图像分割实践报告

蒋颜丞，自动化(电气)1903，3190102563

1 实验内容和要求

自选一张内容简单的灰度图像，用一种方法实现图像前景、背景分割，并提取前景区域边缘；同时，给出边缘的链码表示。要求给出灰度图像、分割后二值化图像、边缘提取结果图像，以及边缘的链码表示，上述结果可以是运行结果截屏图像；同时，提交核心代码。

本次实验将以以下六图为例，进行前后景分割。

Figure1

Figure2

Figure3

Figure4

Figure5

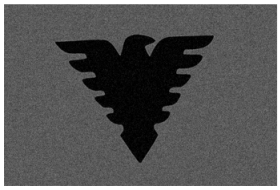


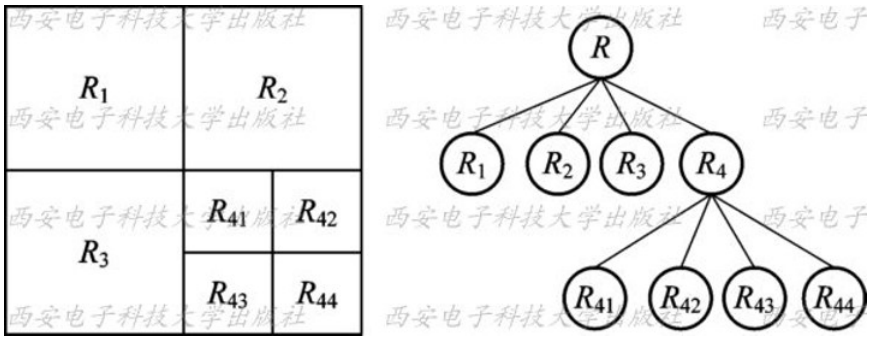
Figure6

其中，Figure 1为ZJU的logo图片，Figure 2为ZJUI的logo图片；Figure 3和Figure 4在Figure 1和Figure 2的基础上让背景的灰度变得不均匀；Figure 5和Figure 6在Figure 1和Figure 2的基础上添加了高斯噪声。

2 实验原理

2.1 区域分裂与合并

区域分裂与合并的核心思想是将图像分成若干个子区域，对于任意一个子区域，如果不满足某种一致性准则(一般用灰度均值和方差来度量)，则将其继续分裂成若干个子区域，否则该子区域不再分裂。如果相邻的两个子区域满足某个相似性准则，则合并为一个区域。直到没有可以分裂和合并的子区域为止。通常基于下图所示的四叉树来表示区域分裂与合并，每次将不满足一致性准则的区域分裂为四个大小相等且互不重叠的子区域。



2.1.1 分裂

在本例中，分裂时的一致性准则为： 如果某个子区域的灰度均方差大于一定值，则将其分裂为4个子区域，否则不分裂。我使用了一个四叉树来实现分裂过程，以初始节点（整张图）为起点，不断进行递归分裂，直到没有一个叶子节点符合分裂准则。每个节点都使用ImgNode对象表示，其中存储有该节点的父节点、子节点、上下左右邻节点等信息，这些信息在分裂的过程中维护，方便在后续的合并过程中使用。

2.1.2 合并

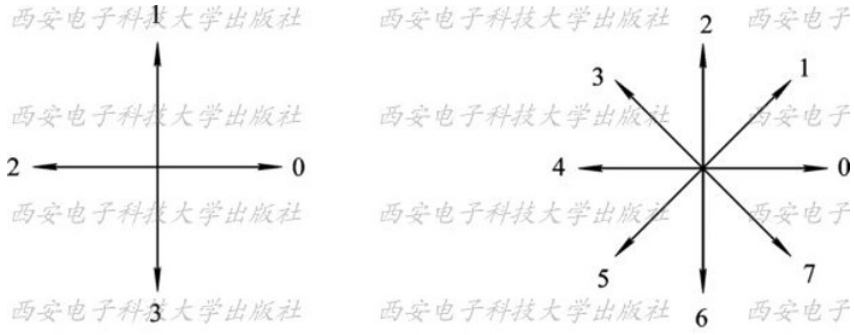
在本例中，合并时的相似性准则为：若相邻两个子区域的灰度均值之差不大于一定值，则合并为一个区域。我采用递归的方法进行合并：先找到一个最小叶子节点（最小区域），并由这个点出发，递归扫描近邻点，直到搜索过所有的联通点，并将这些节点合并为一个区域。

2.2 轮廓提取

二值图像的轮廓提取算法非常简单， 就是掏空目标区域的内部点。 在本例中，图像的目标像素为黑色， 背景像素为白色， 则如果图像中某个像素为黑色， 且它的8个邻点都是黑色时，表明该点是内部点， 否则为边界点。 将判断出的内部像素置为背景色， 对所有内部像素执行该操作便可完成图像轮廓的提取。

2.3 轮廓跟踪

轮廓跟踪就是顺序找出边界点，不仅可以跟踪出边界，还可以同时记录边界信息，如生成边界链码，为图像分析做准备。轮廓跟踪可以基于4方向码和8方向码分别跟踪出4连通的轮廓和8连通的轮廓。在实际应用中，常用基于8方向码的轮廓跟踪。



(a) 4连通方向码　　(b) 8连通方向码

STEP1 首先从上到下、从左到右顺序扫描图像，寻找第一个目标点作为边界跟踪的起始点，记为A。A点一定是最左角上的边界点，其相邻的边界点只可能出现在它的左下、 下、右下、右四个邻点中。定义一个搜索方向变量dir，用于记录从当前边界点搜索下一个相邻边界点时所用的搜索方向码。dir初始化为：dir=5，即从方向5开始搜索与A相邻的下一个边界点。如果当前搜索方向dir上的邻点不是边界点，则依次使搜索方向逆时针旋转一个方向，更新dir，直到搜索到一

个边界点为止。如果所有方向都未找到相邻的边界点，则该点是一个孤立点。dir的更新用公式可表示为：dir=(dir+1) mod 8。

STEP2 把上一次搜索到的边界点作为当前边界点，在其3×3邻域内按逆时针方向搜索新的边界点，它的起始搜索方向设定如下：对基于8方向的轮廓跟踪，若上次搜索到边界点的方向dir为奇数，则使dir=(dir + 6) mod 8，即将上次的搜索方向顺时针旋转两个方向；若dir为偶数，则使dir=(dir + 7) mod 8，即将上次的搜索方向顺时针旋转一个方向。如果起始搜索方向没有找到边界点，则依次使搜索方向逆时针旋转一个方向，更新dir，直到搜索到一个新的边界点为止。

STEP3 如果搜索到的边界点就是第一个边界点A，则停止搜索，结束跟踪，否则重复步骤2继续搜索。由依次搜索到的边界点系列就构成了被跟踪的边界，并可以使用链码表示。

当图像中有多个轮廓时，可以使用循环遍历的方法，即当一个轮廓跟踪完毕后，继续寻找下一个轮廓的起始点，直到所有边缘都已被纳入轮廓中。

2.4 轮廓滤波

在得到轮廓的链码表示后，可以对轮廓进行滤波。这里采用的方法是均值滤波，滤波窗口的大小为11，其主要方法是使用轮廓上当前点附近的11个轮廓点的坐标均值来代替当前点坐标。

3 源代码

imgnode.py（图像节点类，用于构成四叉树）

```python
import cv2
import numpy as np

class ImgNode():
    '''
    图像节点类，用于构成四叉树
    '''
    Visited_List = []  # 已访问的节点，在merge方法中维护

    def __init__(self, img, father_node, h0, h1, w0, w1):
        self.img = img  # 原始图像
        self.father_node = father_node  # 父节点
        self.sub_node1 = None
        self.sub_node2 = None
        self.sub_node3 = None
        self.sub_node4 = None  # 子节点
        self.left_node = []  # 左节点
        self.right_node = []  # 右节点
        self.up_node = []  # 上节点
        self.down_node = []  # 下节点
        self.h0 = h0
        self.h1 = h1  # 当前节点在img上的h范围
        self.w0 = w0
        self.w1 = w1  # 当前节点在img上的w范围
        self.isleaf = True  # 用于存储当前节点是否为叶子节点
```

```
26
27      def split_judge(self):
28          '''
29          判断当前节点是否需要分裂
30          输入：当前节点
31          输出：是否需要分裂
32          '''
33          var_value = self.cal_var()  # 计算当前节点的灰度方差
34          if var_value > 3.6:  # 判断标准
35              return True  # 需要分裂
36          else:
37              return False  # 不需要分裂
38
39      def cal_var(self):
40          '''
41          计算当前节点的灰度方差
42          输入：当前节点
43          输出：方差
44          '''
45          img = self.img
46          h0 = self.h0
47          h1 = self.h1
48          w0 = self.w0
49          w1 = self.w1
50          area = img[h0: h1, w0: w1]
51          var_value = np.var(area)  # 计算方差
52          return var_value
53
54      def cal_mean(self):
55          '''
56          计算当前节点的灰度均值
57          输入：当前节点
58          输出：均值
59          '''
60          img = self.img
61          h0 = self.h0
62          h1 = self.h1
63          w0 = self.w0
64          w1 = self.w1
65          area = img[h0: h1, w0: w1]
66          mean_value = np.mean(area)  # 计算均值
67          return mean_value
68
69      def draw_region_img(self, region_img):
70          '''
71          绘制当前节点的区域图像
72          输入：当前节点，区域图像
73          输出：区域图像
74          '''
```

```
        h0 = self.h0
        h1 = self.h1
        w0 = self.w0
        w1 = self.w1
        # for h in range(h0-1, h1):
        #     for w in range(w0-1, w1): # 遍历当前节点范围内的所有像素
        #         region_img[h][w] = 255  # 填充为白色
        region_img[h0:h1, w0:w1] = 255
        return region_img

    def node_split(self):
        '''
        对当前节点进行分裂
        子节点说明:
         1 | 2
        ———————
         3 | 4
        '''
        self.isleaf = False  # 当前节点不再是叶子节点
        sub_node1 = ImgNode(self.img, self, self.h0, int(
            (self.h0+self.h1)/2), self.w0, int((self.w0+self.w1)/2))  # 创建子节点1
        sub_node2 = ImgNode(self.img, self, self.h0, int(
            (self.h0+self.h1)/2), int((self.w0+self.w1)/2), self.w1)  # 创建子节点2
        sub_node3 = ImgNode(self.img, self, int(
            (self.h0+self.h1)/2), self.h1, self.w0, int((self.w0+self.w1)/2))  # 创建子节点3
        sub_node4 = ImgNode(self.img, self, int(
            (self.h0+self.h1)/2), self.h1, int((self.w0+self.w1)/2), self.w1)  # 创建子节点4

        # 链接各个子节点的上下左右节点
        sub_node1.left_node.extend(self.left_node)
        sub_node1.right_node.append(sub_node2)
        sub_node1.up_node.extend(self.up_node)
        sub_node1.down_node.append(sub_node3)

        sub_node2.left_node.append(sub_node1)
        sub_node2.right_node.extend(self.right_node)
        sub_node2.up_node.extend(self.up_node)
        sub_node2.down_node.append(sub_node4)

        sub_node3.left_node.extend(self.left_node)
        sub_node3.right_node.append(sub_node4)
        sub_node3.up_node.append(sub_node1)
        sub_node3.down_node.extend(self.down_node)

        sub_node4.left_node.append(sub_node3)
```

```
        sub_node4.right_node.extend(self.right_node)
        sub_node4.up_node.append(sub_node2)
        sub_node4.down_node.extend(self.down_node)

        # 链接当前节点的左节点的右节点
        for ln in self.left_node:
            if self in ln.right_node:
                ln.right_node.remove(self)
                if ln.h0 < sub_node1.h1 and ln.h1 > sub_node1.h0:
                    ln.right_node.append(sub_node1)
                if ln.h0 < sub_node3.h1 and ln.h1 > sub_node3.h0:
                    ln.right_node.append(sub_node3)

        # 链接当前节点的上节点的下节点
        for un in self.up_node:
            if self in un.down_node:
                un.down_node.remove(self)
                if un.w0 < sub_node1.w1 and un.w1 > sub_node1.w0:
                    un.down_node.append(sub_node1)
                if un.w0 < sub_node2.w1 and un.w1 > sub_node2.w0:
                    un.down_node.append(sub_node2)

        # 链接当前节点的下节点的上节点
        for dn in self.down_node:
            if self in dn.up_node:
                dn.up_node.remove(self)
                if dn.w0 < sub_node3.w1 and dn.w1 > sub_node1.w0:
                    dn.up_node.append(sub_node3)
                if dn.w0 < sub_node4.w1 and dn.w1 > sub_node4.w0:
                    dn.up_node.append(sub_node4)

        # 链接当前节点的右节点的左节点
        for rn in self.right_node:
            if self in rn.left_node:
                rn.left_node.remove(self)
                if rn.h0 < sub_node2.h1 and rn.h1 > sub_node2.h0:
                    rn.left_node.append(sub_node2)
                if rn.h0 < sub_node4.h1 and rn.h1 > sub_node4.h0:
                    rn.left_node.append(sub_node4)

        # 链接当前节点与各个子节点
        self.sub_node1 = sub_node1
        self.sub_node2 = sub_node2
        self.sub_node3 = sub_node3
        self.sub_node4 = sub_node4

    def is_leaf_father(self):
        '''
        判断当前节点是否是叶子节点的父节点
```

```
        '''
        if self.isleaf is False and self.sub_node1.isleaf and
self.sub_node2.isleaf and self.sub_node3.isleaf and self.sub_node4.isleaf:
            return True
        else:
            return False

    def find_leaf_father(self):
        '''
        寻找一个叶子节点的父节点
        输入：一个起始节点（不能是叶子节点）
        输出：叶子节点的父节点
        '''
        if self.is_leaf_father():
            return self
        elif self.isleaf:
            return None
        else:  # 从子节点出发递归调用
            res1 = self.sub_node1.find_leaf_father()
            if res1 is not None:
                return res1
            res2 = self.sub_node2.find_leaf_father()
            if res2 is not None:
                return res2
            res3 = self.sub_node3.find_leaf_father()
            if res3 is not None:
                return res3
            res4 = self.sub_node4.find_leaf_father()
            if res4 is not None:
                return res4

    def draw_node(self, img):
        '''
        在img中绘制当前节点
        输入：当前节点，欲绘制的图像
        输出：绘制后的图像
        '''
        point_color = (255, 255, 255)  # 颜色
        thickness = 1  # 粗细
        lineType = 4  # 线型
        cv2.rectangle(img, (self.w0, self.h0), (self.w1, self.h1),
                      point_color, thickness, lineType)  # 绘制矩形
        return img

    def split(self, draw_img, min_area=(1, 1)):
        '''
        区域分裂
        输入：欲绘制的图像，最小区域大小
        输出：绘制好的图像
```

```
        '''
        if self.split_judge() and self.h1-self.h0 >= 2*min_area[0] and self.w1-self.w0 >= 2*min_area[1]:  # 符合分裂条件
            self.node_split()  # 分裂当前节点
            # 递归分裂当前节点的子节点
            draw_img = self.sub_node1.split(draw_img, min_area)
            draw_img = self.sub_node2.split(draw_img, min_area)
            draw_img = self.sub_node3.split(draw_img, min_area)
            draw_img = self.sub_node4.split(draw_img, min_area)

        if self.h1-self.h0 >= min_area[0] and self.w1-self.w0 >= min_area[1]:
            draw_img = self.draw_node(draw_img)  # 绘制当前节点

        return draw_img

    def merge(self, region_img, threshold=5.0):
        '''
        区域合并
        输入：当前节点，欲绘制的区域二值图像，相似性判断阈值（若两个区域的灰度均值之差小于threshold，则认为这两块区域可以合并）
        输出：绘制的区域二值图像
        '''
        if self in ImgNode.Visited_List:
            return region_img  # 如果当前节点已被访问过，则不再访问
        else:
            ImgNode.Visited_List.append(self)  # 将当前节点加入访问表

        region_img = self.draw_region_img(region_img)  # 绘制区域二值图像
        self_mean = self.cal_mean()  # 计算当前节点灰度均值

        for rn in self.right_node:  # 遍历所有右侧节点
            # 右侧节点与当前节点类似，且未被访问过
            if abs(self_mean - rn.cal_mean()) <= threshold and rn not in ImgNode.Visited_List:
                # print('right', self_mean - rn.cal_mean())
                region_img = rn.merge(region_img, 5)  # 对右侧节点进行递归合并

        for dn in self.down_node:  # 遍历所有下方节点
            # 下方节点与当前节点类似，且未被访问过
            if abs(self_mean - dn.cal_mean()) <= threshold and dn not in ImgNode.Visited_List:
                # print('down', self_mean - dn.cal_mean())
                region_img = dn.merge(region_img, 5)  # 对下方节点进行递归合并

        for un in self.up_node:  # 遍历所有上方节点
            # 上方节点与当前节点类似，且未被访问过
            if abs(self_mean - un.cal_mean()) <= threshold and un not in ImgNode.Visited_List:
```

```
                # print('up', self_mean - un.cal_mean())
                region_img = un.merge(region_img, 5)  # 对上方节点进行递归合并

        for ln in self.left_node:  # 遍历所有左侧节点
            # 左侧节点与当前节点类似，且未被访问过
            if abs(self_mean - ln.cal_mean()) <= threshold and ln not in ImgNode.Visited_List:
                # print('left', self_mean - ln.cal_mean())
                region_img = ln.merge(region_img, 5)  # 对左侧节点进行递归合并

        return region_img
```

SplitMerge.py（实现图像分割、轮廓提取与跟踪）

```
import cv2
import numpy as np
import sys
from imgnode import ImgNode
import matplotlib.pyplot as plt

def region_split_merge(img, min_area=(1,1), threshold=5.0):
    '''
    区域分裂合并算法，主要依靠ImgNode类实现
    输入：待处理图像，分裂的最小区域，合并的相似性判断阈值
    输出：前后景分割后的二值化图像
    '''
    draw_img = img.copy() # 用于绘制分裂结果的图像

    start_node = ImgNode(img, None, 0, img.shape[0], 0, img.shape[1]) # 创建起始节点，即整幅图像

    draw_img = start_node.split(draw_img, min_area) # 区域分裂

    leaf_father = start_node.find_leaf_father() # 寻找开始合并的节点
    region_img = np.zeros((int(img.shape[0]), int(img.shape[1]))) # 二值化图像初始化
    region_img = leaf_father.sub_node3.merge(region_img, threshold) # 区域合并

    return region_img,draw_img

def extract_contour(region_img):
    '''
    轮廓提取，某一像素周边若有背景像素，则认为其为轮廓
    输入：二值化图像，目标像素为黑色，背景像素为白色
    输出：轮廓图像，轮廓为黑色，背景为白色
    '''
    contour_img = region_img.copy() # 初始化轮廓图像

```

```
    for h in range(1,region_img.shape[0]-1):
        for w in range(1,region_img.shape[1]-1): # 遍历图像中的每一点
            if np.sum(region_img[h-1:h+2, w-1:w+2]) == 0: # 如果该点为黑色且周围全为白色，则认为该点为轮廓，8邻域
            # if region_img[h][w] == 0 and region_img[h-1][w] == 0 and region_img[h+1][w] == 0 and region_img[h][w-1] == 0 and region_img[h][w+1] == 0: #4邻域
                contour_img[h][w] = 255 # 若像素本身及其周围像素均为黑色，则其为内部点，将其置为白色
    return contour_img

def track_contour(img, start_point, all_cnts):
    '''
    轮廓跟踪
    输入：边界图像，当前轮廓起始点，已被跟踪的轮廓点集合
    输出：当前轮廓freeman链码
    '''
    neibor = [(0, 1), (-1, 1), (-1, 0), (-1, -1), (0, -1), (1, -1), (1, 0), (1, 1)]  # 8连通方向码
    dir = 5  # 起始搜索方向
    freeman = [start_point] # 用于存储轮廓方向码

    current_point = start_point # 将轮廓的开始点设为当前点

    neibor_point = tuple(np.array(current_point) + np.array(neibor[dir])) # 通过当前点和邻域点集以及链码值确定邻点

    if neibor_point[0] >= img.shape[0] or neibor_point[1] >= img.shape[1] or neibor_point[0] < 0 or neibor_point[1] < 0: # 若邻点超出边界，则轮廓结束
        return freeman

    while True:  # 轮廓扫描循环
        # print('current_point',current_point)
        while img[neibor_point[0], neibor_point[1]] != 0:  # 邻点不是边界点
            dir += 1  # 逆时针旋转45度进行搜索
            if dir >= 8:
                dir -= 8
            neibor_point = tuple(np.array(current_point) + np.array(neibor[dir])) # 更新邻点

            if neibor_point[0] >= img.shape[0] or neibor_point[1] >= img.shape[1] or neibor_point[0] < 0 or neibor_point[1] < 0: # 若邻点超出边界，则轮廓结束
                return freeman

        else:
            current_point = neibor_point # 将符合条件的邻域点设为当前点进行下一次的边界点搜索

```

```
            if current_point in all_cnts: # 如果当前点已经在轮廓中，则轮廓结束
                return freeman

            freeman.append(dir) # 将当前方向码加入轮廓方向码list
            if (dir % 2) == 0:
                dir += 7
            else:
                dir += 6
            if dir >= 8:
                dir -= 8 # 更新方向
            neibor_point = tuple(np.array(current_point) +
np.array(neibor[dir])) # 更新邻点

            if neibor_point[0] >= img.shape[0] or neibor_point[1] >=
img.shape[1] or neibor_point[0] < 0 or neibor_point[1] < 0: # 若邻点超出边
界，则轮廓结束
                return freeman

        if current_point == start_point:
            break # 当搜索点回到起始点，搜索结束，退出循环

    return freeman

def draw_contour(img, contours, color=(0, 0, 255)):
    '''
    在img上绘制轮廓
    输入：欲绘制的图像，轮廓链码，颜色
    输出：绘制好的图像
    '''
    for (x, y) in contours: # 绘制轮廓
        img[x-1:x+1, y-1:y+1] = color  # 粗
        # img_cnt[x,y] = color # 细
    return img

def find_start_point(img, all_cnts):
    '''
    寻找起始点
    输入：边界图像，已被识别到的轮廓list
    输出：起始点
    '''
    start_point = (-1, -1) # 初始化起始点

    # 寻找起始点
    for i in range(img.shape[0]):
        for j in range(img.shape[1]):
            if img[i, j] == 0 and (i, j) not in all_cnts: # 点为黑色且不在已
识别到的轮廓list中
                start_point = (i, j) # 找到新的起始点
                break
```

```
116         if start_point != (-1, -1):
117             break
118     return start_point
119 
120 def find_cnts(img):
121     '''
122     寻找轮廓集合
123     输入：边界图像
124     输出：轮廓集合（list,每一项都是一个轮廓链码）
125     '''
126     contours = [] # 当前边界轮廓初始化
127     cnts = [] # 轮廓集合初始化
128     freemans = [] # 轮廓方向码集合初始化
129     all_cnts = [] # 所有已找到的轮廓点
130 
131     while True:
132         start_point = find_start_point(img, all_cnts) # 寻找当前边界的轮廓起始点
133 
134         if start_point == (-1, -1): # 若找不到新的起始点，则说明所有的轮廓点都已被找到，退出循环
135             break
136 
137         freeman = track_contour(img, start_point, all_cnts) # 寻找当前边界的轮廓
138         contours = freeman2contour(freeman) # 将轮廓方向码转换为轮廓链码
139 
140         cnts.append(contours) # 将找到的轮廓加入轮廓集合中
141         freemans.append(freeman) # 将找到的轮廓方向码加入轮廓方向码集合中
142 
143         all_cnts = all_cnts + contours # 将找到的轮廓点加入轮廓点集合中
144 
145     # 去掉短轮廓（干扰轮廓）
146     fms = []
147     for fm in freemans:
148         if len(fm) >= 10:
149             fms.append(fm)
150 
151     return fms
152 
153 def draw_cnts(cntlists, img, color=(0, 0, 255), mode='freeman'):
154     '''
155     绘制所有轮廓
156     输入：轮廓集合，欲绘制的图像，颜色
157     输出：绘制好的图像
158     '''
159     if mode == 'freeman':
160         for freeman in cntlists:
161             cnt = freeman2contour(freeman)
```

```
            img = draw_contour(img, cnt, color) # 逐一绘制每个轮廓
    elif mode == 'contour':
        for cnt in cntlists:
            img = draw_contour(img, cnt, color) # 逐一绘制每个轮廓
    return img

def contours_filter(freemans, windows_size = 13):
    '''
    对轮廓进行滤波（均值滤波）
    输入：轮廓集合，滤波窗口大小
    输出：滤波后的轮廓集合
    '''
    if (windows_size % 2) == 0:
        windows_size += 1 # 保证windows_size为奇数
    cnts_filter = [] # 初始化滤波后的轮廓集合
    for freeman in freemans:
        cnt = freeman2contour(freeman) # 将轮廓方向码转换为轮廓链码
        for i in range(int((windows_size-1)/2), len(cnt)-int((windows_size-1)/2)):
            ix = np.mean([cnt[j][0] for j in range(i-int((windows_size-1)/2), i+int((windows_size-1)/2)+1)])
            iy = np.mean([cnt[j][1] for j in range(i-int((windows_size-1)/2), i+int((windows_size-1)/2)+1)])
            cnt[i] = (int(ix), int(iy)) # 均值滤波
        cnts_filter.append(cnt) # 将滤波后的轮廓添加到集合中
    return cnts_filter

def freeman2contour(freeman):
    '''
    轮廓方向码转换为轮廓
    输入：轮廓方向码
    输出：轮廓
    '''
    neibor = [(0, 1), (-1, 1), (-1, 0), (-1, -1), (0, -1), (1, -1), (1, 0), (1, 1)]  # 8连通方向码
    cnt = [freeman[0]] # 初始化轮廓
    for i in range(1,len(freeman)):
        cnt.append(tuple(np.array(cnt[-1]) + np.array(neibor[freeman[i]])))
    return cnt

def contour2freeman(cnt):
    '''
    轮廓转换为轮廓方向码
    输入：轮廓
    输出：轮廓方向码
    '''
    neibor = [(0, 1), (-1, 1), (-1, 0), (-1, -1), (0, -1), (1, -1), (1, 0), (1, 1)]  # 8连通方向码
    freeman = [] # 初始化轮廓方向码
```

```
    for i in range(len(cnt)-1):
        freeman.append(neibor.index(tuple(np.array(cnt[i+1]) - np.array(cnt[i]))))
    return freeman


if __name__ == '__main__':
    sys.setrecursionlimit(100000) # 设置最大允许递归深度
    read_path = 'zju_logo.png' # 设置读取图像的路径

    save_path = read_path[:-4]+'_results.png' # 设置保存图像的路径

    print('save the result to '+save_path)

    img = cv2.imread(read_path, 0) # 读入图像

    origin_img = img.copy() # 备份原始图像

    region_img,draw_img = region_split_merge(img, min_area=(1,1), threshold=5.0) # 5.0 # 区域分裂合并
    cv2.imwrite('draw_img.png', draw_img)
    cv2.imwrite('region_img.png', region_img)

    contour_img = extract_contour(region_img) # 轮廓提取
    cv2.imwrite('contour_img.png', contour_img)

    freemans = find_cnts(contour_img) # 轮廓跟踪

    print('freemans:')
    print(freemans)

    img_cnt = 255*np.ones([img.shape[0], img.shape[1], 3])
    img_cnt = draw_cnts(freemans, img_cnt, color = (0, 0, 255), mode='freeman') # 绘制轮廓跟踪结果
    cv2.imwrite('img_cnt.png', img_cnt)

    cnts_filter = contours_filter(freemans, windows_size = 11) # 轮廓链码滤波

    img_cnt_filter = 255*np.ones([img.shape[0], img.shape[1], 3])
    img_cnt_filter = draw_cnts(cnts_filter, img_cnt_filter, color=(255, 0, 0), mode='contour') # 绘制轮廓链码滤波结果
    cv2.imwrite('img_cnt_filter.png', img_cnt_filter)

    plt.figure(figsize=(9, 9.5))
    title_size = 12
    plt.subplot(321)
    plt.axis('off')
    plt.imshow(origin_img,cmap='gray')
```

```
250     plt.title("Figure 1: Original image",fontdict={'weight':'normal','size':
    title_size})
251 
252     plt.subplot(322)
253     plt.axis('off')
254     plt.imshow(draw_img,cmap='gray')
255     plt.title("Figure 2: Splited image",fontdict={'weight':'normal','size':
    title_size})
256 
257     plt.subplot(323)
258     plt.axis('off')
259     plt.imshow(region_img,cmap='gray')
260     plt.title("Figure 3: Merged image",fontdict={'weight':'normal','size':
    title_size})
261 
262     plt.subplot(324)
263     plt.axis('off')
264     plt.imshow(contour_img,cmap='gray')
265     plt.title("Figure 4: Contours",fontdict={'weight':'normal','size':
    title_size})
266 
267     plt.subplot(325)
268     plt.axis('off')
269     plt.imshow(cv2.cvtColor(img_cnt.astype(np.float32),cv2.COLOR_BGR2RGB))
270     plt.title("Figure 5: Contours tracked by ChainCode",fontdict=
    {'weight':'normal','size': title_size})
271 
272     plt.subplot(326)
273     plt.axis('off')
274 
     plt.imshow(cv2.cvtColor(img_cnt_filter.astype(np.float32),cv2.COLOR_BGR2RGB
    ))
275     plt.title("Figure 6: Filtered Contours",fontdict=
    {'weight':'normal','size': title_size})
276 
277     plt.savefig(save_path, bbox_inches='tight')
278     plt.show()
279 
280     cv2.waitKey(0)
```

4 实验结果与分析

4.1 实验结果

4.1.1 单连通域

程序运行结果如下图所示（注：Figure 1与Figure 2及前文原图的背景灰度值看上去不同是由于matplotlib与opencv的兼容性所致，若均采用opencv的imshow来显示，其观感上没有区别）：

Figure 1: Original image

Figure 2: Splited image

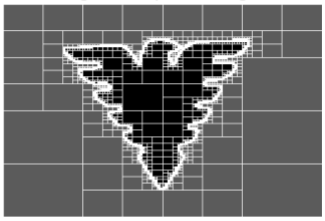


Figure 3: Merged image

Figure 4: Contours

Figure 5: Contours tracked by ChainCode

Figure 6: Filtered Contours

从图中可以看到，我实现的区域分裂与合并算法很好地把前景从背景中分离出来。同时，尽管我采用轮廓跟踪提取出来的边缘有一定毛刺（噪声）存在，但在经过我的滤波算法后，毛刺（噪声）得到了较好的去除。

边缘的链码表示如下图所示（第一项为起始点，随后是方向码）：


```
[[(49, 210), 5, 4, 5, 4, 5, 4, 5, 5, 4, 5, 5, 6, 5, 4, 6, 0, 7, 5, 4, 6, 5, 6, 7, 5, 6, 5, 7, 6, 5, 4, 6, 6, 7, 5, 6, 6, 6, 6, 5, 4,
6, 5, 4, 3, 2, 3, 2, 2, 4, 3, 2, 3, 3, 4, 3, 2, 2, 3, 2, 2, 4, 3, 2, 4, 3, 3, 3, 4, 4, 3, 4, 3, 4, 3, 2, 5, 4, 3, 4, 4, 4, 4, 4, 4, 4
, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 5, 4, 4, 4, 4, 3, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 6, 5, 4, 4, 4, 3, 4, 4, 4, 4,
 4, 4, 4, 6, 5, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 5, 6, 6, 6, 7, 0, 7, 6, 7, 7, 0, 7, 7,
6, 6, 1, 7, 6, 6, 0, 7, 0, 7, 7, 0, 7, 0, 7, 0, 0, 1, 0, 7, 0, 0, 0, 7, 6, 1, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 7
, 1, 7, 6, 6, 6, 0, 7, 6, 5, 4, 4, 6, 5, 4, 3, 4, 5, 4, 5, 4, 4, 4, 4, 4, 4, 4, 5, 4, 3, 4, 4, 4, 4, 4, 4, 4, 6, 5, 5, 4, 6, 6, 6, 6,
 0, 7, 6, 7, 0, 0, 7, 5, 4, 0, 7, 7, 7, 7, 0, 7, 0, 7, 0, 7, 1, 7, 6, 1, 7, 7, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 7, 0, 0, 0, 0, 0, 0,
1, 0, 7, 6, 7, 0, 7, 6, 6, 6, 5, 7, 5, 4, 4, 3, 4, 4, 4, 4, 4, 5, 4, 4, 5, 4, 3, 4, 4, 6, 5, 4, 5, 4, 4, 5, 0, 7, 6, 6, 6, 7, 7, 7, 0, 7
, 5, 4, 0, 7, 7, 6, 0, 0, 0, 7, 5, 7, 0, 7, 7, 0, 0, 0, 0, 0, 0, 7, 7, 0, 1, 0, 0, 0, 0, 0, 0, 7, 0, 0, 0, 0, 1, 7, 7, 0, 0, 7, 5, 4, 6, 7,
 5, 4, 4, 6, 5, 3, 4, 4, 4, 4, 4, 4, 5, 4, 4, 4, 0, 7, 5, 4, 5, 4, 6, 0, 7, 5, 0, 7, 7, 6, 0, 7, 5, 4, 0, 7, 0, 0, 7, 6, 0, 7, 6, 0,
7, 7, 0, 7, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 7, 6, 0, 0, 0, 7, 5, 4, 7, 6, 5, 5, 4, 4, 5, 4, 4, 4, 5, 6, 5, 5, 6, 6, 6, 1, 7, 7, 5, 4
, 0, 7, 6, 0, 7, 6, 0, 7, 7, 7, 7, 0, 7, 0, 0, 7, 0, 1, 0, 7, 6, 0, 1, 7, 0, 0, 0, 7, 0, 7, 6, 6, 6, 6, 6, 6, 6, 5, 4, 5, 4, 6, 6,
 5, 7, 6, 6, 6, 7, 7, 5, 4, 0, 7, 6, 7, 7, 6, 0, 0, 7, 5, 4, 0, 7, 0, 7, 5, 4, 7, 7, 6, 0, 7, 6, 7, 6, 6, 0, 7, 6, 7, 1, 7, 7, 6, 6,
6, 7, 0, 7, 6, 6, 7, 7, 0, 7, 5, 4, 7, 7, 6, 0, 7, 6, 7, 7, 0, 0, 0, 0, 1, 0, 1, 0, 2, 2, 1, 2, 1, 1, 2, 1, 1, 2, 1, 0, 2, 2, 1, 0, 2
, 2, 1, 1, 1, 4, 3, 1, 0, 1, 0, 4, 3, 1, 0, 2, 2, 1, 0, 1, 2, 2, 2, 1, 0, 2, 2, 1, 0, 2, 1, 0, 2, 2, 1, 2, 2, 2, 2, 1, 3, 2, 3, 2, 2,
 4, 3, 2, 2, 4, 3, 2, 4, 3, 2, 2, 1, 0, 2, 1, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 1, 0, 2, 1, 7, 1, 2, 1, 0, 0, 1, 0, 2, 1, 0, 4, 3, 1, 1,
7, 2, 3, 1, 0, 2, 1, 2, 2, 2, 3, 3, 5, 3, 2, 4, 4, 4, 4, 4, 4, 4, 4, 4, 3, 2, 2, 2, 1, 0, 1, 0, 0, 0, 2, 1, 7, 0, 0, 0, 0, 0, 1, 0, 0
, 2, 1, 7, 0, 3, 1, 0, 7, 1, 0, 2, 1, 0, 2, 1, 0, 2, 1, 2, 2, 1, 0, 4, 3, 1, 0, 2, 2, 2, 2, 2, 2, 2, 4, 3, 4, 4, 5, 4, 4, 4, 4, 4, 4,
 4, 4, 3, 3, 3, 4, 5, 3, 2, 1, 0, 1, 1, 2, 1, 7, 0, 0, 0, 2, 1, 0, 0, 0, 0, 0, 0, 0, 7, 0, 0, 0, 1, 0, 1, 1, 0, 2, 1, 0, 2, 2, 1, 1,
1, 1, 0, 1, 1, 2, 1, 0, 2, 2, 2, 2, 2, 2, 4, 3, 4, 3, 1, 0, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 3, 2, 2, 1
, 0, 2, 1, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 2, 1, 0, 0, 7, 0, 0, 0, 1, 0, 0, 0, 2, 1, 7, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 2, 2, 1, 0, 0,
 2, 1, 0, 4, 3, 1, 1, 7, 2, 3, 1, 0, 2, 2, 2, 2, 2, 4, 3, 5, 3, 2, 4, 4, 4, 4, 4, 4, 4, 4, 5, 3, 4, 4, 4, 4, 4, 4, 5, 3, 4, 4, 4, 3,
4, 4, 4, 4, 4, 4, 3, 2, 2, 1, 0, 1, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 1, 0, 7, 0, 1, 0, 0, 0, 1, 1, 0
, 2, 1, 7, 1, 0, 1, 3, 1, 1, 0, 0, 1, 2, 2, 1, 0, 1, 2, 3, 1, 2, 1, 0, 2, 2, 4, 3, 3, 4, 4, 4, 4, 4, 4, 3, 5, 4, 4, 4, 4, 4, 4, 4, 3, 5, 4, 4,
 4, 4, 4, 5, 5, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 5, 4, 4, 4, 4,
4, 4, 3, 4, 4, 4, 4, 4, 4, 4, 4, 6, 5, 4, 3, 4, 4, 5, 4, 4, 5, 4, 7, 5, 4, 3, 5, 4, 0, 7, 5, 4, 4, 4, 5, 4, 6, 5, 5, 6, 5, 5, 4, 0, 7
, 5, 5, 4, 7, 5, 4, 5, 6, 5, 7, 5, 4, 6, 6, 6, 5, 4, 6, 5, 6, 5, 4, 4, 3, 1, 4, 3, 2, 2, 4, 3, 2, 2, 2, 2, 2, 2, 2, 2, 2, 1, 0, 1, 0,
 1, 0, 0, 1, 0, 1, 0, 2, 1, 7, 1, 1, 0, 2, 2, 2, 4, 4, 4, 3, 2, 3, 3, 4, 4, 4, 4, 3, 2, 5, 4, 3, 2, 4, 5, 4, 3, 4, 4, 4, 3, 4, 4, 3,
4, 4, 4, 4, 4, 4, 4, 5, 4, 3, 4, 4, 4, 4, 4]]
```

4.1.2 多连通域

我的算法同样适用于多连通域：

Figure 1: Original image

Figure 2: Splited image

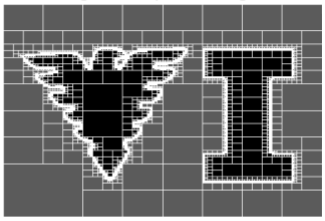


Figure 3: Merged image

Figure 4: Contours

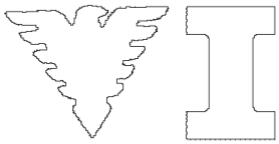
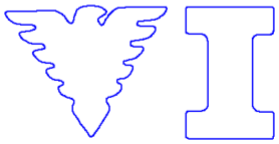


Figure 5: Contours tracked by ChainCode

Figure 6: Filtered Contours



边缘的链码表示如下图所示：


```
[[(49, 210), 5, 4, 5, 4, 5, 4, 5, 5, 4, 5, 5, 6, 5, 4, 6, 0, 7, 5, 4, 6, 5, 6, 7, 5, 6, 5, 7, 6, 5, 4, 6, 6, 7, 5, 6, 6, 6, 6, 5, 4,
6, 5, 4, 3, 2, 3, 2, 2, 4, 3, 2, 3, 3, 4, 3, 2, 2, 3, 2, 2, 4, 3, 2, 4, 3, 3, 3, 4, 4, 3, 4, 3, 4, 3, 2, 5, 4, 3, 4, 4, 4, 4, 4, 4, 4
, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4,
 4, 4, 4, 6, 5, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 5, 6, 6, 6, 7, 0, 7, 6, 7, 7, 0, 7, 7,
6, 6, 1, 7, 6, 6, 0, 7, 0, 7, 7, 0, 7, 0, 7, 0, 0, 1, 0, 7, 0, 0, 0, 7, 6, 1, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 7
, 1, 7, 6, 6, 6, 0, 7, 6, 5, 4, 4, 6, 5, 4, 3, 4, 5, 4, 5, 4, 4, 4, 4, 4, 4, 4, 4, 5, 4, 3, 4, 4, 4, 4, 4, 4, 4, 4, 4, 6, 5, 5, 5, 4, 6, 6, 6, 6, 6,
 0, 7, 6, 7, 0, 0, 7, 5, 4, 0, 7, 7, 7, 7, 0, 7, 0, 7, 0, 7, 0, 7, 1, 7, 6, 1, 7, 7, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 7, 0, 0, 0, 0, 0, 0, 0,
1, 0, 7, 6, 7, 0, 7, 6, 6, 6, 5, 7, 5, 4, 4, 3, 4, 4, 4, 4, 4, 5, 4, 4, 5, 4, 3, 4, 4, 6, 5, 4, 5, 4, 4, 5, 0, 7, 6, 6, 6, 7, 7, 0, 7
, 5, 4, 0, 7, 7, 6, 0, 0, 0, 7, 5, 7, 0, 7, 7, 0, 0, 0, 0, 0, 7, 7, 0, 1, 0, 0, 0, 0, 0, 0, 7, 0, 0, 0, 0, 1, 7, 7, 0, 0, 7, 5, 4, 6, 7,
 5, 4, 4, 6, 5, 3, 4, 4, 4, 4, 4, 4, 5, 4, 4, 4, 0, 7, 5, 4, 5, 4, 6, 0, 7, 5, 0, 7, 7, 6, 0, 7, 5, 4, 0, 7, 0, 0, 7, 6, 0, 7, 6, 0,
7, 7, 0, 7, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 7, 6, 0, 0, 0, 7, 5, 4, 7, 6, 5, 5, 4, 4, 5, 4, 4, 4, 5, 6, 5, 5, 6, 6, 6, 1, 7, 7, 5, 4
, 0, 7, 6, 0, 7, 6, 0, 7, 7, 7, 7, 0, 7, 0, 0, 7, 0, 1, 0, 7, 6, 0, 1, 7, 0, 0, 0, 7, 0, 7, 6, 6, 6, 6, 6, 6, 6, 6, 5, 4, 5, 4, 6, 6,
 5, 7, 6, 6, 6, 7, 7, 5, 4, 0, 7, 6, 7, 7, 6, 0, 0, 7, 5, 4, 0, 7, 0, 7, 5, 4, 7, 7, 6, 0, 7, 6, 7, 6, 6, 0, 7, 6, 7, 1, 7, 7, 6, 6,
6, 7, 0, 7, 6, 6, 7, 7, 0, 7, 5, 4, 7, 7, 6, 0, 7, 6, 7, 7, 0, 0, 0, 0, 1, 0, 1, 0, 2, 2, 1, 2, 1, 1, 2, 1, 1, 2, 1, 0, 2, 2, 1, 0, 2
, 2, 1, 1, 1, 4, 3, 1, 0, 1, 0, 4, 3, 1, 0, 2, 2, 1, 0, 1, 2, 2, 2, 1, 0, 2, 2, 1, 0, 2, 1, 0, 2, 2, 1, 2, 2, 2, 2, 1, 3, 2, 3, 2, 2,
 4, 3, 2, 2, 4, 3, 2, 4, 3, 2, 2, 1, 0, 2, 1, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 1, 0, 2, 1, 7, 1, 2, 1, 0, 0, 1, 0, 2, 1, 0, 4, 3, 1, 1,
 0, 2, 2, 1, 0, 1, 1, 2, 3, 1, 0, 2, 2, 1, 0, 2, 2, 1, 0, 2, 2, 1, 2, 1, 0, 1, 3, 1, 0, 2, 1, 2, 1, 1, 3, 1, 2, 1, 0, 4, 3, 1, 2, 3,
3, 2, 3, 3, 4, 3, 2, 2, 2, 2, 2, 2, 1, 0, 1, 0, 0, 0, 0, 0, 0, 1, 0, 1, 0, 0, 0, 0, 2, 2, 1, 0, 1, 2, 1, 0, 1, 1, 2, 2, 2, 2, 2, 2, 2
, 2, 4, 3, 4, 4, 4, 4, 4, 4, 3, 2, 2, 2, 2, 1, 0, 0, 0, 0, 1, 0, 7, 0, 0, 0, 0, 1, 0, 1, 0, 1, 0, 2, 1, 1, 0, 2, 1, 1, 4, 3, 1, 0, 2,
 2, 1, 0, 2, 3, 2, 1, 4, 3, 2, 4, 4, 4, 5, 4, 4, 4, 4, 4, 4, 4, 3, 2, 4, 3, 2, 2, 2, 1, 0, 0, 0, 0, 0, 0, 0, 2, 1, 7, 0, 0, 1, 0,
0, 0, 1, 0, 0, 0, 2, 2, 1, 0, 2, 1, 0, 2, 1, 0, 1, 0, 4, 3, 1, 0, 1, 2, 1, 2, 2, 3, 2, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4,
, 4, 4, 4, 3, 2, 2, 3, 1, 1, 0, 2, 2, 1, 0, 7, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 2, 1, 0, 0, 0, 1, 0, 1, 1, 0,
 4, 3, 1, 0, 1, 2, 1, 0, 1, 0, 2, 4, 3, 1, 0, 2, 2, 3, 2, 4, 3, 4, 4, 4, 4, 4, 4, 4, 4, 5, 4, 4, 4, 4, 4, 4, 3, 4, 4, 4, 4, 4, 4, 4,
4, 4, 3, 2, 2, 1, 0, 1, 0, 4, 3, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 7, 1, 0, 0, 1, 7, 0, 1, 0, 0, 0, 0, 1, 0, 1, 2, 1
, 0, 2, 1, 0, 0, 0, 1, 2, 2, 2, 2, 1, 0, 1, 2, 3, 1, 0, 2, 3, 2, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4,
 4, 4, 4, 4, 4, 4, 4, 4, 5, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 3, 5, 4, 4, 4, 4, 4, 4, 3, 4, 4, 4, 4, 4, 4, 4, 4, 6, 5, 4, 4, 4, 4, 4, 4, 4, 3,
2, 6, 5, 4, 4, 4, 5, 4, 5, 5, 4, 5, 7, 5, 4, 4, 4, 0, 7, 5, 4, 6, 5, 3, 7, 5, 4, 5, 4, 0, 7, 5, 4, 6, 6, 5, 4, 4, 6, 6, 6, 6, 6, 5, 4
, 6, 6, 5, 4, 5, 3, 3, 2, 2, 2, 2, 3, 4, 3, 1, 2, 3, 2, 2, 1, 0, 1, 0, 0, 1, 0, 1, 0, 0, 1, 2, 1, 7, 3, 1, 0, 2, 3, 1, 4, 4, 4, 3, 3,
 4, 3, 2, 5, 3, 2, 4, 4, 4, 4, 4, 4, 4, 3, 4, 4, 4, 4, 4, 4, 4, 3, 4, 4, 4, 4, 4, 4, 4], [(63, 283), 6, 5, 6, 6, 7, 6, 5, 6, 6, 7, 6, 5, 6, 6,
 6, 6, 6, 6, 7, 6, 5, 6, 6, 6, 6, 6, 6, 7, 6, 5, 6, 6, 6, 6, 6, 6, 7, 6, 5, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 7, 7, 0, 7, 7, 5, 0, 7, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6
, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6,
 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6,
4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 6, 7, 6, 5, 6, 6, 7, 6, 5, 6, 6, 7, 6, 5, 6, 6, 6, 6, 6, 6, 7, 6, 5, 6, 6, 6, 6, 6, 6, 7, 6, 5
, 6, 6, 6, 6, 6, 6, 7, 0, 0, 0, 0, 7, 1, 0, 0, 0, 0, 0, 7, 1, 0, 0, 0, 0, 0, 7, 1, 0, 0, 0, 0, 0, 7, 1, 0, 0, 0, 0, 0, 7, 1, 0, 0, 0,
 0, 0, 7, 1, 0, 0, 0, 0, 0, 7, 1, 0, 0, 0, 0, 0, 7, 1, 0, 0, 0, 0, 0, 7, 1, 0, 0, 0, 0, 0, 7, 1, 0, 0, 0, 0, 0, 7, 1, 0, 0, 0, 0, 0, 7, 1, 0, 0, 0, 0, 0,
7, 1, 0, 0, 0, 0, 0, 7, 1, 0, 0, 0, 0, 0, 7, 1, 0, 0, 0, 0, 0, 7, 1, 0, 0, 0, 0, 0, 7, 1, 0, 0, 0, 0, 0, 7, 1, 0, 0, 0, 0, 0, 7, 7, 2, 2
, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2,
2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2,
, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2,
 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2,
2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2,
, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4,
 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4,
4, 4, 4, 4, 4, 4, 4, 4, 4]]
```

4.1.3 背景灰度不均匀

运行结果：

Figure 1: Original image

Figure 2: Splited image

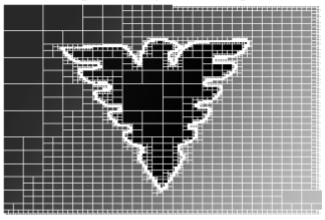


Figure 3: Merged image

Figure 4: Contours

Figure 5: Contours tracked by ChainCode

Figure 6: Filtered Contours

Figure 1: Original image

Figure 2: Splited image

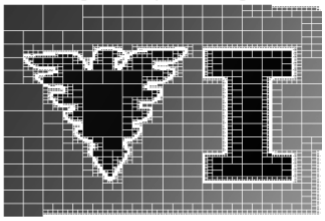


Figure 3: Merged image

Figure 4: Contours

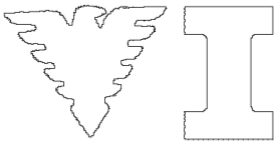
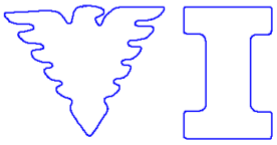


Figure 5: Contours tracked by ChainCode

Figure 6: Filtered Contours



从图中可以看到，尽管背景灰度不均匀，但是算法仍然可以将前景分割出来，只是在背景灰度渐变处会多分裂一些区域。

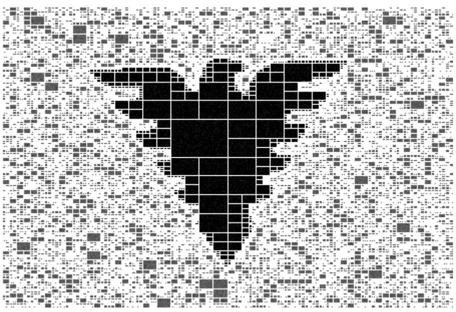
边缘的链码表示如下图所示：

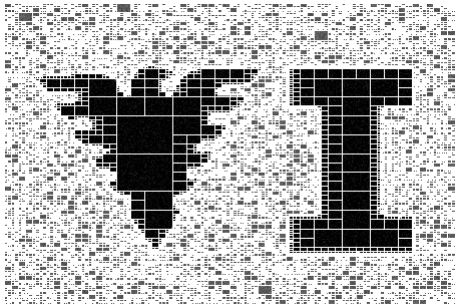
```
[[(49, 210), 5, 4, 5, 4, 5, 4, 5, 5, 4, 5, 5, 6, 5, 4, 6, 0, 7, 5, 4, 6, 5, 6, 7, 5, 6, 5, 7, 6, 5, 4, 6, 6, 7, 5, 6, 6, 6, 6, 5, 4,
6, 5, 4, 3, 2, 3, 2, 2, 4, 3, 2, 3, 3, 4, 3, 2, 2, 3, 2, 2, 4, 3, 2, 4, 3, 3, 3, 4, 4, 3, 4, 3, 4, 3, 2, 5, 4, 3, 4, 4, 4, 4, 4, 4, 4
, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 5, 4, 4, 4, 4, 3, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 6, 5, 4, 4, 4, 3, 4, 4, 4, 4,
 4, 4, 4, 4, 4, 4, 4, 5, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 7, 5, 6, 6, 6, 7, 0, 7, 6, 7, 7, 0, 7,
7, 6, 6, 1, 7, 6, 6, 0, 7, 0, 7, 7, 0, 7, 0, 7, 0, 0, 1, 0, 7, 0, 0, 0, 7, 6, 1, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
, 7, 1, 7, 6, 6, 6, 0, 7, 6, 5, 4, 4, 6, 5, 4, 3, 4, 5, 4, 5, 4, 4, 4, 4, 4, 4, 4, 5, 4, 3, 4, 4, 4, 4, 4, 4, 4, 6, 5, 5, 4, 6, 6, 6,
 6, 0, 7, 6, 7, 0, 0, 7, 5, 4, 0, 7, 7, 7, 7, 0, 7, 0, 7, 0, 7, 1, 7, 6, 1, 7, 7, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 7, 0, 0, 0, 0, 0,
0, 1, 0, 7, 6, 7, 0, 7, 6, 6, 6, 5, 7, 5, 4, 4, 3, 4, 4, 4, 4, 4, 5, 4, 4, 5, 4, 3, 4, 4, 6, 5, 4, 5, 4, 4, 5, 0, 7, 6, 6, 6, 7, 7, 0
, 7, 5, 4, 0, 7, 7, 6, 0, 0, 0, 7, 5, 7, 0, 7, 7, 0, 0, 0, 0, 0, 7, 7, 0, 1, 0, 0, 0, 0, 7, 0, 0, 0, 0, 1, 7, 7, 0, 0, 7, 5, 4, 6,
 7, 5, 4, 4, 6, 5, 3, 4, 4, 4, 4, 4, 4, 5, 4, 4, 4, 0, 7, 5, 4, 5, 4, 6, 0, 7, 5, 6, 7, 6, 0, 0, 0, 7, 5, 4, 0, 7, 7, 6, 0, 0, 0, 7,
6, 0, 7, 7, 0, 7, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 7, 6, 0, 0, 0, 7, 5, 4, 7, 6, 5, 5, 4, 4, 5, 4, 4, 4, 5, 6, 5, 5, 6, 6, 6, 1, 7, 7
, 5, 4, 0, 7, 6, 0, 7, 6, 0, 7, 7, 7, 7, 0, 7, 0, 0, 7, 0, 1, 0, 7, 5, 0, 0, 1, 7, 0, 0, 0, 7, 0, 7, 6, 6, 6, 6, 6, 6, 6, 6, 5, 4, 5,
 4, 6, 6, 5, 7, 6, 6, 6, 7, 7, 5, 4, 0, 7, 6, 7, 7, 6, 0, 0, 7, 5, 4, 0, 7, 0, 7, 5, 4, 7, 7, 6, 0, 7, 6, 7, 6, 6, 0, 7, 6, 7, 7, 0,
7, 7, 5, 7, 0, 7, 6, 6, 7, 7, 0, 7, 5, 4, 7, 7, 6, 0, 7, 6, 7, 7, 1, 7, 0, 0, 1, 0, 1, 0, 2, 2, 1, 2, 1, 1, 2, 1, 1, 2, 1, 0, 2, 2, 1
, 0, 2, 2, 1, 1, 1, 4, 3, 1, 0, 1, 0, 4, 3, 1, 0, 2, 2, 1, 0, 1, 2, 2, 2, 1, 0, 2, 2, 1, 2, 1, 2, 1, 0, 1, 2, 2, 2, 2, 1, 3, 2, 3, 4,
 3, 2, 2, 4, 3, 2, 2, 2, 4, 3, 2, 2, 1, 0, 2, 1, 0, 2, 1, 0, 7, 0, 0, 0, 0, 1, 0, 1, 0, 0, 1, 1, 2, 1, 0, 0, 1, 0, 2, 1, 0, 4, 3, 1,
1, 7, 2, 3, 1, 0, 2, 1, 2, 2, 2, 3, 3, 5, 3, 2, 4, 4, 4, 4, 4, 4, 4, 4, 4, 3, 2, 2, 2, 1, 0, 1, 0, 0, 0, 2, 1, 7, 0, 0, 0, 0, 0, 1, 0
, 0, 2, 1, 7, 0, 3, 1, 0, 7, 1, 0, 2, 1, 0, 2, 2, 2, 1, 1, 2, 1, 2, 1, 0, 2, 2, 2, 2, 2, 2, 2, 4, 3, 4, 4, 5, 4, 4, 4, 4, 4, 4, 4, 4,
 3, 3, 3, 4, 5, 3, 2, 1, 0, 1, 1, 0, 1, 0, 0, 0, 2, 1, 0, 0, 0, 0, 0, 0, 7, 0, 0, 0, 0, 1, 0, 1, 1, 0, 2, 1, 0, 2, 2, 1, 1, 1, 1, 0,
1, 1, 2, 2, 2, 1, 0, 2, 2, 2, 4, 3, 4, 3, 1, 0, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 3, 4, 3, 2, 1, 0, 2, 1, 0
, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 0, 7, 0, 0, 0, 1, 0, 0, 0, 2, 1, 7, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 2, 2, 1, 0, 0, 2, 1, 0, 2, 1,
 2, 3, 1, 0, 2, 2, 2, 2, 2, 4, 3, 5, 3, 2, 4, 4, 4, 4, 4, 4, 4, 4, 5, 3, 4, 4, 4, 4, 4, 4, 3, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 3, 2, 2,
1, 0, 1, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 1, 0, 7, 0, 1, 0, 0, 0, 1, 1, 0, 2, 1, 7, 1, 1, 0, 1, 0
, 4, 3, 1, 0, 0, 1, 2, 1, 0, 2, 1, 2, 3, 1, 2, 1, 0, 2, 2, 4, 3, 3, 4, 4, 4, 4, 4, 3, 5, 4, 4, 4, 4, 4, 3, 5, 4, 4, 4, 4, 4, 5, 5, 4,
 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 5, 4, 4, 4, 4, 4, 4, 3, 4, 4, 4,
4, 4, 4, 4, 4, 6, 5, 4, 3, 4, 4, 5, 4, 4, 5, 4, 7, 5, 4, 3, 5, 4, 6, 5, 4, 5, 4, 6, 5, 5, 6, 5, 5, 4, 0, 7, 5, 5, 4, 7, 5, 4, 5, 6, 5
, 7, 5, 4, 5, 4, 6, 0, 7, 5, 4, 5, 4, 5, 3, 2, 4, 3, 4, 3, 2, 2, 2, 3, 1, 2, 2, 2, 2, 1, 0, 1, 0, 1, 0, 0, 1, 0, 1, 0, 2, 1, 7, 1, 1,
 0, 2, 2, 2, 4, 4, 4, 3, 2, 3, 3, 4, 4, 4, 4, 3, 2, 5, 4, 3, 2, 4, 5, 4, 3, 4, 4, 4, 3, 4, 4, 3, 4, 4, 4, 4, 4, 4, 4, 5, 4, 3, 4, 4,
4, 4, 4]]
```

```
[[(61, 143), 5, 4, 4, 4, 4, 5, 4, 4, 5, 5, 3, 5, 6, 5, 4, 5, 5, 6, 5, 6, 7, 6, 5, 6, 6, 6, 5, 4, 6, 6, 6, 6, 6, 6, 6, 5, 4, 6, 5, 4,
3, 2, 3, 4, 3, 2, 2, 4, 3, 2, 2, 4, 3, 3, 1, 3, 2, 4, 3, 3, 4, 4, 3, 3, 4, 3, 4, 3, 4, 3, 4, 4, 4, 4, 4, 4, 4, 5, 4, 4, 4, 4, 4, 3, 4
, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 0, 7, 5, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 5, 4, 4, 4, 3, 4, 4, 4, 4, 4, 4, 4, 4,
 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 6, 5, 4, 6, 0, 7, 6, 6, 7, 6, 0, 7, 6, 0, 7, 6, 7, 7, 7, 0, 7, 0, 7, 0, 7, 0, 0, 7, 0,
1, 0, 0, 7, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 7, 6, 6, 0, 0, 7, 6, 5, 4, 4, 6, 5, 4, 4, 4, 4, 4, 5, 4, 4, 4, 4
, 4, 4, 4, 5, 4, 4, 4, 4, 4, 4, 5, 4, 4, 6, 7, 6, 0, 7, 5, 7, 6, 0, 0, 7, 6, 6, 0, 7, 0, 0, 7, 0, 7, 0, 7, 0, 7, 6, 0, 0, 0, 0, 0, 0,
 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 7, 0, 7, 6, 6, 6, 7, 6, 6, 5, 4, 4, 3, 4, 4, 4, 4, 4, 4, 6, 5, 4, 3, 4, 6, 5, 4, 3, 6, 5, 4, 6,
7, 5, 6, 6, 0, 7, 0, 7, 7, 6, 0, 7, 7, 6, 7, 0, 7, 7, 0, 7, 0, 0, 1, 0, 7, 0, 0, 0, 0, 0, 1, 7, 0, 0, 0, 7, 6, 0, 7, 6, 7, 6, 5, 4, 4
, 4, 4, 4, 4, 4, 5, 4, 3, 4, 6, 5, 4, 5, 6, 6, 6, 7, 7, 7, 6, 0, 7, 6, 0, 7, 7, 6, 0, 7, 0, 0, 7, 7, 0, 0, 0, 0, 0, 0, 0, 1, 0, 7, 6,
 0, 7, 0, 7, 6, 6, 6, 5, 4, 3, 4, 5, 4, 4, 5, 5, 3, 5, 6, 6, 7, 6, 7, 6, 6, 7, 7, 7, 0, 7, 6, 0, 0, 0, 7, 0, 0, 7, 6, 0, 7, 0, 0, 0,
0, 0, 0, 7, 0, 7, 6, 6, 6, 6, 6, 6, 5, 4, 5, 4, 6, 7, 5, 6, 6, 6, 6, 0, 7, 5, 4, 7, 7, 0, 7, 6, 6, 7, 7, 7, 0, 7, 5, 4, 7, 7, 0, 7, 7
, 5, 7, 6, 6, 0, 7, 6, 7, 7, 0, 7, 6, 6, 7, 0, 7, 6, 6, 0, 7, 6, 0, 7, 5, 4, 0, 7, 6, 7, 0, 0, 0, 2, 2, 1, 0, 0, 2, 2, 2, 1, 1, 2, 1,
 0, 2, 2, 1, 0, 1, 1, 2, 3, 1, 0, 2, 2, 1, 0, 2, 2, 1, 0, 2, 2, 1, 2, 1, 0, 1, 3, 1, 0, 2, 1, 2, 1, 1, 3, 1, 2, 1, 0, 4, 3, 1, 2, 3,
3, 2, 3, 3, 4, 3, 2, 2, 2, 2, 2, 2, 1, 0, 1, 0, 0, 0, 0, 0, 0, 1, 0, 1, 0, 0, 0, 0, 2, 2, 1, 0, 1, 2, 1, 0, 1, 1, 2, 2, 2, 2, 2, 2, 2
, 2, 4, 3, 4, 4, 4, 4, 4, 4, 3, 2, 2, 2, 2, 1, 0, 0, 0, 0, 1, 0, 7, 0, 0, 0, 0, 1, 0, 1, 0, 1, 0, 2, 1, 1, 0, 2, 1, 1, 4, 3, 1, 0, 2,
 2, 1, 0, 2, 3, 2, 1, 4, 3, 2, 4, 4, 4, 5, 4, 4, 4, 4, 4, 4, 4, 3, 2, 4, 3, 2, 2, 2, 1, 0, 0, 0, 0, 0, 0, 0, 2, 1, 7, 0, 0, 1, 0,
7, 1, 1, 0, 0, 0, 2, 1, 0, 2, 1, 0, 2, 2, 1, 0, 1, 0, 4, 3, 1, 0, 1, 2, 1, 2, 2, 3, 2, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4
, 4, 4, 4, 3, 2, 2, 3, 1, 1, 0, 2, 2, 1, 0, 7, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 2, 1, 0, 0, 0, 1, 0, 1, 1, 0,
 4, 3, 1, 0, 1, 2, 1, 0, 1, 0, 2, 2, 2, 2, 2, 3, 2, 4, 3, 4, 4, 4, 4, 4, 4, 4, 4, 5, 4, 4, 4, 4, 4, 4, 3, 4, 4, 4, 4, 4, 4, 4, 4, 4,
3, 2, 2, 1, 0, 1, 0, 4, 3, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 7, 1, 0, 0, 1, 7, 0, 1, 0, 0, 0, 0, 1, 0, 1, 2, 1, 0, 2
, 1, 0, 0, 0, 1, 2, 2, 2, 1, 0, 1, 2, 3, 1, 0, 2, 3, 2, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4,
 4, 4, 4, 4, 4, 4, 4, 4, 4, 5, 4, 4, 4, 4, 4, 4, 4, 4, 4, 3, 5, 4, 4, 4, 4, 4, 3, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 5, 4, 3, 6, 5, 4,
4, 4, 5, 4, 5, 5, 4, 5, 7, 5, 4, 4, 4, 0, 7, 5, 4, 6, 5, 3, 7, 5, 4, 5, 4, 0, 7, 5, 4, 5, 4, 6, 5, 6, 6, 6, 6, 6, 5, 4, 6, 6, 5, 4, 4
, 4, 3, 2, 2, 2, 2, 3, 4, 3, 1, 2, 3, 2, 2, 1, 0, 1, 0, 0, 1, 0, 1, 0, 0, 1, 2, 1, 7, 3, 1, 0, 2, 3, 1, 4, 4, 4, 3, 3, 4, 3, 2, 5, 3,
 2, 4, 4, 4, 4, 4, 4, 3, 4, 4, 4, 4, 4, 4, 3, 4, 4, 4, 4, 4, 4], [(63, 283), 6, 5, 6, 6, 7, 6, 5, 6, 6, 7, 6, 5, 6, 6, 6, 6, 6, 6, 7,
 6, 5, 6, 6, 6, 6, 6, 6, 7, 6, 5, 6, 6, 6, 6, 6, 6, 7, 6, 5, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 7, 7, 0, 7, 7, 5, 0, 7, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6
, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6,
 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 5, 4, 6, 5, 4, 5, 5, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4,
4, 4, 4, 4, 4, 4, 4, 4, 6, 7, 6, 5, 6, 6, 7, 6, 5, 6, 6, 7, 6, 5, 6, 6, 6, 6, 6, 6, 7, 6, 5, 6, 6, 6, 6, 6, 6, 7, 6, 5, 6, 6, 6, 6, 6
, 6, 7, 0, 0, 0, 0, 7, 1, 0, 0, 0, 0, 0, 7, 1, 0, 0, 0, 0, 0, 7, 1, 0, 0, 0, 0, 0, 7, 1, 0, 0, 0, 0, 0, 7, 1, 0, 0, 0, 0, 0, 7, 1, 0,
 0, 0, 0, 0, 7, 1, 0, 0, 0, 0, 0, 7, 1, 0, 0, 0, 0, 0, 7, 1, 0, 0, 0, 0, 0, 7, 1, 0, 0, 0, 0, 0, 7, 1, 0, 0, 0, 0, 0, 7, 1, 0, 0, 0,
0, 0, 7, 1, 0, 0, 0, 0, 0, 7, 1, 0, 0, 0, 0, 0, 7, 1, 0, 0, 0, 0, 0, 7, 1, 0, 0, 0, 0, 0, 7, 1, 0, 0, 0, 0, 0, 7, 2, 2, 2, 2, 2, 2, 2
, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4,
 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 3, 4, 3, 3, 2, 2, 3, 2, 3, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2,
2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2
, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 1, 0, 2, 2, 1, 1, 0, 1, 0, 0, 0, 0, 0, 0, 0,
 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2,
2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4
, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4,
 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4,
4, 4, 4, 4]]
```

4.1.4 带有高斯噪声

分割结果：





从图中可以看出，带有高斯噪声的图像非常容易过分割，分离效果并不好，因此，对于含有高斯噪声的图像，可以滤波后再使用区域分裂与合并法。

4.2 对比分析

根据以上结果，我们可以得出以下结论：区域分裂与合并算法可以较好地适应单连通域、多连通域、背景灰度不均匀等多种情况。但对于含有噪声的图像，其分割效果不佳。此外，这种算法对分裂/合并原则的选取要求较高，只有选择了适当的原则，算法才能较好地完成分割，不适于处理多张差距过大的图片。