Digital Image Processing Laboratory: Neighborhoods and Connected Components

February 10, 2011

Introduction

This laboratory illustrates the concepts of pixel neighborhoods and connected pixel sets. Unless otherwise specified, you should implement all your algorithms using the C programming language.

In order to simplify notation, we will denote the set of 2-D lattice points by S and individual lattice points by $s \in S$. When necessary will we explicitly denote the 2-D coordinates of a lattice point by $s = (s_1, s_2)$, where s_1 is the horizontal coordinate (column) and s_2 is the vertical coordinate (row). Also the upper left-most pixel of the image will correspond to lattice location $s = (0, 0)$.

In this laboratory, we will use the following definitions for neighborhood and connectedness.

- We will use a 4 point neighborhood, so the neighbors of a lattice point (s_1, s_2) are

$$\partial(s_1, s_2) = \{(s_1 - 1, s_2), (s_1 + 1, s_2), (s_1, s_2 - 1), (s_1, s_2 + 1)\} \ .$$

  We will also use a free boundary, so pixels along the boundary of the image have less than 4 neighbors each.

- Two neighboring lattice points $r \in \partial s$ are said to be connected neighbors if

$$|x_s - x_r| \leq T \tag{1}$$

  where x_s is the pixel value at lattice point s and T is a fixed threshold. We will denote the connected neighbors of s by the set $c(s) \subset \partial s$. More specifically, for this application

$$c(s) = \{r \in \partial s | \ |x_s - x_r| \leq T\} \ .$$

- Pixels s and r are said to be connected if there is a sequence of M pixels $s_1, s_2, \cdots, s_M$ such at $s \in c(s_1)$, $s_1 \in c(s_2)$, $\cdots$, $s_{M-1} \in c(s_M)$, $s_M \in c(r)$.

Questions or comments concerning this laboratory should be directed to Prof. Charles A. Bouman, School of Electrical and Computer Engineering, Purdue University, West Lafayette IN 47907; (765) 494-0340; bouman@ecn.purdue.edu

1 Area Fill

In this section, you will write a C program that fills in an area of connected pixels in an image. To do this, you will compute the set of all pixels which are connected to a specified pixel s.

1. First write a C subroutine to find the connected neighbors of a pixel s. The structure of the subroutine call should be as follows:

```
struct pixel {
  int m,n;        /* m=row, n=col */
}

void ConnectedNeighbors(
  struct pixel s,
  double T,
  unsigned char **img,
  int width,
  int height,
  int *M,
  struct pixel c[4])
```

   - Subroutine inputs:

     `struct pixel s` - This data structure contains the location of the pixel `s` whose connected neighbors will be computed.

     `double T` - This is the threshold used in equation (1).

     `unsigned char **img` - This is the 2-D array of pixels `img[m][n]` denoted as x_s in equation (1).

     `int width` - This is the width of `img[height][width]`.

     `int height` - This is the height of `img[height][width]`.

   - Subroutine outputs:

     `int *M` - This is a pointer to the number of neighbors connected to the pixel `s`.

     `struct pixel c[4]` - This is an array containing the `M` connected neighbors to the pixel `s`. Here `M` is assumed to always be less than or equal to 4.

2. In this step, you will write a C subroutine to find all the pixels connected to s_0. The subroutine, which will be called `ConnectedSet`, may be implemented by maintaining 1) a list of pixels B which are known to be connected to s_0, but have not yet been searched, and 2) a segmentation image Y_s which is equal to 1 for pixel's which are known to be connected to s and is zero otherwise.

   Below is a simple description of an algorithm for doing this.

Initialize $Y_r = 0$ for all $r \in S$

$ClassLabel = 1$

ConnectedSet$(s_0, Y, ClassLabel)$ {

  $B \leftarrow \{s_0\}$

  While B is not empty {

    $s \leftarrow$ any element of B

    $B \leftarrow B - \{s\}$

    $Y_s \leftarrow ClassLabel$

    $B \leftarrow B \bigcup \{c(s) \cap \{r : Y_r = 0\}\}$

  }

  return(Y)

}

At termination, the set Y_s is 1 at all pixels that are connected to s_0. This type of algorithm is sometimes referred to as region growing because segmented region grows out from the initial seed point s_0. At each step, B contains pixels at the boundary of the set whose neighbors have not yet been checked.

The structure of the subroutine call should be as follows:

```
void ConnectedSet(
  struct pixel s,
  double T,
  unsigned char **img,
  int width,
  int height,
  int ClassLabel,
  unsigned int **seg,
  int *NumConPixels)
```

- Subroutine inputs:

  `struct pixel s` - This data structure contains the location of the pixel `s` that will serve as the seed.

  `double T` - This is the threshold used in equation (1).

  `unsigned char **img` - This is the 2-D array of pixels `img[m][n]` denoted as x_s in equation (1).

  `int width` - This is the width of img[height][width].

  `int height` - This is the height of img[height][width].

  `int ClassLabel` - This the integer value that will be used to label any pixel which is connected to `s`.

- Subroutine outputs:

`unsigned int **seg` - This is a 2-D array of integers which contains the class of each pixel and is passed to `ConnectedSet` from the main routine. If a pixel at location `i`, `j` is found to be connected to `s`, then `seg[i][j]`$\leftarrow$ *ClassLabel*. Otherwise the value of `seg[i][j]` is left unchanged.

`int *NumConPixels` - This is the number of pixels which were found to be connected to `s`.

Some additional notes:

- The memory for the array `unsigned int **seg` should be allocated by the main routine that calls `ConnectedSet`.
- The value of `NumConnectedPixels` needs to be passed back as a pointer, so the subroutine call for `ConnectedSet` might look like

```
ConnectedSet(s,T,img,width,height,ClassLabel,seg,&NumConPixels);
```

3. Down load the image *img22gd2.tif*. Apply the subroutine `ConnectedSet` to extract the connected set of pixels for $s = (67, 45)$, and $T = 2$. Note from our lattice definition that 67 is the column index, and 45 is the row index. Print out an image with the elements in the connected set printed as black and the remaining pixels printed as white.

Section 1 Report:
Hand in:

1. A print out the image img22gd2.tif.
2. A print out of the image showing the connected set for $s = (67, 45)$, and $T = 2$.
3. A print out of the image showing the connected set for $s = (67, 45)$, and $T = 1$.
4. A print out of the image showing the connected set for $s = (67, 45)$, and $T = 3$.
5. A listing of your C code.

2 Image Segmentation

In this section, you will use the subroutines for region filling to segment the image into connected components.

1. Use the subroutine `ConnectedSet` to extract all the connected sets in the image *img22gd2.tif*. You can do this by indexing through the image in raster order and appling the `ConnectedSet` subroutine at each pixel that does not yet belong to a connected set. Note that for a small threshold, the size of most connected sets for this image will be small, resulting in a large number of connected sets in the segmentation.

2. Generate a segmentation of the image consisting of connected sets containing greater than 100 pixels. Number each of these large connected sets sequentially starting at 1. All remaining connected sets should be labeled as 0. There will be fewer than 255 large connected sets, so you can store the label for each pixel as a 2-D unsigned character array. Save this 2-D array as a monochrome TIFF image, `segmentation.tif`.

3. To view your segmentation clearly, you will need to scramble the colormap to provide contrast between the distinct regions. You can do this in Matlab with the following commands:

```
x=imread('segmentation.tif');
N=max(x(:));
image(x)
colormap(rand(N,3))
axis('image')
```

   Print or export this color segmentation of the image.

Section 2 Report:
Hand in:

1. Print outs of the randomly colored segmentation for $T = 1$, $T = 2$, and $T = 3$.
2. A listing of the number of regions generated for each of the values of $T = 1$, $T = 2$, and $T = 3$.
3. A listing of your C code.