Julio Soldevilla
EECS 545 Winter 2018 — Problem Set 4

Problem 1 Problem 1

Proof:

1. After running the code for problem 1.a, we get that the energy for image 1 is 36 and the energy for image 2 is 42, according to the Potts Model and we get the following images:

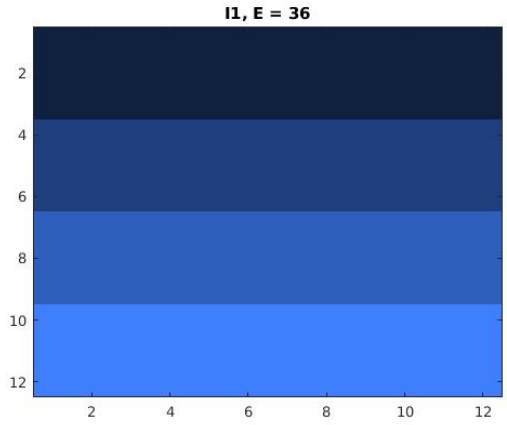

Figure 1:

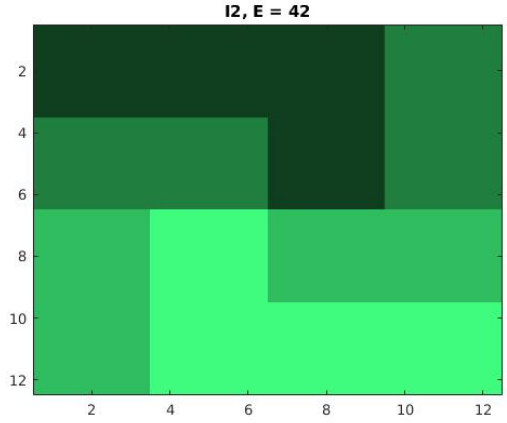

Figure 2:

The code of problem 1.a is:

```
% W18 EECS 504 HW4p1 Mumford-Shah Piecewise Constant
% Siyuan Chen

```

```
function E = potts(I,beta)

if nargin==1
    beta=1;
end

%%% FILL IN YOUR CODE HERE

double_im = im2double(I);
E = 0;

% In this part of the code we are forming the 3d kernel, where the vertical
% kernel has entries 1 and -1 in column form, so this kernel is 3x1x3 of size
% and the horizontal kernel has entries 1 and -1 in verticla form, so this
% kernel has 1x3x3 of size.

hor_mask = [1,-1];
ver_mask = [1;-1];
hor_mask_3d(:,:,1) = hor_mask;
hor_mask_3d(:,:,2) = hor_mask;
hor_mask_3d(:,:,3) = hor_mask;
ver_mask_3d(:,:,1) = ver_mask;
ver_mask_3d(:,:,2) = ver_mask;
ver_mask_3d(:,:,3) = ver_mask;

hor_conv = convn(double_im, hor_mask_3d,'valid');
ver_conv = convn(double_im,ver_mask_3d,'valid');
hor_sum = sum(hor_conv(:) ~= 0);
ver_sum = sum(ver_conv(:) ~= 0);
E = hor_sum + ver_sum;
end
```

2. After running the code for problem 1.b, we get that the energy of the image following the Mumford Shah model is 67.6103. The code for this problem is:

```
function Epc = mumford_shah(original_function,estimated_function)

% This function will compute the discrete piecewise Mumford Shah model. The
% INPUT: an original funcction and an approximation function.
% Output: The energy of the image with respect to the approximation and
% certain boundary term.

alpha = 1;
gamma = 0.1;

double_orig_fxn = im2double(original_function);
double_estimated_fxn = im2double(estimated_function);

```

```
squared_difference = (double_estimated_fxn - double_orig_fxn).*(
    double_estimated_fxn - double_orig_fxn);

integral = sum(squared_difference(:));

hor_mask = [1,-1];
ver_mask = [1; -1];
hor_conv = conv2(double_estimated_fxn,hor_mask,'valid');
ver_conv = conv2(double_estimated_fxn,ver_mask,'valid');
hor_sum = sum(hor_conv(:) ~= 0);
ver_sum = sum(ver_conv(:) ~= 0);
boundary = hor_sum + ver_sum;

Epc = alpha*integral + gamma * boundary;

end
```

3. In the equation above, we can see that the boundary terms comes from the pott's model. Also, we can split the image (the squared difference of images) in the problem into regions based on the pixel values on each of the regions. However, given the images we have, each region would have its own unique minimum. Thus, since each region has its own minimum, we cannot say that the f-hat given is the minimizer of the whole image. All we can do to in a sense minimize the energy in the image is to minimize the sum of the energies in each part of the regions. Thus minimize the double integral and the boundary term.

■

Problem 2 Problem 2

Proof:

1. For part a of problem 2.1, after running the q1.m file we get the following output

   If we just show the 88 superpixel from the figure (the superpixel used to compute the histograms), we can see that it corresponds to one of the yellow peppers that is right below the 3.99 dollar sign.

2. This is problem 2 part 2

   (a) For part a of problem 2.2, after running the q2.m file we get the following output:

   (b) This is part b of problem 2.2. The average node degree is 8.8571. The code for computing this is:

```
function avg_degree = node_degree(adj_matrix)

%%% We can obtain the degree of a vertex from an adjacency matrix by
    adding
%%% the entries in the corresponding column of the vertex in the
    matrix. We
```

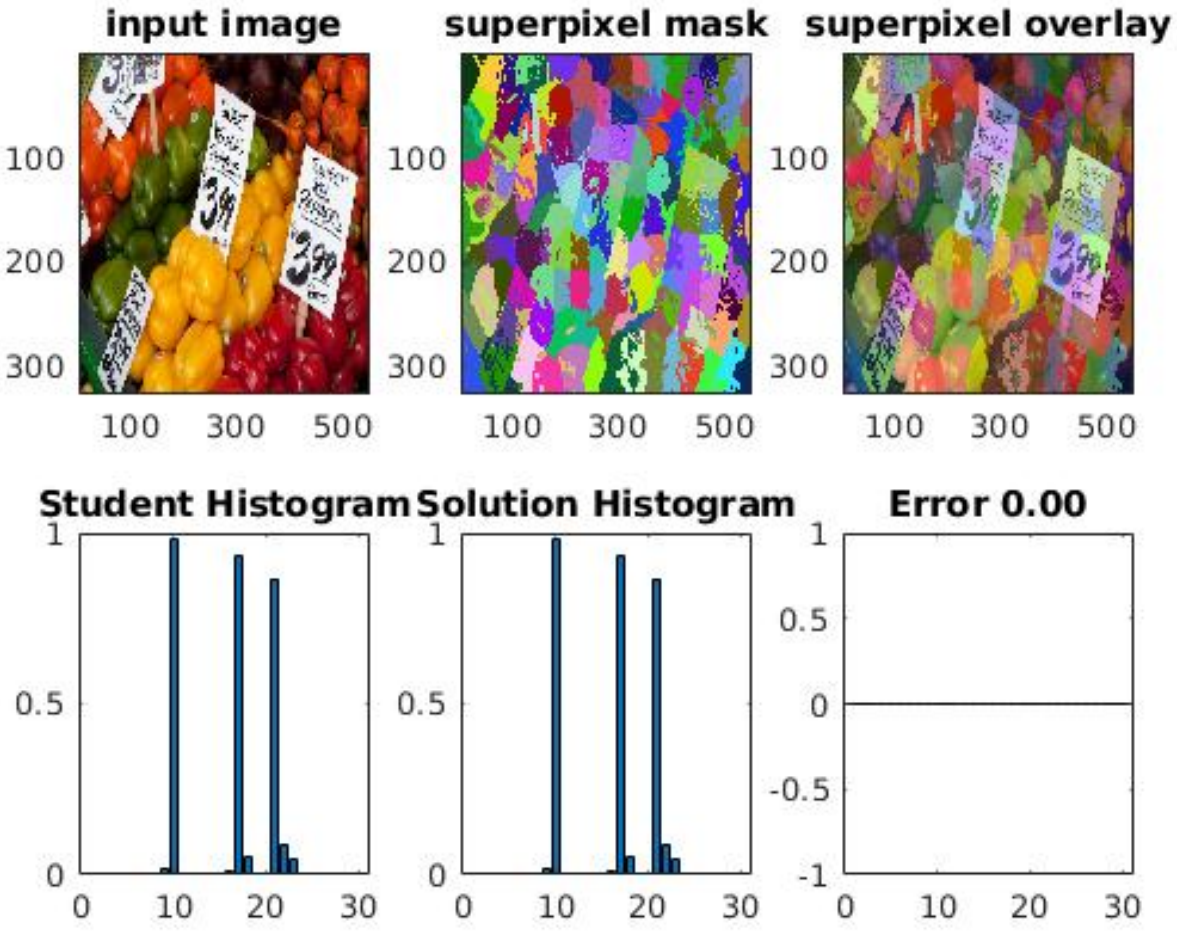

Figure 3: Problem 2.1

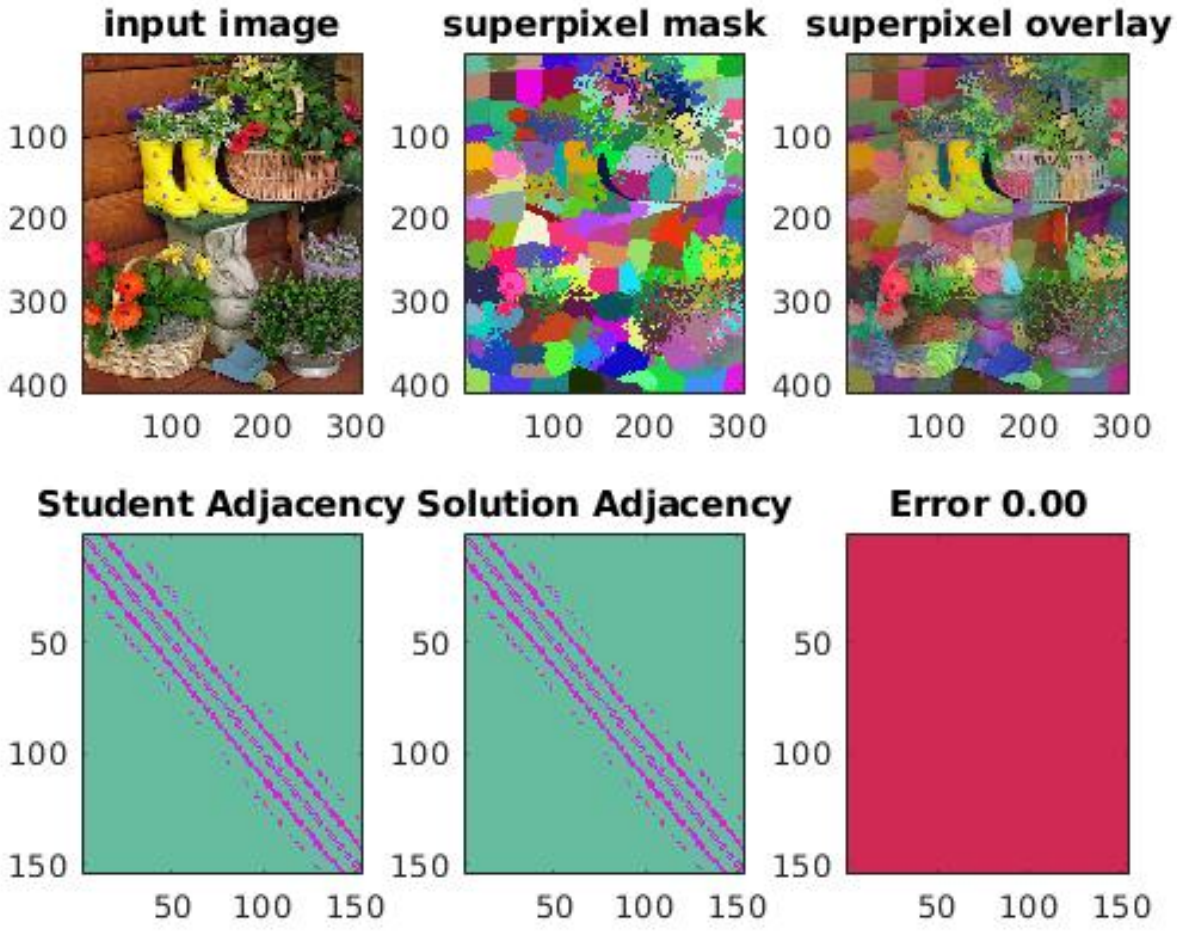

Figure 4: Problem 2.2 part a

```
%%% will add the entries in every column and from these numbers we
    can get
%%% the average node degree.

size_matrix = size(adj_matrix);
no_columns = size_matrix(2);

% This part of the code adds the entries in each column and adds the
    value
% of each column.
degree_sum = 0;
for i = 1:no_columns
    column = adj_matrix(:,i);
    pre_sum = sum(column);
    degree_sum = degree_sum + pre_sum;
end

% Finding the degree.

avg_degree = degree_sum/no_columns;

end
```

(c) This is part c of problem 2.2. The adjacency graph is not a perfect banded diagonal matrix because of two reasons: first of all, if the adjacency graph was a perfect banded diagonal matrix, then that would imply that every superpixel would have the same number of adjacent edges and second of all, it would imply that if superpixel i was neighbor with superpixels $(i+1, ..., i+j)$ for some real number j, then super pixel $i+1$ would be neighbor with superpixels $(i+2, ..., i+(j+1))$, so basically the indices of the superpixels and their neighbors would increase linearly for all the superpixels. In this case, we can see from the picture of superpixels, that the superpixels obtained clearly don't satisfy these conditions, not every superpixel has the same number of neighbors and we can probably inspect the code to see that the indices of the superpixel and their neighbors don't increase as was described above. Because of these reasons, we see that why we don't obtain a perfect banded diagonal matrix.

3. This is problem 2 part 3

(a) For part a of problem 2.3 after running the q3.m file we get the following output in figure 5:

(b) This is part b of problem 2.3. This is an image of the capacity matrix before running graph cuts, shown in figure 6:

We can clearly see that the capacity image is related to the adjacency in the sense that the capacity has a very big 105×150 window that is also an (almost perfect) banded diagonal matrix. Also, we can see that whereas in the adjacency matrix the entries of the matrix are either 0 or 1, in the capacity matrix we have entries that can bee between 0 and 1 and so we can see some of the possible weights that the edges between the

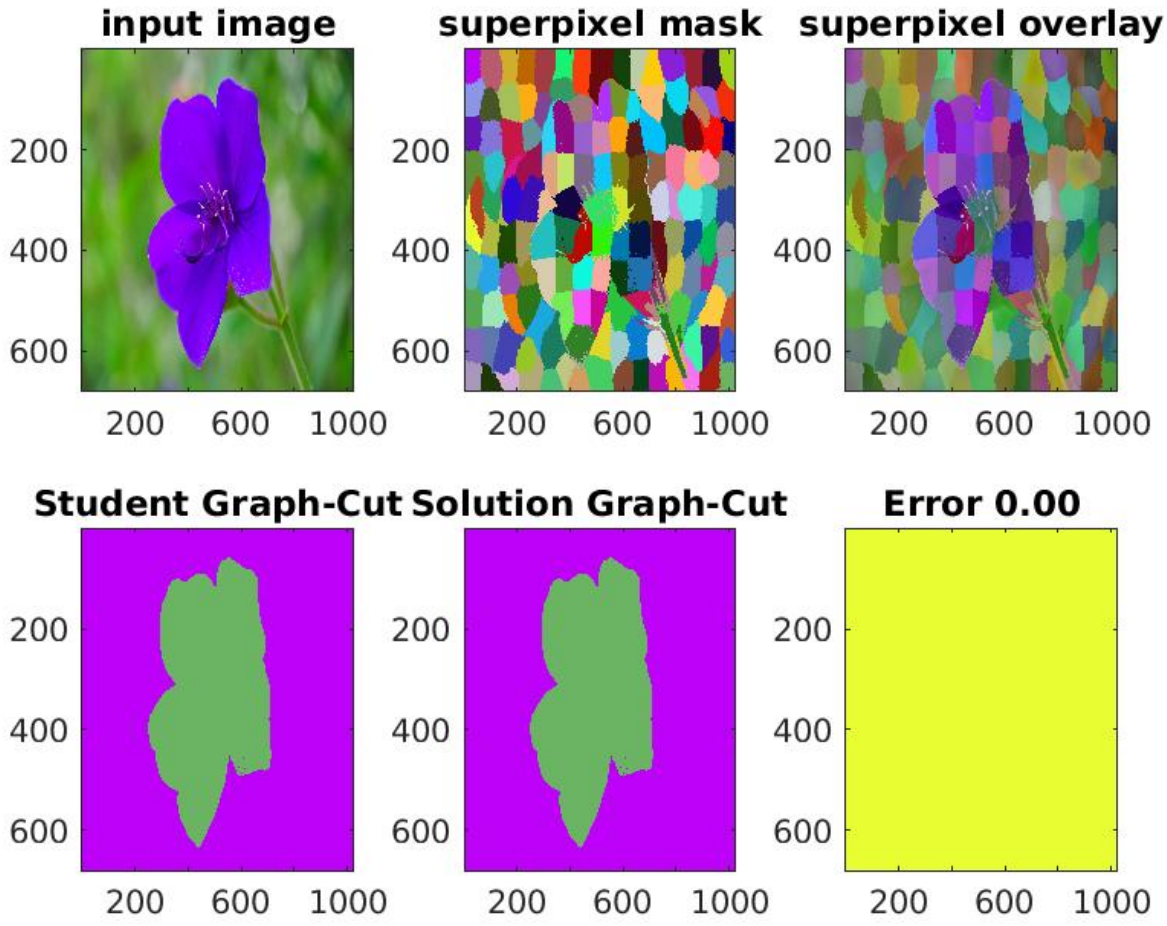

Figure 5: Problem 2.3 part a

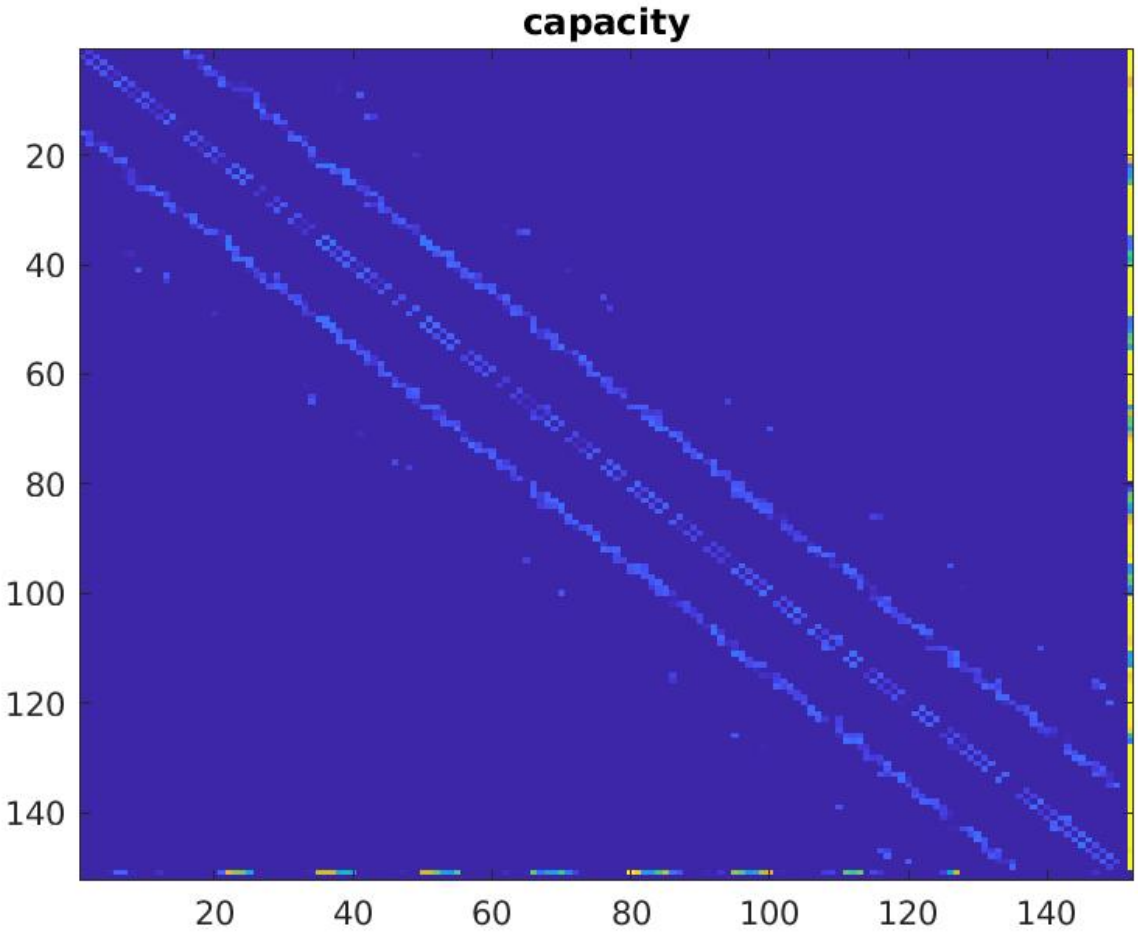

Figure 6: Problem 2.3 part b

superpixels can have. Finally, we can see that the capacity matrix is of size bigger than 150×150 and from the formulation of this matrix and looking at the left and down edges of the image, we can see that these capacities correspond to the capacity of the edges connecting the source (on the right part of the matrix) and the capacity of the edges on the edges connectin with the sink (on the bottom part of the matrix). These rows and columns related to the source and sink of the graph are clearly recognizable from the capacity image.

(c) This is problem 2.3 part c. In this problem, we are downeighting the capacity between vertices (not considering the source and the sink) or in a sense having higher capacity between any edge and the source or sink compared connection between edges because in this way we are reducing the probability that when we compute the residual graph, we run out of edges (basically having edges of capacity 0) from the source (or sink) to any other edge in the graph. Also, by downweighting the capacity in edges between non sink and non source vertices, then we are reducing the probability that the cut in our graph goes through edges connecting the source or the sinnk to any other edge, since these edges will in general have higher capacity and so a higher chance of a higher flow than the edges between non source vertices. Thus, this reduces the probability that we lose connections from the source or sink to the rest of the graph.

4. For problem 2 part 4 we have the following results:

(a) This is the output we get from running the example.m file once:

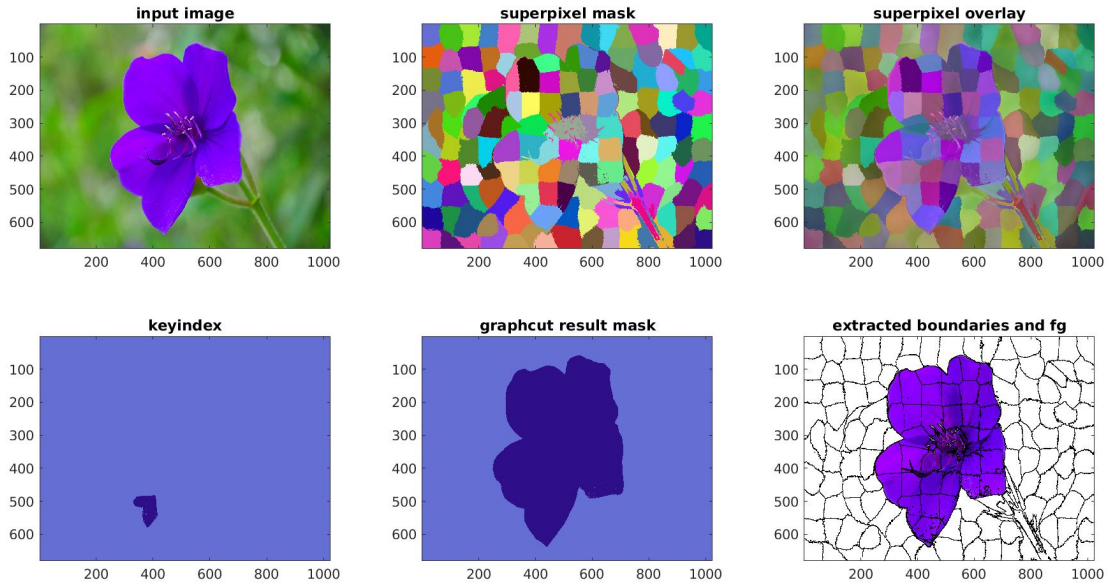

Figure 7: Problem 2.4 a

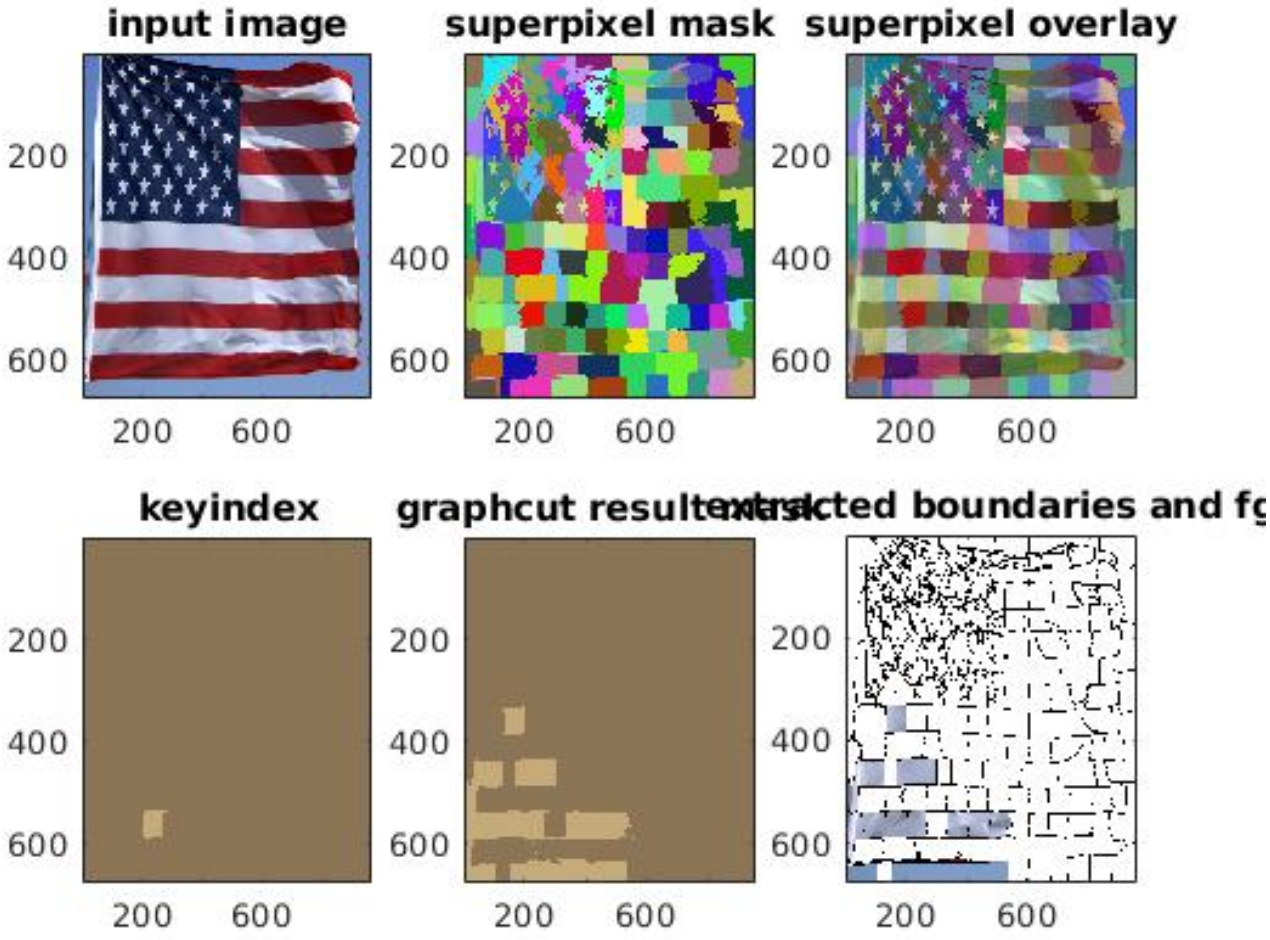

Figure 8: Problem 2.4 b

(b) This is the output for running the example.m file with the flag image:

In this figure, I didn't get the whole stripe. I think to getting just one stripe (either white or red) is hard because of two reasons: First, getting just one stripe of the flag is hard because (even though there are some shadows in the picture), the white stripes or red stripes in the flag have a very similar color histogram on average (either with variations of white or variations of red) and so this will make it difficult to get just one stripe. Now, it is also hard to get a whole stripe (either white or red), because along one stripe there is also a good amount of variation in the color of the stripe because of the shadows of the flag. Thus, this would mean there are also different color histograms along one stripe of the flag which would make it difficult to get just one stripe of the flag.

(c) This is the output for running the example.m file with the basket image and trying to get the boots in the foreground:

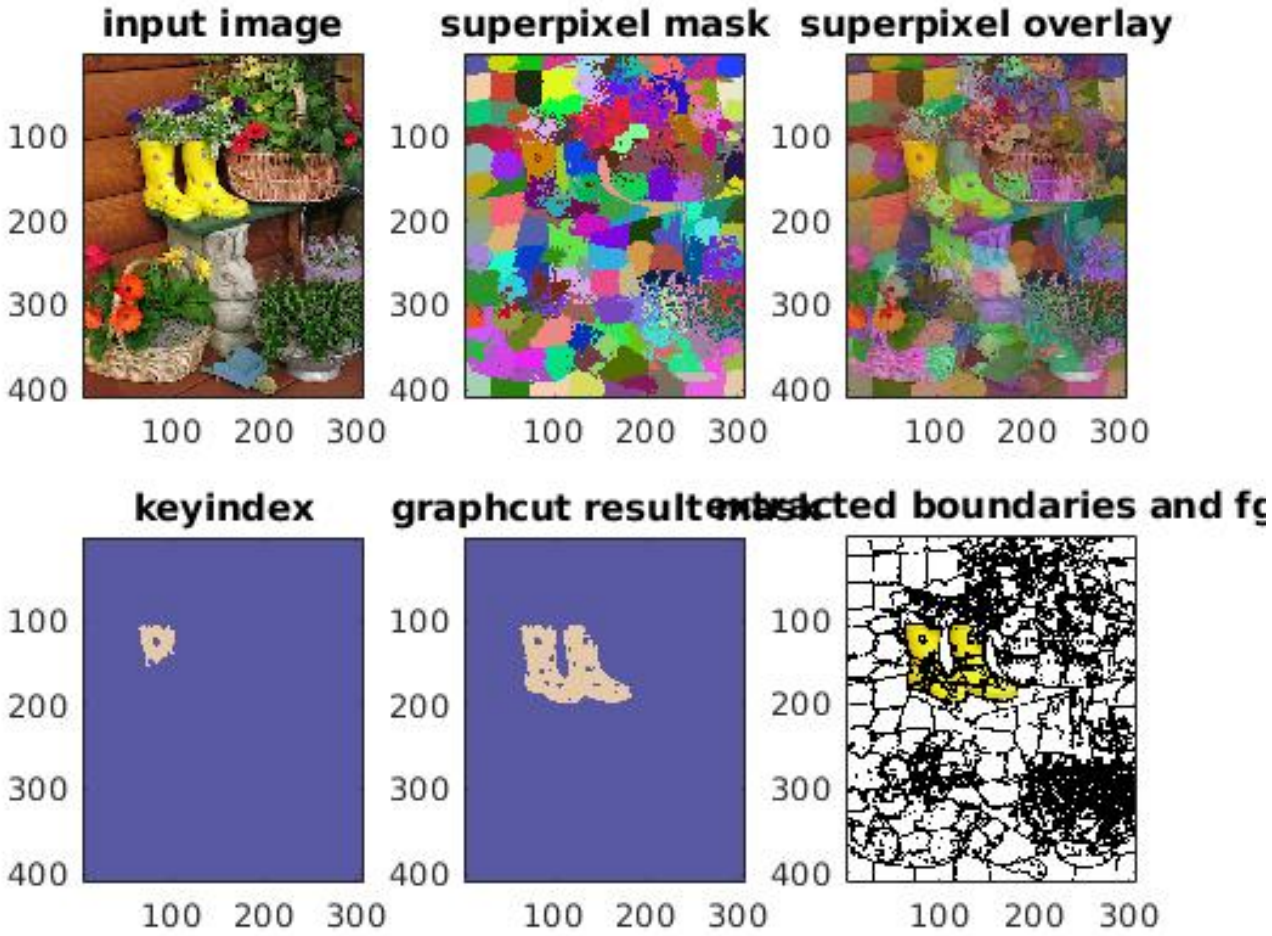

Figure 9: Problem 2.4 c

In this part of the problem, I didn't get the whole boots, I missed the pattern on the

boots. In this case, I think I didn't get the whole boots image because the pattern on the boots has very different color and so it will have a very different color histogram. Thus, when doing the graph cut, these regions of the pattern will be on the other end of the cut, not where the boots are. Also, maybe the pattern of the flowers (not smooth regions) makes it more difficult to recognize them as part of the boots.

(d) This is the output for running the example.m file with the basket image and trying to get the basket in the foreground:

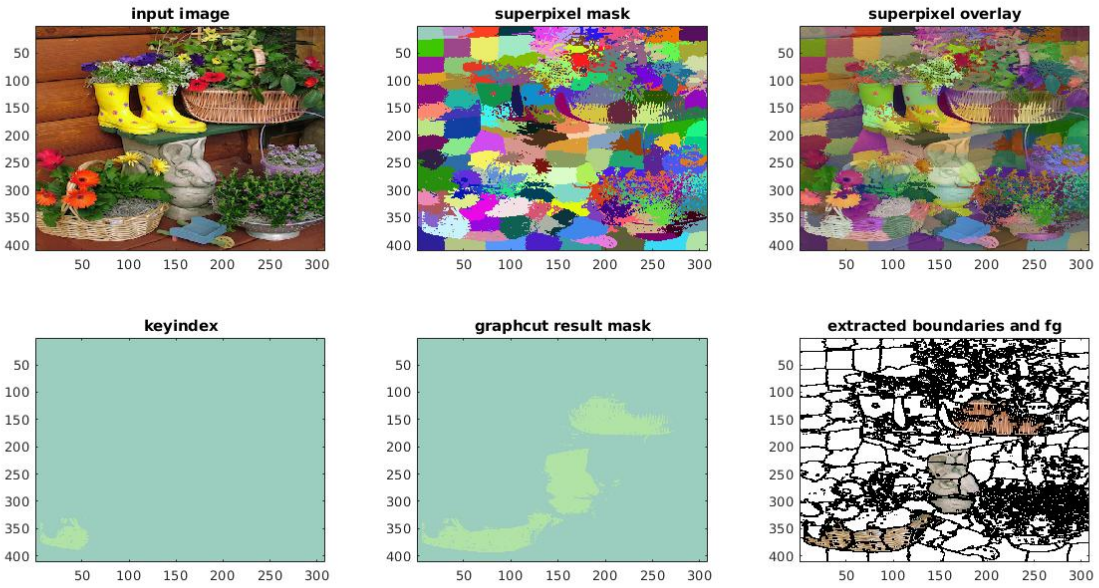

Figure 10: Problem 2.4 d

In this region I didn't get thw whole basket. In the system we have, I would change the bin size in the histogram to be more sensitive to variations in color. In my images, I distinguished parts of the basket but also parts of the rabbit and parts of the other basket. In this case, I think what is happening is that because of the width of the bins in the histograms, we are probably putting together the color of the rabbit and the baskets and so that's why we are getting all these regions. To get more of the basket and less of the rabbit or the other basket, increasing the bin of the histogram would help us detect better the color on the basket.

■

Problem 3 Problem 3

Proof:

1. For this image, we see the default result, with hyperparameter $k = 4$ is:

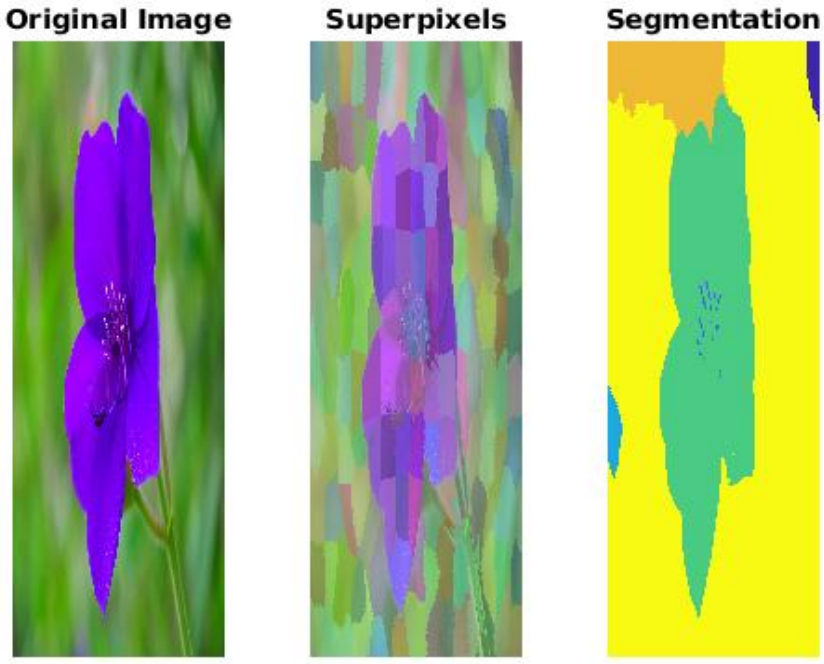

Figure 11: Problem 3.1 a, k =4

2. In this problem, we computed the weights differently because first of all, in problem 3.1, we are not working with a graph cut and so we don't have a source or sink edge. Thus, we don't have the need to increase the probability of not losing some edges (in problem 2 we wante to save many edges conected to source or sink). Thus, we don't need to downweight any edge capacity.

3. For this problem, after implementing the post - processing we get:

   The implementation of the function is the following, it is the whole code for fh.m, but the important part implementing this function is in the final lines of code.

```
function seg = fh(G, k, seg_img)
%
% W18 EECS 504 HW4p3 Image Segmentation with Minimum Spanning Forest
%
% Felzenszwalb-Huttenlocher algorithm implementation, which is a modified
% version of the Kruskal algorithm.
% Input  G: mx3 matrix, the adjacency list of a graph, of which each row
    is
%           an edge (node1, node2, weights). m is the number of edges.
%        k: hyperparameter for F-H algorithm
% Output seg: 1xn vector, the segment id assigned to each node in the
    graph
%             n is the number of nodes, id is from 1 to number of
```

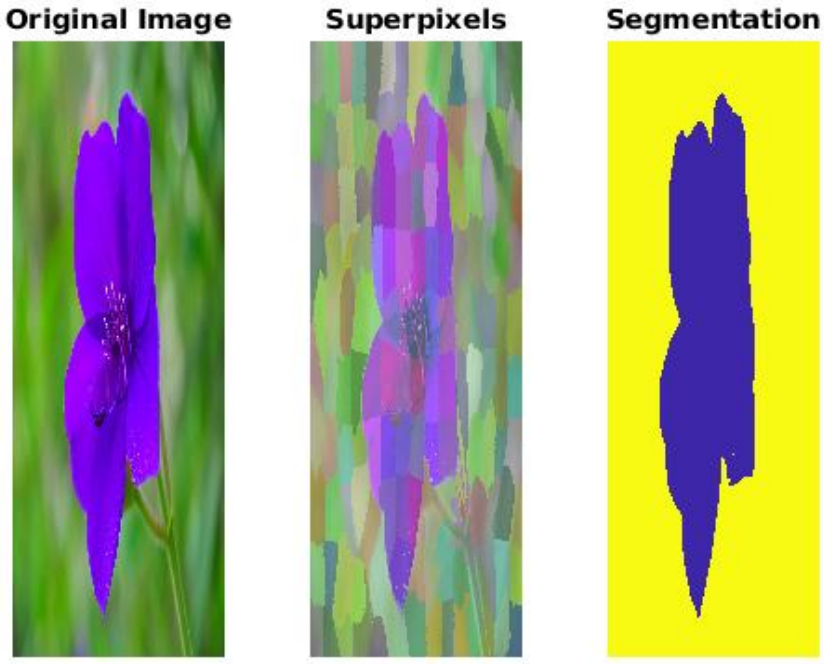

Figure 12: Problem 3.1 c, k =4

```
    segments.
%

sorted_matrix = sortrows(G,3);
size_matrix = size(sorted_matrix);
no_edges = size_matrix(1);

pre_edges = [G(:,1);G(:,2)];
edges = unique(pre_edges);
no_vertices = length(edges);

% Creation of components
components_vector = {};
for t = 1:no_vertices
    components_vector{t} = [edges(t), 0];
end
%components_vector

% In this part we will construct the segments
p = 1
for q = 1 : no_edges
    current_edge = sorted_matrix(q,:);
    current_v1 = current_edge(1);
    current_v2 = current_edge(2);
```

```
    current_weight = current_edge(3);

    for s = 1:no_vertices
        %if p == 150
         %  a = 3;
        %end
        %p = p + 1
        if ismember(current_v1,components_vector{s}) == 1
            component_1_index = s;
        end
        if ismember(current_v2,components_vector{s}) == 1
            component_2_index = s;
        end
    end

    component_1 = components_vector{component_1_index};
    component_2 = components_vector{component_2_index};

    %internal_comp1 = component_1(end);
    %internal_comp2 = component_1(end);
    %tau_comp1 = k/(length(component_1) - 1);
    %tau_comp2 = k/(length(component_2) - 1);

    %minimum_internal = min(internal_comp1 + tau_comp1,internal_comp2 +
    tau_comp2);

    if isequal(component_1(1:end-1),component_2(1:end-1)) == 0

        internal_comp1 = component_1(end);
        internal_comp2 = component_2(end);
        tau_comp1 = k/(length(component_1) - 1);
        tau_comp2 = k/(length(component_2) - 1);

        minimum_internal = min(internal_comp1 + tau_comp1,internal_comp2
    + tau_comp2);

        if current_weight <= minimum_internal
            nC = {};
            n = 1;
            for j = 1 : length(components_vector)
                if j ~= component_1_index && j ~= component_2_index
                    nC{n} = components_vector{j};
                    n = n + 1;
                end
            end

            max_weight_edge = max([internal_comp1,internal_comp2,
    current_weight]);
            nC{n} = [component_1(1:end-1),component_2(1:end-1),
    max_weight_edge];

            %nC{n}
```

```
            %components_vector{component_1_index} = [component_1(1:end-1)
    , component_2(1:end-1),max_weight_edge];
            %components_vector(component_2_index) = [];
            %no_vertices = no_vertices - 1;
        end
    end
    components_vector = nC;
    size(components_vector)
    %if size(components_vector,2) <= 52
    %    a = 2
    %end
    no_vertices = size(components_vector,2);
end

no_vertices = length(edges);
%components_vector;
size(components_vector);
no_components = length(components_vector);
zsegment_vertex_vector = zeros(1,no_vertices);

for j = 1 : no_components
    current_component = components_vector{j};
    %j
    for p = 1 : length(current_component(1:end-1))
       %p
       segment_vertex_vector(current_component(p)) = j;
    end
end

%To run the algorithm with the post_processing, just uncomment the line
%calling the post_processing function and rename the first seg in line 88
%to be pre_seg.

seg = segment_vertex_vector;
%c = 1;
%seg = post_processing(sorted_matrix,pre_seg,seg_img,3);
%seg;
%w = 1;
end

%%% This part is the implementation of the post_processing function.

function seg_final = post_processing(sorted_matrix,seg,seg_img,min_size)

% IN this function we take the sorted matrix, the result from the
    function
% above and the whole image with the segment on each pixel.
   size_matrix = size(sorted_matrix);
   no_edges = size_matrix(1);
   size_segments = size(seg);
   for j = no_edges : -1 : 1
       edge_considered = sorted_matrix(j,:);
```

```
        vertex1 = edge_considered(1);
        vertex2 = edge_considered(2);
        component_vertex_1 = seg(vertex1);
        vector_spixels_1 = find(seg(:) == component_vertex_1);
        component_vertex_2 = seg(vertex2);
        vector_spixels_2 = find(seg(:) == component_vertex_2);
        size_comp_1 = length(vector_spixels_1);
        size_comp_2 = length(vector_spixels_2);

        if component_vertex_1 ~= component_vertex_2
           if size_comp_1 < min_size
                for x = 1 : size_comp_1
                    seg(1,vector_spixels_1(x)) = component_vertex_1;
                end
           elseif size_comp_2 < min_size
                for x = 1 : size_comp_2
                    seg(1,vector_spixels_2(x)) = component_vertex_1;
                end
           end
        end
    end
    seg_final = seg;
end
```

4. Finally, after commenting out the post - processing function, we can play around with the parameter k and we get the following results:

   We can see that as we change the value of k, then the higher the value of k we don't get as many components but we get a primitive shadow of the main things of the picture, in this case the petals of the flower. As k decreases, we get many more components and some more details of the picture, in this case the pistils (the thingies in the center of the flower). So k is playing a role of quantity of components in the algorithm (and in a sense the level of detail we can see). The lower the k the more components. This makes sense sinse k is some element in the condition for combining components or not.

■

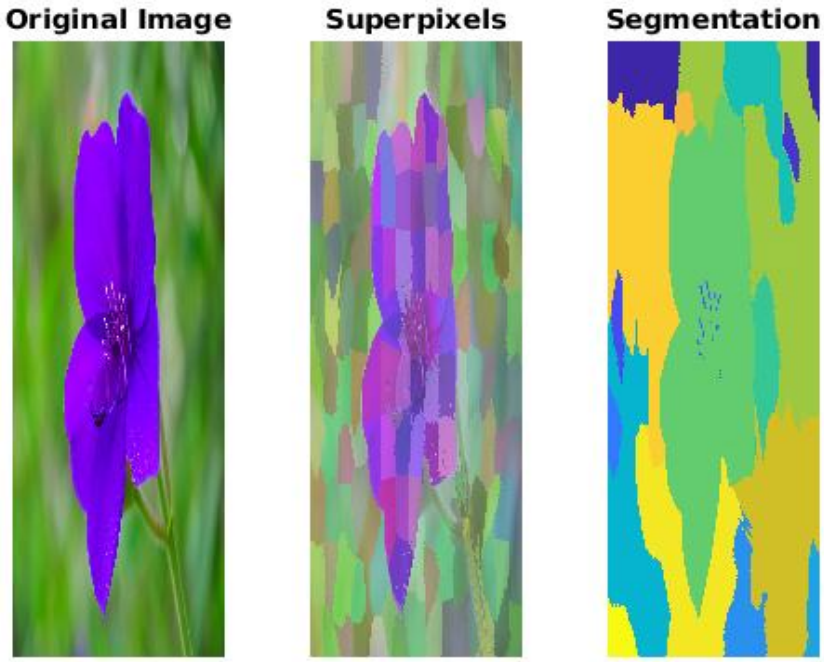

Figure 13: Problem 3.1 a, k = 1

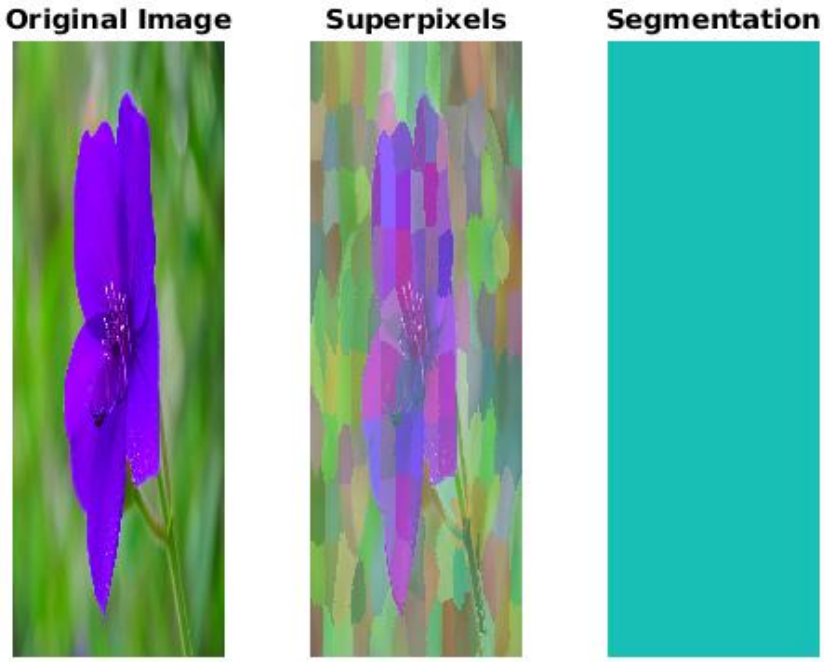

Figure 14: Problem 3.1 a, k = 20