv2.
       COLOR_BGR2RGB))
   #plt.show()


#img_intensity = cv2.cvtColor(img, cv2.
    COLOR_BGR2GRAY)

#cv2.mixChannels(img, img_intensity)

#print(img_intensity)
#
start_time = time.time()
procesar(img, mask)
end_time = time.time()
print("se calculo en:" ,(end_time-start_time)
    /60," minutos")
#
# plt.imshow(img2, cmap="gray")

#sobelx = cv2.Sobel(img_intensity, cv2.CV_64F,
     1, 0, ksize=9)
#sobely = cv2.Sobel(img_intensity, cv2.CV_64F,
     0, 1, ksize=9)

#print(img)

#plt.imshow(cv2.cvtColor(img, cv2.
    COLOR_BGR2RGB))

#plt.imshow(sobelx, cmap="gray")



#plt.show()
```

REFERENCIAS

[1] A. Criminisi, P. Perez, and K. Toyama, “Region filling and object ´ removal by exemplar-based image inpainting,” IEEE T. Image Process., vol. 13, no. 9, pp. 1200–1212, Sep. 2004.

[2] Pierre Buyssens, Maxime Daisy, David Tschumperlé, Olivier Lézoray. Exemplar-based Inpainting: Technical Review and new Heuristics for better Geometric Reconstructions. IEEE Transactions on Image Processing, Institute of Electrical and Electronics Engineers, 2015, 24 (6), pp.1809 - 1824. ff10.1109/TIP.2015.2411437ff. ffhal-01147620f