Computer Vision Homework 2

B03902042 宋子維

Description

In this homework, we are asked to implement the programs to satisfy the following things:

1. Binarize lena.bmp with the threshold 128 (0-127, 128-255)

2. The histogram of lena.bmp

3. Connected components of the binary image of lena.bmp

Programming

I use python to implement part 1. There are totally three python programs, binarize.py, histogram.py and connect.py, where I use **pillow** to process basic image I/O. In the three programs, there are some similar functions:

1. `PIL.Image.open(img)`: load the image `img` and return a pillow **Image** object.

2. `pix = Image.load()`: return the image access object of **Image** object to `pix`, which offers us to use `pix[x, y]` to access the pixel value at position `(x, y)`.

3. `PIL.Image.new(mode, size)`: create a new image with given mode and size, and return a piilow **Image** object.

4. `Image.width, Image.height`: width and height of **Image** object.

1. Binarize lena.bmp with the threshold 128 (0-127, 128-255)

Run `python3.5 binarize.py $IMG_IN $IMG_OUT`, and the program will generate a binary image of `IMG_IN` with threshold 128, called `IMG_OUT` with **BMP** format.

Figure 1-1: Binarize lena.bmp with the threshold 128

The algorithm I use is to iterate all pixels and check whether the value of pixel is larger or equal to 128. If it is, set that value to 1, otherwise, 0.

```
for x in range(0, width):
    for y in range(0, height):
        pix_b [x, y] = 1 if pix_ori[x, y] >= 128 else 0
```

Figure 1-2: binarize.py

where `pix_b` (`pix_ori`) is the image access object of **IMG_OUT**(**IMG_IN**).

2. The histogram of lena.bmp

Run `python3.5 histogram.py $IMG_IN $IMG_OUT`, and the program will generate the histogram of `IMG_IN`, called `IMG_OUT`. To draw the histogram, I use auxiliary python library, **matplotlib**, to assist me in drawing the bar graph.

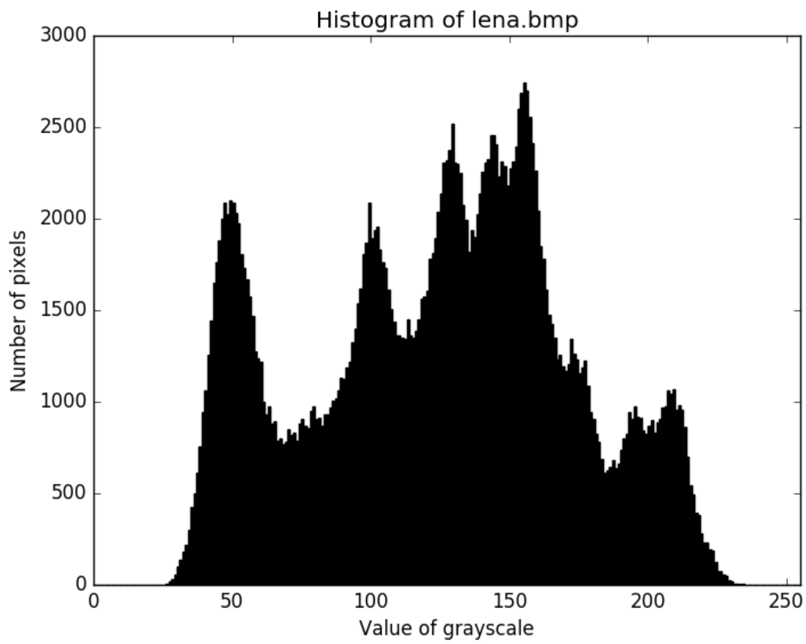

Figure 1-1: Binarize lena.bmp with the threshold 128

The algorithm I use is to create a list `sum`, where the index `i` is the value of gray scale and `sum[i]` is the number of pixels with pixel value `i`, and iterate all pixels to calculate it.

```
N = 256

sum = [0] * N

for x in range(0, width):
    for y in range(0, height):
        sum[pix_ori[x, y]] += 1
```

Figure 2-1: histogram.py

where `pix_ori` is the image access object of the **IMG_IN**.

As for plotting, `matplotlib.pyplot.bar(left, height, color)` can help us. `left` is sequence of scalars, which are the x coordinates , `height` is also the sequence of scalars, which are the heights of the bars, and `color` is the color of the bars.

```
ind = range(N)
plt.bar(ind, sum, color='black')
```

Figure 2-1: histogram.py

where `ind` is the list from 0 to 255, `sum` is the list with the statistic data mentioned above.

3. Connected components of the binary image of lena.bmp

Run `python3.5 connect.py $IMG_IN $IMG_OUT`, and the program will generate the image **IMG_OUT** containing the bounding box of each connected component, called `IMG_OUT` with **BMP** format. **I use 4-connected**.

Figure 3-1: Connected components of the binary image of lena.bmp

The algorithm I use is the classical algorithm in the Chapter 2 slide, but I merge the equivalence classes with disjoint set rather than DFS.

First, iterate all pixels (x, y) of `img`. If the value is 0 (background), assign 0 to the label of pixel (x, y) and continue; If both left and top label of pixel (x, y) are 0 (background), assign a new label to it and make a new disjoint set; If both left and top label of pixel (x, y) are nonzero, the smaller one propagates and merge those two disjoint set. If one of left and top label of pixel (x, y) is zero and the other is not, the nonzero one propagates. Since my class `DisJoint` maintains the size of each set, it is necessary to increase the size by 1 when the old label propagates and `Disjoint.increase()` can do this.

The following figures is the first step of the algorithm and the class `DisJoint`.

```python
def Label(img):
    pix = img.load()
    width, height = img.size

    Eq = DisJoint()

    labels = {}
    stamp = 1

    for y in range(height):
        for x in range(width):
            if pix[x, y] == 0:
                labels[x, y] = 0 # background
                continue

            left = labels[x-1, y] if x-1 >= 0 else 0
            top = labels[x, y-1] if y-1 >= 0 else 0

            if left == 0 and top == 0:
                labels[x, y] = stamp
                Eq.make(stamp)
                stamp += 1
            elif left != 0 and top != 0:
                labels[x, y] = left if left < top else top
                if left != top:
                    Eq.union(left, top)
                Eq.increase(labels[x, y])
            else:
                labels[x, y] = left if left != 0 else top
                Eq.increase(labels[x, y])

    return Eq, labels
```

Figure 3-2: Label

```python
class DisJoint:
    def __init__(self):
        self.p = {}
        self.size = {}

    def make(self, x):
        self.p[x] = x
        self.size[x] = 1

    def find(self, x):
        if x == self.p[x]:
            return x
        else:
            self.p[x] = self.find(self.p[x])
            return self.p[x]

    def union(self, x, y):
        X, Y = self.find(self.p[x]), self.find(self.p[y])
        if X == Y:
            return

        if self.size[X] >= self.size[Y]:
            self.size[X] += self.size[Y]
            self.p[Y] = X
        else:
            self.size[Y] += self.size[X]
            self.p[X] = Y

    def getsize(self, x):
        return self.size[self.find(x)]

    def increase(self, x):
        self.size[self.find(x)] += 1
```

Figure 3-3: DisJoint

Second, solve the equivalence classes. In first step, the labels in same equivalence class have been merged. Hence, all we need to do is to perform a translation on it. Iterate all pixels (x, y) and find which disjoint set it belongs to. Since `DisJoint` keeps track of the size of each set, it is easy to omit the connected component that has less than 500 pixels. If the size of the class that pixel (x, y) belongs to is larger than or equal to 500, add it into the component it belongs to. `Component` is also a class, which maintains the topmost, bottommost, leftmost and rightmost point of pixels and the sum of x, y coordinates and the area of connected component in it. Thus, we can easily get the bounding box and centroid of each connected component.

The following figures is the second step of the algorithm and the class `Component`.

```python
def Last(img, Eq, labels):
    width, height = img.size

    out = Image.new('RGB', img.size)
    pix = out.load()

    Group = {}
    for y in range(height):
        for x in range(width):
            if labels[x, y] != 0:
                n = Eq.find(labels[x, y])
                if Eq.getsize(n) >= 500:
                    if n not in Group:
                        Group[n] = Component()
                    Group[n].add(x, y)
                pix[x, y] = (255, 255, 255)
            else:
                pix[x, y] = (0, 0, 0)

    draw = ImageDraw.Draw(out)
    colorP, colorR = (255, 0, 0), (220, 116, 246)
    for i in Group:
        drawPlus(draw, Group[i].getcentroid(), colorP)
        draw.rectangle(Group[i].getbound(), fill=None, outline=colorR)

    return out
```

Figure 3-4: Last

```python
class Component:
    def __init__(self):
        self.up = self.left = float('inf')
        self.low = self.right = -float('inf')
        self.X = 0
        self.Y = 0
        self.A = 0

    def add(self, x, y):
        self.up = min(self.up, y)
        self.low = max(self.low, y)
        self.left = min(self.left, x)
        self.right = max(self.right, x)
        self.X += x
        self.Y += y
        self.A += 1

    def getbound(self):
        return [self.left, self.up, self.right, self.low]

    def getcentroid(self):
        return ((self.X/self.A), (self.Y/self.A))
```

Figure 3-5: Component