Problem Set 2

CMSC 733

Assigned Tuesday, September 30, Due Tuesday, October 14

Normalized Cut

In this problem, you will implement normalized cut, as described in the Shi and Malik paper that I link to on the class web site. You only need to implement this for grouping points (ie, the first bullet item in Section 4). To assist you, the class web page contains a matlab function called `test_normalized_cut_points`, which tests each of the functions that you are to create, along with two figures that show the output of our implementation on this test.

1. **10 points** Implement a function of the form:

```
[x W] = normalized_cut_cont_points(p,sigma)
```

p will be a 2xn matrix in which each column contains the x and y coordinates of a 2D point. sigma will be the standard deviation used in the Gaussian weighting of distances. x will be a vector of continuous values that indicates the group membership of the points. These values will be turned into binary values by the next routine. W will be the matrix that contains the pairwise affinities of the points. Note that W is not used in `test_normalized_cut_points`, but you may find it convenient to return this value so that it can be used by the next routine. Hand in the code and a figure illustrating the results of your algorithm.

2. **10 points** Implement a function of the form

```
group = normalized_cut_points(p, sigma)
```

p and sigma will be the same as in (1). group will be a binary vector indicating to which group each point belongs. Hand in your code and a figure illustrating the results of your algorithm.

3. **5 points** Create your own set of test data, for which your algorithm produces a puzzling, unintuitive or apparently incorrect grouping. Show the results of your algorithm and briefly describe why you think normalized cut is behaving in this way.

Texture Synthesis

The goal of this problem is to implement the texture synthesis method of Efros and Leung. This is described in the paper: ``Texture Synthesis by Non-parametric Sampling'', by Efros and Leung, in the International Conference on Computer Vision, 1999. There is a link to their web site on the class web site, which includes links to their paper and pseudocode for their algorithm. In this assignment, I have simplified their

algorithm a little bit, to remove some details such as Gaussian weighting, which don't seem to be necessary to achieve good performance.

This algorithm takes a *sample* of some texture and generates a new *image* containing a similar texture. The strategy of the algorithm is to generate each new pixel in the image using a *neighborhood* of already generated pixels. One looks in the sample for similar neighborhoods, selects one of these similar neighborhoods at random, and copies the corresponding pixel into the new image.

1. **SSD 15 points:** S will contain a sample of the texture we want to generate. T contains a small (2n+1)x(2n+1) neighborhood of pixels. Not all the pixels in this neighborhood have been filled in with valid values however. So M (the *mask*) is a (2n+1)x(2n+1) matrix that contains a 1 for each position in which T contains a valid pixel, and a 0 whenever the corresponding pixel in T should be ignored. Computing the SSD is like correlation (or convolution) in that we shift the template over every position in the sample, and compute a separate result for each position. Thus, the output D is the same size as S. To compute D(i,j) we shift T so that its center is right on top of S(i,j). Then we take the difference between each valid pixel in T and the corresponding pixel in S, square the result, and then add all these together.

   To begin, you should write a function called SSD which performs the central step of the algorithm. This will compute the sum of squared difference between a little portion of the new image you are making and every portion of the sample.

   Hint: While you can compute SSD directly, it is interesting to note that it is closely related to correlation or convolution. To implement this algorithm efficiently in matlab you should make use of a function such as imfilter or conv2. It is simpler to implement the algorithm with a quadruple loop, but this will be very slow.

   Test your function using the binary checkerboard image:

   111000111
   111000111
   111000111
   000111000
   000111000
   000111000
   111000111
   111000111
   111000111

   The template:
   100

011
011

And the mask:

111
110
110

The output of your routine should be a 9x9 array of distances. If we call this D, then, for example, D(4,4) should be 0, while D(5,5) should be 4. Note that while this test only involves binary images, your program should work for grayscale images.

Turn in a printout of the results, along with your code.

You may be worried about how to compute SSD when the template goes outside the boundary. Don't worry about this. You can do anything that produces reasonable results. For example, you can just treat pixels outside S as if they were 0. This will work fine, since these areas then will generally not match your template very well.

2. **15 points:** Using SSD and the pseudocode below, implement the function FindMatches. This should find all candidate pixels in the sample that have a neighborhood that is sufficiently similar to the template.

You might want to write FindMatches so that it takes three inputs: the sample image, the template, and the mask. Test your program on the same example that you used in problem 1. If you test it using the error threshold listed below, you will get two possible matches. Also test it using a different template and threshold which produce more than one result. Hand in printouts of the results, and of your program.

3. **25 points:** Now complete your program. I am including below the pseudocode given by Efros and Leung. Test your program using a checkerboard pattern, and using the bricks image in brickbw.jpg. Try different window sizes. Hand in printouts of the results, and a brief explanation, with illustrations, of how the window size affects the results.

4. **10 points:** Use two more images of your own choosing as sample textures and generate new textures using these samples. Turn in printouts of the original images and of the textures you generate. Also, include these images in your electronic submission.

5. **10 points** Modify your program so that it will work with color images, and generate color textures.

Appendix: Below is pseudocode provided by Efros and Leung.

Algorithm details

Let `SampleImage` contain the image we are sampling from and let `Image` be the mostly empty image that we want to fill in (if synthesizing from scratch, it should contain a 3-by-3 seed in the center randomly taken from `SampleImage`, for constrained synthesis it should contain all the known pixels). `WindowSize`, the size of the neighborhood window, is the only user-settable parameter. The main portion of the algorithm is presented below. I have removed the part of the pseudocode that deals with Gaussian filtering, for simplicity.

```
function GrowImage(SampleImage,Image,WindowSize)
  while Image not filled do
    progress = 0
    PixelList = GetUnfilledNeighbors(Image)
    foreach Pixel in PixelList do
      Template = GetNeighborhoodWindow(Pixel)
      BestMatches = FindMatches(Template, SampleImage)
      BestMatch = RandomPick(BestMatches)
      Pixel.value = BestMatch.value
  end
  return Image
end
```

Function `GetUnfilledNeighbors()` returns a list of all unfilled pixels that have filled pixels as their neighbors (the image is subtracted from its morphological dilation). The list is randomly permuted and then sorted by decreasing number of filled neighbor pixels. `GetNeigborhoodWindow()` returns a window of size `WindowSize` around a given pixel. `RandomPick()` picks an element randomly from the list. `FindMatches()` is as follows:

```
function FindMatches(Template,SampleImage)
  ValidMask = 1s where Template is filled, 0s otherwise
  TotWeight = sum i,j ValidMask(i,j)
  for i,j do
    for ii,jj do
      dist = (Template(ii,jj)-SampleImage(i-ii,j-jj))^2
      SSD(i,j) = SSD(i,j) + dist*ValidMask(ii,jj)
    end
    SSD(i,j) = SSD(i,j) / TotWeight
  end
```

```
  PixelList = all pixels (i,j) where SSD(i,j) <=
min(SSD)*(1+ErrThreshold)
  return PixelList
end
```

In our implementation the constant were set as follows: `ErrThreshold = 0.1`. Pixel values are in the range of 0 to 1.