# hw2_p3

February 17, 2018

```
In [1]: import numpy as np
        import pandas as pd
        import tensorflow as tf
        import time
        from scipy.optimize import linear_sum_assignment

        %matplotlib inline
        import matplotlib
        import matplotlib.pyplot as plt

        import pymesh
        from pyntcloud import PyntCloud

        from tf_emddistance import emd_distance

In [2]: # This section of code is copied from problem 2
        def triangle_area(x):
            a = x[:,0,:] - x[:,1,:]
            b = x[:,0,:] - x[:,2,:]
            cross = np.cross(a, b)
            area = 0.5 * np.linalg.norm(np.cross(a, b), axis=1)
            return area

        def euclidean_distance_matrix(x):
            r = np.sum(x*x, 1)
            r = r.reshape(-1, 1)
            distance_mat = r - 2*np.dot(x, x.T) + r.T
            #return np.sqrt(distance_mat)
            return distance_mat

        def update_farthest_distance(far_mat, dist_mat, s):
            for i in range(far_mat.shape[0]):
                far_mat[i] = dist_mat[i,s] if far_mat[i] > dist_mat[i,s] else far_mat[i]
            return far_mat, np.argmax(far_mat)

        def init_farthest_distance(far_mat, dist_mat, s):
            for i in range(far_mat.shape[0]):
                far_mat[i] = dist_mat[i,s]
            return far_mat

        def farthest_point_sampling(obj_file, num_samples=1000):
            mesh = pymesh.load_mesh(obj_file)
            faces = mesh.vertices[mesh.faces]
            area = triangle_area(faces)
            total_area = np.sum(area)

            set_P = []
            for i in range(faces.shape[0]):
                num_gen = area[i] / total_area * 10000
                for j in range(int(num_gen)+1):
                    r1, r2 = np.random.rand(2)
                    d = (1-np.sqrt(r1)) * faces[i,0] + np.sqrt(r1)*(1-r2) * faces[i,1] + np.sqrt
                    set_P.append(d)

            set_P = np.array(set_P)
            num_P = set_P.shape[0]

            distance_mat = euclidean_distance_matrix(set_P)

            set_S = []
            s = np.random.randint(num_P)
            far_mat = init_farthest_distance(np.zeros((num_P)), distance_mat, s)

            for i in range(num_samples):
                set_S.append(set_P[s])
                far_mat, s = update_farthest_distance(far_mat, distance_mat, s)

            return np.array(set_S, dtype=np.float32)

        def pointcloud_distance_matrix(x1, x2):
            a = tf.expand_dims(x1, axis=1)
            b = tf.expand_dims(x2, axis=0)
            distance_mat = tf.norm(a - b, axis=-1)
            return distance_mat
```

```
In [3]: # Sample cloud points -- same procedure as problem 2
        teapot_pts = farthest_point_sampling('../teapot.obj', 200)
        violin_pts = farthest_point_sampling('../violin_case.obj', 200)
```

```
In [4]: # compute distance matrix between 2 cloud points
        with tf.Session() as sess:
            distance_mat = sess.run(pointcloud_distance_matrix(teapot_pts, violin_pts))
```

```
In [5]: class hungrary():
```

```
def __init__(self, weight):
    self.n = weight.shape[0]

    self.w = np.copy(weight)
    # cost matrix
    self.c = np.copy(weight)
    self.m = np.zeros((self.n, self.n), dtype=int)

    # record row and col covers
    self.RowCover = np.zeros((self.n), dtype=bool)
    self.ColCover = np.zeros((self.n), dtype=bool)
    # record augment paths
    self.path = np.zeros((2*self.n, 2), dtype=int)

# main program, run the algo through steps
def run_hungrary(self):
    done = False
    step = 1
    while not done:
        if step == 1:
            step = self.step1()
        elif step == 2:
            step = self.step2()
        elif step == 3:
            step = self.step3()
        elif step == 4:
            step = self.step4()
        elif step == 5:
            step = self.step5()
        elif step == 6:
            step = self.step6()
        elif step == 7:
            done = True

# Each row subtract smallest elements
def step1(self):
    self.c -= np.min(self.c, axis=1, keepdims=True)
    return 2

# star zeros
def step2(self):
    for u in range(self.n):
        for v in range(self.n):
            if self.c[u,v] == 0 and not self.RowCover[u] and not self.ColCover[v]:
                self.m[u, v] = 1
                self.RowCover[u] = True
                self.ColCover[v] = True
                break
        self.clear_covers()
        return 3

    # cover cols with starred zeros. check if done
    def step3(self):
        for u in range(self.n):
            for v in range(self.n):
                if self.m[u, v] == 1:
                    self.ColCover[v] = True

        colcnt = np.sum(self.ColCover)

        if colcnt >= self.n:
            return 7
        else:
            return 4

    # find noncovered zero and prime it (starred as 2)
    def step4(self):
        while True:
            row, col = self.find_a_zero()
            if row == -1:
                return 6
            else:
                self.m[row, col] = 2
                if self.star_in_row(row):
                    col = self.find_star_in_row(row)
                    self.RowCover[row] = True
                    self.ColCover[col] = False
                else:
                    self.path_row_0 = row
                    self.path_col_0 = col
                    return 5

    # use augment algo to increase matches
    def step5(self):
        done = False
        self.path_count = 1
        self.path[self.path_count-1, 0] = self.path_row_0
        self.path[self.path_count-1, 1] = self.path_col_0

        while not done:
            row = self.find_star_in_col(self.path[self.path_count-1, 1])
            if row > -1:
                self.path_count += 1
                self.path[self.path_count-1, 0] = row
                self.path[self.path_count-1, 1] = self.path[self.path_count-2, 1]
            else:
                    done = True
                if not done:
                    col = self.find_prime_in_row(self.path[self.path_count-1, 0])
                    self.path_count += 1
                    self.path[self.path_count-1, 0] = self.path[self.path_count-2, 0]
                    self.path[self.path_count-1, 1] = col

        self.augment_path()
        self.clear_covers()
        self.erase_prime()
        return 3

    # add minval val to double covered elements and subtract it to noncovered elements
    def step6(self):
        minval = self.find_smallest()
        for u in range(self.n):
            for v in range(self.n):
                if self.RowCover[u]:
                    self.c[u,v] += minval
                if not self.ColCover[v]:
                    self.c[u,v] -= minval
        return 4

    # find first uncovered zero
    def find_a_zero(self):
        for u in range(self.n):
            for v in range(self.n):
                if self.c[u,v] == 0 and not self.RowCover[u] and not self.ColCover[v]:
                    return u, v
        return -1, -1

    def star_in_row(self, row):
        for v in range(self.n):
            if self.m[row, v] == 1:
                return True
        return False

    def find_star_in_row(self, row):
        for v in range(self.n):
            if self.m[row, v] == 1:
                return v
        return -1

    def find_star_in_col(self, col):
        for u in range(self.n):
            if self.m[u, col] == 1:
                return u
        return -1
    def find_prime_in_row(self, row):
        for v in range(self.n):
            if self.m[row, v] == 2:
                return v
        return -1

    def augment_path(self):
        for p in range(self.path_count):
            if self.m[self.path[p,0], self.path[p,1]]  == 1:
                self.m[self.path[p,0], self.path[p,1]] = 0
            else:
                self.m[self.path[p,0], self.path[p,1]] = 1

    def clear_covers(self):
        self.RowCover = np.zeros((self.n), dtype=bool)
        self.ColCover = np.zeros((self.n), dtype=bool)

    def erase_prime(self):
        for u in range(self.n):
            for v in range(self.n):
                if self.m[u,v] == 2:
                    self.m[u,v] = 0

    def find_smallest(self):
        minval = np.max(self.c)
        for u in range(self.n):
            for v in range(self.n):
                if self.c[u,v] < minval and not self.RowCover[u] and not self.ColCover[v
                    minval = self.c[u,v]
        return minval
```

```
In [7]: H = hungrary(distance_mat)
        H.run_hungrary()
```

```
In [8]: # EMD computed by my hungarian algorithm
        np.sum(H.w * H.m)
```

```
Out[8]: 10703.283664703369
```

```
In [9]: # EMD computed by scipy linear_sum_assignment
        row_ind, col_ind = linear_sum_assignment(distance_mat)
        print(distance_mat[row_ind, col_ind].sum())
```

```
10703.3
```