Yoshika Takezawa

CS 558, Computer Vision

Hw2

5/4/19

Code:

```
# yoshika takezawa

# 5/1/19

# Computer Vision hw2

# I pledge my honor that i have abided by the stevens honor system

# kms -> week 6 pg 10

# slic -> week 6 pg 41


import math

import random

import sys

import numpy as np

import cv2


# given a pixel's bgr values, and the array of centroids,

# will output the index of the shortest distance



def bgr_distance(pix, bgrlist):

    # distance funct: sqrt((r-r1)^2+(g-g1)^2+(b-b1)^2)

    b, g, r = pix

    smallest = 200000
    for l in range(len(bgrlist)):
        b1, g1, r1 = bgrlist[l]
        # temp skips the sqrt b/c we are just comparing
        temp = (r-r1)**2+(g-g1)**2+(b-b1)**2
        if (temp < smallest):
            smallest = temp
            index = l
    return index



# k means segmentation takes image, randomly selects points as
# initial colors, processes image and colors until convergence



def kmeans(img, k=10):
    mean_lst = []
    cluster_lst = [[] for _ in range(k)]
    rows = img.shape[0]
    columns = img.shape[1]

    # getting first k unique random samples
    while(len(mean_lst) < k):
        x = random.randint(0, columns-1)
        y = random.randint(0, rows-1)
        mean = [img.item(y, x, 0), img.item(y, x, 1), img.item(y, x, 2)]
        if (mean in mean_lst):
            continue
        mean_lst.append(mean)
    # iterating through pixels until convergence
    while (1):
        temp_lst = []
        # iterating through pixels
        for i in range(columns):
            for j in range(rows):
                p = [img.item(j, i, 0), img.item(j, i, 1), img.item(j, i, 2)]
                ind = bgr_distance(p, mean_lst)
                cluster_lst[ind].append(p)
        # get mean for all clusters
        for c in range(len(cluster_lst)):
            temp = [(sum(l)//len(l)) for l in zip(*cluster_lst[c])]
            temp_lst.append(temp)
        # breaking condition
        if np.all(temp_lst == mean_lst):
            break
        else:
            mean_lst = temp_lst
            for cl in cluster_lst:
                cl.clear()
        print(mean_lst)
    # setting each pixel to its respective avg cluster color
    for n in range(columns):
        for m in range(rows):
            curbgr = [img.item(m, n, 0), img.item(m, n, 1), img.item(m, n, 2)]
            i = bgr_distance(curbgr, mean_lst)
```

```
            img.itemset((m, n, 0), mean_lst[i][0])
            img.itemset((m, n, 1), mean_lst[i][1])
            img.itemset((m, n, 2), mean_lst[i][2])
    return img

# given an image, square filter matrix and the length,
# outputs the filtered image


def filter(img, fm, len, channel):
    height, width = img.shape
    # create black image
    res = np.zeros((height, width), dtype=img.dtype)
    off = len//2
    for j in range(off, height-off):
        for i in range(off, width-off):
            fval = block_sum(img, fm, len, j, i)
            res.itemset((j, i), fval)
    return res

# using the filtermatrix's weight distribution, adds all the values


def block_sum(img, fm, len, y, x):
    total = 0.0
    off = len//2
    # iterating through each matrix element
    for j in range(len):
        for i in range(len):
            curx = x + i - off
            cury = y + j - off
            total += img.item(cury, curx) * fm[j][i]
    return total



def eudist(pix, clist):
    y, x, b, g, r = pix
    smallest = -100
    index = 0
    for l in range(len(clist)):
        y1, x1, b1, g1, r1 = clist[l]
        # temp skips the sqrt b/c we are just comparing
        temp = ((x-x1)/2)**2 + ((y-y1)/2)**2 + (r-r1)**2+(g-g1)**2+(b-b1)**2
        if (temp < smallest) | (smallest < 0):
            smallest = temp
            index = l
    return index



def kmeans_5d(img, cent, S):
    cluster_lst = [[] for _ in range(len(cent))]
    rows = img.shape[0]
    columns = img.shape[1]
    iter = 0
```

```
while (1):
    iter += 1
    print(cent)
    new_cent = []
    # iterating through pixels
    for i in range(columns):
        for j in range(rows):
            p = [j, i, img.item(j, i, 0), img.item(
                j, i, 1), img.item(j, i, 2)]
            # get list of centers within 2Sx2S neighborhood
            eligible = []
            for w in cent:
                if (w[0] in range(p[0]-(2*S), p[0]+(2*S)+1)) & (w[1] in range(p[1]-(2*S), p[1]+(2*S)+1)):
                    eligible.append(w)
            if (eligible == []):
                print("hey")
                continue
            ind = eudist(p, eligible)
            cluster_lst[cent.index(eligible[ind])].append(p)
    # get mean for all clusters
    for cc in range(len(cluster_lst)):
        tempt = [(sum(l)//len(l)) for l in zip(*cluster_lst[cc])]
        # ty, tx, _,_,_ = tempt
        # tempt = [ty, tx, img.item(ty, tx, 0), img.item(ty, tx, 1), img.item(ty, tx, 2)]
        new_cent.append(tempt)
    # breaking condition, residual error
    E = 0
```

```
        for k in range(len(cent)):
            E += abs(cent[k][0]- new_cent[k][0])
            E += abs(cent[k][1]- new_cent[k][1])
            E += abs(cent[k][2]- new_cent[k][2])
            E += abs(cent[k][3]- new_cent[k][3])
            E += abs(cent[k][4]- new_cent[k][4])
        if np.all(new_cent == cent) | (iter==10) | (E <=15):
            break
        else:
            cent = new_cent.copy()
            for cl in cluster_lst:
                cl.clear()
    # setting each pixel to its respective avg cluster color
    for gi in range(len(cluster_lst)):
        for pix in cluster_lst[gi]:
            py,px,_,_,_ = pix
            img.itemset((py,px, 0), cent[gi][2])
            img.itemset((py,px, 1), cent[gi][3])
            img.itemset((py,px, 2), cent[gi][4])
    return img



def SLIC(image, S=50):
    height = image.shape[0]
    width = image.shape[1]

    # initializing centroids in middle of 50x50 blocks
```

```
centroids_coords = []
for y in range(S//2, height, S):
    for x in range(S//2, width, S):
        centroids_coords.append([y, x, image.item(y, x, 0),
                                image.item(y, x, 1), image.item(y, x, 2)])
# compute bgr gradients and move centroids to the smallest gradient
hsob = [[1, 2, 1],
        [0, 0, 0],
        [-1, -2, -1]]
vsob = [[1, 0, -1],
        [2, 0, -2],
        [1, 0, -1]]
# skipped sqrt in B/G/R_comb because will be squared later
B_x = filter(image[:, :, 0], hsob, 3, 0)
B_y = filter(image[:, :, 0], vsob, 3, 0)
B_comb = np.power(B_x, 2.0) + np.power(B_y, 2.0)

G_x = filter(image[:, :, 1], hsob, 3, 1)
G_y = filter(image[:, :, 1], vsob, 3, 1)
G_comb = np.power(G_x, 2.0) + np.power(G_y, 2.0)

R_x = filter(image[:, :, 2], hsob, 3, 2)
R_y = filter(image[:, :, 2], vsob, 3, 2)
R_comb = np.power(R_x, 2.0) + np.power(R_y, 2.0)

combined = B_comb + G_comb + R_comb
# in the 3x3 windows surrounding center, finds smallest gradient
```

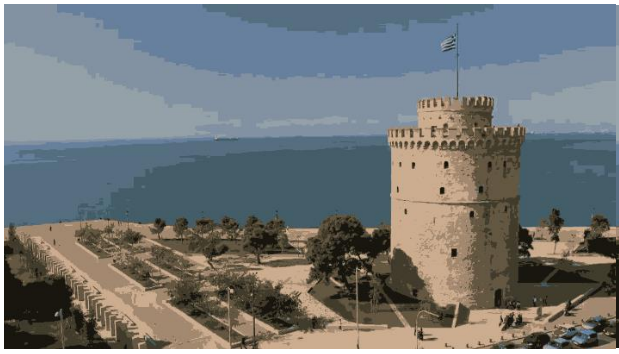
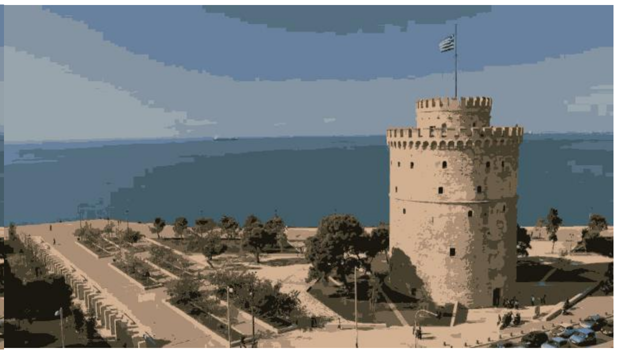
```
    for b in range(S//2, height, S):
        for a in range(S//2, width, S):
            index = (a//S) + ((b//S)*(width//S))
            small_temp = -1
            for t in range(b-1,b+2):
                for u in range(a-1, a+2):
                    cur_grad = combined.item(t, u)
                    if (cur_grad < small_temp) | (small_temp < 0):
                        small_temp = cur_grad
                        centroids_coords[index] = [
                            t, u, image.item(t, u, 0), image.item(t, u, 1), image.item(t, u, 2)]
    # use centroids as start for 5d kmeans
    kmeans5d = kmeans_5d(image, centroids_coords, S)

    return kmeans5d



if __name__ == '__main__':
    # k means segamnetation (part 1)

    wt_image = cv2.imread("white-tower.png", cv2.IMREAD_COLOR)
    seg_result = kmeans(wt_image, 10)
    cv2.imwrite('kmeans.png', seg_result)

    # # SLIC (part 2)
    slic_image = cv2.imread("wt_slic.png", cv2.IMREAD_COLOR)
    slic_res = SLIC(slic_image, 50)
```

```
cv2.imwrite('slic.png', slic_res)
```

K-means Segmentation:

The k-means algorithm starts by randomly selecting pixels in the photo, extracting its RGB values, and making sure the other values do not coincide with other initial values. This is done to ensure that the initial seeds are within the photo's color scheme. The program then iterates through the photo and decides which cluster is closest (in RGB values) to each pixel's values. After the photo is processed, each cluster finds the average RGB value. If the old cluster means match the new means, the loop terminates; otherwise, the process is repeated. This was done in order to ensure that the resulting images would be consistent in respect to the colors

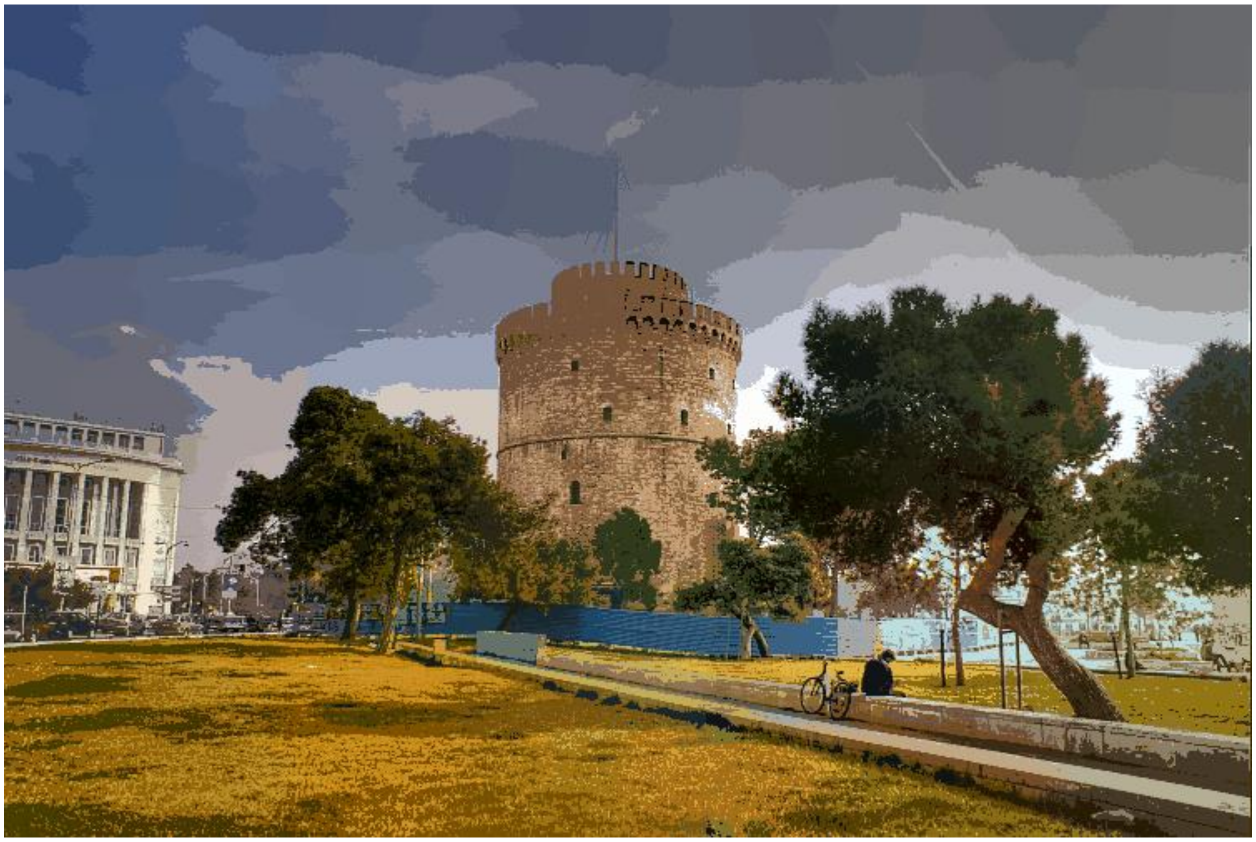
As shown from the photos, the same algorithm results in slightly different photos due to its random nature. Due to the random nature, the runtimes vary greatly such as the first photo taking 7 iterations, whereas the second photo took 23 iterations.

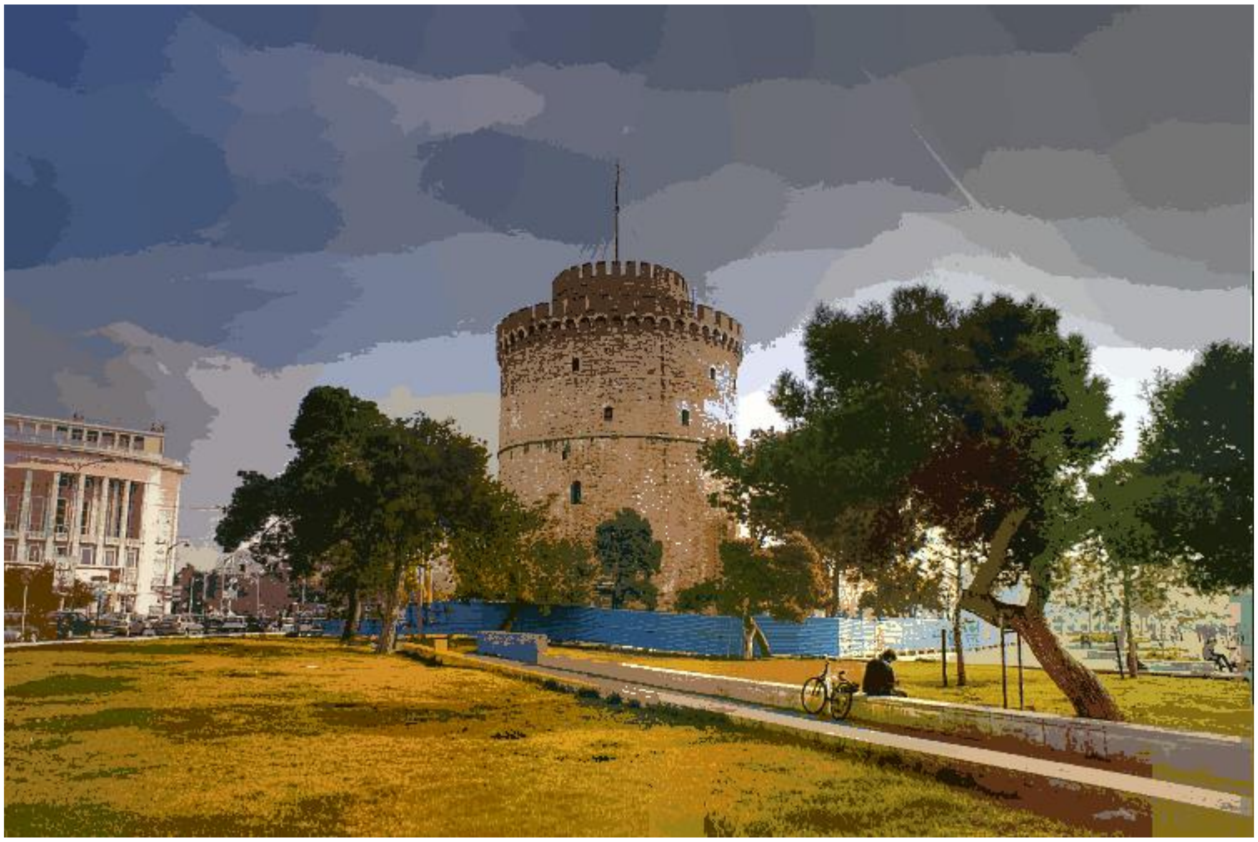
SLIC:

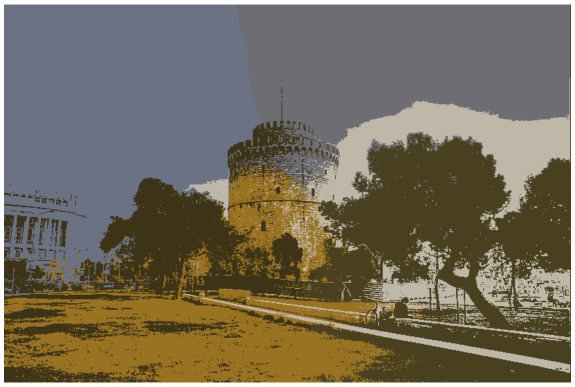
The concept behind SLIC is simple, but efficiently implementing it was tricky to handle. Following the algorithm outline from the assignment, centroids were instantiated at the centers of 50x50 blocks. These are later moved to the neighboring pixel with the smallest gradient in all color channels (RGB). The k-means part of the algorithm was modified to take in account of pixel distance and the RBG values. Additionally, once cluster means were decided, I chose to stick with the mean y,x,r,g,b values as opposed to y,x, img(y,x)'s rgb values because it produced stable mean values.

The following is the result of S=50, averaging all the values…



The following is the result of S=50, keeping the mean xy's rgb values…





The image to the left is the result of S=250.

From the article introducing the SLIC algorithm (link), on page 3, there is a high level implementation of it. I did attempt to replicate it, but even with S=150, the algorithm was too slow to process in a timely manner. (This code is in the zip under hw2_fail.py)

To make the process faster, the photo was iterated, identified which cluster centers were close to the pixel, and 5d Euclidean distance allowed the program to make a final decision on which cluster to go into.