EE6222-Machine Vision Assignment 1

Thinning / Skeletonization

Student Name: Wei Zhifeng
Student ID: G2002825F

Feature extraction is an important topic in image processing. Among these, thinning, which is a process to obtain the skeleton of a binary object (0 or 1 image) region.

Obtaining binary images

The image is composed of three-color channels (RGB). After being converted into a gray image, each pixel will be changed to a number between 0-255. 0 represents the black and 255 represents the white.

Binary image is the image only contains 0 and 1. 0 is the background and 1 is the object.

To perform thinning, we should first convert the input image to the binary image. Therefore, we use OTSU algorithm to calculate an adaptive threshold.

If we have L grey level, the number of pixels in grey level i is N_i, therefore:

$$N = N_1 + N_2 + \cdots + N_{L-1} \tag{1}$$

The probability of grey level i is:

$$P_i = \frac{N_i}{N} \tag{2}$$

We assume the threshold is grey level T, so we divide the image into wo group: w_0 and w_1,representing background and object respectively. The probability of them is:

$$w_0 = \sum_{i=0}^{T} P_i \tag{3}$$

$$w_1 = 1 - w_0 \tag{4}$$

The mean gray value of the image and w_0 and w_1 is:

$$\mu_T = \sum_{i=1}^{L-1} i * p_i \tag{5}$$

$$\mu_0 = \frac{\sum_{i=1}^{T} i * p_i}{w_0} \tag{6}$$

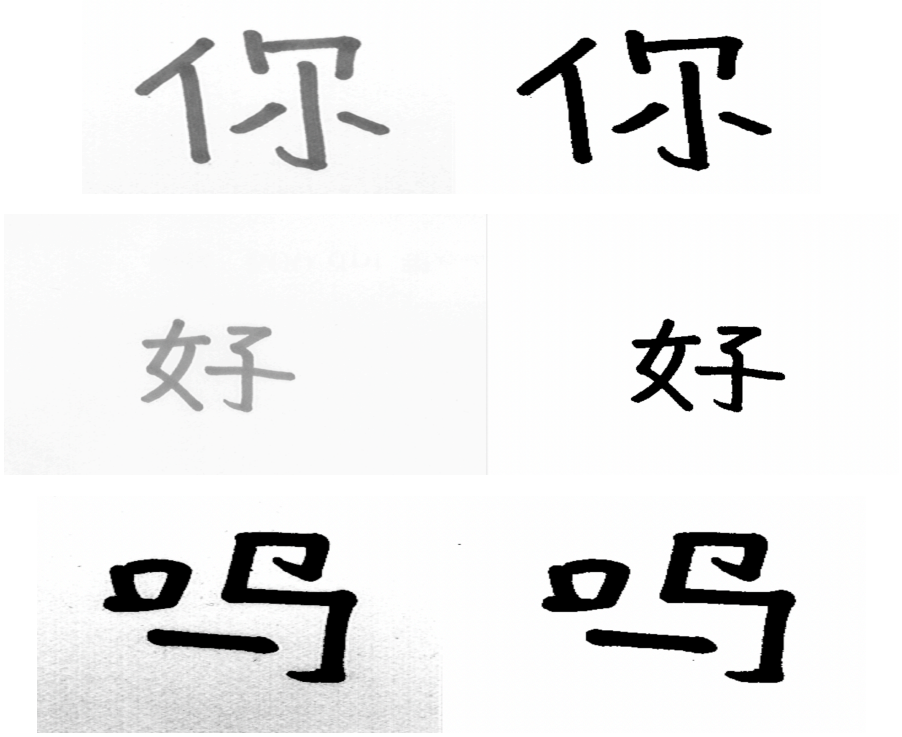
$$\mu_1 = \frac{\mu_T - \sum_{i=1}^{T} i * p_i}{w_1} \tag{7}$$

Therefore, we can get the best threshold by:

$$\sigma^2 = w_0 * (\mu_T - \mu_0)^2 + w_1 * (\mu_1 - \mu_T)^2 \tag{8}$$

$$T = \max_{0 \le T \le L-1} \{\sigma^2\} \tag{9}$$

Applying this algorithm to the images, we can obtain the binary image. We demonstrate it by converting the 1 to 255 in grey level.



Original_image. Binary_image

Figure 1.

From the conversion result, we could see the OTSU algorithm works well and find a suitable threshold for the image.

The OTSU algorithm finds a proper threshold that maximum the distance between classes and minimum the distance inside class. Therefore, this threshold is adaptive to different image and suitable for binary threshold.
From the result, we could also find out that this algorithm performs well in these images.

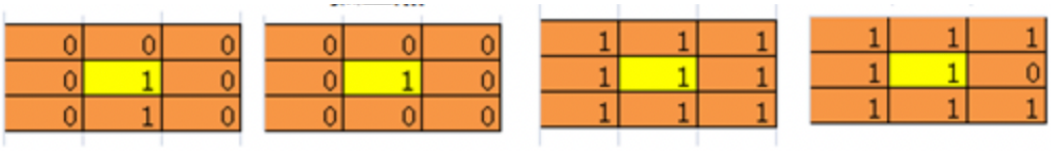
Code:

```
def OTSU(gray):
    hist = cv2.calcHist([gray], [0], None, [256], [0, 256])  # 256*1 grey histogram
    gray_size = gray.size  # pixel num
    k = 0  # initial threshold
    best_k = 0  # best threshold
    best_M = 0  # measure the threshold

    p = []  # probability array

    # for k in range(30,150):
    for i in range(len(hist)):
        p.insert(i, hist[i][0] / gray_size)  # add the probability to the array

    for k in range(30, 200):# range of k
        u = 0
        u_t = 0
        σ2_0 = 0
        σ2_1 = 0
        σ2_t = 0
        sum_0 = np.sum(hist[0:k + 1:], axis=0)

        sum_1 = np.sum(hist[k + 1:256:], axis=0)

        w_0 = np.sum(p[0:k + 1:])
        w_1 = np.sum(p[k + 1:256:])

        for i in range(k + 1):
            u = i * p[i] + u

        for i in range(len(hist)):
            u_t = i * p[i] + u_t

        u0 = u / w_0
        u1 = (u_t - u) / w_1

        for i in range(k + 1):
```

```
            σ2_0 = (p[i] / w_0) * np.square(i - u0) + σ2_0
        for i in range(k + 1, 256):
            σ2_1 = (p[i] / w_1) * np.square(i - u1) + σ2_1
        for i in range(256):
            σ2_t = p[i] * np.square(i - u_t) + σ2_t

        σ2_w = w_0 * σ2_0 + w_1 * σ2_1
        σ2_b = w_0 * w_1 * np.square(u1 - u0)

        M = σ2_b / σ2_t
        if M > best_M:
            best_M = M;
            best_k = k;
    return best_M, best_k
```

2-step thinning

After getting the binary images, we could perform 2-step thinning in these images. For an image, we want to remove the redundant boundary points and keep some important points such as connection points, end points, and isolated points.



end points isolated points. Connection point. Connection point

Figure 2.

Therefore, the process of 2 step thinning algorithm is explain as follow:

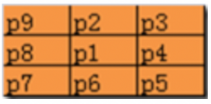


Figure 3.

1. For each boundary pixel in the image (p1 = 1), if it satisfies the following requirement, we will flag this point for removal.
   - $2 \leq N(p1) \leq 6$
   - $S(p1) = 1$
   - $P2 \,\&\, P4 \,\&\, P6 = 0$
   - $P4 \,\&\, P6 \,\&\, P8 = 0$

   N(P1) is the number of non-zero neighbors of P1.

   S(P1) is the number of 0 to 1 transitions in the order sequence if P2,P3,P4,P5⋯P8, P9.

2. Remove the flagged point in step 1and then flag the boundary pixel satisfies the following requirement.
   - $2 \leq N(p1) \leq 6$
   - $S(p1) = 1$
   - $P2 \,\&\, P4 \,\&\, P8 = 0$
   - $P2 \,\&\, P6 \,\&\, P8 = 0$
3. Remove the flagged point after step 2, and then repeat these 2 steps until no more pixels are flagged.

N(P1) =0 is an isolated point, N(P1) =1 means this point is the end of skeleton and delete point when N(p1) =7 will cause the erosion. Therefore, we should keep these points.

When $S(p1) > 1$,the point might be the connection point and the region is more than 1 pixel wide. we should retain it. If $S(p1)=1$, it might be a boundary point that could be removed.

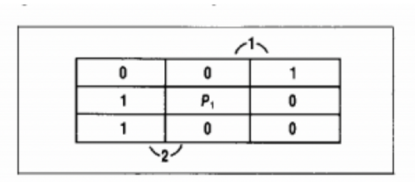


Figure 4.

For condition 3 and 4 in step1 and 2, we could remove the SE and NW boundary point respectively.

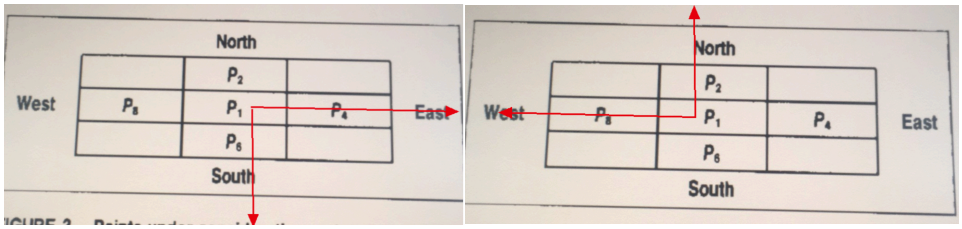
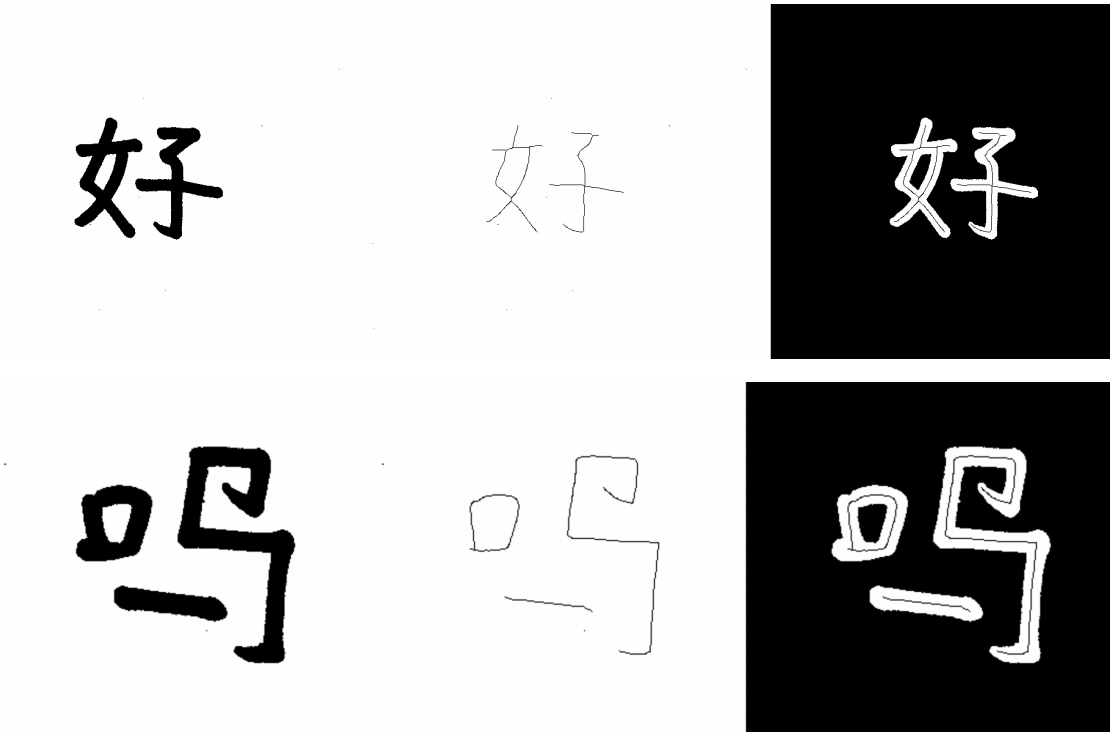


Figure 5.

Applying this algorithm to the images, we could get the skeleton of these images.



Binary_image. Thinning_image Combined_image

Figure 6.

From the result, we could find that the 2-step thinning algorithm works well and extracts the skeleton of the images. The important points are retained, and other redundant points are deleted.

Code:

```
def thining(image):
    """
     repeat the 2-
step thining until no element to remove
     """
    changing1 = changing2 = [(-1, -1)] # initial statue
    while changing1 or changing2:
        # Step 1  flag the points to be remove
        changing1 = []
        for x in range(1, len(image) - 1):  # row
            for y in range(1, len(image[0]) - 1):  # col
                # Traverse the matric
                P2, P3, P4, P5, P6, P7, P8, P9 = n = find_neighbour
s(x, y, image)
                if (image[x][y] == 1 and  # (Condition 0 p1
== 1)
                        P4 & P6 & P8 == 0 and  # Condition 4
                        P2 & P4 & P6 == 0 and  # Condition 3
                        freq(n) == 1 and  # Condition 2
```

```
                    2 <= sum_neighbour(n) <= 6):  # Conditi
on 1 non-zero>2&&non-zero<6
                changing1.append((x, y))  # flag the remo
ve_point
        # remove the flag point
        for x, y in changing1:
            image[x][y] = 0
        # Step 2  flag the points to be removed
        changing2 = []
        for x in range(1, len(image) - 1):
            for y in range(1, len(image[0]) - 1):
                P2, P3, P4, P5, P6, P7, P8, P9 = n = find_neighbour
s(x, y, image)
                if (image[x][y] == 1 and  # (Condition 0)
                        P2 & P6 & P8 == 0 and  # Condition 4
                        P2 & P4 & P8 == 0 and  # Condition 3
                        freq(n) == 1 and  # Condition 2
                        2 <= sum_neighbour(n) <= 6):  # Conditi
on 1
                    changing2.append((x, y))
        # remove the point in step 2 until no point
to be removed
        for x, y in changing2:
            image[x][y] = 0
    return image
```

Medial Axis Transformation-based thinning

Medial axis transform (MAT) is another thinning algorithm based on distance transform.

Distance transform will produce a gray level image in which the intensities of the image point inside foreground represents the distance to the closet boundary in vertical or horizontal direction.

After Distance transform, the process of MAT is:

1. For each point in foreground, we find its closet neighbor in boundary.
2. If this point has more than one such neighbors, it is belong to the medial axis of this image.

The distance transform is like:

	0	1	2	3	4	5	6	7	8	9
0	0	0	0	0	0	0	0	0	0	0
1	0	1	1	1	1	1	1	1	1	0
2	0	1	1	1	1	1	1	1	1	0
3	0	1	1	1	1	1	1	1	1	0
4	0	1	1	1	1	1	1	1	1	0
5	0	1	1	1	1	1	1	1	1	0
6	0	0	0	0	0	0	0	0	0	0

Input_array

Figure 7.

	0	1	2	3	4	5	6	7	8	9
0	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000
1	0.00000	1.00000	1.00000	1.00000	1.00000	1.00000	1.00000	1.00000	1.00000	0.00000
2	0.00000	1.00000	2.00000	2.00000	2.00000	2.00000	2.00000	2.00000	1.00000	0.00000
3	0.00000	1.00000	2.00000	3.00000	3.00000	3.00000	3.00000	2.00000	1.00000	0.00000
4	0.00000	1.00000	2.00000	2.00000	2.00000	2.00000	2.00000	2.00000	1.00000	0.00000
5	0.00000	1.00000	1.00000	1.00000	1.00000	1.00000	1.00000	1.00000	1.00000	0.00000
6	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000

Distance matric

Figure 8.

From the image, we can find out that the DT calculates the distance to the nearest neighbor boundary.

The MAT result is shown as below:

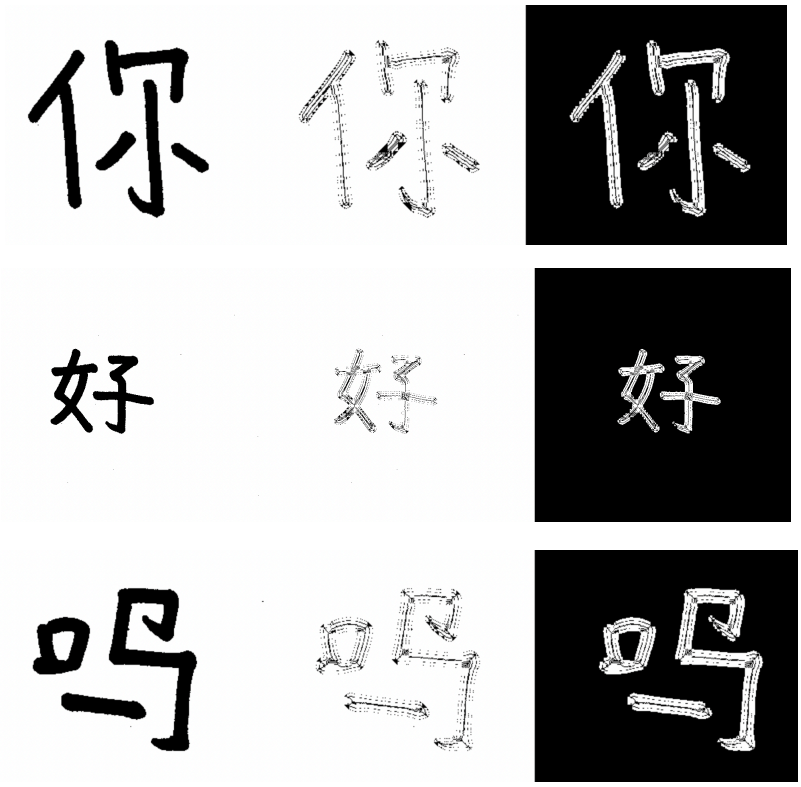
	0	1	2	3	4	5	6	7	8	9
0	0	0	0	0	0	0	0	0	0	0
1	0	1	0	0	0	0	0	0	1	0
2	0	0	1	0	0	0	0	1	0	0
3	0	0	0	1	1	1	1	0	0	0
4	0	0	1	0	0	0	0	1	0	0
5	0	1	0	0	0	0	0	0	1	0
6	0	0	0	0	0	0	0	0	0	0

MAT matric

Figure 9.

We could find that MAT extracts the skeleton of the input array.

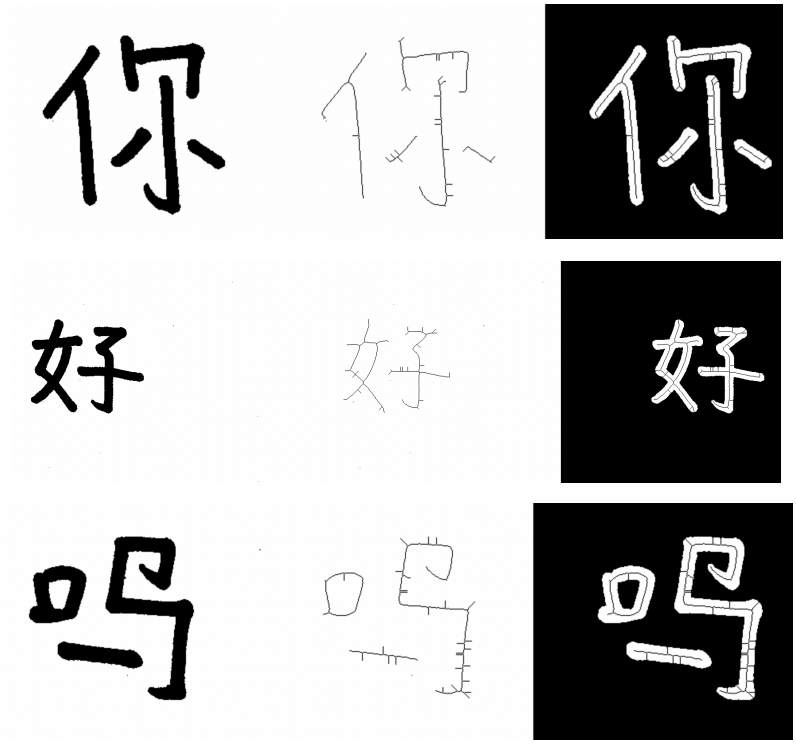
We then perform DT-based MAT to these images. The result is shown as follow:



Binary_image. MAT_image Combined_image

Figure 10.

From the result, we could find that the skeleton is retained after mat transform. However, there are still some redundant points are kept. In order to remove these points. We use the official mat function(morphology.medial_axis) to get the mat image.
The result is shown as follow:



Binary_image. MAT_image Combined_image

Figure 11.

From the result, we could see the official MAT is better than my implement of MAT, but there are still some boundary points retained.

Code：

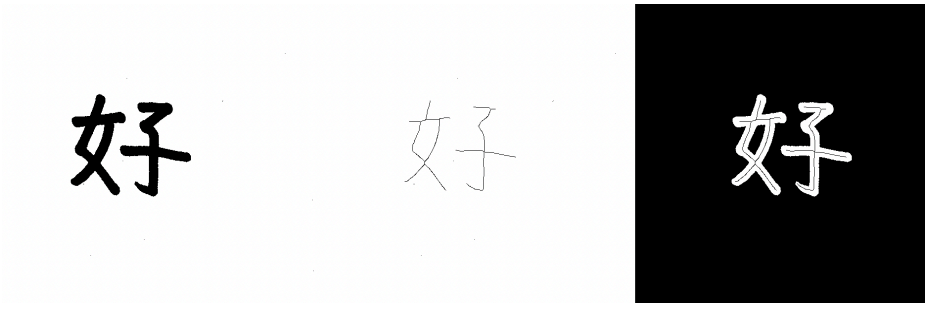
```
def medial_axis(image):
    distance_img=copy.deepcopy(image).astype(float)  #distance image
    transform_img = copy.deepcopy(image) #image after transform
    a=np.argwhere(image == 0) # boundary
    b=np.argwhere(image == 1)  # object
    ''''
     DT
     '''
    for p in b:
        x=p[0]
        y=p[1]
        #dis_x =np.sqrt(np.sum(np.asarray(a- p) ** 2, axis=1))  #  nearest boudary
        dis_x = np.sqrt(np.sum(np.asarray(a[np.where(a[:,0] == x)] - p) ** 2, axis=1))  # nearest boudary
```

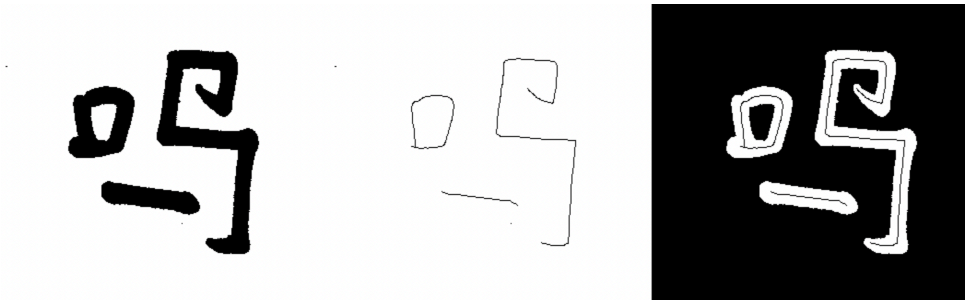
```
        dis_y = np.sqrt(np.sum(np.asarray(a[np.where(a[:,1] == y)] - p) ** 2, axis=1))
        if dis_x.size == 0 and dis_y.size == 0:
        #     transform_img[x][y] = 1
            continue
        elif dis_x.size ==0:
        #     transform_img[x][y] = 1
            nearset_nei =np.min(dis_y)
        elif dis_y.size == 0:
            nearset_nei = np.min(dis_x)
        else:
            nearset_nei=min(np.min(dis_x),np.min(dis_y))
        distance_img[x][y] = nearset_nei  #dis
    ''''
          mat
     '''
    for p in b:
        x = p[0]
        y = p[1]
        dis = np.sqrt(np.sum(np.asarray(a - p) ** 2, axis=1))  # nearest boudary
        nearst_counts = np.sum(dis == distance_img[x][y])
        # nearst_counts +=np.sum(dis_y == nearset_nei)
        if nearst_counts > 1 or mat_sum(x,y,distance_img) == 1:
            transform_img[x][y] = 1
        else:
          # if mat_sum(x, y, distance_img) > 2:
          #     transform_img[x][y] = 1
          # else:
              transform_img[x][y] = 0

    return   transform_img.astype(int), distance_img
```

Result

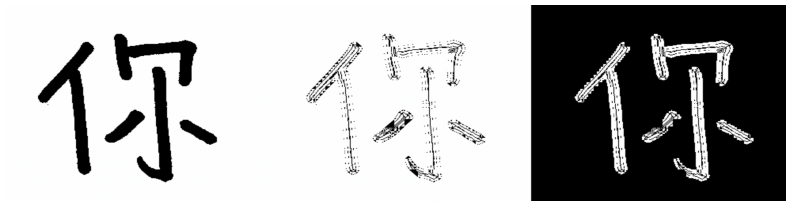
The experiment result is shown as below:

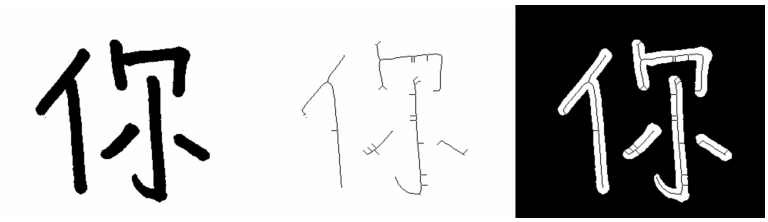
1、2-step thinning:

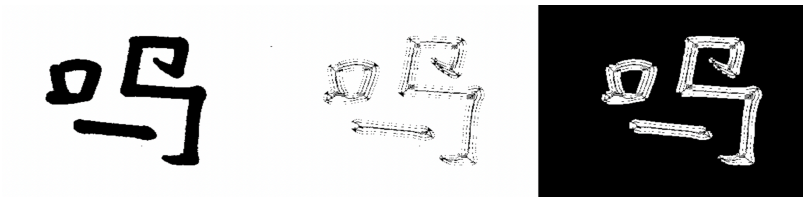


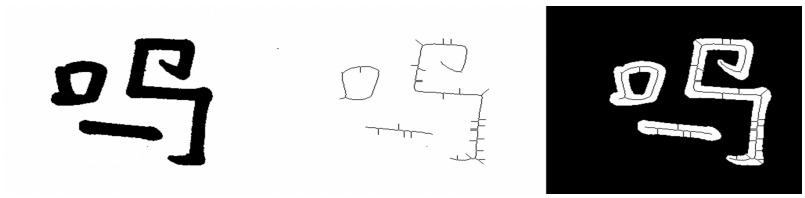


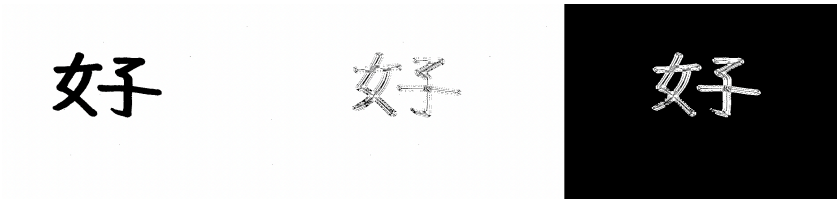
2、MAT implement and official MAT function:

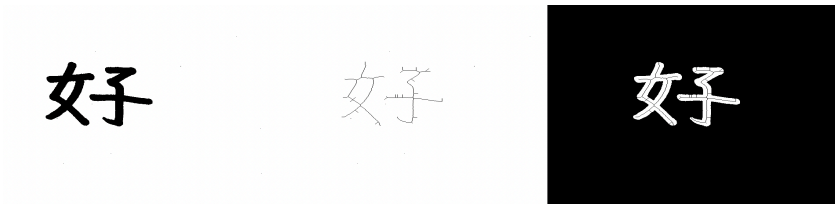


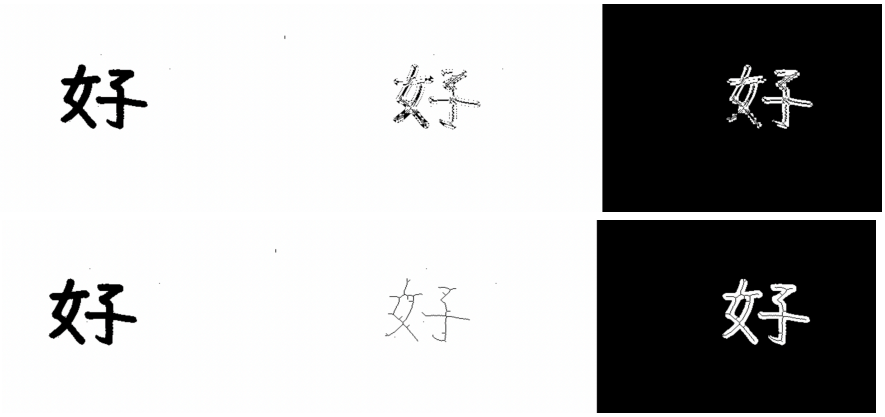












Conclusion

Thinning is a process to extract the skeleton of a region, In this experiment, we use 2 methods to perform thinning: 2-step thinning will repeat the steps to remove the redundant boundary points; the MAT will extract the skeleton based on distance transformation. From the experiment result, we could see that the 2-step thinning perform better than MAT, it could remove the redundant point perfectly. Mat also work well in skeleton, but it will leave some removable points.