CV-MidSem

Q1

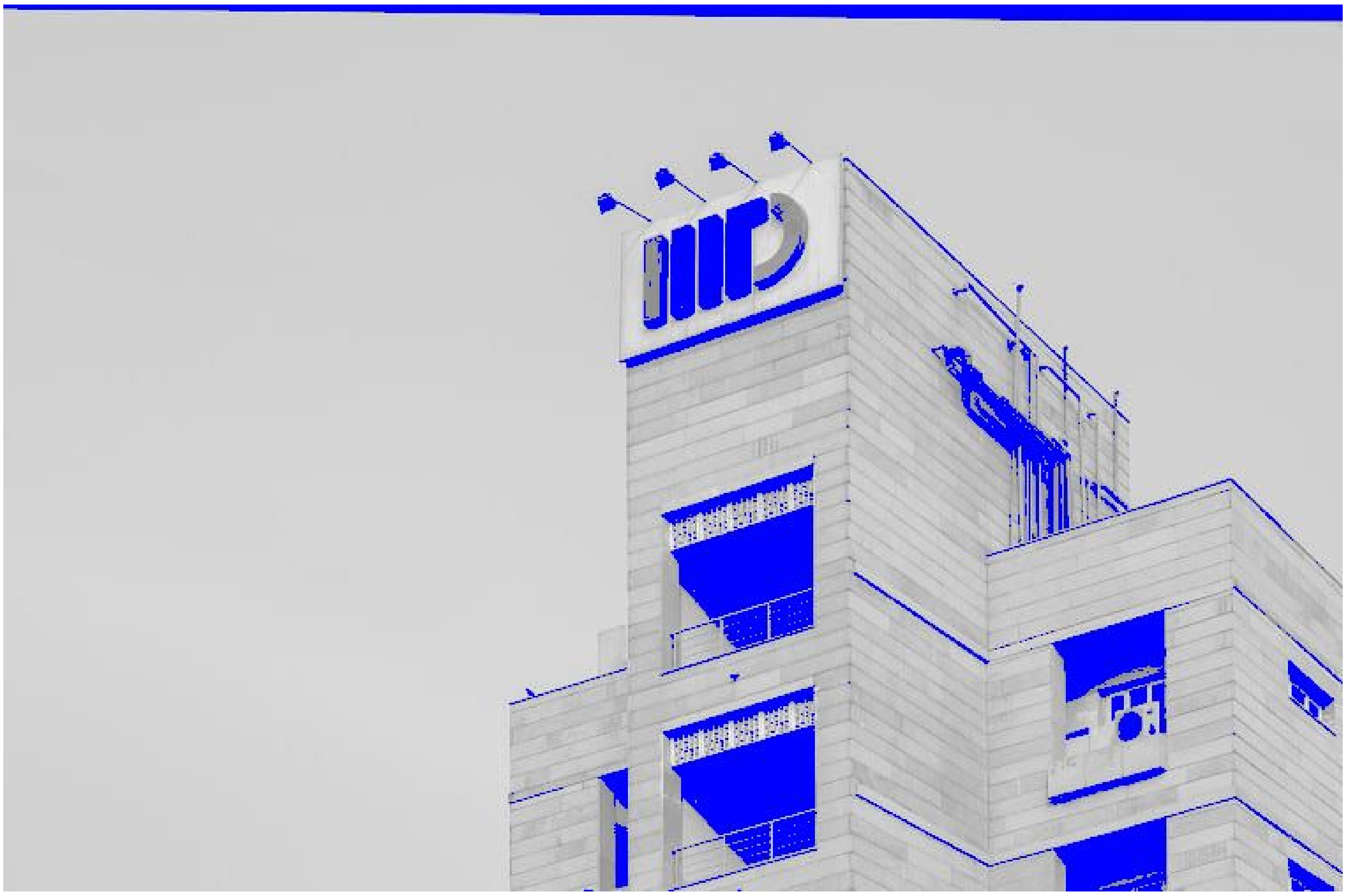
Approach:

- Make a histogram of the pixels of the given image.
- For every integer **i** in [0, 255), calculate the sum of TSSs of the two partitions formed and take the i corresponding to minimum TSS. That i will be our **optimal threshold**.
- Use the assumption that the object is present at the center of the image to find which of the two classes is foreground and which is background.
- Color the background with blue color.
- Display the final image.

Output:

```
^[shubham@inspiron-5590:~/Documents/IIIT DELHI/SEM 6/CV/CV_Homeworks/Mid-sem$ py q1.py iiitd1.png
Optimal Threshold = 138
Using Assumption: Object will be present at the center of the image
shubham@inspiron-5590:~/Documents/IIIT DELHI/SEM 6/CV/CV_Homeworks/Mid-sem$
```

Blue color denotes **background pixels.**



Code with Approach in comments:

```
# Otsu's algorithm for TSS

'''
Assumption: Object will be present at the center of the image.
'''


import cv2
import sys


def apply_assumption(grayImage, min_threshold):
    '''
    Assumption: Object will be present at the center of the image.
    1) Consider the rectangle whose length and width are ⅕ of given image dimensions and its center
coincides with that of the given image's center.
    2) Find the median of the pixels which are present in that rectangle. Let it be denoted by the
median_pixel.
    3) If median_pixel lies to the left of the min_threshold then the left side is foreground
otherwise the right side is foreground
    Note: This function will return which side is background.
    '''

    print("Using Assumption: Object will be present at the center of the image")
    rows, cols = grayImage.shape

    center_x, center_y = rows // 2, cols // 2

    # length and breadth are 20% of the original image
    l_box = int(0.2 * cols)
    b_box = int(0.2 * rows)

    #(x, y) denotes the top left coordinate of the rectangle we are considering
    x = center_x - b_box // 2
    y = center_y - l_box // 2

    count = 0 # count number of pixels in the sub-rectangle
    arr = [] # store the pixel values in the sub-rectangle

    for i in range(x, x + l_box):
       for j in range(y, y + b_box):
             arr.append(grayImage[i][j])
             count += 1

    arr.sort()
    median_pixel = arr[count // 2]

    if median_pixel <= min_threshold:
       return "right" # we are returning which side will be background

    return "left"


def extract_foreground(grayImage, originalImage, min_threshold):
    '''
    This will make the background of blue color according to optimal threshold returned by otse
    '''
    rows, cols = grayImage.shape

    background_side = apply_assumption(grayImage, min_threshold)
```

```python
    for row in range(rows):
       for col in range(cols):
             if (grayImage[row][col] <= min_threshold and background_side == "left") or
(grayImage[row][col] >= min_threshold and background_side == "right"):
                   originalImage[row][col][0] = 255 # blue
                   originalImage[row][col][1] = 0 # green
                   originalImage[row][col][2] = 0 # red

    cv2.imshow('Otsu\'s Output', originalImage)
    cv2.waitKey()
    cv2.destroyAllWindows()


def get_frequency_distribution(grayImage):
    '''
    returns a mapping of pixel values with their frequency in the grayImage of original Image
    '''
    rows, cols = grayImage.shape

    frequency = [0] * 256 # index denotes the pixel values so frequency[i] denotes the count of
pixels of value i in the grayImage

    for row in range(rows):
       for col in range(cols):
             frequency[grayImage[row][col]] += 1

    return frequency


def calculate_tss(frequency, start, end):
    '''
    'end' not inclusive in the range
    '''
    mean = 0
    count = 0 # number of pixel in given range

    for i in range(start, end):
       mean += frequency[i] * i
       count += frequency[i]

    if count != 0:
       mean /= count

    tss = 0
    for i in range(start, end):
       tss += frequency[i] * ((mean - i)**2)

    return tss


def otsu(grayImage, originalImage):
    '''
    Apply Otsu Algorithm on given Image
    Class 1: [0, minthreshold)
    Class 2: [minthreashold, 256]
    '''

    frequency = get_frequency_distribution(grayImage) # a mapping of pixel values with their
frequency in the grayImage of original Image

    min_threshold = 0
```

```
    min_tss = 0

    for i in range(1, 255):
       tss0 = calculate_tss(frequency, 0, i)
       tss1 = calculate_tss(frequency, i, 256)
       tss =  tss0 + tss1

       if tss <= min_tss or i == 1:
             min_tss = tss
             min_threshold = i

    print("Optimal Threshold =", min_threshold)
    return min_threshold


def main(image_name):
    originalImage = cv2.imread(image_name)
    grayImage = cv2.cvtColor(originalImage, cv2.COLOR_BGR2GRAY)
    min_threshold = otsu(grayImage, originalImage)
    extract_foreground(grayImage, originalImage, min_threshold)


if __name__ == "__main__":

    if len(sys.argv) != 2:
       print("Usage: python3 <script_name.py> <path_of_image>")
       exit(1)

    image_name = sys.argv[1]
    main(image_name)
```

Q3

Approach:

- Convert every pixel to the **center** of the **subcube** it belongs to and then find **histogram** of frequencies and store it in a python dictionary.(described in detail in the **Optimisations** section)
- Use **multiprocessing** to find the chebyshev distance for every pixel.
- **Normalise** the distances by dividing each with the maximum distance to get the saliency map.
- Now replace every pixel with its **saliency** values and display the final image.

Optimisations:

- Use of **multiprocessing** on each pixel to find its chebyshev distance to all pixels
- Used **histogram** approach to find and store frequency of every pixel.
- We know maximum 256*256*256 rgb values are possible and if we make a little change in rgb value of some colour then it does not make much difference.
  For example, (12, 231, 123) can be written as (11, 233, 121) and the corresponding color will not make much difference to our eyes.
  So, I divided the 256*256*256 cube of rgb values into smaller cube of length **SUB_CUBE_LEN** and map any pixel in any cube to the center pixel of that corresponding sub-cube.
  - For example: Let **SUB_CUBE_LEN = 8**, and we have pixel1 = (25, 53, 124) and pixel2 = (231, 121, 2). We will map these pixels to **(28, 52, 124)** and **(228, 124, 4)** respectively.

Complexity:

$$\left(\frac{1}{num\_of\_processors}\right)\left(\frac{256}{SUB\_CUBE\_LEN}\right)^2 + num\_pixels$$

Output:

As expected, not much difference can be seen in below two outputs with different SUB_CUBE_LEN and our code is executing in less than 3s with SUB_CUBE_LEN = 8, otherwise it would have taken more than a minute with SUB_CUBE_LEN = 1.

SUB_CUBE_LEN = 8

```
shubham@inspiron-5590:~/Documents/IIIT DELHI/SEM 6/CV/CV_Homeworks/Mid-sem$ py q3.py iiitd2.png 8
Execution Time = 2.9601895809173584 sec
shubham@inspiron-5590:~/Documents/IIIT DELHI/SEM 6/CV/CV_Homeworks/Mid-sem$
```

SUB_CUBE_LEN = 4

```
shubham@inspiron-5590:~/Documents/IIIT DELHI/SEM 6/CV/CV_Homeworks/Mid-sem$ py q3.py iiitd2.png 4
Execution Time = 5.537802219390869 sec
shubham@inspiron-5590:~/Documents/IIIT DELHI/SEM 6/CV/CV_Homeworks/Mid-sem$
```

Code:

```
import cv2
import sys
import numpy as np
import time
from multiprocessing import Pool

# length of cube
SUB_CUBE_LEN = 4
rgb_frequency_dic = {}
pixel_frequency_list = []
saliency = {}

num_pixels = 0

def process_pixel(pixel1):
    '''
    for the pixel1, calculate the chebyshev distance
    '''
    pixel1_b, pixel1_g, pixel1_r = map(int, pixel1.split())

    chebyshev_distance = 0
    for i in pixel_frequency_list:
       #  i = b, g, r, frequency of bgr
       chebyshev_distance += i[-1] * max(abs(pixel1_r - i[2]), \
                                abs(pixel1_b - i[0]), abs(pixel1_g - i[1]))

    return (pixel1, chebyshev_distance)


def calculate_saliency(rgb_frequency_dic, originalImage):
    '''
       calculate saliency map using multiprocessing
    '''
    pool = Pool(4)    # Create a multiprocessing Pool with 4 workers
    pixel_saliency = pool.map(process_pixel, rgb_frequency_dic.keys())

    max_sal = 0

    # find maximum saliency in map
    for x, y in pixel_saliency:
       max_sal = max(max_sal, y)

    if max_sal == 0:
       max_sal = 1

    # normalise saliency map
    for x, y in pixel_saliency:
       saliency[x] = y / max_sal

    return saliency


def map_saliency(originalImage, saliency):
    '''
    convert every pixel to its saliency
    '''

    height, width = originalImage.shape[:2]

    image = [[0] * width for _ in range(height)]
```

```python
    for i in range(height):
       for j in range(width):
             image[i][j] = saliency[str(originalImage[i][j][0]) + " " \
                   + str(originalImage[i][j][1]) + " " + str(originalImage[i][j][2])]

    image = np.float64(image)
    return image


def find_rgb_frequency(originalImage):
    '''
    find frequency of every pixel and store in dictionary
    '''
    global num_pixels
    height, width = originalImage.shape[:2]
    num_pixels = height * width

    # find which pixel belongs to which cube
    # and map it to median of that sub-cube
    map_pixel_to_cube = [0] * 256
    for i in range(256):
       map_pixel_to_cube[i] = i - i % SUB_CUBE_LEN + (SUB_CUBE_LEN >> 1)

    for i in range(height):
       for j in range(width):
             for k in range(3):
                   originalImage[i][j][k] = map_pixel_to_cube[originalImage[i][j][k]]

             pixel = str(originalImage[i][j][0]) + " " + \
                         str(originalImage[i][j][1]) + " " + str(originalImage[i][j][2])

             try:
                   rgb_frequency_dic[pixel] += 1
             except:
                   rgb_frequency_dic[pixel] = 1

    for i, j in rgb_frequency_dic.items():
       b, g, r = map(int, i.split())
       pixel_frequency_list.append([b, g, r , j])


def main(image_path):
    originalImage = cv2.imread(image_path)
    find_rgb_frequency(originalImage)
    calculate_saliency(rgb_frequency_dic, originalImage)
    final_image = map_saliency(originalImage, saliency)
    return final_image


if __name__ == '__main__':
    if len(sys.argv) != 3:
       print("Usage: python3 code.py <path_of_image> <SUB_CUBE_LEN>")
       exit(1)

    image_path = sys.argv[1]
    SUB_CUBE_LEN = int(sys.argv[2])

    # start timer
    start = time.time()
    final_image = main(image_path)
```

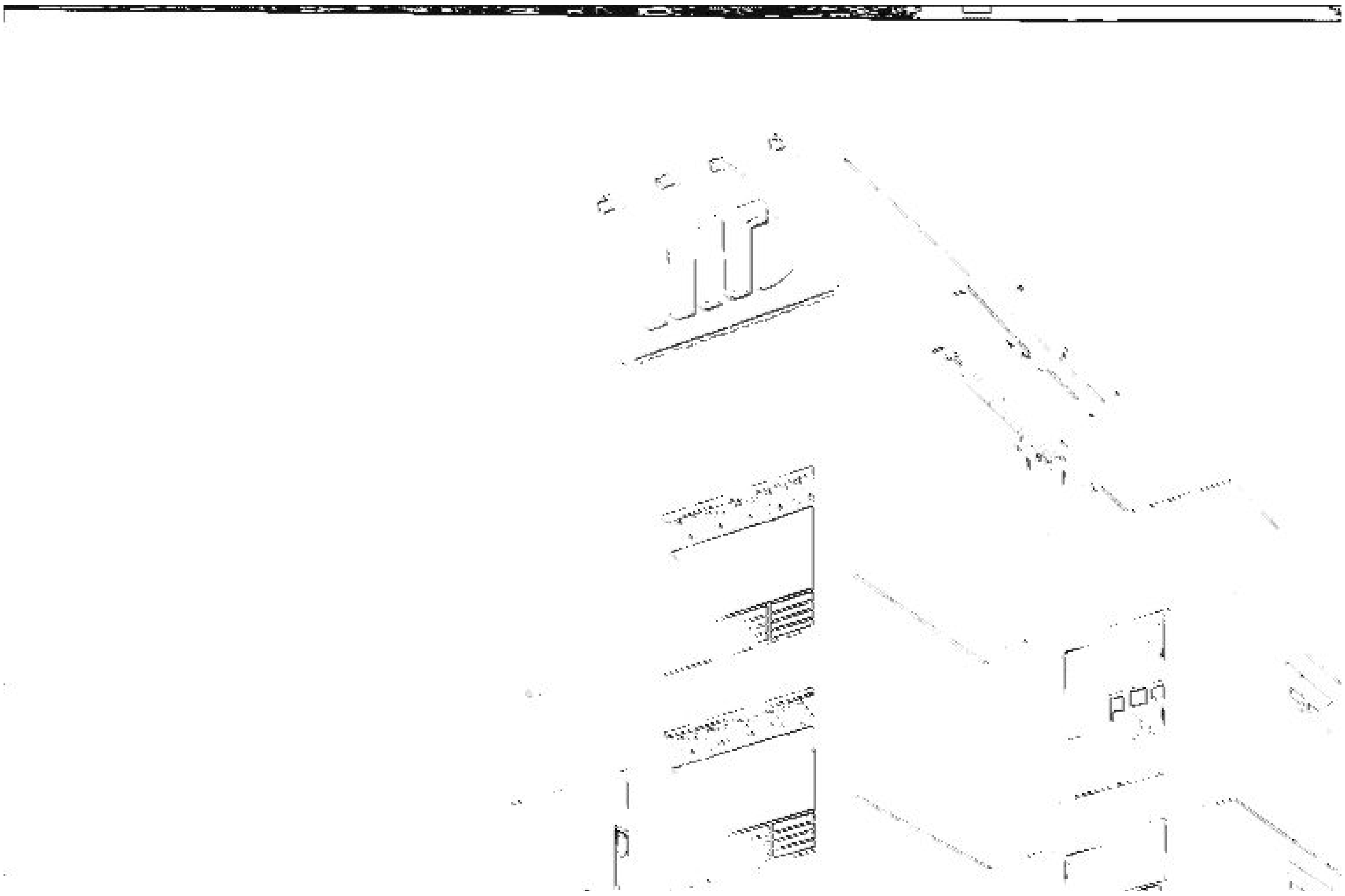
```python
# end timer
end = time.time()
print("Execution Time =", end - start,"sec")

# display image
cv2.imshow('saliency image', final_image)
cv2.waitKey()
cv2.destroyAllWindows()
```

Q4

Approach:

- Find the min-max ratio and round it off to get a 8 bit number for every pixel with its neighbours in clockwise direction.
- Display the final image.

Output:



Code:

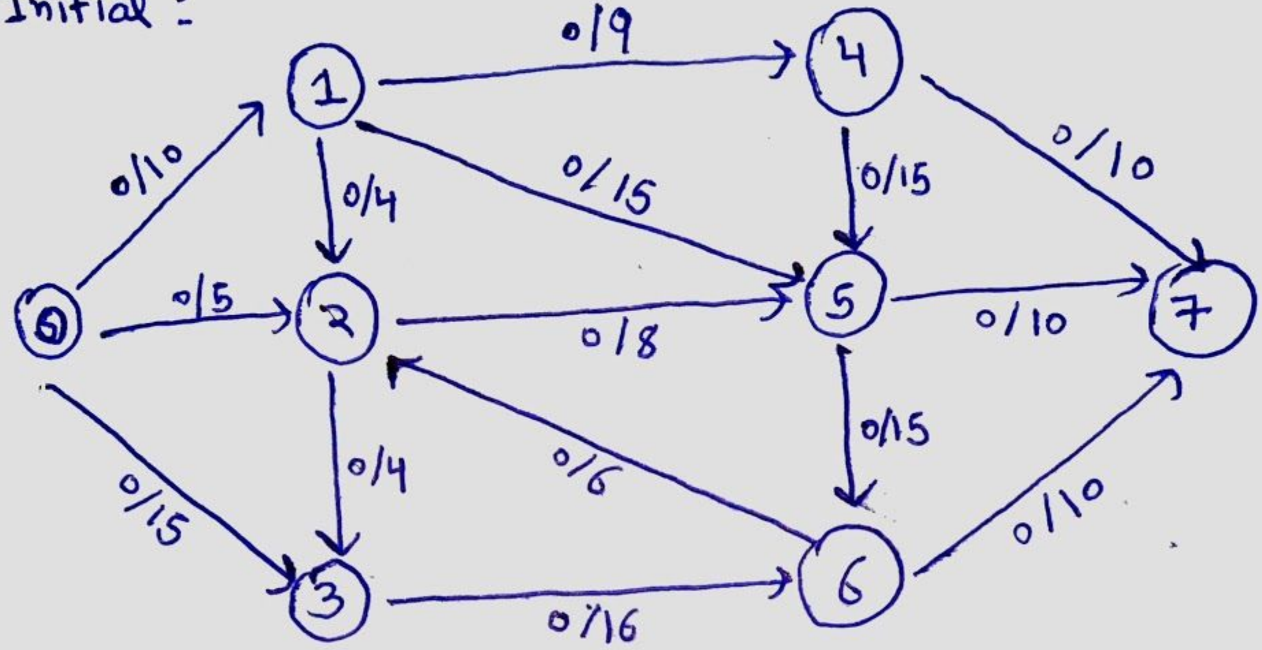
```
import cv2
import sys
import copy
import numpy as np
import itertools

def min_max_ratio_to_binary_num(a, b):
    # to avoid NANs
    a += 0.00001
    b += 0.00001
    return round(min(a, b) / max(a, b))

def find_lbp_feature_map(grayImage):
    height, width = grayImage.shape

    image = np.zeros(grayImage.shape, dtype = np.uint8)

    for i in range(height):
       for j in range(width):
            # find decimal for every pixel grayImage[i][j]

            decimal = 0
            directions = [(-1, -1), (-1, 0), (-1, 1), (0 , 1), (1, 1), (1, 0), (1, -1), (0, -1)]

            for x, y in directions:
                  try:
                        z = min_max_ratio_to_binary_num(grayImage[i + x, j + y], grayImage[i][j])
                  except:
                        # this will excecute when concerned pixel is at boundary of patch
                        z = 1
                  decimal = 2 * decimal + z

            image[i][j] = decimal

    return image

def main(image_path):
    originalImage = cv2.imread(image_path)
    grayImage = cv2.cvtColor(originalImage, cv2.COLOR_BGR2GRAY)

    final_image = find_lbp_feature_map(grayImage)

    cv2.imshow('lbp', final_image)
    cv2.waitKey()
    cv2.destroyAllWindows()

if __name__ == '__main__':
    if len(sys.argv) != 2:
       print("Usage: python3 code.py <path_of_image>")
       exit(1)

    image_path = sys.argv[1]
    main(image_path)
```

Q5

Q -5

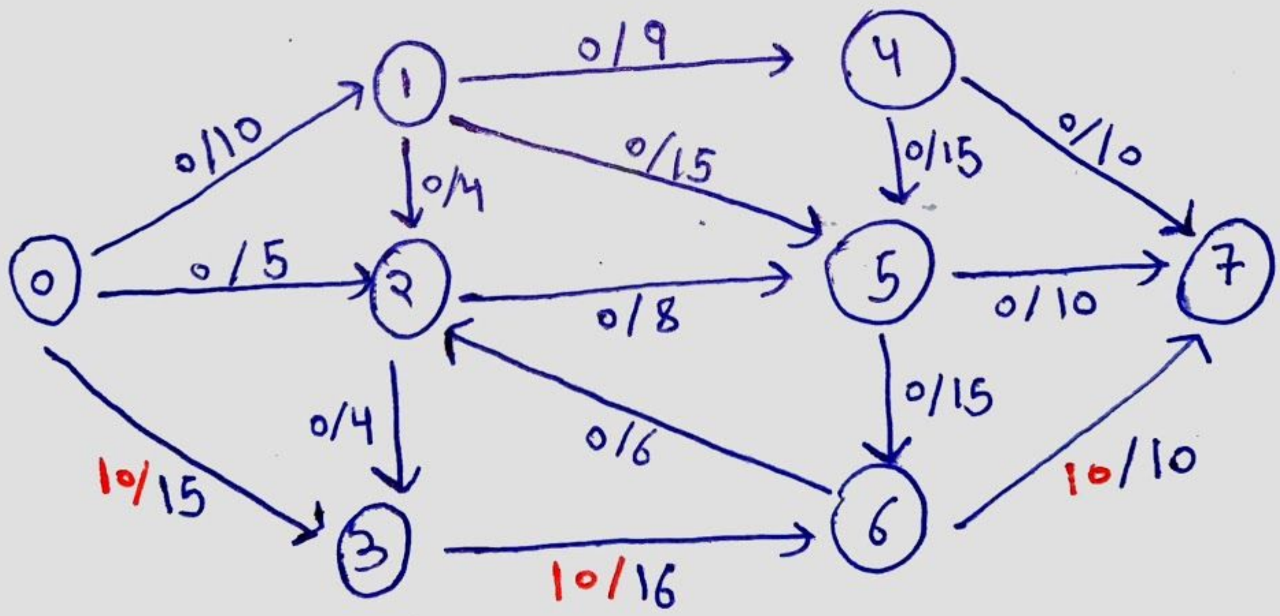
Notation: Flow/Capacity

Initial:



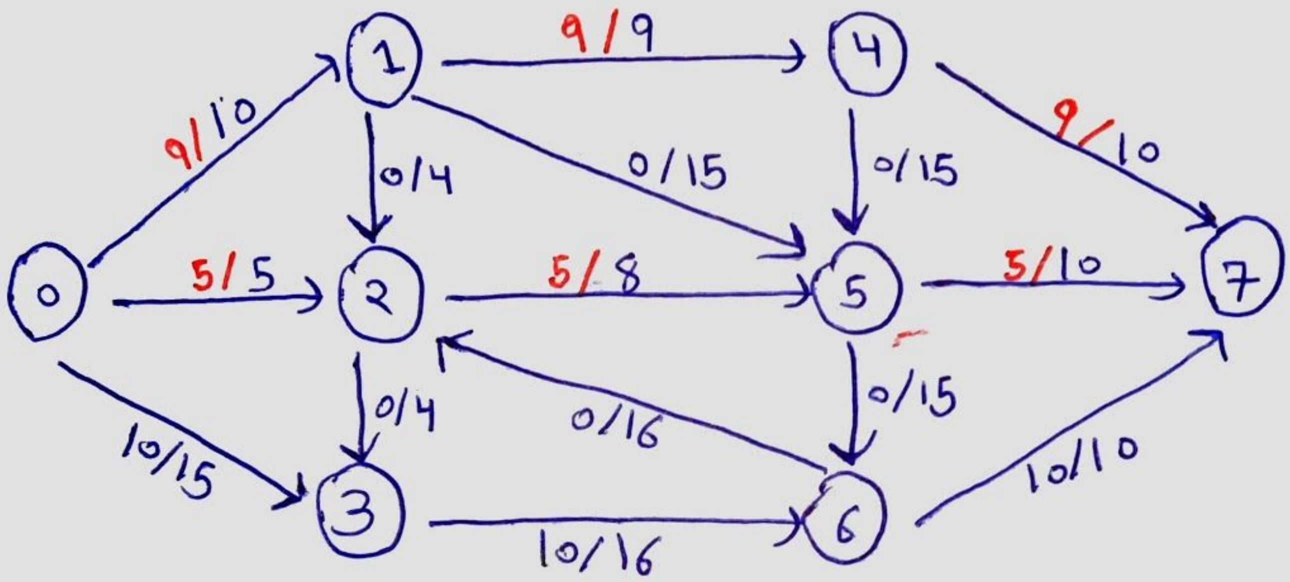
Iteration-1: Path = 0 - 3 - 6 - 7

bottleneck = 10; pipe(6,7)



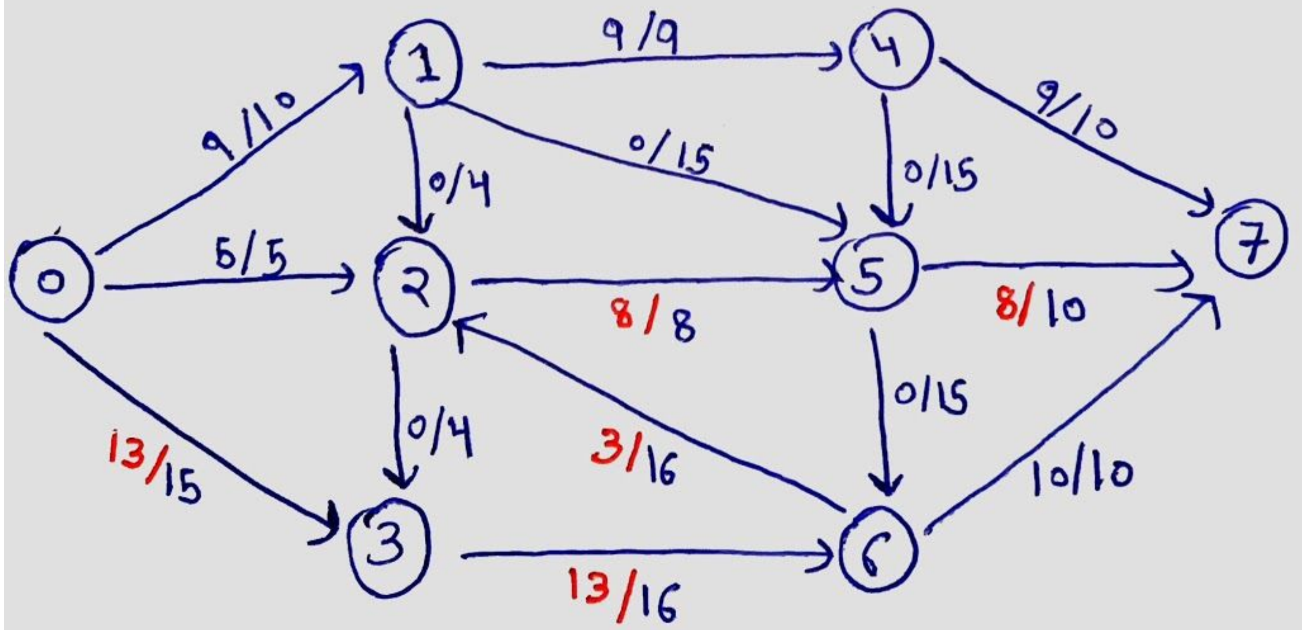
Iteration-3 - Path → 0-1-4-7 bottleneck = 9 pipe(1,4)

Path → 0-2-5-7 bottleneck = 5 pipe(0,2)



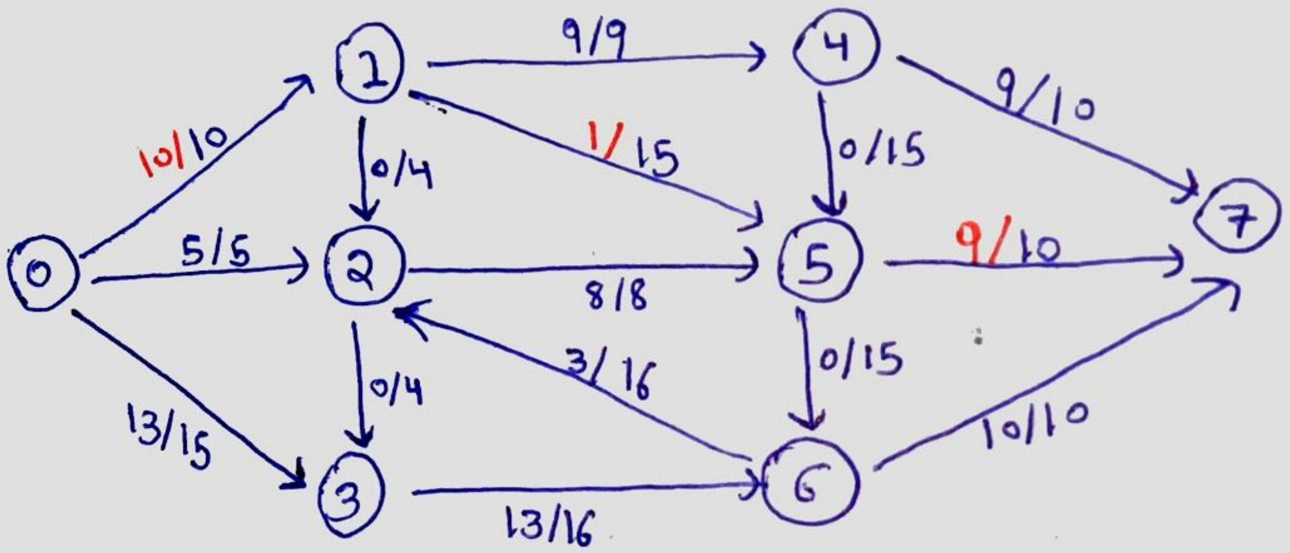
Iteration-4: path - 0-3-6-2-5-7

bottleneck - 3 pipe(2,5)



Iteration-5: path - 0 - 1 - 5 - 7

bottleneck - 1 pipe(0,1)



Maxflow = 10 + 5 + 13 = 28

~~mincutes~~

reachable - 0, 3, 6, 2

non-reachable - 1, 4, 5, 7

Mincut = 10 + 8 + 10

$\downarrow$ pipe(0,1) $\quad \downarrow$ pipe(~~5~~2,5) $\quad$ pipe(6,7)

= 28

pipe(1,2) and pipe(5,6) are also in min-cut with zero flow