Assignment 3 – TDT4195

Mikkel Svagård

Theory:

1. Segmentation

The goal of segmentation is to separate and extract objects in a picture, such as background and object separation. This is done by looking for intensity changes and similarities, creating boundaries and regions for future separation.

Segmentation can be shown difficult as one could only be interested in some object. Identifying these without overanalysing the image is one difficult decision, whereas knowing which objects that are desired is another, as the picture can be lacking in contrast and filled with noise.

2. Split and Merge

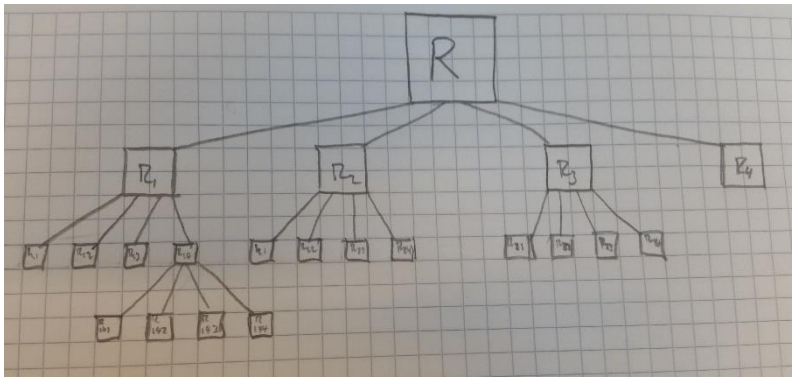
Region Splitting splits the image into regions wherever a set condition of homogeneity is not satisfied. Through iterations the image will eventually be split into homogeneous regions. Further merging these regions with neighbours with the most identical properties, also done in iterations, creates the desired result of a segmented picture.

0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	1	1
0	0	1	0	0	0	1	1
0	0	1	1	1	1	1	1
0	0	1	1	1	1	1	1
0	0	0	0	1	1	1	1
0	0	0	0	1	1	1	1

0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	1	1
0	0	1	0	0	0	1	1
0	0	1	1	1	1	1	1
0	0	1	1	1	1	1	1
0	0	0	0	1	1	1	1
0	0	0	0	1	1	1	1

Result:

0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	1	1
0	0	1	0	0	0	1	1
0	0	1	1	1	1	1	1
0	0	1	1	1	1	1	1
0	0	0	0	1	1	1	1
0	0	0	0	1	1	1	1



3. Morphological Operations

Operator H is said to be linear if

$H[a_i f_i(x,y) + b_j g_j(x,y)] = a_i [H f_i(x,y)] + b_j H[g_j(x,y)]$

In other words, this means that applying an operation on two different pictures and then assembling them would give the same result as assembling them first, then doing the operation.

Since Erosion removes parts of the image where the structure element won't fit, it would not always produce the same result when applying the operation before or after assembling. On example of this is when the two images get fully blacked out by the erosion, while assembled some parts are being kept. Hence, erosion together with closure and opening are not linear operations.

Dilation on the other hand is a linear operation is linear, as the result of dilation before or after assembling the pictures would be the same.

4. Opening and Closing

Opening and closing are operators derived from the fundamental operations erosion and dilation. Erosion is basically to erode away the foreground's regional boundaries, while dilation is enlarging these boundaries.

Closure is a procedure where first a dilation is done, followed by an erosion, where both operations uses the same structure element. This causes preservation of background elements that can contain the structure element, hence filling out the remaining regions that cannot.

Opening is the opposite of closure, by first doing erosion followed by dilation, again with the same structure element. Opening preserves the foreground elements that can contain the structure element.

Doing closure or opening of images several times won't make a difference, as the regions that cannot contain the structure element already have been dealt with. This is known as idempotent operations.

5. Dilation $A \oplus B$

A:

0	0	0	0	0	0
0	1	0	0	1	0
0	1	1	1	1	0
0	0	1	0	1	0
0	0	1	0	0	0
0	0	0	0	0	0

B:

1
1
1

Result: Grey is old foreground pixels, black are the added ones.

0	1	0	0	1	0
0	1	1	1	1	0
0	1	1	1	1	0
0	1	1	1	1	0
0	0	1	0	1	0
0	0	1	0	0	0

→

0	1	0	0	1	0
0	1	1	1	1	0
0	1	1	1	1	0
0	1	1	1	1	0
0	0	1	0	1	0
0	0	1	0	0	0

2. Programming:

1. Segmentation

(A)

I created the following algorithm for calculating the threshold. My program converts the image to a matrix, and further adds all the pixels into one list. Numpy's mean function found the two means pretty simple. As I am working with a grayscale image of depth 255 I'm starting with the initial threshold of 128. The dT was set to 1 as I want an accurate estimate of the true mean.

```
def get_threshold(s,T=128, dT=1):
  #Finds the threshold in the set by using numpys mean function
  Tp = 0
  while(abs(T-Tp)>dT):
      m1, m2 =  np.mean([p for p in s if p>T]),np.mean([p for p in s if p<=T])
      Tp, T = T,int((m1+m2)/2)
  print("Threshold ",T)
  return T
```

(B)

I create the following algorithm for returning the segmentation of the image. My program takes in the image in the form of a matrix, a structuring element n, a threshold T and a seed point.

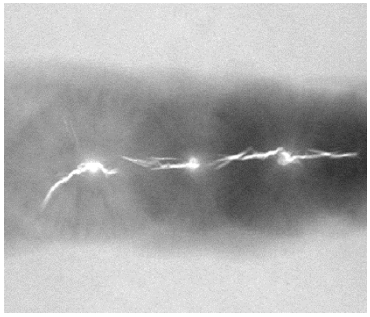
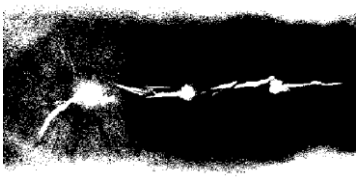
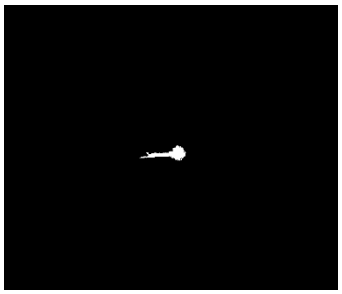
The algorithm works as a queue, where all coordinates set to one gets added to the queue to further be dealt with.

```
def segment_matrix(m,n,T,seed):
  #Segments the matrix with structure n from seed with treshold = T
  M = [[0 for y in range(len(m[0]))]for x in range(len(m))]
  ln,ln2= len(n),int(len(n)/2)
  print("seed: ",m[seed[0]][seed[1]])
  q = [seed]
  i = 0
  while(i<len(q)):
    start_x,start_y = (q[i][0]-ln2),(q[i][1]-ln2)
    for a in range(ln):
      for b in range(ln):
        if(n[a][b]==1):
          x,y = start_x + a, start_y + b
          if(m[x][y]>T):
            M[x][y]=255
            p = (x,y)
            if p not in q: q.append(p)
    i+=1
  return M
```

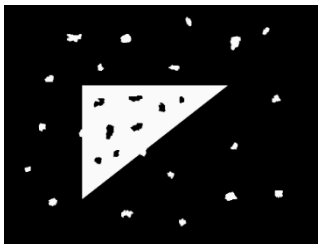
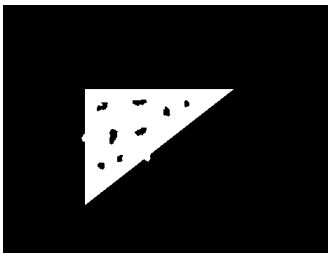
By opening my images in Paint I found the seed points that I wanted.

The following pictures first got translated to binary with the threshold algorithm found in a, then segmented at the set seed point with the Neumann Neighbourhood:

Seed: (290,250)

Seed: (200,150)

2. Mathematical Morphology

A)

I created the following algorithms for erosion and dilation;

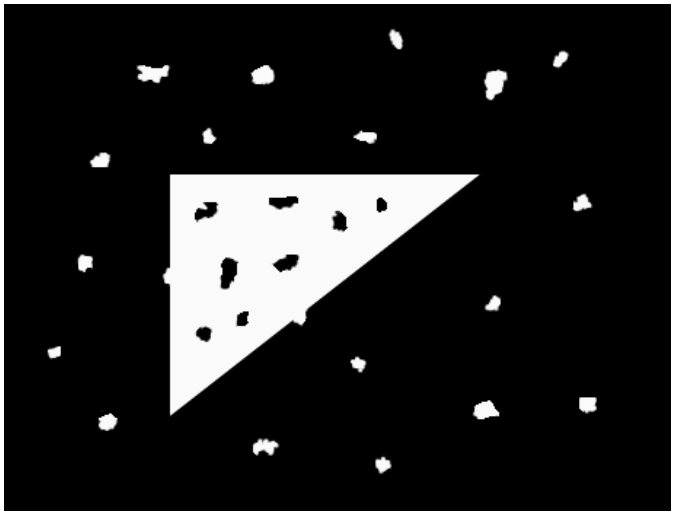
The two functions take in the matrix and the structuring element k.

I also made a functions for creating Disc or Box structuring elements of a desired width.

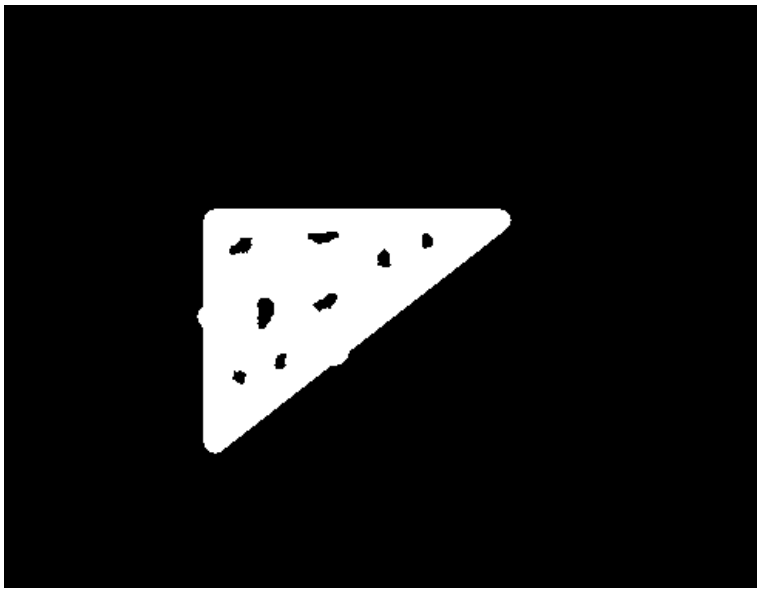
By using the two operations I first did an opening of the picture with a Disc of width diameter 15. This removed all the noise. Then I did a closure, too with a disc of width 15, so that all the holes in the figure got filled. The result looks like this:

```
def perform_dilation(m,k): #Applies kernel through dilation on matrix, returns matrix
  M = [[0 for y in range(len(m[0]))]for x in range(len(m))]
  lx,ly,lk,lk2= len(m),len(m[0]),len(k),int(len(k)/2)
  for x in range(lk2,lx-lk2):
    for y in range(lk2,ly-lk2):
      if(m[x][y]>128):
        #If reference node is high, sets all surrounding to highk
        for a in range(lk):
          for b in range(lk):
            pos_x,pos_y = x-lk2+a,y-lk2+b
            if k[a][b]==1:
              M[pos_x][pos_y]=255
  return M

def perform_erosion(m,k,):
  #Applies kernel through dilation on matrix, returns matrix
  M = [[0 for y in range(len(m[0]))]for x in range(len(m))]
  lx,ly,lk,lk2= len(m),len(m[0]),len(k),int(len(k)/2)
  for x in range(lk2,lx-lk2):
    for y in range(lk2,ly-lk2):
      if(m[x][y]>128):
        M[x][y]=255
        while(True):
          #If any of the surrounding pixels are 0, the reference get set to zero.
          #The while causes the algorithm to break after the reference-node gets set to zero
          for a in range(lk):
            for b in range(lk):
              pos_x,pos_y = x-lk2+a,y-lk2+b
              if k[a][b]==1 and m[pos_x][pos_y]==0:
                M[x][y]=0
                break
          break
  return M
```

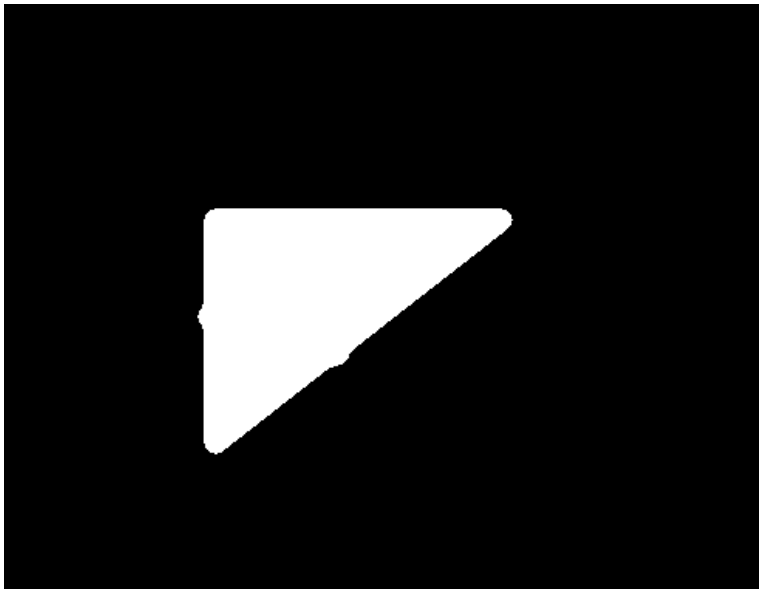
Original image



After opening:



After both opening and closure



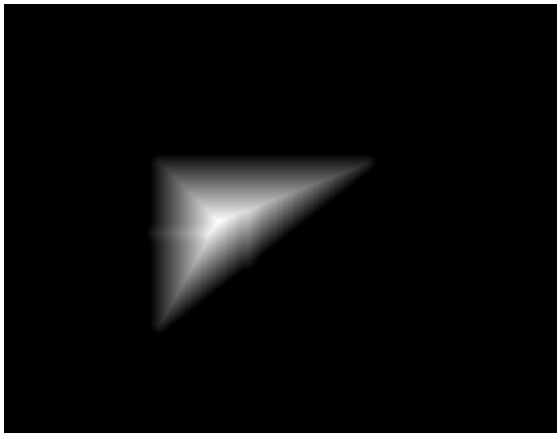
(B)

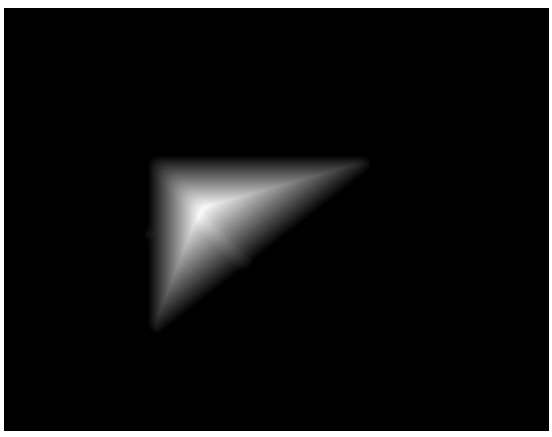
By tweaking the erosion algorithm, a little I made a distance transform function.

By not setting the pixels to 0, but to the current iteration i, I could simply multiply all pixels with 255/i after the image was supposed to be blank to get a visual impression of the distance. The still_pixels returns True if there are any pixels above a value of 200.

```
def perform_erosion2(m,k,i): #Applies kernel through dilation on matrix, returns matrix
  M = copy.copy(m)
  lx,ly,lk,lk2= len(M),len(M[0]),len(k),int(len(k)/2)
  for x in range(lk2,lx-lk2):
    for y in range(lk2,ly-lk2):
      if(m[x][y]==255):
        #Cheks wheter the reference pixel is still high
        while(True):
          #If any of the surrounding pixels are 0, the reference get set to zero.
          for a in range(lk):
            for b in range(lk):
              pos_x,pos_y = x-lk2+a,y-lk2+b
              if k[a][b]==1 and m[pos_x][pos_y]<i:
                #sets the pixel to the current iteration if its supposed to be 0
                M[x][y]=i
                break
          break
  return M

def still_pixels(m): #returns true if there are any pixels above 200
  for l in m:
    for p in l:
      if(p>200):
        return True
  return False

def distance_transform(m,k): #applies distance transofmraton
  M= copy.copy(m)
  i=0
  while(still_pixels(M)): #Applies transformation till all pixels are belove gone
    i+=1
    print("Performing erosion: ",i)
    M = perform_erosion2(M,k,i)

  value = 255/i #scales the pixels
  for x in range(len(M)):
    for y in range(len(M[0])):
      M[x][y]=M[x][y]*value
  return M
```

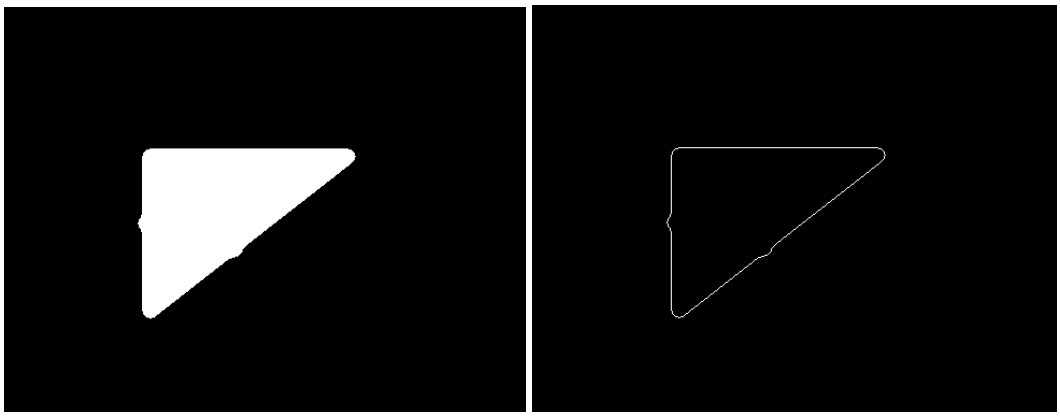
My result looked like this: Disc Box



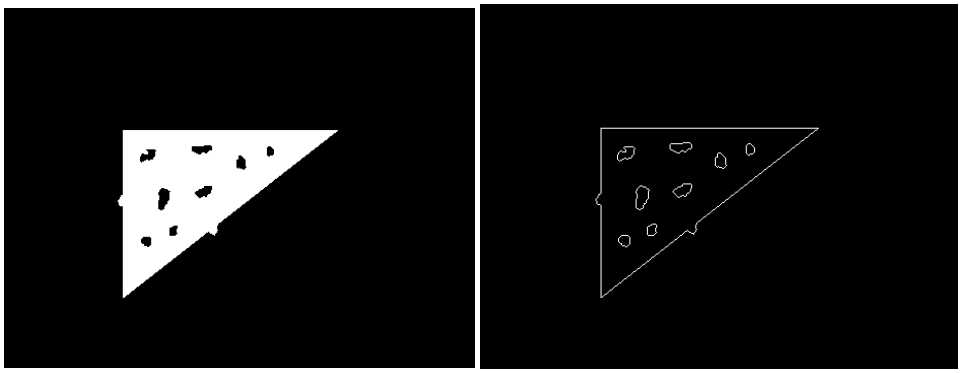


C)

I created here a function that subtracted one matrix from another. By first doing an erosion, then substracting the result from the original this effect was created:

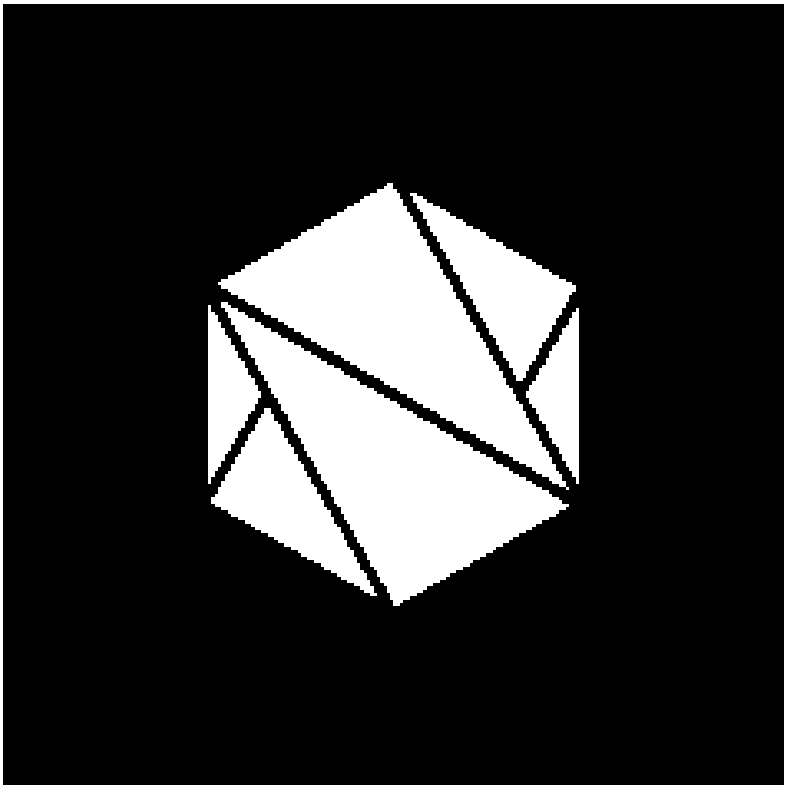


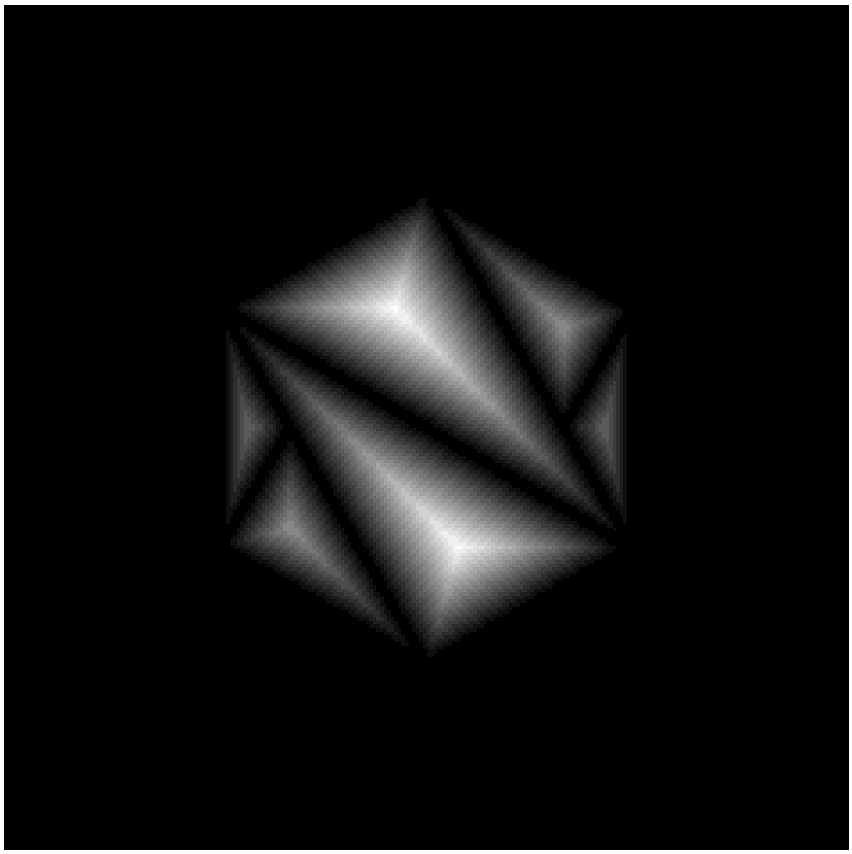
I also tried it with the result of Task 1b)



3. Exam Question

The distance transform is an operation typically applied on a binary image and creates a greyscale image where each foreground pixel shows the distance to the closest boundary pixel. One inefficient way of calculating the distance transform is to use erosion. The following picture have been transformed as such. How many iterations of erosion did it approximately take to complete this transformation, where the structuring element was a 3x3 box filled with 1's? The pictures have the dimensions 236*236. Explain your reasoning:





Answer:

Erosion removes one pixel of the boundaries each iteration. Hence we basically need to know how many iterations of erosion it takes to completely remove these objects.

The bright spots on the big triangular shows the last spot remaining. Calculating the shortest distance from these spots to the boundaries of the figures, knowing the dimensions of the picture to be 236*236, it shows that it's approximately a little short of 20 pixels. Therefor it takes almost 20 iterations. The correct answer in this case is 17.