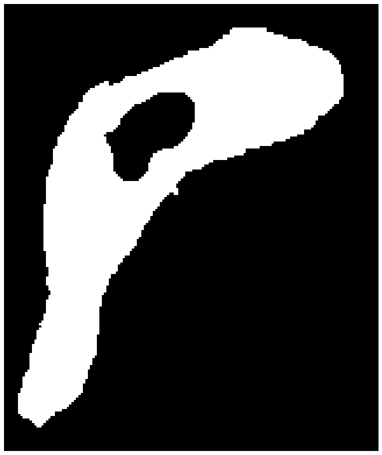
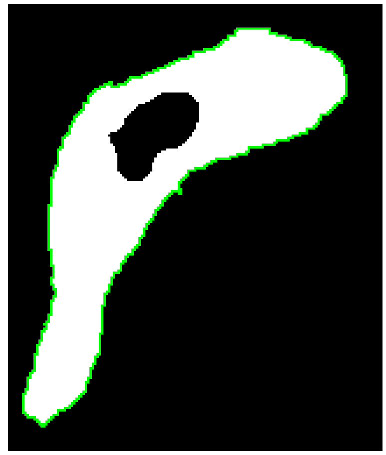
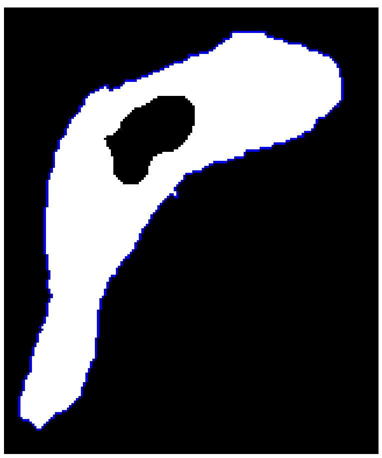
1 Problem 1



(a) Original Region



(b) 4-Connected Edge Tracking



(c) 8-Connected Edge Reduction

(d) Edge Differences

Figure 1: Edge Tracking

Figure 1a shows the initial region under consideration. Using the 4-connected edge tracking algorithm proposed in class, the edge in Figure 1b was created: once the first 'on' pixel (rastering across then down from the upper left) was identified, the region was tracked going counter-clockwise. The edge was reduced using the `reduce_edge()` function, which looked for redundant pixels in an 8-connected sense. When implementing this, it is important to realize that some sets of points are necessary to retain even if they can be substituted by 8-connected points, because these 'redundant' points are part of the region. Therefore, I solved that all redundant steps were at right angles (odd difference) but only when the shorter edge would not exclude a part of the region. This corresponds to cuts that are locally concave with respect to the region when going counterclockwise, or alternately when the difference between successive moves is positive. The other key was to note that when two successive moves involved 3 and 0 (see `problem1.py` for direction maps), the 0 should be treated as a 4 in order to not cut through the region. The result is shown in Figure 1c. Finally, the differences in the edges can be seen in Figure 1d. The overlap of both edges is given in white, and the pixels cut out from the 4-connected edge are shown in red.

Chain code for both edge chains are listed below in `problem1chain#.txt` The perimeters of both regions were calculated based on the chain length, less two for the coordinates of the start pixel. The results are given in `problem1.txt`. As expected, the 8-connected edge is smaller than the 4-connected edge.

problem1chain4.txt

```
10  98  3  2  3  3  2  2  2  3  3  2  2  3  2  3  2  3  2  2  3  2  2  2  2  3  2  2  2  2  2  3  3  2  2  3  2  2  2  2  3
        2  2  3  2  2  3  2  2  3  2  2  2  3  2  2  3  2  2  2  3  2  2  3  2  2  2  2  2  3  2  3  2  3  2  2  2  3  2  2  2
        1  1  2  3  2  2  3  2  2  3  2  2  3  2  3  2  3  2  3  3  3  2  2  3  3  3  2  3  2  3  2  3  2  3  3  2  3  3  2  3
        3  3  2  3  2  3  2  3  3  3  3  2  3  2  3  3  3  3  3  3  3  2  3  3  2  3  3  3  2  3  3  3  3  3  3  3  3  3  3  3
        3  2  3  3  3  3  3  3  3  3  3  3  3  3  3  3  3  3  3  3  3  0  3  3  3  3  3  3  3  0  3  3  3  3  3  2  3  3  3  0
        3  3  0  3  3  3  2  3  3  2  3  3  3  3  3  3  2  3  3  3  2  3  2  0  3  3  2  3  2  3  3  3  3  2  3  3  2  3  3  3
        3  2  3  3  3  3  3  3  3  2  3  2  3  3  2  3  3  3  3  3  2  3  3  2  3  3  3  3  3  3  3  3  0  3  0  3  0  0  0  3
        0  3  0  3  0  3  0  1  0  1  0  1  0  1  0  1  0  0  1  1  0  0  0  1  0  0  1  0  1  0  1  0  1  0  1  0  1  0  1  0
        1  1  1  1  0  1  1  0  1  1  1  1  0  1  1  0  1  1  1  0  1  0  1  1  1  1  1  1  1  1  1  0  1  1  1  1  1  1  1  1
        1  1  1  1  0  1  1  1  1  1  0  1  0  1  1  1  0  1  1  1  0  1  1  0  0  1  0  1  1  1  0  1  0  1  1  0  1  0  0  1
        1  0  1  0  1  0  1  1  1  0  1  0  1  1  1  0  1  1  0  1  0  1  0  1  1  0  1  1  0  1  0  1  0  1  1  0  1  0  1  0
        1  0  1  0  0  0  3  0  1  1  2  1  1  1  0  1  0  1  0  1  0  1  1  0  0  0  1  0  0  0  0  1  0  1  0  0  1  0  0  1
        0  0  0  0  0  0  1  0  0  0  1  0  0  0  0  1  0  1  1  0  0  0  0  0  0  1  0  0  0  0  0  1  0  1  0  0  0  0  0  1
        0  0  1  0  0  0  1  0  0  0  1  1  0  0  0  1  0  1  0  1  1  0  1  1  0  0  0  1  0  1  0  1  0  1  0  1  0  0  1  1
        0  1  0  1  1  1  1  1  1  1  1  1  2  1  1  1  1  2  1  1  2  1  2  1  2  1  2  1  2  2  2  1  2  2  1  2  2  2  2  1
        2  1  2  2  1  2  2  2  2  2  2  1  2  2  1  2  2  2  2  1  2  2  2  1  1  2  2  2  2  2  2  2  2  2  2  2  2  2  2
```

problem1chain8.txt

```
10  98  5  6  5  4  4  6  5  4  5  5  5  4  5  4  4  4  5  4  4  4  4  6  5  4  5  4  4  4  5  4  5  4  5  4  5  4  4  5  4
        5  4  4  5  4  5  4  4  4  4  5  5  5  4  4  5  4  3  2  4  5  4  5  4  5  4  5  5  5  6  6  5  4  6  6  5  5  5  5  6
        5  6  5  6  6  5  5  5  6  6  6  5  5  6  6  6  6  6  6  5  6  5  6  6  5  6  6  6  6  6  6  6  6  6  6  6  5  6  6  6
        6  6  6  6  6  6  6  6  6  6  6  6  6  6  6  6  7  6  6  6  6  6  6  7  6  6  6  5  6  6  6  7  6  7  6  5  6  5  6  6
        6  6  6  5  6  6  5  5  7  5  5  6  6  6  5  6  5  6  6  6  5  6  6  6  6  6  6  5  5  6  5  6  6  6  6  5  6  5  6  6
        6  6  6  6  6  6  7  7  0  0  7  7  7  7  0  1  1  1  1  1  0  2  1  0  0  1  0  1  1  1  1  1  1  1  2  2  2  1  2  1
        2  2  2  1  2  1  2  2  1  1  2  2  2  2  2  2  2  2  1  2  2  2  2  2  2  2  2  2  2  2  1  2  2  2  2  1  1  2  2  1
        2  2  1  2  1  0  1  2  2  1  1  2  1  1  0  2  1  1  1  2  2  1  1  2  2  1  2  1  1  1  2  1  2  1  1  1  2  1  1  1
        1  1  0  7  0  2  2  3  2  1  1  1  1  2  1  0  0  1  0  0  0  1  1  0  1  0  1  0  0  0  0  0  1  0  0  1  0  0  0  1
        2  1  0  0  0  0  0  1  0  0  0  0  1  1  0  0  0  0  1  0  1  0  0  1  0  0  2  1  0  0  1  1  2  1  2  1  0  0  1  1
        1  1  1  0  2  1  1  2  2  2  2  2  2  2  2  3  2  2  2  3  2  3  3  3  3  4  4  3  4  3  4  4  4  3  3  4  3  4  4
        4  4  4  3  4  3  4  4  4  3  4  4  3  2  4  4  4  4  4  4  4  4  4  4  4  4  4
```

problem1.txt

```
4-perim: 638px
8-perim: 466px
```

problem1.py

```
#Athanasios Athanassiadis Jan 2012
from segmentation import *

fig1 = np.fromfile('figure1_problem_set_3',dtype='>i2').reshape((192,161)) / 255.0
edge, chain = track_edge(fig1)
chain8 = reduce_edge(chain)
edge2 = decode_edge8(chain8, (192,161))

imsave('3-1a.png',fig1)
imsave('3-1b.png',(fig1-edge,fig1,fig1-edge))
imsave('3-1c.png',(fig1-edge2,fig1-edge2,fig1))
imsave('3-1d.png',(edge,edge2,edge2))

write_chain('problem1chain4.txt',chain)
write_chain('problem1chain8.txt',chain8)
```

```
#get and write perimiters
#for a chain of directions of length n,
#the perimeter length will be (n-2)+1
#because the first two 2 entries are start pix coords
#and it only takes (m-1) moves to traverse m pixels
with open('problem1.txt','w') as of:
    of.write('4-perim: {}px\n8-perim:
        {}px'.format(len(chain[2:]+1),len(chain8[2:]+1)))
```

2 Problem 2

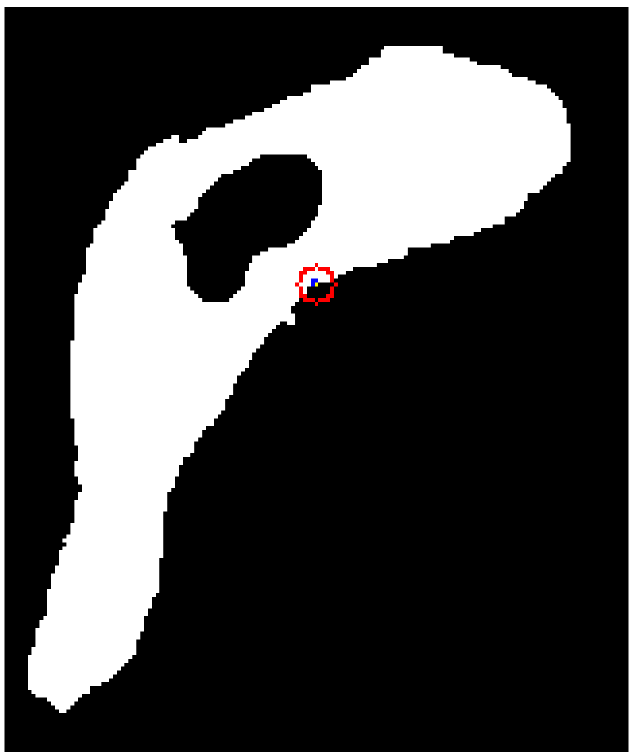


Figure 2: Bad Start Point

The tracking algorithm used in Problem 1 will not successfully track a region's edge for arbitrary start points. The next direction to search is always set as the direction one step counterclockwise from the move just made. Therefore, the algorithm assumes that the first pixel was reached by rastering from upper left, and therefore that the first direction to check is one pixel up from the initial pixel. Thus, if the start point is set somewhere on an underside of the image, the border tracking algorithm will be confused, as is evident in Figure 2. The start point for the tracking algorithm was the yellow point at the center of the red circle. The four blue pixels around the yellow pixel are the border that it tracked, which just form a small 2×2 square including the start pixel.

problem2.py

```
#Athanasios Athanassiadis Jan 2012
from segmentation import *
from ball import make_ball

fig1 = np.fromfile('figure1_problem_set_3',dtype='>i2').reshape((192,161))
fig1 /= 1.0*fig1.max()

#set a new starting point that is on the edge on the other side of the figure
i0,j0 = 71,80
startpoint = np.zeros(fig1.shape)
startpoint[i0,j0] = 1
edge, chain = track_edge(fig1,(i0,j0))
ball = make_ball(5,shell=1,center=(i0,j0),bgsize=(192,161))

#set up colors
r = fig1*(1-edge)*(1-ball)*(1-startpoint)+ball+startpoint
g = fig1*(1-edge)*(1-ball)*(1-startpoint)+startpoint
b = fig1*(1-ball)*(1-startpoint)
imsave('3-2a.png', (r,g,b))
```

3 Problem 3

Depending on the task, there are various ways to fill a region based on the edge in Figure 1c and preserve the hole in the original region. One method is to completely fill the region and then perform an element-wise multiplication of the array containing the original figure, and that containing the filled region. Because the two figures are binary, this will result in another binary image that only retains points common to the two regions. This however seems less useful because it does not give the computer any new information that it did not have from the original figure. This could be used to determine the inner border though: the original region could be element-wise subtracted from the hole-less region. This would produce an image that consisted of the central hole being bright (1) and everything else appearing as background (0). This image could then be used to track the inner edge of the region. This method can be used recursively to track nested regions, and is generalizable when tracking regions in binary images.

Alternately, a second border tracking could be performed to find the border of the interior region. The edge filling algorithm could then take this into account and keep track of ‘entrance’ and ‘exit’ pixels for a region, like the algorithm presented in class. This smart filling would require a bit more computation time, but would give the computer more knowledge of the figure that it is analyzing. Instead of separately tracking the interior boundary, another algorithm could calculate gradients in the original image, and then use the gradient image as a boundary image to perform a similar smart filling algorithm.

4 Problem 4

(a) Original Image (b) Identified Regions

Figure 3: Region Identification

The original image to be segmented is shown in Figure 3a. Initially guessing, a region labeling algorithm scanning from the top left corner should need to allocate 11 regions before adjusting the image using the equivalence table. This is due to the separation of the two edges in the letter H, as well as the bottom curl in the letters C and G. My algorithm processed this image and came up with the 7 distinct regions colored in image 3b. The number of components was determined by the number of entries in the equivalence table that point to themselves. The output, including the equivalence table, are given in `problem4.txt`. As expected, 11 distinct regions were discovered before remapping regions based on the equivalence table.

Note that to run this algorithm, the original image was padded with one pixel on each side, in order to identify regions that potentially include boundary pixels. The padding was removed when returning the labeled image.

`problem4.txt`

```
Number of regions: 7
Equivalence Table:
{0: 0.0, 1: 1.0, 2: 2.0, 3: 2.0, 4: 4.0, 5: 5.0, 6: 6.0, 7: 7.0, 8: 8.0, 9: 7.0,
    10: 1.0, 11: 5.0}
```

`problem4.py`

```python
#Athanasios Athanassiadis Jan 2012
from segmentation import *

fig1 = np.fromfile('figure2_problem_set_3',dtype='>i2').reshape((56,56))
fig1 /= 1.0*fig1.max()

labels,equiv = label_components(fig1)
#number of components is number of entries in the equivalence table equal to
    themselves
#less one because the background (0) is in there
ncomp = len([i for i in equiv if equiv[i]==i])-1

imsave('3-4a.png',fig1)
imsave('3-4b.png',(labels % 3, labels % 4, labels % 5))

```

```
equiv_s = repr(equiv)
with open('problem4.txt','w') as outfile:
    outfile.write('Number of regions: {}\n'.format(ncomp))
    outfile.write('Equivalence Table:\n')
    outfile.write(equiv_s)
```

5 Appendix: Common Code

Common functions used for these problems are contained in `segmentation.py`.

`segmentation.py`

```
#Athanasios Athanassiadis Jan 2012
import numpy as np
from scipy.misc import imsave
import pylab as pl

#establish maps between direction and motion
# 4-connected
#   X 1 X
#   2 X 0
#   X 3 X
map4 = {0: np.array((0,1)),
        1: np.array((-1,0)),
        2: np.array((0,-1)),
        3: np.array((1,0))
        }
# 8-connected
#   3 2 1
#   4 X 0
#   5 6 7
map8 = {0: np.array((0,1)),
        1: np.array((-1,1)),
        2: np.array((-1,0)),
        3: np.array((-1,-1)),
        4: np.array((0,-1)),
        5: np.array((1,-1)),
        6: np.array((1,0)),
        7: np.array((1,1))
        }

#track the edge of a region in an image
#im should be a binary mask
#start is an optional point to start the search
def track_edge(im,start=None):
    edge = np.zeros(im.shape)
    chain = []
    dir = 3

    #if it exists, find first 'on' point in image
    #row,col point encoding
    try:
        on = np.nonzero(im)
        i,j =on[0][0],on[1][0]
        chain = [i,j]
```

```python
        p0 = (i,j)
        point = np.array(p0)
    except:
        print 'No figure found'
        return edge, np.array(chain)

    #unless a start point is specified, use the first 'on' point as the start
    if start!=None:
        p0 = start
        point = np.array(p0)

    #loop until break
    while True:
        #set next direction to look, 1 step counterclockwise from where we came
        checkdir = (dir+3) % 4
        i,j = point + map4[checkdir]

        #if we hit a pixel in the region, move there and add the direction to the list
        if im[i,j]==1:
            dir = checkdir
            point = point+map4[checkdir]
            edge[i,j]=1
            chain.append(dir)

            #if we've hit the start, then end the loop
            if (i,j) == p0:
                break

        #otherwise, check the next dir
        else:
            dir+=1

    return edge,np.array(chain)

#reduce 4-connected edge to be 8-connected where possible
def reduce_edge(chain):
    #keep starting pixel info - it can't be redundant
    chain8 = [chain[i] for i in range(2)]
    i,j = 2,3

    while j<len(chain):
        #based on search direction, the edge is redundant if the difference of the directions gone is a positive odd number
        #also because going counter-clockwise, movement of 0 should be treated as a 4 when paired with a movement of 3, so handle that case individually
        #handle those individually
        if chain[i] == 0 and chain[j]==3:
            newdir = 7
            chain8.append(newdir)
            i+=2
            j+=2
        elif chain[i] == 3 and chain[j] ==0:
            newdir = 6
            chain8.append(newdir)
            i+=1
```

```
                j+=1
            elif (chain[i]-chain[j]>0) and ((chain[i]-chain[j]) % 2 == 1) and
               (chain[i]-chain[j]!=3):
                newdir = chain[i]+chain[j]
                chain8.append(newdir)
                i+=2
                j+=2

            #need to account for the fact that numbers have changed between encodings
            else:
                chain8.append(2*chain[i])
                j+=1
                i+=1

    return np.array(chain8)

#decode 8-connected chain code
def decode_edge8(chain8, shape):
    edge = np.zeros(shape)
    i,j = chain8[:2]
    edge[i,j] = 1

    for dir in chain8[2:]:
        i,j = np.array([i,j])+map8[dir]
        edge[i,j] = 1

    return edge

#write the chain-code
def write_chain(fn, chain):
    with open(fn,'w') as of:
        for i in chain:
            of.write('{} '.format(i))
    print 'Successfully wrote: '+fn

#pad image with zeros
def pad_image(im, pad=1):
    newim = np.zeros(np.array(im.shape) + 2*pad)
    newim[pad:-pad,pad:-pad] = im.copy()

    return newim

#find and label connected components
def label_components(im):
    #initially pad label image so that we can the first row without error
    l_im = pad_image(np.zeros(im.shape))
    equiv = {0:0.0}

    #raster through image
    for i in range(im.shape[0]):
        for j in range(im.shape[1]):
            if im[i,j]==1:
                #find if any neighbors have already been labeled
                #if so, then take the smalles of all neighboring labels
                #otherwise, create a new label
                nearby = []
```

```
                for dir in [1,2,3,4]:
                    dirval = l_im[i+map8[dir][0]+1,j+map8[dir][1]+1]
                    if dirval > 0:
                        nearby.append(dirval)
                if nearby==[]:
                    newval = max(equiv)+1
                    nearby.append(newval)
                else:
                    newval = min(nearby)

                l_im[i+1,j+1] = newval

                #adjust equivalence table, but maintain previous corrections to the table
                for n in nearby:
                    if n in equiv:
                        equiv[n] = min(newval,equiv[n])
                    else:
                        equiv[n] = newval

    #adjust image according to equivalence table
    #go backwards through the keys so that chains of equivalences are handled properly
    for key in equiv.keys()[::-1]:
        l_im[l_im==key] = equiv[key]

    return l_im[1:-1,1:-1],equiv    #return without padding
```