University of Nevada, Reno

CS 302 — Data Structures

Assignment #5

Students:
Joshua Gleason
Josiah Humphrey

Instructor:
Dr. George Bebis

May 3, 2010

Contents

1 Introduction

In this programming assignment, we continued and built upon image processing. We took out previous programming assignment and added more functionality. Our goals were to represent the regions found by connected components in a list and to compute a number of useful properties to characterize their position, orientation, shape, and intensity. This was a challenge that tested our ability to understand the data structures and image processing. We were also asked to use our skills to manipulate items in lists, use templates, and learn about feature extraction and classification.

In this assignment, we learned about geometric properties of objects and how to objectively classify them using mathematical formulas. We were able to implement the equations that found various geometric properties of objects found in images. Since this is a introduction to image processing, we were only asked to implement geometric and intensity properties only. Using more advanced techniques we could have done more complicated things.

To calculate these properties, we used a technique from probability theory called moments. A moment is defined as

$$M_{p,q} = \sum_{i,j \in R} i^p j^q$$

This was the basis for many of our calculations and was implemented as a generic function that would take any numbers for p and q.

2 Use of Code

The use of this code should be fairly straight forward. The main menu is easy to understand, just select the option you want and hit enter. To load images into the programs image registers, you can either do it with the menu option, or on the command line. To load images on the command line, just enter them as arguments. For instance, you could do `$ ./main.out images/hubble1.pgm`. You can enter as many images as you want on the command line like so: `$ ./main.out images/hubble1.pgm images/hubble2.pgm images/hubble3.pgm`

To classify the regions in an image, you need to choose the classify region option in the menu. Once you do this, you will be able to choose which of the classifiers you would like to do. You can do multiple things to the same image. Once you are done classifying the image, make sure you choose the save image option that is in that sub-menu to save all of the changes. You will then need to save the image at the top of the main menu in order to output the image.

One note about the classifiers. There are minimum and maximum values for each function, but to make them easier to use, if the user inputs bounds that are below the minimum and/or above the maximum, it defaults to the minimum and/or maximum value respectively. This was done to create a more user friendly program that is capable of accepting data that the user intends. For instance, if the user wants to include the maximum size, you can just enter "99999999" and it will interpret that input as a maximum value, but the program will automatically make that input sane for the environment.

3 Functions

3.1 Image.h

REGIONTYPE()

```
RegionType();
```

Purpose
Sets all of the data to 0 or 0.0.

Input
None

Output
None

Assumption
None

RegionType(const RegionType¡pType¿&);

OPERATOR>

```
bool operator>(const RegionType<pType> &rhs) const;
```

Purpose
Overloaded function for greater than.

Input
A RegionType object to compare to

Output
Bool value based on the output of the comparison

Assumption
None

OPERATOR<

```
bool operator<(const RegionType<pType> &rhs) const;
```

Purpose
Overloaded function for less than.

Input
A RegionType object to compare to

Output
Bool value based on the output of the comparison

Assumption
None

OPERATOR>=

```
bool operator>=(const RegionType<pType> &rhs) const;
```

Purpose

Overloaded function for greater than or equal

Input

A RegionType object to compare to

Output

Bool value based on the output of the comparison

Assumption

None

OPERATOR<=

```
bool operator<=(const RegionType<pType> &rhs) const;
```

Purpose

Overloaded function for less than or equal

Input

A RegionType object to compare to

Output

Bool value based on the output of the comparison

Assumption

None

OPERATOR==

```
bool operator==(const RegionType<pType> &rhs) const;
```

Purpose

Overloaded function for equal to

Input

A RegionType object to compare to

Output

Bool value based on the output of the comparison

Assumption

None

OPERATOR=

```
RegionTypep& operator=(const RegionType<pType> &rhs);
```

Purpose

The overloaded = sign that will copy the object on the left into the object on the right

Input
The object to be copied from

Output
The object itself to allow chaining

Assumption
None

setData

```
void setData( const ImageType<pType>& );
```

Purpose
The function that calls all of the other functions to set all of the data members

Input
An image of some sort

Output
None

Assumption
Assumes the picture is valid, but it will work for an image as long as the image exists.

getCentroidR

```
double getCentroidR() const;
```

Purpose
Gets the R centroid for the image

Input
None

Output
The double value for the centroid

Assumption
None

getCentroidC

```
double getCentroidC() const;
```

Purpose
Gets the C centroid for the image

Input
None

Output
None

Assumption

None

getSize

```
int getSize() const;
```

Purpose

Gets the size of the region using the moment calculation

Input

None

Output

The integer value for the size

Assumption

None

getOrientation

```
double getOrientation() const;
```

Purpose

Gets the orientation for a region

Input

None

Output

The double value for the orientation of the region

Assumption

None

getEccentricity

```
double getEccentricity() const;
```

Purpose

Gets the eccentricity value for the region

Input

None

Output

The double value for the eccentricity

Assumption

None

getMeanVal

```
pType getMeanVal() const;
```

Purpose

Gets the mean pixel value

Input

None

Output

The average pixel value returned as a pixelType

Assumption None

GETMINVAL

```
pType getMinVal() const;
```

Purpose

Gets the minimum value for the region

Input

None

Output

The pixelType value for the minimum pixel value

Assumption

None

GETMAXVAL

```
pType getMaxVal() const;
```

Purpose

Gets the maximum value for the region

Input

None

Output

The pixelType value for the maximum pixel value

Assumption

None

MOMENT

```
double moment(int, int);
```

Purpose

Calculates the moment using this formula:

$$M_{p,q} = \sum_{i,j \in R} i^p j^q$$

Input

Two ints that correspond to p and q in the equation

Output

The double value that is calculated by the function.

Assumption

A natural number for p and q.

MU

```
double mu(int, int);
```

Purpose

Calculates the equation defined by

$$\mu_{p,q} = \sum_{i,r \in R} (i - \bar{x})^p (j - \bar{y})^q$$

Input

Two ints for p and q

Output

The double value calculated by μ

Assumption Natural numbers for the ints

XYBAR

```
void xyBar();
```

Purpose

Calculates the equation defined by

$$\bar{x} = \frac{M_{1,0}}{M_{0,0}}$$

and

$$\bar{y} = \frac{M_{0,1}}{M_{0,0}}$$

Input

None

Output

None

Assumption

None

LAMBDA

```
void lambda();
```

Purpose

Calculates the equation defined by

$$\lambda_{max} = \frac{1}{2}(\mu_{2,0} + \mu_{0,2}) + \frac{1}{2}\sqrt{\mu_{2,0}^2\mu_{0,2}^2 - 2\mu_{0,2}\mu_{2,0} + 4\mu_{1,1}^2}$$

and

$$\lambda_{min} = \frac{1}{2}(\mu_{2,0} + \mu_{0,2}) - \frac{1}{2}\sqrt{\mu_{2,0}^2\mu_{0,2}^2 - 2\mu_{0,2}\mu_{2,0} + 4\mu_{1,1}^2}$$

Input

None

Output

None

Assumption

None

THETA

```
void theta();
```

Purpose

Calculates the equation defined by

$$\theta = \tan^{-1}\frac{\lambda_{max} - \mu_{2,0}}{\mu_{1,1}}$$

Input

None

Output

None

Assumption

None

EPSILON

```
void epsilon();
```

Purpose

Calculates the eccentricity defined by the equation

$$\varepsilon = \sqrt{\frac{\lambda_{max}}{\lambda_{min}}}$$

Input

Output

Assumption

3.2 sortedList.h

SORTEDLIST()

```
sortedList();
```

Purpose
Sets the data to null and length to 0

Input
None

Output
None

Assumption
None

SORTEDLIST

```
~sortedList();
```

Purpose
Deletes the list

Input
None

Output
None

Assumption
None

GETLENGTH

```
int getLength();
```

Purpose
Returns the length of the list

Input
None

Output
Returns the length of the list

Assumption
None

MAKEEMPTY

```
void makeEmpty();
```

Purpose

Empties the list

Input

None

Output

None

Assumption

None

RETRIEVEITEM

```
bool retrieveItem( T& );
```

Purpose

Checks to see if the item passed to the function is in the list

Input

The item to be checked if it exists in the list

Output

The bool to say if the item was found

Assumption

None

INSERTITEM

```
void insertItem( T );
```

Purpose

Inserts the item into the correct place into the list

Input

The item to be inserted

Output

None

Assumption

None

DELETEITEM

```
void deleteItem( T );
```

Purpose

Deletes the item and relinks the list to preserve the sorted attribute

Input

The item to be deleted

Output

None

Assumption

None

RESET

```
void reset(');
```

Purpose

Resets the head pointer to be at the top of the list. This needs to be done before running through the list.

Input

None

Output

None

Assumption

None

ISEMPTY

```
bool isEmpty();
```

Purpose

A function to tell you if the list is empty

Input

None

Output

Bool telling you if the list is empty

Assumption

None

ATEND

```
bool atEnd();
```

Purpose

Tells the user if the current list pointer is pointing at the last element. Useful for using a while loop to do things to the list.

Input

None

Output

Bool telling you if the current list item is pointing to NULL

Assumption

None

getNextItem

```
T getNextItem();
```

Purpose

Gets the next item in the list

Input

None

Output

The item that comes next in the list

Assumption

None

operator=

```
sortedList<T>& operator=(const sortedList<T>&);
```

Purpose

Copies the list into another list

Input

The list to be copied from

Output

The object to allow for chaining

Assumption

None

3.3 list.h

list()

```
sortedList();
```

Purpose

Sets the data to null and length to 0

Input

None

Output

None

Assumption

None

IST

```
~sortedList() { makeEmpty(); }
```

Purpose
Deletes the list

Input
None

Output
None

Assumption
None

GETLENGTH

```
int getLength();
```

Purpose
Returns the length of the list

Input
None

Output
Returns the length of the list

Assumption
None

MAKEEMPTY

```
void makeEmpty();
```

Purpose
Empties the list

Input
None

Output
None

Assumption
None

RETRIEVEITEM

```
bool retrieveItem( T& );
```

Purpose

Checks to see if the item passed to the function is in the list

Input

The item to be checked if it exists in the list

Output

The bool to say if the item was found

Assumption

None

INSERTITEM

```
void insertItem( T );
```

Purpose

Inserts the item into the list

Input

The item to be inserted

Output

None

Assumption

None

DELETEITEM

```
void deleteItem( T );
```

Purpose

Deletes the item and relinks the list

Input

The item to be deleted

Output

None

Assumption

None

RESET

```
void reset();
```

Purpose

Resets the head pointer to be at the top of the list. This needs to be done before running through the list.

Input
None

Output
None

Assumption
None

isEmpty

```
bool isEmpty();
```

Purpose
A function to tell you if the list is empty

Input
None

Output
Bool telling you if the list is empty

Assumption
None

atEnd

```
bool atEnd();
```

Purpose
Tells the user if the current list pointer is pointing at the last element. Useful for using a while loop to do things to the list.

Input
None

Output
Bool telling you if the current list item is pointing to NULL

Assumption
None

getNextItem

```
T getNextItem();
```

Purpose
Gets the next item in the list

Input
None

Output

The item that comes next in the list

Assumption

None

OPERATOR=

```
sortedList<T>& operator=(const sortedList<T>&);
```

Purpose

Copies the list into another list

Input

The list to be copied from

Output

The object to allow for chaining

Assumption

None

3.4 driver.cpp

COMPUTECOMPONENTS

```
int computeComponents( ImageType<pType>,
        sortedList<RegionType<pType> >& );
```

Purpose

The main calling function that will get all of the regions and classify them

Input

A single image and a list of regions

Output

Fills the refion list with all the regions in the image and returns the total number of regions

Assumption

That the image is a valid image and sorted list is initialized

FINDCOMPONENTSDFS

```
void findComponentsDFS( ImageType<pType>,
        ImageType<pType>&, int , int , pType,
        RegionType<pType>&, const ImageType<pType>& );
```

Purpose

Computes the region's attributes and stores them in the region node

Input

Input image, output image, location of a pixel in the region, a region node to store the data in, the original image

Output

Fills the region object as well as flooding the region in the output image

Assumption

That the input is already thresholded and the location is part of the region.

deleteSmallRegions

```
void deleteSmallRegions( sortedList<RegionType<pType> >&,
        int );
```

Purpose

Finds the regions in an image that are below in size of a certain threshold and deletes them

Input

A list of regions

Output

Prints a summary of all the regions to the screen

Assumption

That the region list is a valid list

printSummary

```
void printSummary( sortedList<RegionType<pType> >& );
```

Purpose

Prints a summary of the regions to the screen

Input

A list of regions

Output

A summary of all of the regions

Assumption

That the list of regions is valid

4 Bugs and Errors

During the creating of this program, there was one single bug that took a very, very long time to track down, following is a detailed explanation of the bug and the methods used to track down and repair it.

The problem originally manifested itself as a segmentation fault when the choice to 'Classify Regions' was selected in the main menu. At first I looked through the classifyRegions function for

any obvious problems, after that search came up empty I began using the GDB debugger to track down the fatal error.

The first thing I needed to know was where the actual error was occurring, so I executed the program in GDB. After the re-creating the segmentation fault I found that the crash was occurring a conditional statement inside of the == operator overload function inside of the Region Type class. By examining parameters passed to the function I discovered that the right hand side was actually an invalid value, printing the address of the parameter I found the value was actually NULL. This seemed very strange, so I used GDB's backtrace command, which indicated that the comparison was taking place in the deleteItem function of the sortedList class or more specific the list of regions for the image.

Before debugging further, I pondered the recently acquired information and came to a hypothesis. I believed that the deleteItem function was not finding the value that it was passed even thought the RegionType values were being directly taken from the list of regions. This was the only way I could conceive the == operator being passed NULL from deleteItem. Some more debugging was definitely needed to verify this claim and also answer some other questions if this was the case.

After setting a breakpoint in the deleteItem function I ran the program and selected the Classify Regions option. The program paused at the first breakpoint where I obtained some very interesting information about the RegionType in question. I ran the command `print *this` in GDB so that I could quickly see all of the private members of the current object. To my surprise one of the values was definitely invalid, which may explain why the == operator never returned true, even if the values had the same data members. What would happen if you tried to compare two invalid double values, even if they were copies of one another? I had to determine the answer to this question. By continuing execution I found where two regions had all the same valid data members and when finished the == function I discovered that the returned value was false, which would explain why deleteItem never found the right value.

At this point I was feeling pretty good about having narrowed down the problem to a calculation issue, but why was I getting invalid values for eccentricity for some of the regions? To determine this I set a breakpoint in the setData function of RegionType and recreated the error yet again. To my surprise the first region had some invalid values, but I also noticed that the value for lambdaMin was zero; I thought I recalled the eccentricity requiring dividing by lambdaMin, so I checked it out. I verified that this was the indeed true, so I decided to find why lambdaMin was being set to zero. After using similar techniques I discovered that lambdaMin was zero because the central moment was returning zero for regions of width or height one. I determined that the calculation was correct because I had received the same value when I did it by hand, so the problem wasn't actually a coding problem, it was a problem with the function equation. To fix this problem, I added an exception to prevent dividing by zero by adding one to the numerator and denominator if the central moment of 1,1 returned zero. This single bug took nearly two hours to track down, but after discovering the cause I at least feel much more compentent with GDB.

5 What was Learned

In this lab, the students learned about classifying regions in images. The students also learned about moments from probability theory. The students combined these aspects of problem solving to come up with a solution to make probabilistic calculations on regions found in an image. The

students used these tools to better understand the application and development of image processing. The students were able to combine these tools successfully to implement a working program that can correctly classify regions found within an image. To store the regions unique data, the students implemented a sorted list and an unsorted list that contained the x, y coordinates of the regions and a sorted list of the regions and their associated data. This helped the students better understand data structures and helped the students to know how to implement and develop a data structure. Since it was suggested to template these data structures, the students choose to do this and had a template for the unsorted and sorted list types. This proved to further the student's knowledge and prowess of templates in C++. The students also learned how to embed other objects into objects. This was done with the region object that had embedded in it an unsorted list to hold the region pixel locations. This was not too hard, but still taught the students how to embed objects in objects.

6 Division of Labor

For this assignment, the labor was divided equally among the partners. Each student contributed equally to the production and development of the code. The work was divided after the assignment was announced, and the students each had about the same work to be done.

Joshua was responsible for writing the sorted list and Josiah wrote the unsorted list. For the calculations Josiah wrote most of the functions in RegionType dealing with moment and central moments and Joshua wrote the intensity calculations. Joshua also wrote most of the extra curses based code in driver including printSummary, both students contributed equally to the classifyRegions function. The documentation was split evenly while both students put their own part into most of the sections and reviewed all of the sections.

7 Extra Credit

Sorted List

The implementation for the sorted list was completed and done using templates. The lists were some of the first things that were templated because they were needed in order to complete the rest of the assignment. The sorted list uses a link list implementation and keeps the list sorted. While a array based implementation would have been faster to search and retrieve items, we choose a linked list implementation because we would not know how large the list would need to be. Therefore the only possible implementation that is efficient is a linked list implementation of the sorted list.

Unsorted List

The implementation for the unsorted list was completed and done using templates. The lists were some of the first things that were templated because they were needed in order to complete the rest of the assignment. The unsorted list uses a link list implementation and inserts at the fastest possible place because it is unsorted and it does not matter where the nodes are inserted. While a array based implementation would have been faster to search and retrieve items, we choose a linked list implementation because we would not know how large the list would need to be. Therefore the only possible implementation that is efficient is a linked list implementation of the unsorted list.