Assignment #4

HARNEET SINGH - 400110275

COMPENG 4TN4: Image Processing

February 20, 2022

Theory

1. Morphological Operations

1.1 - Erosion

Erosion process requires *logical AND* operation between pixels based on the structuring element. Given structuring element has all 1's in the middle column, and a 1 in third column and second row. Consider the highlighted element in the following matrix, and find the elements of interest based on the structuring element.

$$Binary\ Image = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & \boxed{1} & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

$$\text{We get a sub-matrix} = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 1 \\ 0 & 0 & 1 \end{bmatrix}$$

Based on the structuring element, we need to apply logical AND operation on the highlighted elements:

$$\text{Sub-matrix} = \begin{bmatrix} 0 & \boxed{0} & 0 \\ 0 & \boxed{1} & \boxed{1} \\ 0 & \boxed{0} & 1 \end{bmatrix}$$

Erosion Result = {0 AND 1 AND 1 AND 0} = 0

Following Python code uses the same technique to compute erosion values for all the pixels in the binary image: NOTE: Zero-padding is not applied to the given binary image.

```python
import numpy as np
import cv2

# creating image array
img = np.array([[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
                [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
                [0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 0, 0],
                [0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 0, 0],
                [0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 0, 0],
                [0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 0, 0],
                [0, 0, 1, 0, 0, 0, 0, 1, 1, 1, 0, 0],
                [0, 0, 0, 1, 0, 0, 0, 0, 1, 1, 0, 0],
                [0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0],
                [0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0],
                [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
                [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
```

```
                [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]], dtype='uint8')
imgShape = int(img.shape[1])
erodedImg = np.zeros_like(img)

# creating structuring element
structElement = np.array([[0, 1, 0],
                          [0, 1, 1],
                          [0, 1, 0]], dtype='uint8')
structElemShape = int(structElement.shape[1])

# finding elements that need to be considered based on structuring element
elem = np.where(structElement==1)
elements = list(zip(elem[0], elem[1]))
# print(elements)

# erode functionality implemented in the for loop
# first getting the sub-matrix, then using the elements variable,
# determining the elements of interest based on structuring element
for k in range(imgShape-structElemShape+1):
    for p in range(imgShape-structElemShape+1):
        subMatrix = img[k:k+structElemShape, p:p+structElemShape]

        temp = 1
        for loc in elements:
            temp = temp and subMatrix[loc[0]][loc[1]]

        if temp:
            erodedImg[k+1][p+1] = 1

print(erodedImg)

# cv2's erode function to check the functionality of my logic
erodedImg2 = cv2.erode(img, structElement, iterations=1)
print(erodedImg2)
```

We get the following result after executing the above written script:

$$Eroded\ Image = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

1.2 - Dilation

Dilation process requires *logical OR* operation between pixels based on the structuring element. Given structuring element has all 1's along the diagonal. Consider the highlighted element in the following matrix, and find the elements of interest based on the structuring element.

$$Binary\ Image = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & \boxed{0} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

$$\text{We get a sub-matrix} = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Based on the structuring element, we need to apply logical OR operation on the highlighted elements:

$$\text{Sub-matrix} = \begin{bmatrix} \boxed{0} & 0 & 0 \\ 0 & \boxed{0} & 0 \\ 0 & 0 & \boxed{1} \end{bmatrix}$$

Erosion Result = {0 OR 0 OR 1} = 1

Following Python code uses the same technique to compute dilation values for all the pixels in the binary image:
NOTE: Zero-padding is not applied to the given binary image.

```
import numpy as np
import cv2

# creating image array
img = np.array([[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
                [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
                [0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 0, 0],
                [0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 0, 0],
                [0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 0, 0],
                [0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 0, 0],
                [0, 0, 1, 0, 0, 0, 0, 1, 1, 1, 0, 0],
                [0, 0, 0, 1, 0, 0, 0, 0, 1, 1, 0, 0],
                [0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0],
                [0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0],
                [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
                [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]], dtype='uint8')
imgShape = int(img.shape[1])
dilateImg = np.zeros_like(img)

# creating structuring element
structElement = np.array([[1, 0, 0],
                          [0, 1, 0],
                          [0, 0, 1]], dtype='uint8')
structElemShape = int(structElement.shape[1])

# finding elements that need to be considered based on structuring element
elem = np.where(structElement==1)
elements = list(zip(elem[0], elem[1]))
# print(elements)

# dilate functionality implemented in the for loop
# first getting the sub-matrix, then using the elements variable,
# determining the elements of interest based on structuring element
for k in range(imgShape-structElemShape+1):
    for p in range(imgShape-structElemShape+1):
```

```
        subMatrix = img[k:k+structElemShape, p:p+structElemShape]

        temp = 0
        for loc in elements:
            temp = temp or subMatrix[loc[0]][loc[1]]

        if temp:
            dilateImg[k+1][p+1] = 1

print(dilateImg)

# cv2's dilate function to check the functionality of my logic
dilateImg2 = cv2.dilate(img, structElement, iterations=1)
print(dilateImg2)
```

We get the following result after executing the above written script:

$$Dilated\ Image = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

1.3 - Opening

Opening process requires erode and then dilate. Given structuring element has all 1's, therefore we need to consider all the elements around anchor pixel using 3x3 kernel. Since logic of erosion and dilation is explained in part 1.1 and 1.2, Python code and result is shown below.

Following Python code uses the same technique to compute opening values for all the pixels in the binary image: NOTE: Zero-padding is not applied to the given binary image.

```
import numpy as np
import cv2

# creating image array
img = np.array([[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
                [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
                [0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 0, 0],
                [0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 0, 0],
                [0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 0, 0],
                [0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 0, 0],
                [0, 0, 1, 0, 0, 0, 0, 1, 1, 1, 0, 0],
                [0, 0, 0, 1, 0, 0, 0, 0, 1, 1, 0, 0],
                [0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0],
                [0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0],
                [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
                [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]], dtype='uint8')
imgShape = int(img.shape[1])
erodedImg = np.zeros_like(img)
dilatedImg = np.zeros_like(img)

# creating structuring element
```

```
structElement = np.array([[1, 1, 1],
                          [1, 1, 1],
                          [1, 1, 1]], dtype='uint8')
structElemShape = int(structElement.shape[1])

# finding elements that need to be considered based on structuring element
elem = np.where(structElement==1)
elements = list(zip(elem[0], elem[1]))
# print(elements)

# erode functionality implemented in the for loop
def erodeImg(img):
    for k in range(imgShape-structElemShape+1):
        for p in range(imgShape-structElemShape+1):
            subMatrix = img[k:k+structElemShape, p:p+structElemShape]

            temp = 1
            for loc in elements:
                temp = temp and subMatrix[loc[0]][loc[1]]

            if temp:
                erodedImg[k+1][p+1] = 1
    return erodedImg


# dilate functionality implemented in the for loop
def dilateImg(img):
    for k in range(imgShape - structElemShape + 1):
        for p in range(imgShape - structElemShape + 1):
            subMatrix = img[k:k + structElemShape, p:p + structElemShape]

            temp = 0
            for loc in elements:
                temp = temp or subMatrix[loc[0]][loc[1]]

            if temp:
                dilatedImg[k + 1][p + 1] = 1
    return dilatedImg


# opening steps: erode then dilate
stepErode = erodeImg(img)
openedImg = dilateImg(stepErode)
print(openedImg)

# cv2's open function to check the functionality of my logic
openedImg2 = cv2.morphologyEx(img, cv2.MORPH_OPEN, structElement)
print(openedImg2)
```

We get the following result after executing the above written script:

$$Opened\ Image = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

2. Distance and Boundary

2.1

$4-neighbors$: Using N_4 pixels i.e. immediate horizontal and vertical elements to the anchor pixel, we can determine the boundary pixels. In this case, if any of the N_4 pixels about anchor pixel are 0, then it is a boundary pixel. Boundary pixels are highlighted with circles in the following matrix:

$$\begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & \textcircled{1} & 0 & 0 & 0 \\ 0 & 0 & \textcircled{1} & 1 & \textcircled{1} & 0 & 0 \\ 0 & \textcircled{1} & 1 & 1 & 1 & \textcircled{1} & 0 \\ 0 & \textcircled{1} & 1 & 1 & \textcircled{1} & 0 & 0 \\ 0 & \textcircled{1} & \textcircled{1} & \textcircled{1} & 0 & 0 & 0 \\ 0 & \textcircled{1} & 0 & \textcircled{1} & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

$8-neighbors$: Using N_8 pixels i.e. $N_4 \cup N_D$, we can determine the boundary pixels. In this case, if any of the N_8 pixels about anchor pixel are 0, then it is a boundary pixel. Boundary pixels are highlighted with circles in the following matrix:

$$\begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & \textcircled{1} & 0 & 0 & 0 \\ 0 & 0 & \textcircled{1} & \textcircled{1} & \textcircled{1} & 0 & 0 \\ 0 & \textcircled{1} & \textcircled{1} & 1 & \textcircled{1} & \textcircled{1} & 0 \\ 0 & \textcircled{1} & 1 & \textcircled{1} & \textcircled{1} & 0 & 0 \\ 0 & \textcircled{1} & \textcircled{1} & \textcircled{1} & 0 & 0 & 0 \\ 0 & \textcircled{1} & 0 & \textcircled{1} & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

2.2

Boundary pixels are represented by 1 in the following matrices and we need to determine distance between the highlighted pixels (p and q).

$$Boundary\ Image\ (4-neighbors) = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & \textcircled{1} & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & \textcircled{1} & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

$\boldsymbol{D_8} = max\{|x-s|, |s-t|\}$ where (x, y) and (s, t) are the coordinates of two pixels between which the distance is to be computed.
Let's define coordinates of p and q pixels. But note that, assumption is that origin is at the top-left corner and x-axis is to the right ($\rightarrow$) and y-axis is to the bottom ($\downarrow$).
Therefore, p-pixel is at (3, 1) and q-pixel is at (3, 6).
$D_8 = max\{|3-3|, |1-6|\}$
$D_8 = max\{|0|, |-5|\}$
$D_8 = max\{0, 5\}$

$$\boxed{D_8 = 5} \tag{1}$$

$\boldsymbol{D_m}$ = length of the shortest m-path between p and q with V = 1 i.e. traverse from one point to another only if pixel value is 1 (boundary). Following matrix highlights the possible two paths from point p to q: (note that it is not representing all the boundary pixels in the sense of 4-neighbors)

$$\begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

We get distance of 5 from the right branch and distance of 7 from the left branch as shown below:

$$\begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & a_7|a_5 & 0 & 0 & 0 \\ 0 & 0 & a_6 & 0 & a_4 & 0 & 0 \\ 0 & a_5 & 0 & 0 & 0 & a_3 & 0 \\ 0 & a_4 & 0 & 0 & a_2 & 0 & 0 \\ 0 & a_3 & a_2 & a_1 & 0 & 0 & 0 \\ 0 & 1 & 0 & a_0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

$$D_m = min\{5, 7\}$$

$$\boxed{D_m = 5} \tag{2}$$

2.3

Boundary pixels are represented by 1 in the following matrices and we need to determine distance between the highlighted pixels (p and q).

$$Boundary\ Image\ (8-neighbors) = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 & 1 & 1 & 0 \\ 0 & 1 & 0 & 1 & 1 & 0 & 0 \\ 0 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

$\boldsymbol{D_4} = |x - s| + |s - t|$ where (x, y) and (s, t) are the coordinates of two pixels between which the distance is to be computed.
Let's define coordinates of p and q pixels. But note that, assumption is that origin is at the top-left corner and x-axis is to the right ($\rightarrow$) and y-axis is to the bottom ($\downarrow$).
Therefore, p-pixel is at (3, 1) and q-pixel is at (3, 6).
$D_4 = |3 - 3| + |1 - 6|$
$D_4 = |0| + |-5|$
$D_4 = 0 + 5$

$$\boxed{D_4 = 5} \tag{3}$$

$\boldsymbol{D_m}$ = length of the shortest m-path between p and q with V = 1 i.e. traverse from one point to another only if pixel value is 1 (boundary). Following matrix highlights the possible two paths from point p to q: (note that it is not representing all the boundary pixels in the sense of 8-neighbors)

$$\begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 & 1 & 1 & 0 \\ 0 & 1 & 0 & 1 & 1 & 0 & 0 \\ 0 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

We get distance of 7 from the right branch and distance of 9 from the left branch as shown below:

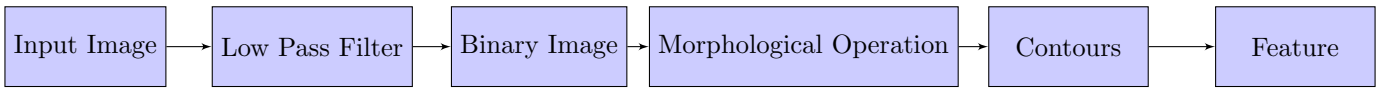
$$\begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & a_9|a_7 & 0 & 0 & 0 \\ 0 & 0 & a_7 & a_8 & a_5 & 0 & 0 \\ 0 & a_5 & a_6 & 0 & a_4 & 1 & 0 \\ 0 & a_4 & 0 & a_2 & a_3 & 0 & 0 \\ 0 & a_3 & a_2 & a_1 & 0 & 0 & 0 \\ 0 & 1 & 0 & a_0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

$$D_m = min\{9, 7\}$$

$$\boxed{D_m = 7} \tag{4}$$

To illustrate the choice of path, pixels at (1,3) and (2,2) are not m-adjacent because the intersection of N_4 of (1,3) and N_4 of (2,2) has 1 at (2,3). This is why we can not move from a_5 directly to a_7 i.e. diagonal movement is not allowed.

3. Extracting Licence Plate



- First, convert the image to gray-scale color to apply subsequent steps and retrieve desired features. This is done for higher performance and accuracy as most operations perform well on gray colored images.
- Next, apply a pre-processing step to remove high frequency noise from the image. This can be achieved through Gaussian filter which is what I will be using in my implementation.
- After filtering the image, we need to convert the image to binary format i.e. pixel values either 0 or 1. In my implementation, I have used a thresholding function, however edge detection can be employed to achieve similar results.
- Next, we need to apply morphological operations to remove small objects or to fill the holes. In my implementation, I will be using Open operation because it serves the purpose of removing unwanted blobs. This step will reduce the number of contours that will be derived in the next step.
- For retrieving the licence plate, contours can be used. Contours is tool used for shape analysis and object detection and recognition which determines all the continuous points (along the boundary) with same color or intensity.
- Finally, based on the characteristics of the object such as area or height or width, we can access the desired feature from all the contours. In my implementation, I am using both contour-area to image-area and contour height to width ratios to make the algorithm robust to changes in the image size, color and rotation.

Implementation

1. Extracting Licence plate

Python Script:

```python
import numpy as np
import cv2
from matplotlib import pyplot as plt

# creating image array
img = cv2.imread('3.png')
grayImg = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
height, width = grayImg.shape
imgSize = height * width

# applying filter to smooth the image - remove high frequency noise
gaussianFilteredImg = cv2.GaussianBlur(grayImg, (3,3), 0)

# binarizing the image using threshold function of OpenCV
# threshold value of 127 is chosen to find the black letters in the licence plate
# OTSU finds the optimal threshold value
thresVal, binImg = cv2.threshold(gaussianFilteredImg, 127, 255, cv2.THRESH_BINARY | cv2.THRESH_OTSU)
print(f'Threshold Value: {thresVal}')
# cv2.imshow("binarized img", binImg)
# cv2.imwrite("impl_1_binarized_img.png", binImg)

# cv2's Open function (erode and then dilate) to remove small blobs
open_kernel = cv2.getStructuringElement(cv2.MORPH_RECT, (2,2))
openedImg = cv2.morphologyEx(binImg, cv2.MORPH_OPEN, open_kernel, iterations=3)

# cv2.imshow("opened img", openedImg)
# cv2.imwrite("impl_1_opened_img.png", openedImg)

'''
using contours to join all the continuous points along the boundary with approximation,
this contour function identifies white objects on black background,
cv2.CHAIN_APPROX_SIMPLE removes redundant points from the boundary and compresses the contour,
using underscore for hierarchy data because it is not needed in this case,
2nd tuple in the contours array stores the hierarchy but we do not need it in this case.
'''
contours, _ = cv2.findContours(openedImg, cv2.RETR_EXTERNAL, cv2.CHAIN_APPROX_SIMPLE)
cv2.drawContours(grayImg, contours[0], -1, (0,255,0), 3)
# cv2.imshow("contours img", grayImg)
# cv2.imwrite('impl_1_contour_img.png', grayImg)

# iterating through each contour to get the area of the contour and find the license plate
# based on the area and height to width ratio in terms of full image size
for cont in contours:
    x, y, w, h = cv2.boundingRect(cont)
    area = cv2.contourArea(cont)
    print(f'w: {w}, h: {h}, area:{area}')
```

```
    if 2 <= w/h <= 2.5 and 7 < imgSize/area < 9:
        print(f'ROI - w: {w}, h: {h}, area:{area}')
        ROI = np.ones_like(grayImg)
        ROI[y:y+h, x:x+w] = openedImg[y:y+h, x:x+w]

cv2.imshow("img", ROI)
# cv2.imwrite('impl_1_extracted_plate.png', ROI)
cv2.waitKey(0)
```

Results from the script shown above:

After converting the image to grayscale and filtering with Gaussian filter, image has been binarized. The binary image result is shown below:

As we can see, the binary image has unwanted blobs and small objects which need to be removed. This occurred because of the uneven lighting conditions in the given image. To remove the unwanted blobs, Open morphology operation is applied as the next step. Result of opened image is shown below:

Upon comparing the binary image with Opened image, we can notice that some of the unwanted blobs have been filled in and original shape of the image is still intact.

Next, we need to determine all the contours in the image so that we can classify them and retrieve the licence plate based on region properties such as area, height and width of the contour. OpenCV's *findContours()* is applied to get all the contours. In the following image, all the contours are highlighted on the grayscale image:

Since, the area of the licence plate is maximum between all the contours and the height to width ratio is unique,

I have used these two properties to make my algorithm robust. Please read instructions between line 43 and 51 to understand the implementation better. Final result is shown below:

Robustness against illumination: As the image is first converted to binary image using thresholding mechanism, change in illumination will not impact the binarization step. This is because optimal threshold value is computed using the OpenCV's library function and the feature extraction steps are applied on the binary image. I am using Otsu method which determines the optimal threshold value from the image histogram. Otsu method tries to find the best threshold value by first creating a bimodal histogram from the image, if possible. Optimal value is chosen in the middle of these two modals from the histogram.

Robustness against Scale: Variant scale of the image will not hinder the detection of licence plate because the image-area to licence-area will be consistent in the scaled image as well. Also, height to width ratio is used to determine the unique contour that fits within the dimensional range of a licence plate.

Robustness against Angle: As for the angular rotation, since we are using contours and contours can be determined regardless of the rotation, this algorithm will not be affected by the rotation of the image.

Bibliography

[1] Contour features. https://docs.opencv.org/3.4/dd/d49/tutorial_py_contour_features.html.

[2] Contour properties. https://docs.opencv.org/3.4/d1/d32/tutorial_py_contour_properties.html.

[3] Threholding. https://docs.opencv.org/4.x/d7/d4d/tutorial_py_thresholding.html.