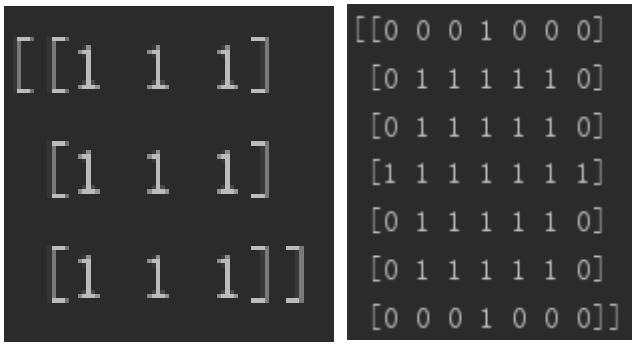
谭树杰 11849060

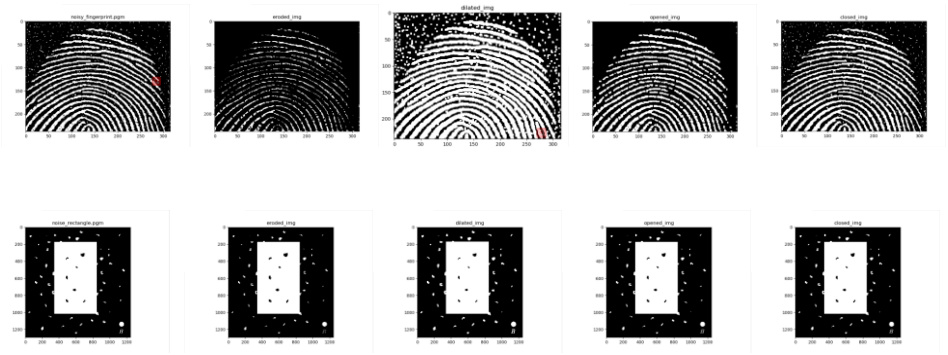
Lab8 Report

1. Define two structural elements, perform binary dilation, erosion, opening, and closing operations on noisy_fingerprint.pgm and noise_rectangle.pgm

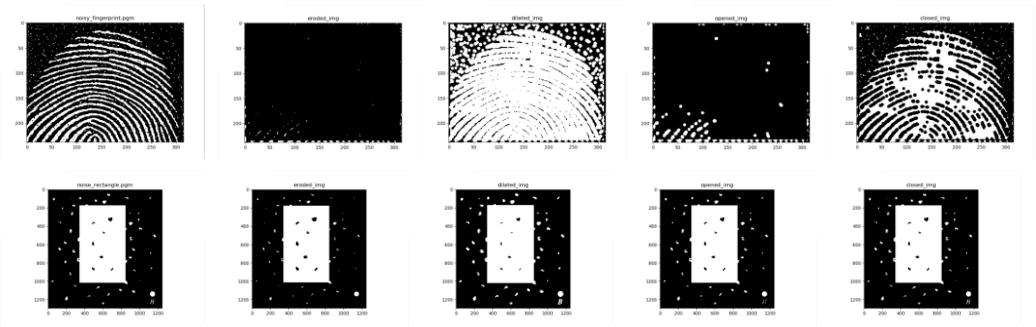
(1) I the structural elements are shown below. The rectangular structuring element is in the left and the disk structuring element is in the right.



(2) noisy_fingerprint.pgm and noise_rectangle.pgm and images after performing erosion, dilation, opening and closing using rectangular structuring element respectively are shown below:



(3) noisy_fingerprint.pgm and noise_rectangle.pgm and images after performing erosion, dilation, opening and closing using disk structuring element respectively are shown below:



Analysis:

We can see that the background noise was completely eliminated after erosion by rectangular structuring element, but the fingerprint is eliminated by using disk structuring element for that the size the structuring element is too larger.

However, the disk SE removes more noise in noise_rectangle.pgm, therefore we should choose appropriate size of SE.

For the same reason, the results of dilation, opening and closing by rectangular SE are better than by disk SE.

The dilation operation could make the fingerprint more connected.

We can observe that the net effect of opening was to eliminate virtually all noise components in both the background and the fingerprint itself.

The closing makes the inward pointing corners more rounded, whereas the outward pointing corners were unchanged.

The implementation code is shown below:

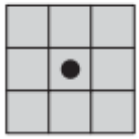
```
import numpy as np
import cv2
from matplotlib import pyplot as plt
from copy import deepcopy
from collections import Counter



def getStructuringElement(errosion_type, ksize, anchor=(-1, -1)):
    if errosion_type == 'rect':
        if anchor == (-1, -1):
            se = np.ones(ksize, dtype=np.uint8)
    elif errosion_type == 'disk':
        se = np.zeros((ksize * 2 + 1, ksize * 2 + 1), dtype=np.uint8)
        for r in range(ksize * 2 + 1):
            for c in range(ksize * 2 + 1):
                dist_square = (r - ksize) ** 2 + (c - ksize) ** 2
                if dist_square <= ksize * ksize:
                    se[r][c] = 1
    return se
```

```
def getset(img):
    img_set = set()
    for r in range(img.shape[0]):
        for c in range(img.shape[1]):
            if img[r, c] != 0:
                img_set.add((r, c))
    return img_set


def erosion(src, kernel):
    c_x = int(kernel.shape[0] / 2)
    c_y = int(kernel.shape[1] / 2)
    dst = np.copy(src)
    kernel_set = getset(kernel)
    for r in range(c_x, src.shape[0]-c_x):
        for c in range(c_y, src.shape[1]-c_y):
            subimg = src[r-c_x:r+c_x+1, c-c_y:c+c_y+1]
            subimg_set = getset(subimg)
            if kernel_set.issubset(subimg_set):
                dst[r, c] = 255
            else:
                dst[r, c] = 0
    return dst


def dilation(src, kernel):
    c_x = int(kernel.shape[0] / 2)
    c_y = int(kernel.shape[1] / 2)
    dst = np.copy(src)
    kernel_set = getset(kernel)
    for r in range(c_x, src.shape[0]-c_x):
        for c in range(c_y, src.shape[1]-c_y):
            subimg = src[r-c_x:r+c_x+1, c-c_y:c+c_y+1]
            subimg_set = getset(subimg)
            if (kernel_set & subimg_set) != set():
                dst[r, c] = 255
            else:
                dst[r, c] = 0
    return dst



def opening(src, kernel):
    tmp = erosion(src, kernel)
    return dilation(tmp, kernel)
```

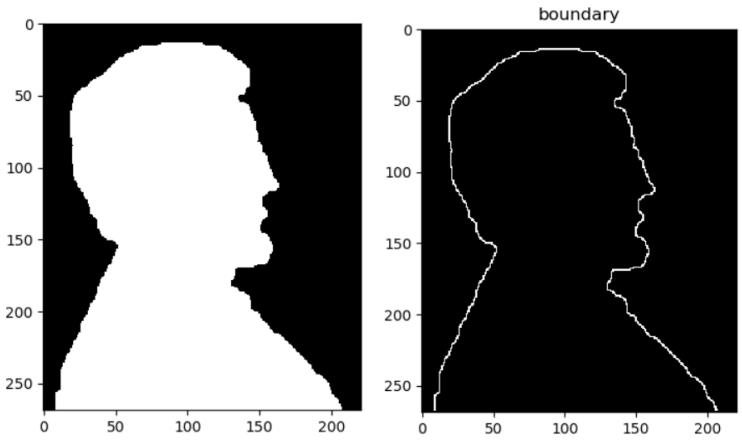
```
def closing(src, kernel):
    tmp = dilation(src, kernel)
    return erosion(tmp, kernel)


# 1.Define two structural elements, perform binary dilation, erosion, opening, and
#   closing operations on noisy_fingerprint.pgm and noise_rectangle.pgm
path_in = './in/noisy_fingerprint.pgm'
# path_in = './in/noise_rectangle.pgm'
img = cv2.imread(path_in, 0)
print(img)
plt.figure(1)
# plt.title("noise_rectangle.pgm")
plt.title("noisy_fingerprint.pgm")
plt.imshow(img, cmap='gray')
# se = np.ones(3 * 3 + 1)
# se = getStructuringElement('rect', (3, 3))
se = getStructuringElement('disk', 3)
print(se)
eroded_img = erosion(img, se)
plt.figure(2)
plt.title('eroded_img')
plt.imshow(eroded_img, cmap='gray')
# cv2.imshow("eroded_img", eroded_img)

dilated_img = dilation(img, se)
plt.figure(3)
plt.title("dilated_img")
plt.imshow(dilated_img, cmap='gray')

opened_img = opening(img, se)
plt.figure(4)
plt.title("opened_img")
plt.imshow(opened_img, cmap='gray')

closed_img = closing(img, se)
plt.figure(5)
plt.title("closed_img")
plt.imshow(closed_img, cmap='gray')
```

2. Extract the boundaries of licoln.pgm and U.pgm

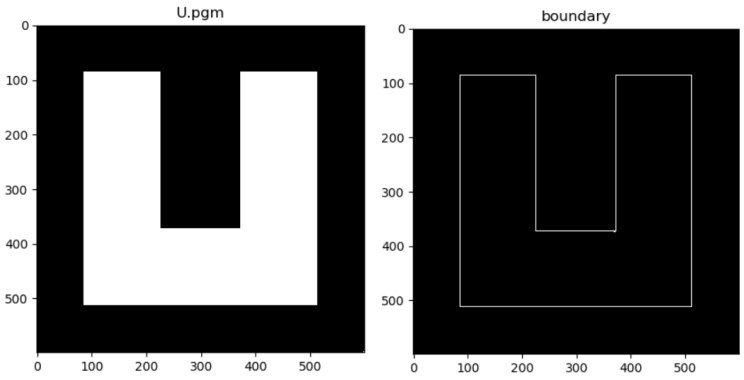
(1) The structural elements are shown below.



(2) The licoln.pgm and its boundary image are shown below



The U.pgm and its boundary image are shown below:



Analysis:

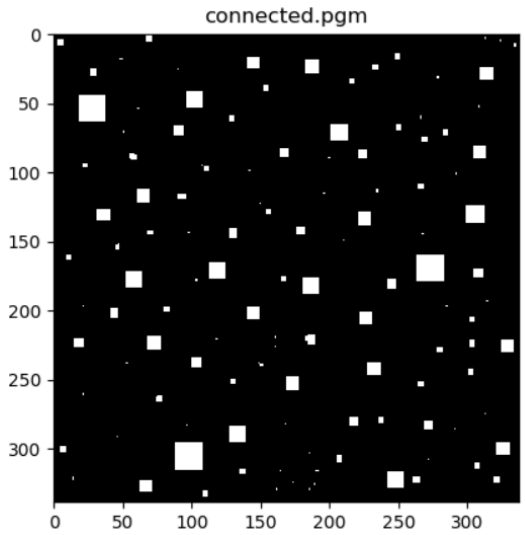
We can see the boundaries of the licoln.pgm and U.pgm clearly. This is achieved by erosion first, then performing the set difference between the original image and its erosion. This is useful when we are interested in boundary feature rather the flatten part of the image.

The implementation code is shown below:

```
# 2. Extract the boundaries of licoln.pgm and U.pgm
def boundary(src, kernel):
    eroded_img = erosion(src, kernel)
    eroded_set = getset(eroded_img)
    dst = np.copy(src)
    for i in eroded_set:
        dst[i] = 0
    return dst


# path_in = './in/licoln.pgm'
path_in = './in/U.pgm'
img = cv2.imread(path_in, 0)
print(img)
plt.figure(1)
# plt.title("licoln.pgm")
plt.title("U.pgm")
plt.imshow(img, cmap='gray')
se = getStructuringElement('rect', (3, 3))
boundary_img = boundary(img, se)
plt.figure(2)
plt.title("boundary")
plt.imshow(boundary_img, cmap='gray')
```

3. Extract the connected components from connected.pgm

(1) The connected.pgm is shown below.



(2) The result is shown below. The left column are the leftmost and topmost pixels's

coordinates of each connected component, the right column are the corresponding number of pixels of the connected component.

left_top,	#pixels
(66, 248)	20
(42, 96)	144
(317, 242)	144
(296, 88)	400
(19, 182)	100
(227, 278)	16
(284, 127)	144
(66, 201)	144
(172, 52)	144
(67, 87)	49
(198, 140)	81
(45, 18)	380
(160, 263)	400
(252, 264)	16
(176, 165)	16
(218, 184)	50
(25, 309)	90
(170, 304)	49
(116, 90)	24
(280, 268)	49
(248, 169)	90
(321, 319)	16
(125, 299)	156
(202, 222)	81
(222, 324)	81
(299, 5)	16
(82, 304)	81
(278, 214)	42
(219, 68)	100
(331, 108)	16
(221, 15)	42
(83, 164)	42
(17, 140)	81
(278, 235)	16
(152, 47)	13

left_top,	#pixels
(84, 221)	42
(87, 55)	23
(177, 180)	144
(323, 62)	81
(127, 31)	90
(198, 80)	16
(166, 113)	144
(33, 214)	16
(160, 9)	16
(143, 68)	12
(238, 227)	90
(140, 176)	36
(243, 300)	16
(5, 3)	16
(141, 127)	42
(129, 221)	90
(305, 205)	24
(296, 321)	90
(262, 75)	24
(113, 60)	100
(321, 260)	24
(199, 41)	42
(96, 109)	16
(311, 305)	16
(235, 100)	49
(222, 301)	20
(178, 242)	42
(144, 97)	2
(2, 67)	16
(109, 264)	16
(113, 234)	2
(315, 135)	16
(38, 152)	16
(23, 231)	16
(94, 21)	12

left_top,	#pixels
(303, 185)	1
(26, 27)	20
(60, 127)	16
(197, 21)	1
(308, 271)	1
(127, 154)	16
(250, 128)	16
(75, 267)	16
(70, 282)	16
(283, 96)	1
(238, 148)	7
(197, 284)	2
(60, 266)	2
(205, 301)	16
(18, 48)	2
(15, 247)	16
(7, 333)	6
(219, 160)	2
(3, 312)	2
(316, 190)	2
(26, 90)	1
(52, 308)	2
(221, 117)	2
(282, 168)	1
(321, 14)	2
(149, 210)	1
(5, 323)	1
(95, 107)	1
(99, 141)	2
(193, 313)	2
(145, 267)	1
(316, 164)	1
(325, 189)	2
(286, 290)	1
(329, 185)	2

left_top,	#pixels
(324, 173)	1
(260, 21)	2
(115, 195)	2
(54, 60)	2
(238, 52)	2
(291, 46)	1
(31, 278)	2
(329, 161)	2
(123, 149)	1
(178, 103)	2
(71, 50)	1
(101, 291)	2
(275, 312)	1
(90, 199)	2
(226, 160)	1

Analysis:

We can see connected components of different size and position in the picture above. We can use this information to analysis the image and perform more enhancement based on that.

The implementation code is shown below:

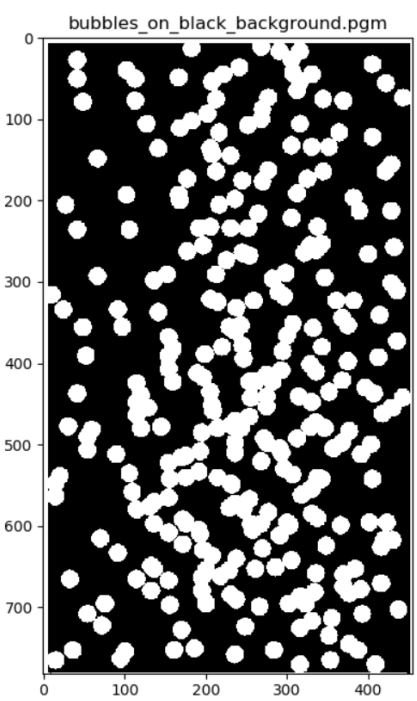
```
# 3. Extract the connected components from connected.pgm
# return component set determined by point, 8-connected
def get_component(img_set, point):
    result = set()
    queue = {point}
    tmp_set = deepcopy(img_set)
    while queue != set():
        p = queue.pop()
        result.add(p)
        for r in range(-1, 2):
            for c in range(-1, 2):
                a_point = (p[0]+r, p[1]+c)
                if a_point in tmp_set:
                    queue.add(a_point)
                    tmp_set = tmp_set.difference({a_point})
    return result


# return list of connectivity components(represented by set of points (row, col))
def connectivity(src):
    src_set = getset(src)
    result = []
    while src_set != set():
        i = src_set.pop()
        current_component = get_component(src_set, i)  ##
        src_set = src_set.difference(current_component)
        result.append(current_component)
    return result



path_in = './in/connected.pgm'
img = cv2.imread(path_in, 0)
print(img)
plt.figure(1)
plt.title("connected.pgm")
plt.imshow(img, cmap='gray')

con = connectivity(img)
```

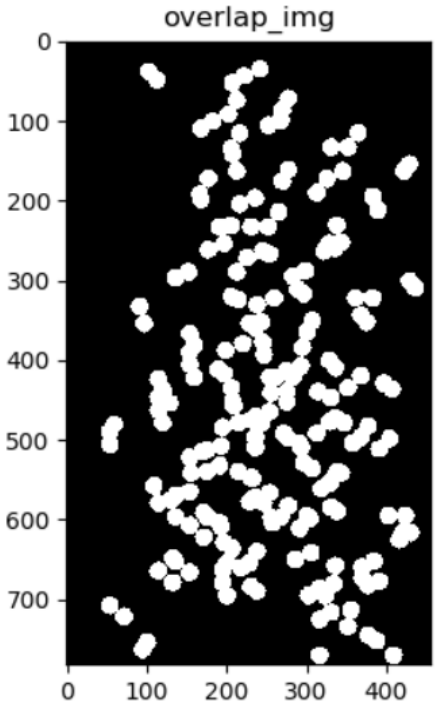
```
test_img = np.zeros_like(img)
for c in con:
    for p in c:
        test_img[p] = 255
plt.figure(2)
plt.title("reconstructed img")
plt.imshow(test_img, cmap='gray')


i = 0
for c in con:
    num = len(c)
    left_top = c.pop()
    for p in c:
        if p[0] < left_top[0]:
            left_top = p
        elif (p[0] == left_top[0]) & (p[1] < left_top[1]):
            left_top = p
    if i % 35 == 0:
        print("left_top,  #pixels")
    i += 1
    print(str(left_top) + "    " + str(num))
```

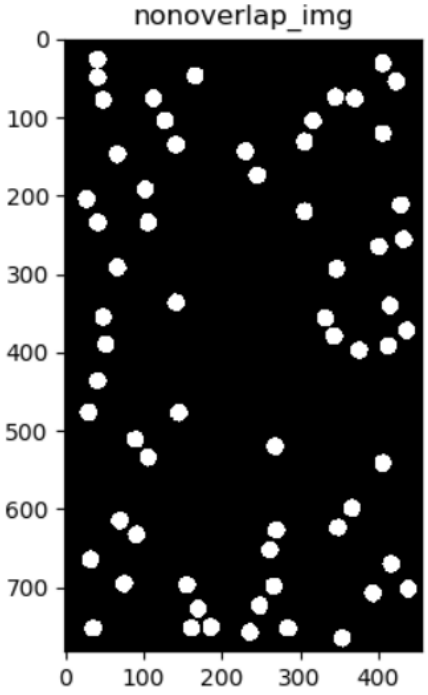
4. Problem 9.36, write a program to separate the three required sets from bubbles_on_black_background.pgm

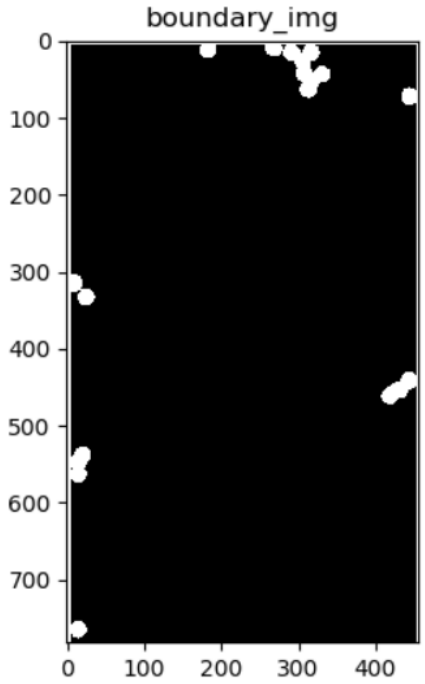
Results:

The bubbles_on_black_background.pgm and overlapping particles, nonoverlapping particles and particles that have merged with the boundary of the image are shown below.









Analysis:

After process by connectivity components analysis, we can see overlapping particles, nonoverlapping particles and particles that have merged with the boundary of the image clearly. Then we can analysis three different kind of particles separately. Of course this is helpful when we analysis the cells in microscopy.

The implementation code is shown below:

```
# 4. Problem 9.36, write a program to separate the three required sets from
# bubbles_on_black_background.pgm
path_in = './in/bubbles_on_black_background.pgm'
img = cv2.imread(path_in, 0)
print(img)
plt.figure(1)
plt.title("bubbles_on_black_background.pgm")
plt.imshow(img, cmap='gray')

con = connectivity(img)


def set_to_img(src_set, img):
    test_img = np.zeros_like(img)
    for p in src_set:
        test_img[p] = 255
    return test_img


i = 0
num = []
for c in con:
    num.append(len(c))
pixels_count = Counter(num)
most_pixels = pixels_count.most_common(1)[0][0]
print(pixels_count.most_common(10))


overlap_set = set()
nonoverlap_set = set()
boundary_set = set()
for c in con:
    if len(c) == max(num):
        boundary_set = boundary_set | c
    elif len(c) >  most_pixels + 5:
        overlap_set  = overlap_set | c
    else:
        nonoverlap_set = nonoverlap_set | c
```

```
overlap_img = set_to_img(overlap_set, img)
plt.figure(2)
plt.title("overlap_img")
plt.imshow(overlap_img, cmap='gray')



nonoverlap_img = set_to_img(nonoverlap_set, img)
plt.figure(3)
plt.title("nonoverlap_img")
plt.imshow(nonoverlap_img, cmap='gray')

boundary_img = set_to_img(boundary_set, img)
plt.figure(4)
plt.title("boundary_img")
plt.imshow(boundary_img, cmap='gray')
```