1. Out put images `q1_Superpixels.png` and `q1_result.png` are shown in Figure 1 and 2 correspondingly. This super-pixel covers a part of two yellow peppers of the lower middle of the input image.

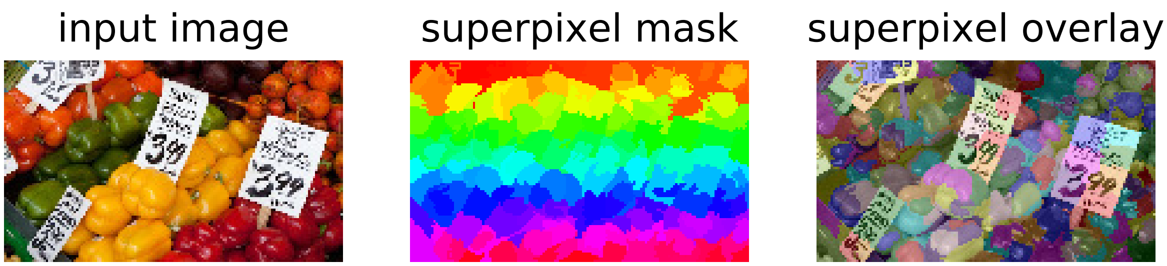

Figure 1: Superpixels of Question 1

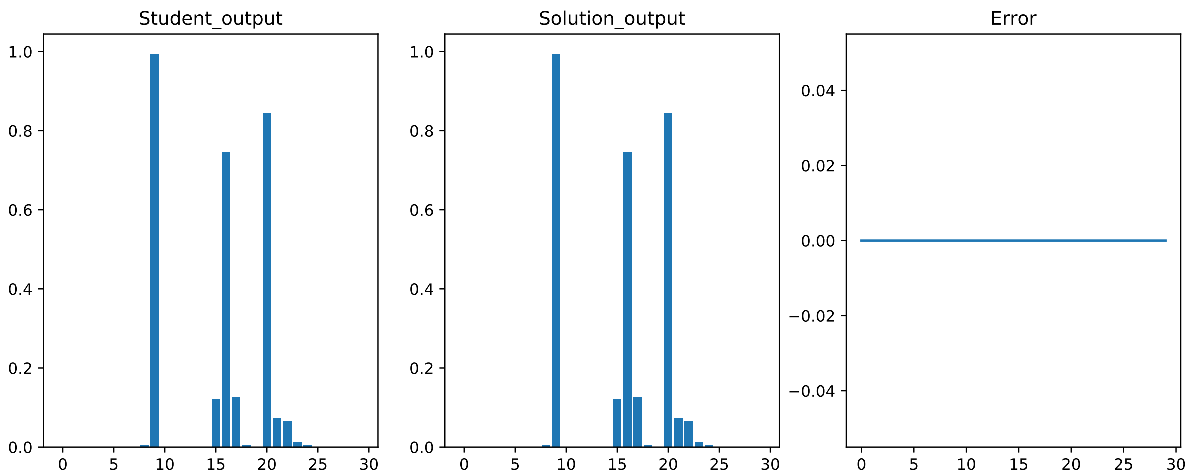

Figure 2: Results of Question 1

2. (a) Out put images `q2_Superpixels.png` and `q2_result.png` are shown in Figure 3 and 4 correspondingly.

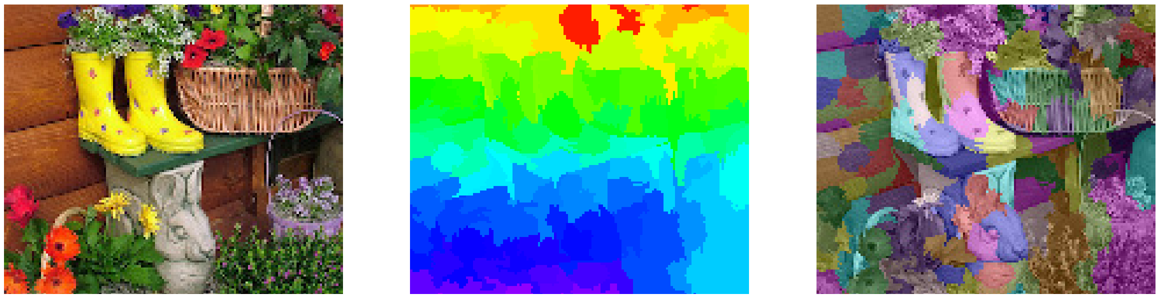

Figure 3: Superpixels of Question 2

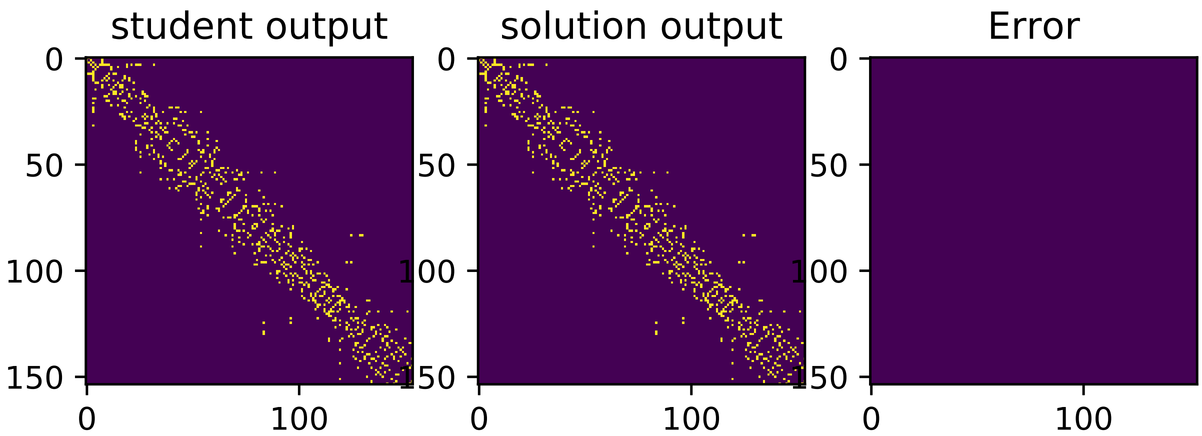

Figure 4: Results of Question 2

(b) We find the average node degree is 5.33766 for the adjacency matrix in this problem.

(c) The adjacency graph is not perfectly banded diagonal matrix because superpixels in images have various sizes. We can see that some superpixels are cover a lot of space along the y direction, while narrow along x direction. In this way, it is able to adjunct to some superpixel, whose centroid is far from its centroid. Moreover, some superpixel are relatively small and only adjunct to four superpixels around it.

3. (a) Out put images `q3_Superpixels.png` and `q3_result.png` are shown in Figure 5 and 6 correspondingly.

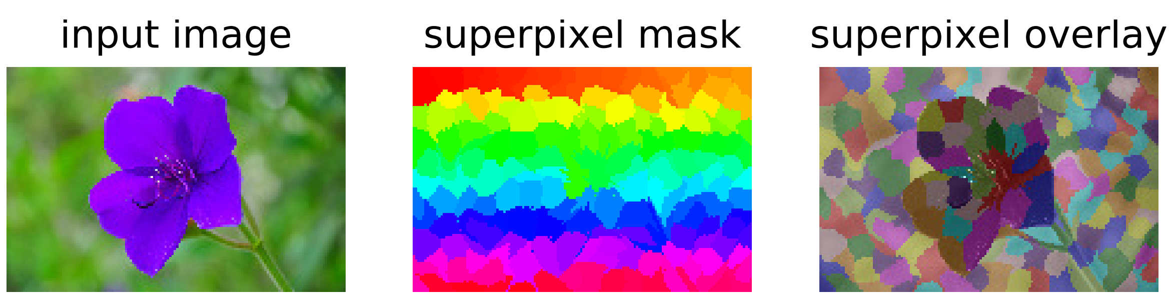

Figure 5: Superpixels of Question 3

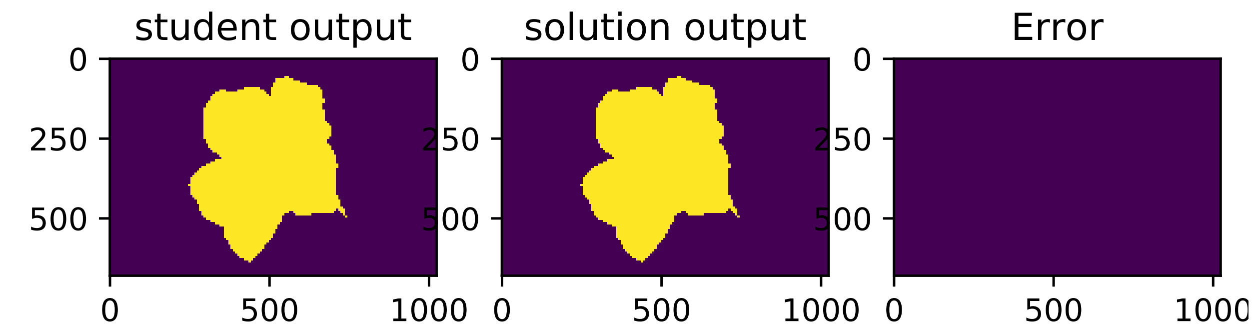

Figure 6: Results of Question 3

(b) We have shown Adjacency matrix and capacity image in the figure 7. We know that adjacency matrix is an unweighted bidirected graph, and capacity matrix is an weighted bidirected graph. Capacity image has two extra nodes, source and sink than the adjacency matrix. All edges in adjacency matrix are preserved in capacity matrix, and source and sink are connected to all nodes besides each other.

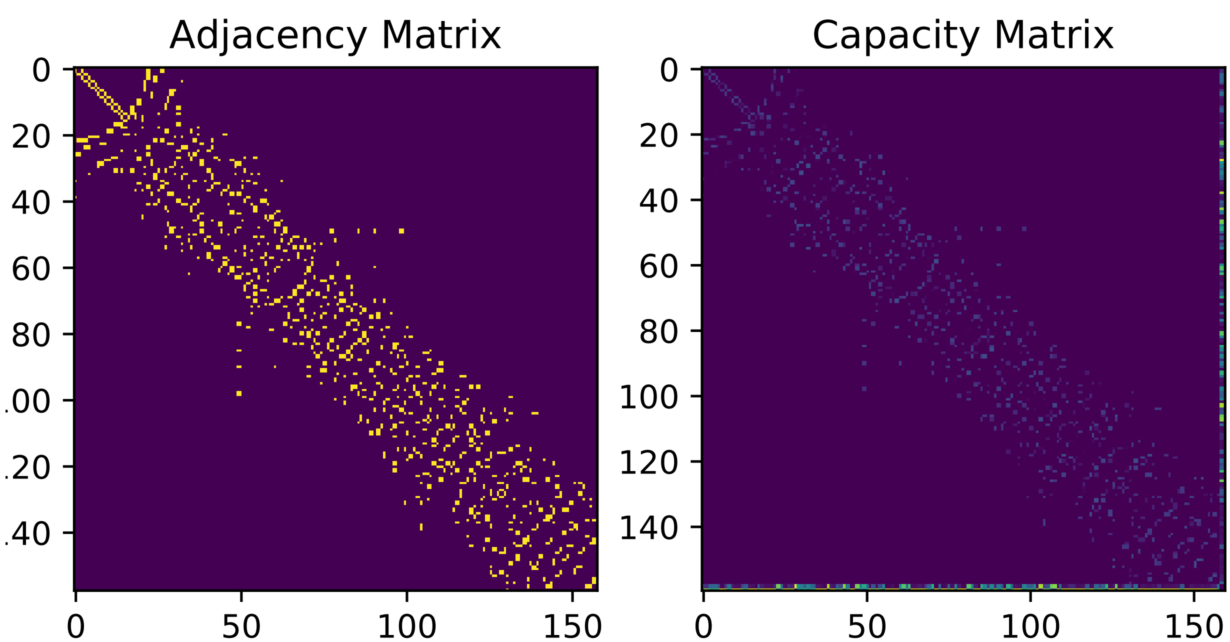

Figure 7: Capacity Image of Question 3

(c) One possible reason to downweight the capacity between adjacent nodes is to avoid the minimum cut at the edge between source and nodes with large capacity.

4. (a) Flower segmentation is shown in Figure 8 and 9.

Zixu Zhang
zixu@umich.edu

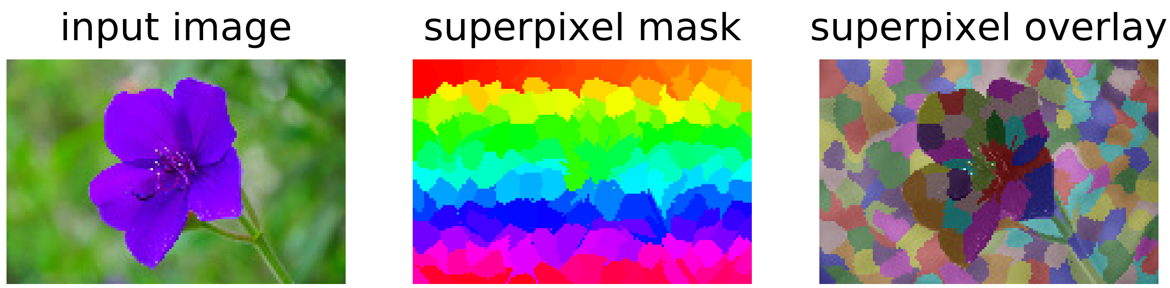

Figure 8: Superpixels of Question 4a

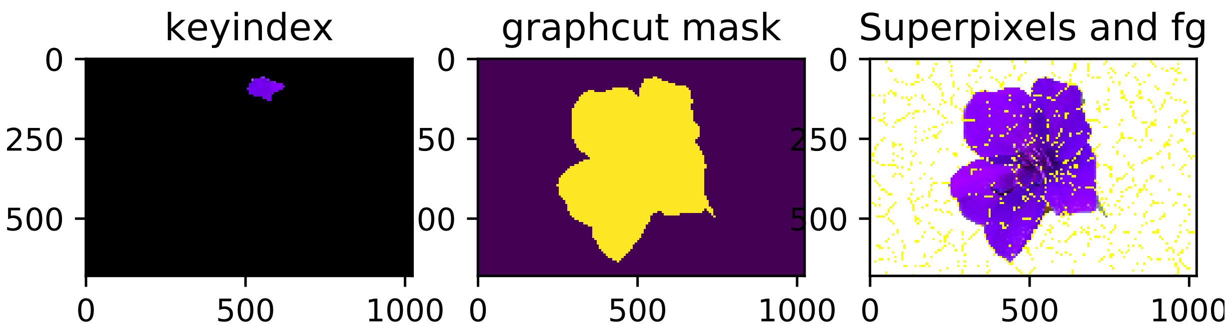

Figure 9: Results of Question 4a

(b) Flag segmentation is shown in Figure 10 and 11. We can see that if we click on a red stripe, we are not able to segment all red stripes in the images. The shadow in on the flag makes some superpixel of red strip darker, so that it is hard to be segmented with the clicked section. Moreover, we can see from superpixel overlay, each superpixel in one strip is only connected to two other superpixel in the same strip. So that in order to segment a full strip, the residual graph can only reach those superpixel, even though there are many other superpixel has high capacity with the source. Thus, it is hard.

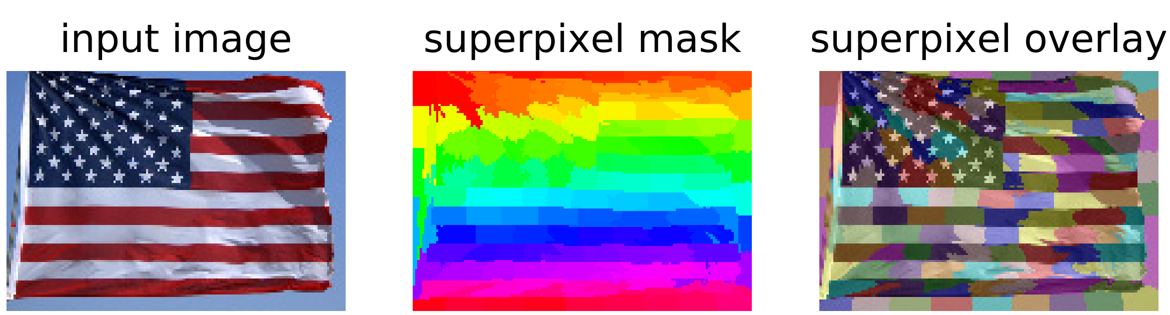

Figure 10: Superpixels of Question 4b

Zixu Zhang
zixu@umich.edu

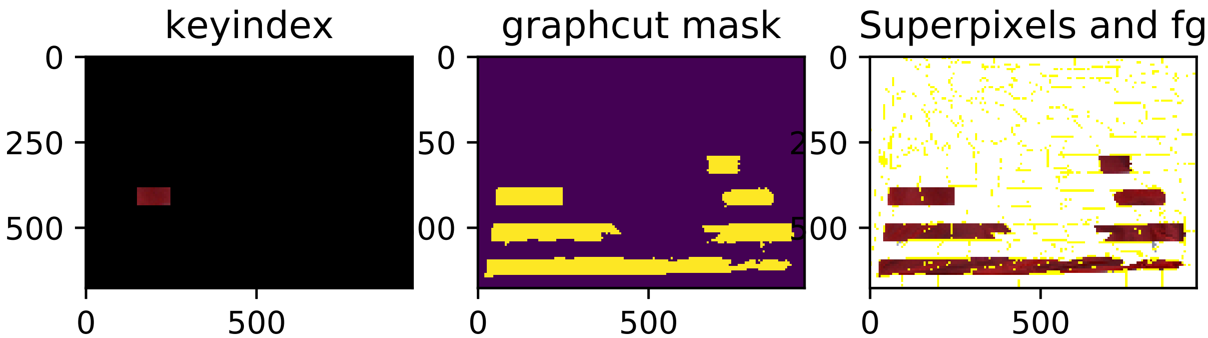

Figure 11: Results of Question 4b

(c) Porch segmentation is shown in Figure 12 and 13. It is clear that we can segment two boots perfectly, since the color of both boots are quite constant, and it is unique than other section in the image.

Figure 12: Superpixels of Question 4c

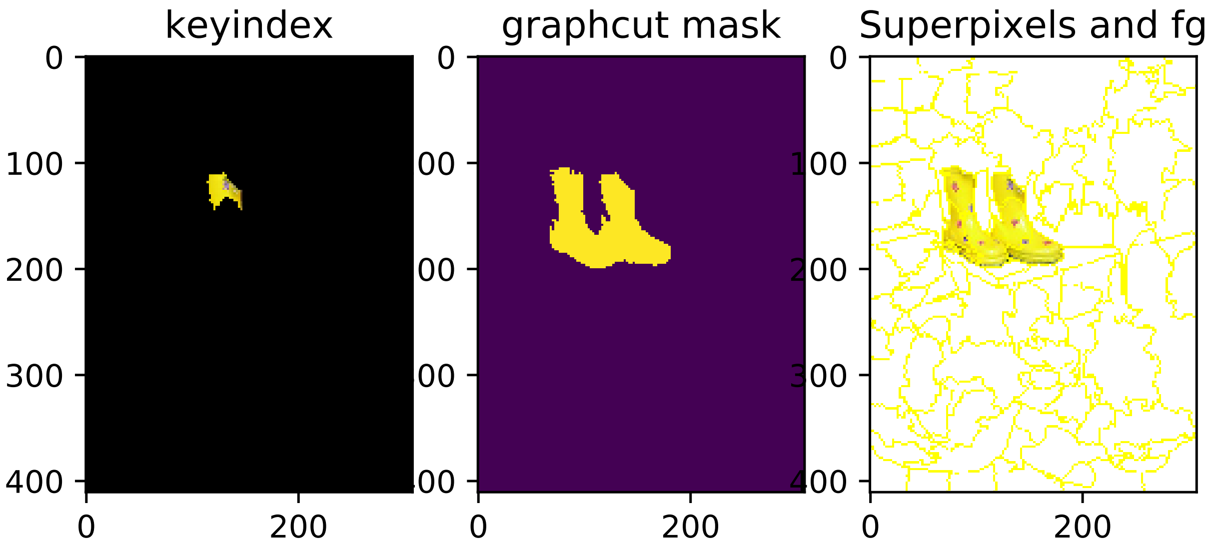

Figure 13: Results of Question 4c

(d) If we try to segment the basket of porch image, the result is shown in Figure 14.

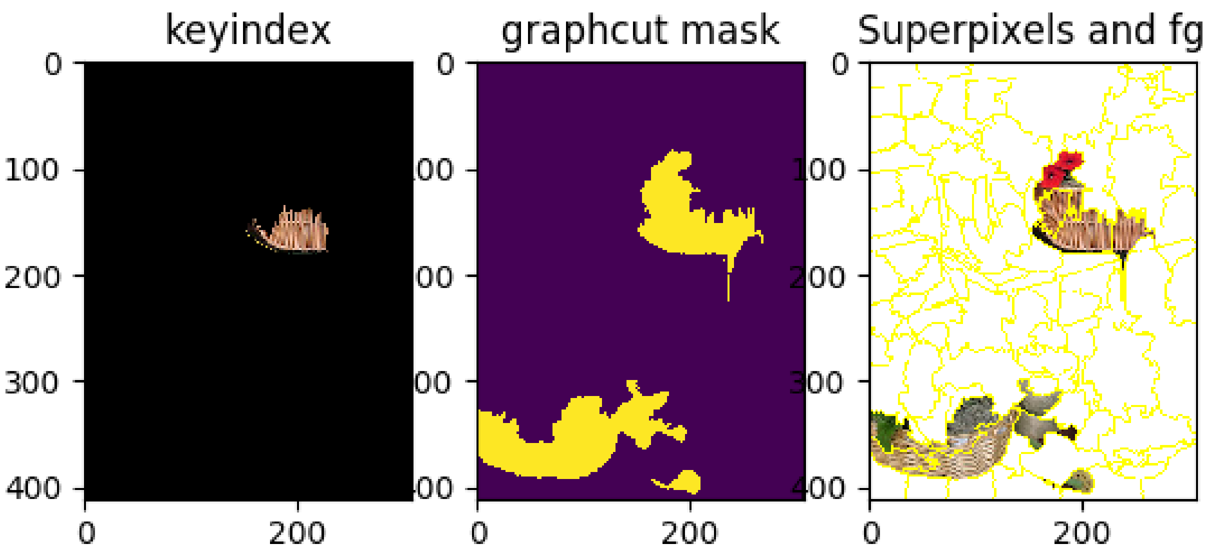

Figure 14: Results of Question 4c

It is clear that we are not able to segment either of the baskets completely. This is caused by the perceptual similarity of two basket even if they are spatially away from each other. One way to solve this is to make sure that all segmented section are connected in adjacency graph. In this way we can separate two similar items but are not spatially connected.

Appendix

Q1:

```python
def histvec(img,mask,b):
    '''
    Function to find the color histogram of the image.

    Args:
    -----
    img: input image
    mask: Super pixel mask. Each pixel location will have the
       superpixel label corresponding to it
    b: number of bins in the histogram
    Return:
    --------
    hist_vector: 1-D vector having the histogram of all three
       channels appended
    '''

    img_in_SP = img[mask,:].astype(dtype=np.int64)
    total_location = img_in_SP.shape[0]

    hist_vector = np.zeros(3*b)
    ub_unit = 256.0/b

    '''loop through all bins'''
    for i in range(b):
        ub_cur = ((i+1)*ub_unit)
        '''loop through rgb channels'''
        for j in range(3):
            cur_idx = np.argwhere(img_in_SP[:,j]<=ub_cur)
            hist_vector[j*b+i]+=len(cur_idx)
            img_in_SP[cur_idx,j] = 300

    '''Normalize Histogram'''
    hist_vector=hist_vector/total_location
    return hist_vector
```

Q2:

```python
def seg_neighbor(svMap):
    '''
    Function to find adjacency matrix
    Args:
    ----
    svMap: Super pixel mask. Each pixel location will have the
       superpixel label corresponding to it.

    Return:
    ------
    Bmap: a binary adjacency matrix NxN (N being the number of
       superpixels  in svMap).
    '''
    segmentList = np.unique(svMap)
    segmentNum = segmentList.shape[0]
    # FILL IN THE CODE HERE to calculate the adjacency
    Bmap = np.zeros([segmentNum, segmentNum])
    height,width = svMap.shape
    for i in range(height):
        for j in range(width):
            '''check eight connectivity'''
            y_u = min(i+1, height-1)
            x_u = min(j+1, width-1)
            x_l = max(j-1, 0)
            ''' check lower'''
            if svMap[i,j] != svMap[y_u,j]:
                Bmap[svMap[i,j],svMap[y_u,j]] = 1
                Bmap[svMap[y_u,j],svMap[i,j]] = 1
            ''' check left'''
            if svMap[i,j] != svMap[i,x_u]:
                Bmap[svMap[i,j],svMap[i,x_u]] = 1
                Bmap[svMap[i,x_u],svMap[i,j]] = 1
            ''' check lower left'''
            if svMap[i,j] != svMap[y_u,x_u]:
                Bmap[svMap[i,j],svMap[y_u,x_u]] = 1
                Bmap[svMap[y_u,x_u],svMap[i,j]] = 1
            ''' check lower right'''
            if svMap[i,j] != svMap[y_u,x_l]:
                Bmap[svMap[i,j],svMap[y_u,x_l]] = 1
                Bmap[svMap[y_u,x_l],svMap[i,j]] = 1
    return Bmap
```

```python
def ave_deg(adj_mat):
    '''
    Calculate average node of an adj_mat
    '''
    total_deg = np.sum(adj_mat)
    num_nodes = adj_mat.shape[0]
    return total_deg/num_nodes
```

Q3:

```python
def graphcut(S,C,hist_values, keyindex, plt_img=False):

    dnorm = 2*np.square(np.prod(np.divide(S.shape,2)))

    k = len(C)
    # Generate capacity matrix
    capacity = np.zeros((k+2,k+2)) # initialize the zero-valued
        capacity matrix
    source = k # set the index of the source node
    sink = k+1 # set the index of the sink node

    # FILL IN CODE HERE to generate the capacity matrix using the
        description above.
    capacity[k+1,keyindex]=0
    capacity[keyindex,k+1]=0
    capacity[k,keyindex]=3
    capacity[keyindex,k]=3
    for i in range(k):
        for j in range((i+1),k):
            His_sim_cur = hist_intersect(hist_values[i],
                hist_values[j])
            dis_cen = np.array(C[i])-np.array(C[j])
            Spa_sim_cur = np.exp(-1*np.linalg.norm(dis_cen, ord=2)
                /dnorm)
            Capa_cur = His_sim_cur*Spa_sim_cur
            if adjacency[i,j]==1:
                capacity[i,j] = 0.25*Capa_cur
                capacity[j,i] = 0.25*Capa_cur
            if i==keyindex:
                capacity[k,j] = Capa_cur
                capacity[j,k] = Capa_cur
                capacity[k+1,j] = 3-Capa_cur
                capacity[j,k+1] = 3-Capa_cur
            elif j==keyindex:
```

```
                capacity[k,i] = Capa_cur
                capacity[i,k] = Capa_cur
                capacity[k+1,i] = 3-Capa_cur
                capacity[i,k+1] = 3-Capa_cur

    #Compute the cut (this code is provided to you)
    _,current_flow = ff_max_flow(source, sink, capacity, k+2)

    reachable_node = bfs_residual_reachable(source, current_flow, capacity)

    '''create mask'''
    B = np.zeros_like(S)
    for i in range(len(reachable_node)):
    cur_idx = (S==reachable_node[i])
    B[cur_idx] = 1
    return B

def bfs_residual_reachable(start, current_flow, capacity):

    num_node = capacity.shape[0]-2
    q = deque([])
    q.append(start)
    node_hist = np.zeros(num_node,dtype=bool)
    visited_node = []

    while len(q) != 0:
        u = q.popleft()
        for i in range(num_node):
            if~node_hist[i]:
                if current_flow[u,i]<capacity[u,i]:
                    node_hist[i]=True
                    visited_node.append(i)
                    q.append(i)
    return  visited_node
```