922 U0610 電腦視覺
Computer Vision

Homework 7

授課教師： 傅楸善 教授

學生系級： 資工所一年級

學生姓名： 姚嘉昇

學生學號： R06922002

I. INTRODUCTION

1.1. Descriptions of Problem

This homework is to do thinning operation with following rules:

A. Please binarize leba.bmp with threshold 128.

B. Do thinning operation on binary image.

1.2. Programming Tools

1.2.1. Programming Language: Python3

1.2.2. Programming IDE: Visual Studio Code

II. METHOD

2.1. Algorithm of Thinning Operator

2.1.1. Step 1

- input: original symbolic image
- marked-interior/border-pixel operator
- output: interior/border image

2.1.2. Step 2

- input: interior/border image
- pair relationship operator
- output: marked image

2.1.3. Step 3

- input: original symbolic image + marked image
- marked-pixel connected shrink operator
  - removable (by connected shrink operator on original symbolic image)
  - marked (by marked image)
  - delete those pixels satisfied the two conditions mentioned above
- output: thinned output image

2.1.4. Step 4

- use thinned output image as next original symbolic image
- repeat step 1, step 2, step 3 until the last output never changed
- In my homework, it converged after 47 iterations.

2.2. Code Fragments

2.2.1. Code fragments of this homework

```python
if __name__ == '__main__':
    from PIL import Image
    import numpy as np

    # Load image from file.
    originalImage = Image.open('lena.bmp')
    # Get binary image.
    binaryImage = getBinaryImage(originalImage, 128)
    # Save binary image fo file.
    binaryImage.save('binary.bmp')

    # Define kernel for dilation.
    kernel = np.array([
        [1, 1, 1],
        [1, 1, 1],
        [1, 1, 1]])

    # Clear iteration counter.
    i = 0
    # Pass binary image to thinning image.
    thinningImage = binaryImage
    while True:
        # Get Yokoi Connectivity Number.
        YokoiConnectivityNumber = getYokoiConnectivityNumber(thinningImage)
        # Get interior image from Yokoi Connectivity Number.
        interiorImage = getInteriorImage(YokoiConnectivityNumber)
        # Get marked image by dilation of interoir image..
        markedImage = dilation(interiorImage, kernel)
        # Get thinning image.
        tempImage = getThinningImage(thinningImage, YokoiConnectivityNumber, markedImage)
        # If this iteration doesn't change image.
        if (isEqualImage(tempImage, thinningImage)):
            # Jump out this while loop.
            break
        # Update thinning image.
        thinningImage = tempImage
        # Increase iteration counter.
        i = i + 1
        # Show iteration counter on console.
        print ('Iteraion: ', i)
        # Save thinning image fo file.
        thinningImage.save('thinning' + str(i) + '.bmp')

    # Save Yokoi Connectivity Number to file.
    np.savetxt('YokoiConnectivityNumber.txt',
        YokoiConnectivityNumber.T,
        delimiter='', fmt='%s')
```

Figure 2.2.1.1. Code of main function.

```python
def getBinaryImage(originalImage, threshold):
    """
    :type originalImage: Image (from PIL)
    :type threshold: int
    :return type: Image (from PIL)
    """
    from PIL import Image
    # New image with the same size and 'binary' format.
    binaryImage = Image.new('1', originalImage.size)
    # Scan each column in original image.
    for c in range(originalImage.size[0]):
        # Scan each row in original image.
        for r in range(originalImage.size[1]):
            # Get pixel value in original image at (c, r).
            originalPixel = originalImage.getpixel((c, r))
            if (originalPixel >= threshold):
                # Put pixel value '1'  to binary image.
                binaryImage.putpixel((c, r), 1)
            else:
                # Put pixel value '0'  to binary image.
                binaryImage.putpixel((c, r), 0)
    # Return binary image.
    return binaryImage
```

Figure 2.2.1.2. Code of binarize.

```python
def downsampling(originalImage, sampleFactor):
    """
    :type originalImage: Image (from PIL)
    :type sampleFactor: int
    :return type: Image (from PIL)
    """
    from PIL import Image
    # Calculate the width and height of downsampling image.
    downsamplingWidth = int(originalImage.size[0] / sampleFactor)
    downsamplingHeight = int(originalImage.size[1] / sampleFactor)
    # New image with the downsampling size and 'binary' format.
    downsamplingImage = Image.new('1', (downsamplingWidth, downsamplingHeight))
    # Scan each column in downsampling image.
    for c in range(downsamplingImage.size[0]):
        # Scan each row in downsampling image.
        for r in range(downsamplingImage.size[1]):
            # Get pixel value in original image at (c * sampleFactor, r * sampleFactor).
            originalPixel = originalImage.getpixel((c * sampleFactor, r * sampleFactor))
            # Put pixel to downsampling image.
            downsamplingImage.putpixel((c, r), originalPixel)
    # Return downsampling image.
    return downsamplingImage
```

Figure 2.2.1.3. Code of down sampling.

```python
def getNeighborhoodPixels(originalImage, position):
    """
    :type originalImage: Image (from PIL)
    :type position: tuple
    :return type: numpy array
    """
    # Allocate memory space of neighborhoodPixels.
    neighborhoodPixels = np.zeros(9)
    # Get x and y of position.
    x, y = position
    # Scan dx from -1 to 1.
    for dx in range(3):
        # Scan dy from -1 to 1.
        for dy in range(3):
            # Calculate destination x, y position.
            destX = x + (dx - 1)
            destY = y + (dy - 1)
            # Avoid out of image range.
            if ((0 <= destX < originalImage.size[0]) and \
                (0 <= destY < originalImage.size[1])):
                # Get neighborhood pixel values.
                neighborhoodPixels[3 * dy + dx] = originalImage.getpixel((destX, destY))
            # It is out of image range.
            else:
                # Padding zeros when it is out of image range.
                neighborhoodPixels[3 * dy + dx] = 0
    # Original order:   [[x0, x1, x2], [x3, x4, x5], [x6, x7, x8]]
    # Sort pixels in    [[x7, x2, x6], [x3, x0, x1], [x8, x4, x5]] order.
    neighborhoodPixels = [
        neighborhoodPixels[4], neighborhoodPixels[5], neighborhoodPixels[1],
        neighborhoodPixels[3], neighborhoodPixels[7], neighborhoodPixels[8],
        neighborhoodPixels[2], neighborhoodPixels[0], neighborhoodPixels[6]]
    # Return Neighborhood Pixels.
    return neighborhoodPixels
```

Figure 2.2.1.4. Code of getting neighborhood pixels.

```python
def hFunctionYokoi(b, c, d, e):
    """
    :type b: int
    :type c: int
    :type d: int
    :type e: int
    :return type: str
    """
    if ((b == c) and (b != d or b != e)):
        return 'q'
    if ((b == c) and (b == d and b == e)):
        return 'r'
    if (b != c):
        return 's'
```

Figure 2.2.1.5. Code of Yokoi h function.

```python
def fFunctionYokoi(a1, a2, a3, a4):
    """
    :type a1: str
    :type a2: str
    :type a3: str
    :type a4: str
    :return type: str
    """
    # a1 == a2 == a3 == a4 == r
    if ([a1, a2, a3, a4].count('r') == 4):
        # Return label 5 (interior).
        return 5
    else:
        # Return count of 'q'.
        # 0: Isolated, 1: Edge, 2: Connecting, 3: Branching, 4: Crossing.
        return [a1, a2, a3, a4].count('q')
```

Figure 2.2.1.6. Code of Yokoi f function.

```python
def getYokoiConnectivityNumber(originalImage):
    """
    :type originalImage: Image (from PIL)
    :return type: numpy array
    """
    # Allocate memory space of Yokoi Connectivity Number.
    YokoiConnectivityNumber = np.full(originalImage.size, ' ')

    # Scan each column in original image.
    for c in range(originalImage.size[0]):
        # Scan each row in original image.
        for r in range(originalImage.size[1]):
            # This point is object.
            if (originalImage.getpixel((c, r)) != 0):
                # Get neighborhood pixel values.
                neighborhoodPixels = getNeighborhoodPixels(originalImage, (c, r))
                YokoiConnectivityNumber[c, r] = fFunctionYokoi(
                    hFunctionYokoi(neighborhoodPixels[0], neighborhoodPixels[1], neighborhoodPixels[6], neighborhoodPixels[2]),
                    hFunctionYokoi(neighborhoodPixels[0], neighborhoodPixels[2], neighborhoodPixels[7], neighborhoodPixels[3]),
                    hFunctionYokoi(neighborhoodPixels[0], neighborhoodPixels[3], neighborhoodPixels[8], neighborhoodPixels[4]),
                    hFunctionYokoi(neighborhoodPixels[0], neighborhoodPixels[4], neighborhoodPixels[5], neighborhoodPixels[1]))
            # This point is background.
            else:
                YokoiConnectivityNumber[c, r] = ' '

    # Return Yokoi Connectivity Number.
    return YokoiConnectivityNumber
```

Figure 2.2.1.7. Code of getting Yokoi number function.

```python
def getInteriorImage(YokoiConnectivityNumber):
    """
    :type YokoiConnectivityNumber: numpy array
    :return type: Image (from PIL)
    """
    from PIL import Image
    # New image with the same size as Yokoi connectivity number and 'binary' format.
    interiorImage = Image.new('1', YokoiConnectivityNumber.shape)
    # Scan each column in interior image.
    for c in range(interiorImage.size[0]):
        # Scan each row in interior image.
        for r in range(interiorImage.size[1]):
            # If this pixel is interior(label = 5).
            if (YokoiConnectivityNumber[c, r] == '5'):
                # Put white pixel to interior image.
                interiorImage.putpixel((c, r), 1)
            # If this pixel is not interior.
            else:
                # Put black pixel to interior image.
                interiorImage.putpixel((c, r), 0)
    # Return interior image.
    return interiorImage
```

Figure 2.2.1.8. Code of getting interior image function.

```python
def dilation(originalImage, kernel):
    """
    :type originalImage: Image (from PIL)
    :type kernel: numpy array
    :return type: Image (from PIL)
    """
    from PIL import Image
    # Get center position of kernel.
    centerKernel = tuple([x // 2 for x in kernel.shape])
    # New image with the same size and 'binary' format.
    dilationImage = Image.new('1', originalImage.size)
    # Scan each column in original image.
    for c in range(originalImage.size[0]):
        # Scan each row in original image.
        for r in range(originalImage.size[1]):
            # Get pixel value in original image at (c, r).
            originalPixel = originalImage.getpixel((c, r))
            # If this pixel is object (1, white).
            if (originalPixel != 0):
                # Paste kernel on original image at (c, r).
                # Scan each column in kernel.
                for x in range(kernel.shape[0]):
                    # Scan each row in kernel.
                    for y in range(kernel.shape[1]):
                        # Only paste '1' value from kernel.
                        if (kernel[x, y] == 1):
                            # Calculate destination x, y position.
                            destX = c + (x - centerKernel[0])
                            destY = r + (y - centerKernel[1])
                            # Avoid out of image range.
                            if ((0 <= destX < originalImage.size[0]) and \
                                (0 <= destY < originalImage.size[1])):
                                # Paste '1' value on original image.
                                dilationImage.putpixel((destX, destY), 1)
    # Return dilation image.
    return dilationImage
```

Figure 2.2.1.9. Code of dilation function.

```python
def getThinningImage(originalImage, YokoiConnectivityNumber, markedImage):
    """
    :type originalImage: Image (from PIL)
    :type YokoiConnectivityNumber: numpy array
    :type markedImage: Image (from PIL)
    :return type: Image (from PIL)
    """
    from PIL import Image
    # New image with the same size as original image and 'binary' format.
    thinningImage = Image.new('1', originalImage.size)
    # Scan each column in thinning image.
    for c in range(thinningImage.size[0]):
        # Scan each row in thinning image.
        for r in range(thinningImage.size[1]):
            # If this point is removable(label = 1) and is in marked image.
            if (YokoiConnectivityNumber[c, r] == '1' and markedImage.getpixel((c, r)) != 0):
                # Remove this pixel from original image.
                thinningImage.putpixel((c, r), 0)
            else :
                thinningImage.putpixel((c, r), originalImage.getpixel((c, r)))
    # Return thinning image.
    return thinningImage
```

Figure 2.2.1.10. Code of getting thinning image function.

```python
def isEqualImage(image1, image2):
    """
    :type image1: Image (from PIL)
    :type image2: Image (from PIL)
    :return type: bool
    """
    from PIL import ImageChops
    return ImageChops.difference(image1, image2).getbbox() is None
```

Figure 2.2.1.11. Code of equal image function.

III. RESULTS

3.1. Original Image

Figure 3.1. Original lena.bmp.

3.2. Results of binary and down sampling

Figure 3.2.1. binary.bmp. Figure 3.2.2. downsampling.bmp.

3.3. Results of Thinning Operator

Figure 3.3.1. Iteration 1.

Figure 3.3.2. Iteration 2.

Figure 3.3.3. Iteration 3.

Figure 3.3.4. Iteration 4.

Figure 3.3.5. Iteration 5.

Figure 3.3.6. Iteration 6.

Figure 3.3.7. Iteration 7.

Figure 3.3.8. Iteration 8.

Figure 3.3.9. Iteration 9.

Figure 3.3.10. Iteration 10.

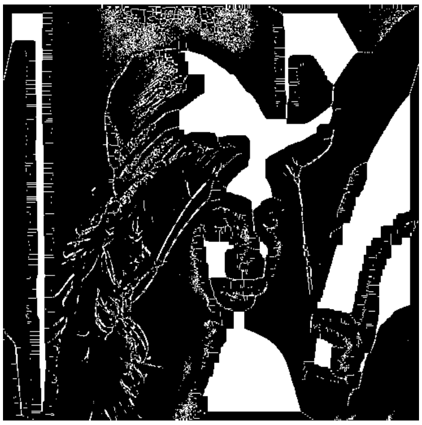

Figure 3.3.11. Iteration 11.

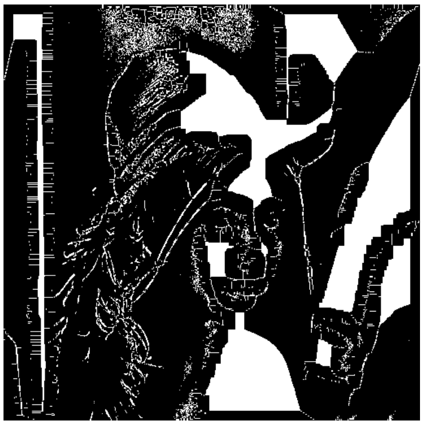

Figure 3.3.12. Iteration 12.

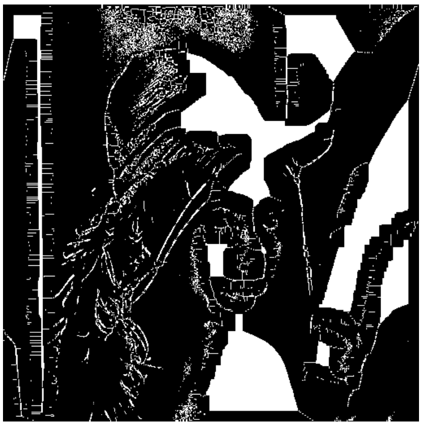

Figure 3.3.13. Iteration 13.

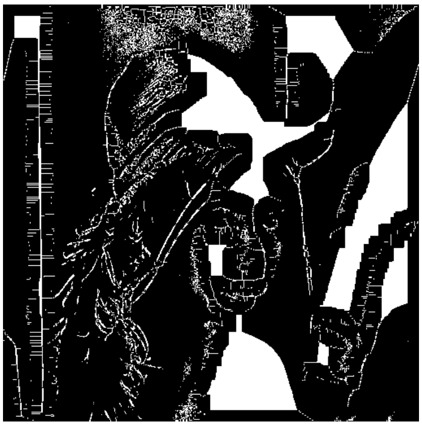

Figure 3.3.14. Iteration 14.

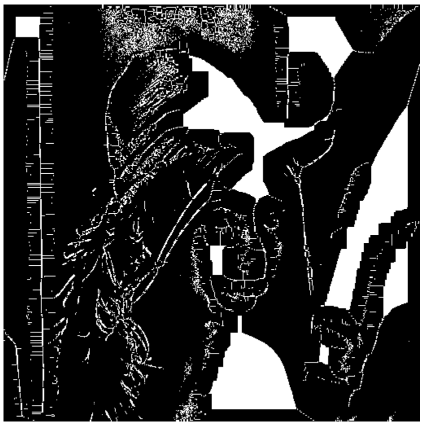

Figure 3.3.15. Iteration 15.

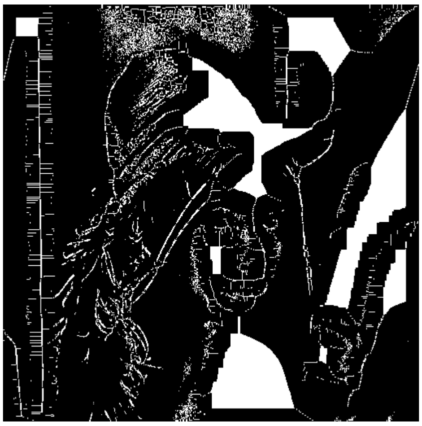

Figure 3.3.16. Iteration 16.

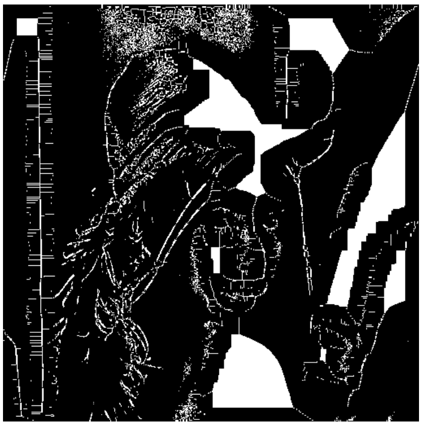

Figure 3.3.17. Iteration 17.

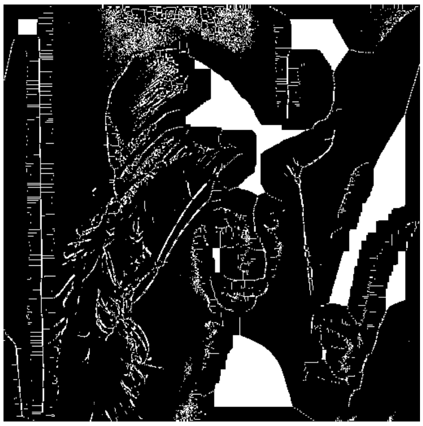

Figure 3.3.18. Iteration 18.

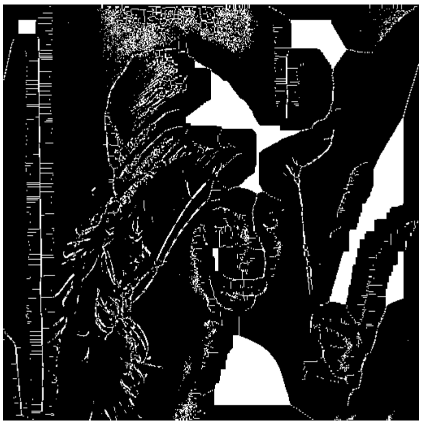

Figure 3.3.19. Iteration 19.

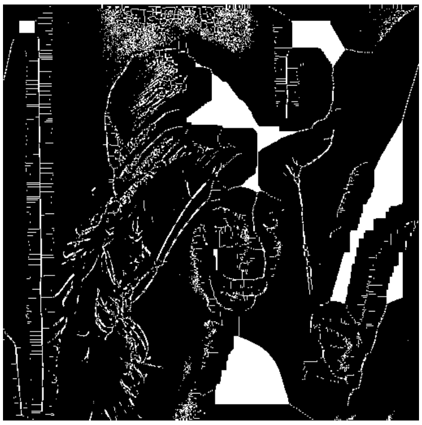

Figure 3.3.20. Iteration 20.

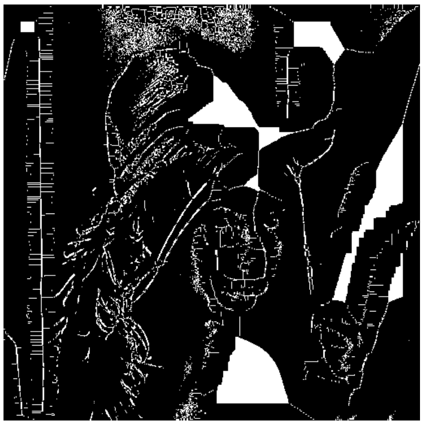

Figure 3.3.21. Iteration 21.

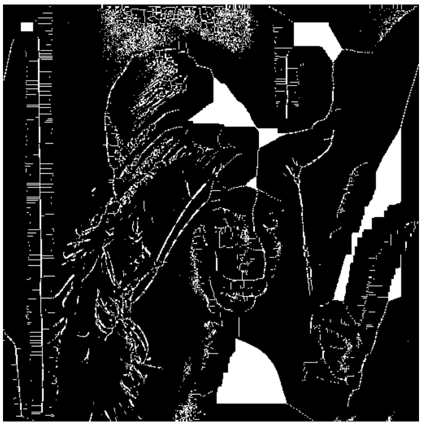

Figure 3.3.22. Iteration 22.

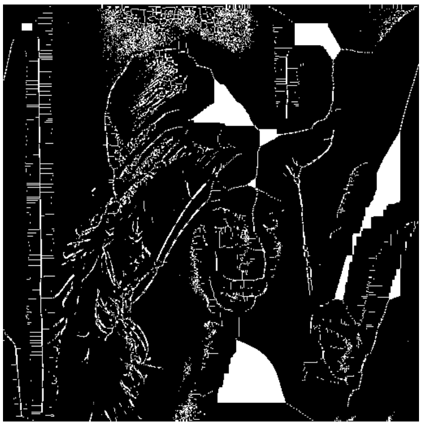

Figure 3.3.23. Iteration 23.

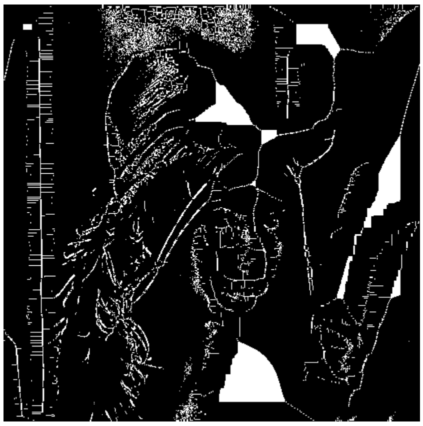

Figure 3.3.24. Iteration 24.

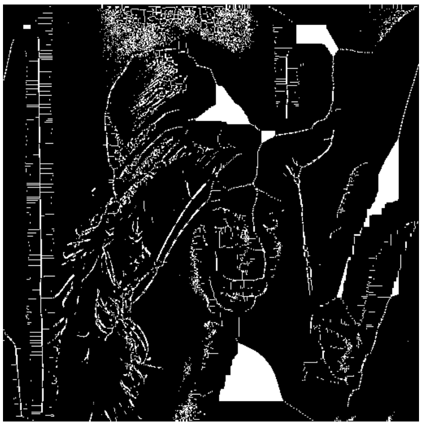

Figure 3.3.25. Iteration 25.

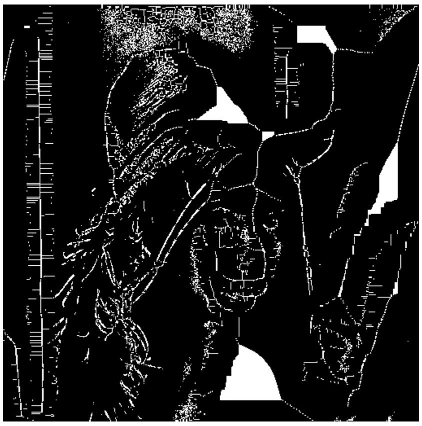

Figure 3.3.26. Iteration 26.

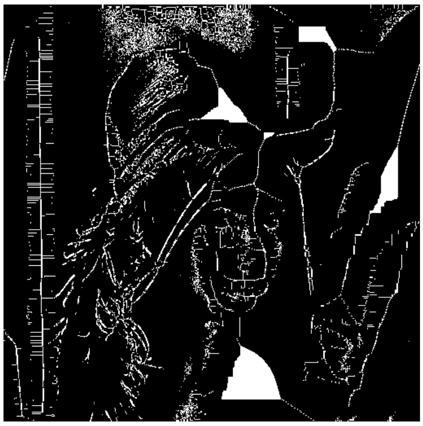

Figure 3.3.27. Iteration 27.

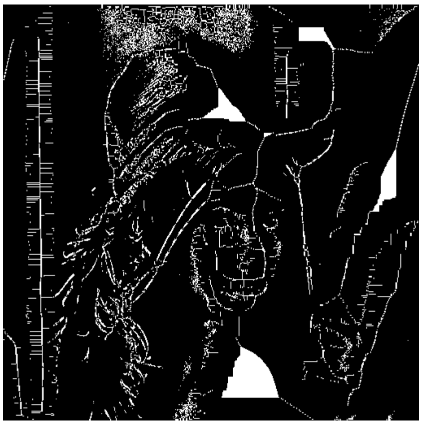

Figure 3.3.28. Iteration 28.

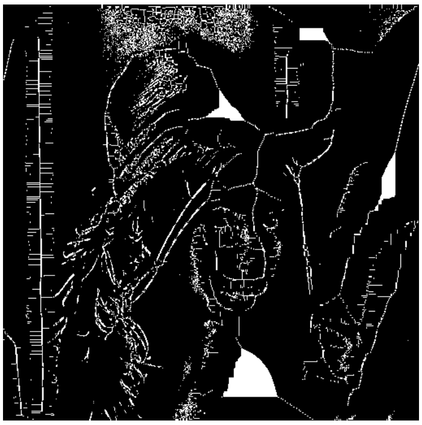

Figure 3.3.29. Iteration 29.

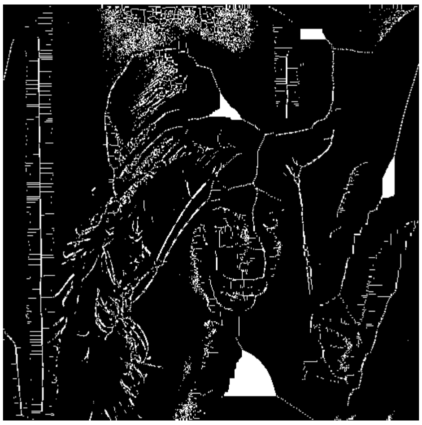

Figure 3.3.30. Iteration 30.

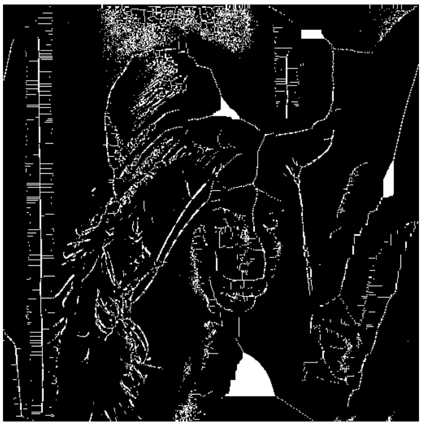

Figure 3.3.31. Iteration 31.

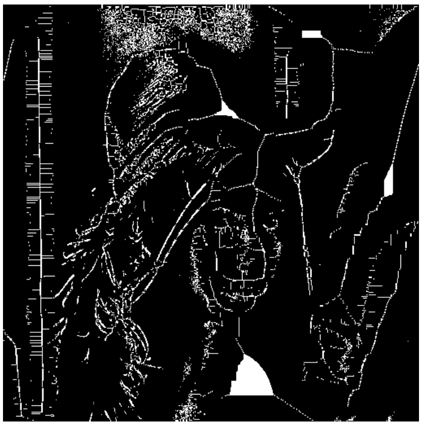

Figure 3.3.32. Iteration 32.

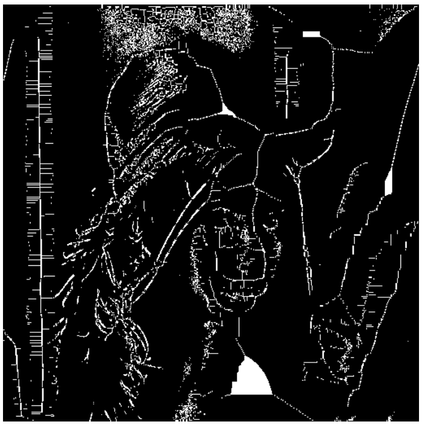

Figure 3.3.33. Iteration 33.

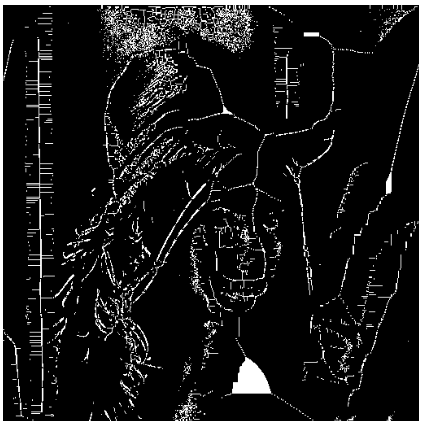

Figure 3.3.34. Iteration 34.

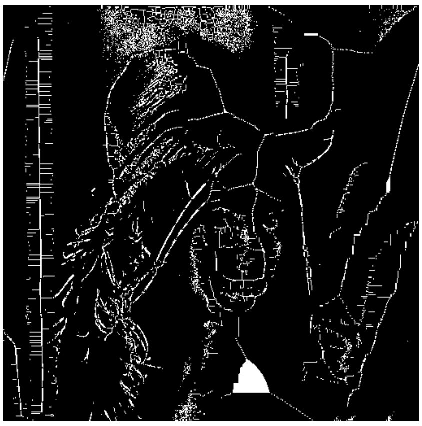

Figure 3.3.35. Iteration 35.

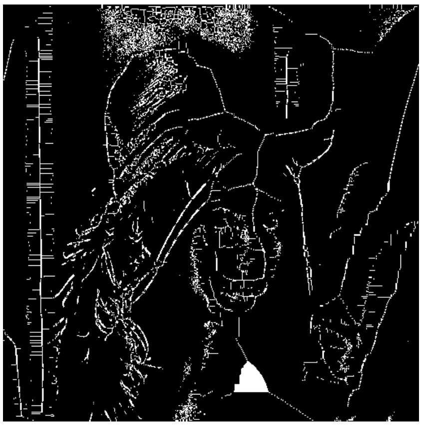

Figure 3.3.36. Iteration 36.

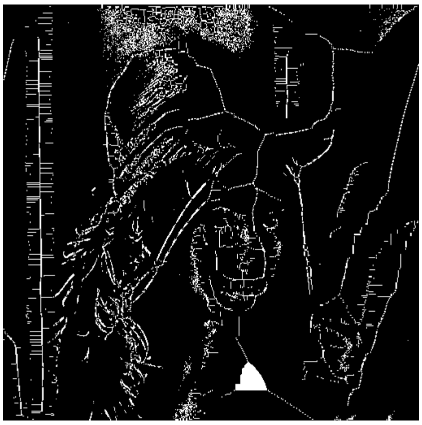

Figure 3.3.37. Iteration 37.

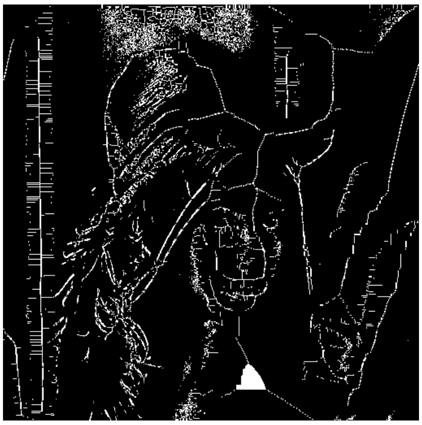

Figure 3.3.38. Iteration 38.

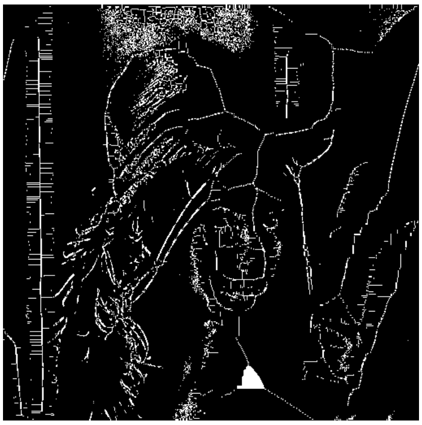

Figure 3.3.39. Iteration 39.

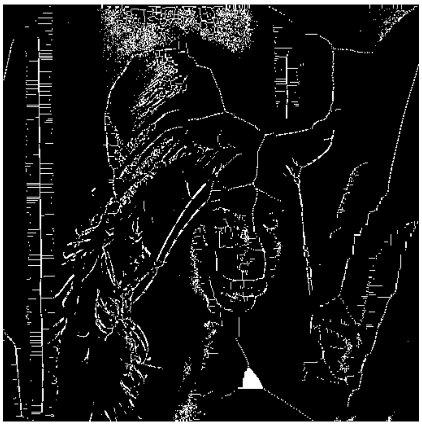

Figure 3.3.40. Iteration 40.

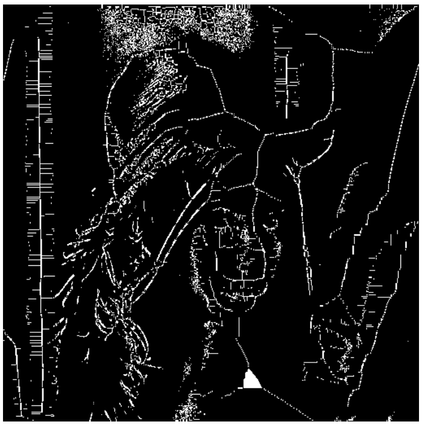

Figure 3.3.41. Iteration 41.

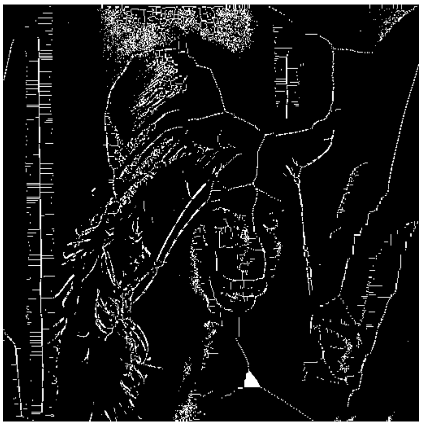

Figure 3.3.42. Iteration 42.

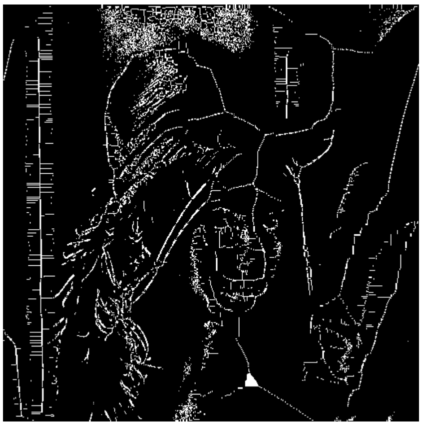

Figure 3.3.43. Iteration 43.

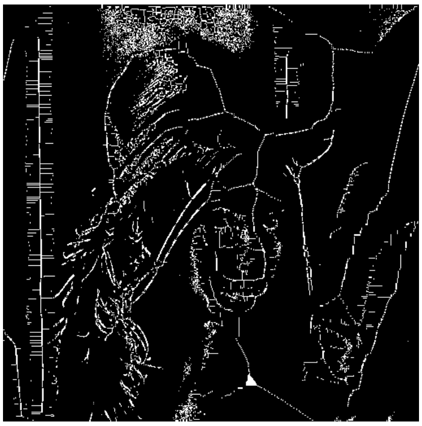

Figure 3.3.44. Iteration 44.

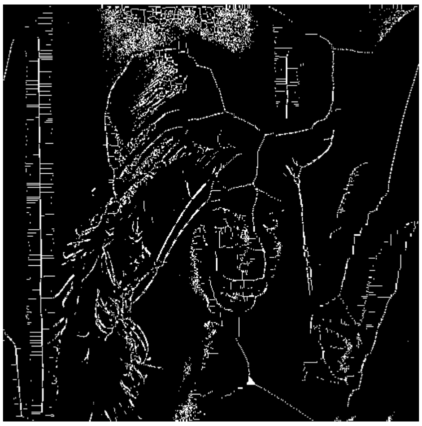

Figure 3.3.45. Iteration 45.

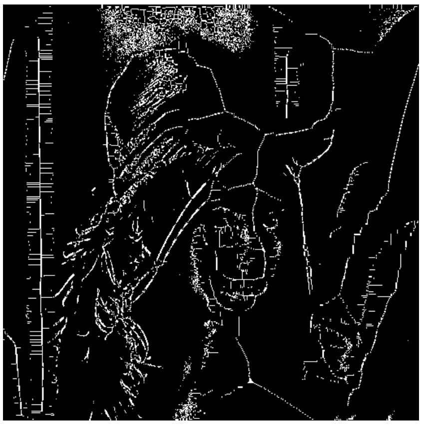

Figure 3.3.46. Iteration 46.

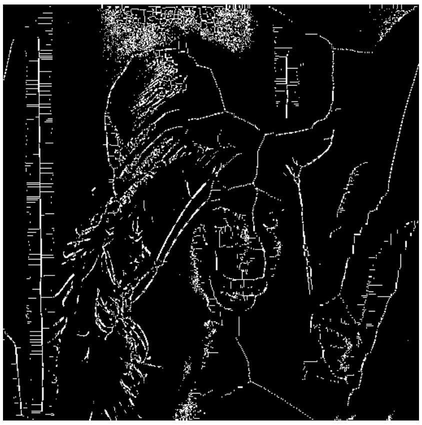

Figure 3.3.47. Iteration 47.

3.4. Final Results of Thinning Operator

Figure 3.4.1. binary.bmp.

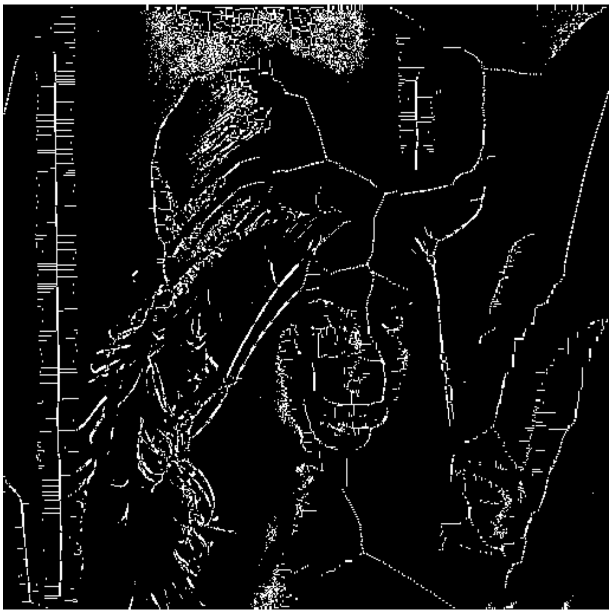

Figure 3.4.2. thinning47.bmp.