Cp467 A6 report

By Phoebe Schulman, Afaq Shad, Arvind Sahota

Purpose

The purpose of this project was to create a program that thins an image.

Important Concepts

A “Binary image” consists of only black and white pixels. Where a value of 0 represents black pixels and 255 represents white.

While a Binary Large Object (**BLOB**) is a group of connected pixels inside of a binary image.

Method 1: **Skeletonization** is a process that makes a BLOB very thin (like a skeleton). It will aim to keep the general shape and connectivity of the BLOB but make it only 1 pixel in width *(3).*

Method 2: The **Zhang-Suen thinning algorithm** is used to thin binary images *(2).*

Implementation of method 1 and 2

Method 1: By using the skeletonize API *(1).*

```
from skimage.morphology import skeletonize
```

```
thinned_image_1 = skeletonize(blobs)
```

Method 2: modifying a zhangSuen function which has a binary image as its input and output. Additionally, it calls the helper functions “neighbours” and “transitions” *(2).*

It takes the input of a binary image (as a N x M array), where black pixels are represented with a 1 and white pixels are 0.

Any pixel P1 can have eight **neighbours** (excluding pixels near the boundary of the image):

P9	**P2**	**P3**
P8	**P1**	**P4**

P7 P6 P5

The neighbors can be found by looking at the rows and columns adjacent to P1.

```
def neighbours(x, y, image):
    '''Return 8-neighbours of point p1 of picture, in order'''

    x1, y1= x+1, y-1
    x_1, y_1 = x-1, y+1

    p2 = image[y1][x]
    p3= image[y1][x1]

    p4 = image[y][x1]
    p5=image[y_1][x1]

    p6=image[y_1][x]
    p7=image[y_1][x_1]

    p8 = image[y][x_1]
    p9 = image[y1][x_1]

    n = [p2,p3,p4,p5,p6,p7,p8,p9]

    return n
```

Let A(P1) = number of **transitions from white to black** (0->1), in the sequence P2,P3,P4,P5,P6,P7,P8,P9,P2.

```
def transitions(neighbours):
    n = neighbours + neighbours[0:1]    # P2, ... P9, P2

    s = 0 #sum
    for n1, n2 in zip(n, n[1:]):
        if (n1, n2) == (0, 1) : #white -> black
          s+=1
    return s
```

Let B(P1) = number of black pixel neighbours of P1 = sum(P2, P3, P4, …, P9).

Step 1 conditions for selecting black points to remove

(0) **The pixel is black** and has eight neighbours
(1) **2 <= B(P1) <= 6**
(2) **A(P1) = 1**
(3) At least one of P2 and P4 and P6 is white -> P2*P4*P6 = 0
(4) At least one of P4 and P6 and P8 is white -> P4*P6*P8 =0

After traversing the image, pixels that satisfy all these conditions are set to white.

```python
def zhangSuen(image):

    changing1 = changing2 = [(-1, -1)]
    while changing1 or changing2:

        #==================
        # Step 1
        changing1 = []

        for y in range(1, len(image) - 1): #traversal
            for x in range(1, len(image[0]) - 1):

                #neighbours
                P2,P3,P4,P5,P6,P7,P8,P9 = n = neighbours(x, y, image)
```

```python
#Conditions
cond0=(image[y][x] == 1)
cond1 = (2 <= sum(n) <= 6)
cond2 =(transitions(n) == 1)

cond3 =(P2 * P4 * P6 == 0)
cond4 =(P4 * P6 * P8 == 0)


if (cond0 and cond1 and  cond2 and cond3 and cond4):
    changing1.append((x,y))
```

```python
for x, y in changing1: #set to white
  image[y][x] = 0
```

Step 2 conditions for selecting black points to remove

(0) The pixel is black and has eight neighbours
(1) 2 <= B(P1) <= 6
(2) A(P1) = 1
(3) At least one of P2 and P4 and P8 is white -> P2*P4*P8 =0
(4) At least one of P2 and P6 and P8 is white -> P2*P6*P8 = 0

Again, after traversing the image, pixels that satisfy all these conditions are set to white.

Note: step 1 is similar to step 2, except for the conditions 3 and 4. Step 1 checks for the right and bottom side of the pixel. While step 2 checks for the top and left side.

P9	**P2**	P3
P8	P1	**P4**
P7	**P6**	P5

P9	**P2**	P3
P8	P1	**P4**
P7	P6	P5

```
    #=================
    # Step 2
    changing2 = []

    for y in range(1, len(image) - 1):#traversal
        for x in range(1, len(image[0]) - 1):

            #neighbours
            P2,P3,P4,P5,P6,P7,P8,P9 = n = neighbours(x, y, image)


            #Conditions
            cond0=(image[y][x] == 1)
            cond1 = (2 <= sum(n) <= 6)
            cond2 =(transitions(n) == 1)

            cond3 =(P2 * P4 * P8 == 0 )
            cond4 =(P2 * P6 * P8 == 0)


            if (cond0 and cond1 and  cond2 and cond3 and cond4):
                changing2.append((x,y))


    for x, y in changing2: #set to white
      image[y][x] = 0


return image
```

If any pixels were set in this round from either step, then all steps are repeated until no pixels are changed.

Final results

Input: 3 different BLOB examples.

```
#3 example inputs
print("3 examples:\n\n")

blobs = data.binary_blobs(100, blob_size_fraction=.2,  volume_fraction=.40, seed=15)
main(blobs)


blobs = data.binary_blobs(100, blob_size_fraction=.2, volume_fraction=.35, seed=1)
main(blobs)

blobs = data.binary_blobs(50, blob_size_fraction=.2, volume_fraction=.35, seed=2)
main(blobs)
```

Output: original BLOB, output of method 1, and output of method 2.

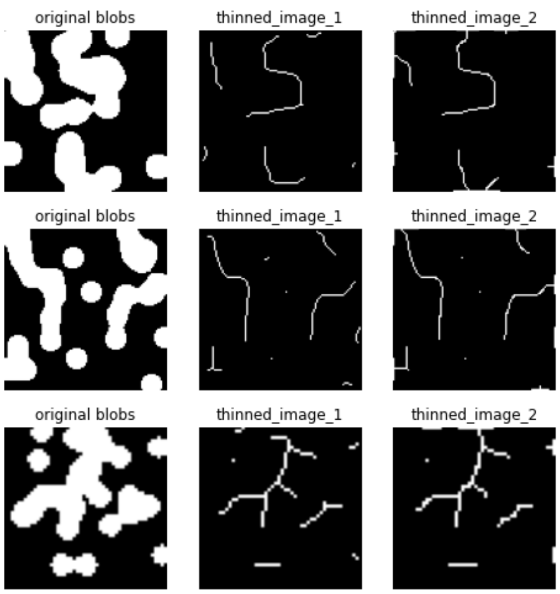
```
def main(blobs):

    #display
    figure, axes = plt.subplots(1, 3, figsize=(8, 5), sharex=True, sharey=True)
    display = axes.ravel()

    #display orig
    display[0].imshow(blobs, cmap=plt.cm.gray)
    display[0].set_title('original blobs')
    display[0].axis('off')
```

```
#===========================
#thinning -version 1 - with api #(does not mutate blobs var)
thinned_image_1 = skeletonize(blobs)

#display thinned 1
display[1].imshow(thinned_image_1, cmap=plt.cm.gray)
display[1].set_title('thinned_image_1')
display[1].axis('off')
```

```
#===========================
#thinning -version 2 - with function #(does  mutate blobs var)
thinned_image_2 =zhangSuen(blobs)

#display thinned 2
display[2].imshow(thinned_image_2, cmap=plt.cm.gray)
display[2].set_title('thinned_image_2')
display[2].axis('off')
```

Visually, we can display 3 rows (of input blobs) and 3 columns (the original output, method1 output, and method2 output).

```
3 examples:
```



In conclusion, method1 (Skeletonization) and method2 (Zhang-Suen thinning algorithm) are both thinning the original image well. But it's hard to determine if one is “better” than the other.

Sources:

1. Skeletonize by scikit-image
   https://scikit-image.org/docs/dev/auto_examples/edges/plot_skeleton.html
2. Zhang-Suen thinning algorithm by Rosetta Code
   https://rosettacode.org/wiki/Zhang-Suen_thinning_algorithm#Python
3. Skeletonization in Python using OpenCV by Neeramitra Reddy
   https://medium.com/analytics-vidhya/skeletonization-in-python-using-opencv-b7fa16867331
4. Zhang-Suen Thinning Algorithm, Java Implementation by Nayefreza
   https://nayefreza.wordpress.com/2013/05/11/zhang-suen-thinning-algorithm-java-implementation/