University of Nevada, Reno

CS 302 — Data Structures

Assignment #3

Students:
Joshua Gleason
Josiah Humphrey

Instructor:
Dr. George Bebis

March 30, 2010

Contents

List of Figures

1 Introduction

In this assignment, the students were asked to take an image of the night sky and count the number of regions, or galaxies, in the picture. This was to be done using stacks and queues. The students were also asked to implement the counting of th pixels in an iterative and recursive way and to see the difference that both methods take.

The first problem that the students ran into was the use of stacks and queues. The students wanted to get the core project done first, so they utilized the STL version of the stacks and queues first. After the core program was found to be complete, the students then implemented their own stack and queue templates for the assignment.

Before the stacks and queues could be used, however, the students had to implement a breadth first and depth first search that would search for the regions in the image. One of the searches used stacks while the other used queues to implement the region finding.

The assignment also required the use of constructors, destructors, copy constructors, and operator overloading, These topics were review from CS 202, but the use of them refreshed the students minds on how they work and to their purposes.

The students also extensively documented their program and made it as easy as possible to understand what was happening in the various algorithms that were implemented. The students realize that commenting and documenting the source code for projects is extremely important and is essential to the success of a powerful programmer.

2 Use of Code

The use of the program should be very intuitive. The user must use the arrow keys to scroll in the menu and press enter to select some option. The main program will first have the student decide to choose if the user wants to edit color or gray scale images. This is a limitation and was done to prevent the code from becoming too complex. The menu will then pop up windows and message boxes depending on the user's selection and the user's input. This provides for a very effective setup and takes the burden out of trying to look at the keyboard and the menu for key choices in a simple menu. Our menu uses ncurses to accomplish the scrolling menu. The students adapted a ncurses API written by Micheal Leverington from CS 135 to implement a simple curses menu at the beginning, but after all of the functions were completed, the students decided to make a more robust menu capable of scrolling. This was not required, but was done for esthetic reasons and for the experience that it allowed the students.

The image class that was implemented has all of the requirements for the functions. Each function is defined exactly like Dr. Bebis wanted them to be defined and the coders adapted their coding style to match what Dr. Bebis expected. The code is built fairly modular and most of the time expects the class user to bounds and error check before information is sent to the class. The image class has also been templated. This allows the user to use any image type as long as there is a read and write function provided. The RGB class was also made to hold the data for the color images. As long as the read and write functions hold their info in a Red, Green, Blue setup, the image class should be able to handle the image. There is some rudimentary error checking in the class itself, but it only prevents the most foul and gross errors. The class is also setup to throw string objects in the case of an error. These thrown strings must be caught in the driver, so it is up to the class user to implement the catches for these thrown strings. Strings were used as the errors because of their ease of use and ease of manipulation by the class user.

The file IO functions that were provided by Dr. Bebis have been combined into a single file pair (.cpp and .h) called imageIO. This allows a much more unified and modular approach to the use of the image IO features. It also makes including the functions much easier to include into multiple source files. This file has been modified from the last lab to have the ability to read and write PPM and PGM files. The PPM functions depend on the RGB class to store the data.

Whenever a new image is needed by the user of the class, the user should always use the setImageInfo function that is included in the image class. This function takes the rows, columns, and levels and creates a blank image to manipulate. Another feature of this function is that it creates a checkered background of 25x25 squares that make it easy to see the dimensions of the image if it is saved without modification. It is also helpful for functions that take an image and move them somehow. For instance, if an image was translated by 55 pixels, the program user could see that there is 2.2 squares that are uncovered, showing that the images has been translated 55 pixels. This was a design choice that came about as a result of how other image manipulation programs work and the students attempt at creating a robust image manipulation resource.

The count regions functionality has been added to the menu and can be used quite easily. To use it, select its option from the menu and select which image to count the regions. After the counting has been done, the image is modified in the register so that the user has the option of saving the output file to the disk so that the user may view the regions in an external image viewer program. To choose between the different methods of searching for the regions, the user must uncomment the appropriate function in driver.cpp.

One last area that needs some mention to properly use is the function of operator+. This function includes a coefficient that determines the weight that each image has when adding them together. The students mimicked Dr. Bebis name for this coefficient and called it 'a'. When 'a' is large, the first image has more weight in the addition; when 'a' is small, the second image has more weight, and when 'a' is .5, the images have equal weight in the addition. The coefficient can be between 0 and 1.

3 Functions

3.1 Image.h

DILATE

```
void ImageType<pType>::dilate()
```

Purpose

This function changes any pixel to black that is touching a black pixel on any of its 8 sides.

Input

None

Output

None

Assumptions

Nothing is assumed but it makes sense to actually have a defined picture.

ERODE

```
void ImageType<pType>::erode(){
```

Purpose

This function changes any pixel to white that is touching a white pixel on any of its 8 sides.

Input

None

Output

None

Assumptions

Nothing is assumed but it makes sense to actually have a defined picture.

THRESHOLD

```
void ImageType<pType>::threshold(){
```

Purpose

Applies this formula to each pixel:

$$O(i,j) = \begin{cases} 255 & \text{if } I(i,j) > T \\ 0 & \text{if } I(i,j) <= T \end{cases}$$

But must first find the correct value for T by using a number derived from the average pixel value.

Input

None

Output

None

Assumptions

Nothing is assumed but it makes sense to actually have a defined picture.

3.2 driver.cpp

COUNTREGIONS

```
void countRegions(ImageType<pType> img[],
        bool loaded[], char name[][NAME_LEN])
```

Purpose

This prompts the user for the image in which to count the number of regions.

Input

- img[]
  The image to be region counted.

- loaded[ ]
  The bool value that sets whether or not the image has been loaded.
- name[ ][NAME_LEN]
  The char array that holds the image name.

Output

None

Assumptions

Assumes that there has been a image loaded into the register, but does not require that.

FINDCOMPONENTSDFS

```
void findComponentsDFS(ImageType<pType> inputImg,
        ImageType<pType>& outputImg, int startRow,
        int startCol, pType label)
```

Purpose

Uses a depth first search to count regions. Uses a stack to fill regions at the deepest point and working its way back.

Input

- inputImg
  The image to have its regions counted.
- outputImg
  The image that will be saved to once the original image has had its regions counted.
- startRow
  The row at which to start the counting of regions.
- startCol
  The column at which to start the counting of regions.
- label
  The label to label each region as it is counted. Should be unique.

Output

None

Assumptions

Assumes nothing but it makes sense if there is a image that is being counted and that the image has a region.

FINDCOMPONENTSBFS

```
void findComponentsBFS(ImageType<pType> inputImg,
        ImageType<pType>& outputImg, int startRow,
        int startCol, pType label)
```

Purpose

Uses a breadth first search to count regions. Uses a queue to fill regions at the shallowest point and works its way from there.

Input

- inputImg
  The image to have its regions counted.
- outputImg
  The image that will be saved to once the original image has had its regions counted.
- startRow
  The row at which to start the counting of regions.
- startCol
  The column at which to start the counting of regions.
- label
  The label to label each region as it is counted. Should be unique.

Output

None

Assumptions

Assumes nothing but it makes sense if there is a image that is being counted and that the image has a region.

FINDCOMPONENTSREC

```
void findComponentsRec(const ImageType<pType>& inputImg,
        ImageType<pType>& outputImg, int startRow,
        int startCol, pType label)
```

Purpose

Uses a depth first recursive function search that recursively floods the current region with the value of the label.

Input

- inputImg
  The image to have its regions counted.
- outputImg
  The image that will be saved to once the original image has had its regions counted.
- startRow
  The row at which to start the counting of regions.
- startCol
  The column at which to start the counting of regions.
- label
  The label to label each region as it is counted. Should be unique.

Output

None

Assumptions

Assumes nothing but it makes sense if there is a image that is being counted and that the image has a region.

3.3 queue.h

CONSTRUCTOR

```
queue();
```

Purpose

Constructs the queue by setting its two pointers to `NULL`

Input

None

Output

None

Assumptions

None

DESTRUCTOR

```
~queue();
```

Purpose

Calls makeEmpty in order to empty and destroy the entire list.

Input

None

Output

None

Assumptions

None

MAKEEMPTY

```
void makeEmpty();
```

Purpose

Deletes the nodes of the list until it reaches the end of the queue.

Input

None

Output

None

Assumptions

None

EMPTY

```
bool empty( )const;
```

Purpose

Returns a bool to show if the list has any elements in it.

Input

None

Output

Bool showing if the list is empty. False for not empty, true for empty.

Assumptions

None

PUSH

```
void push(const T&);
```

Purpose

Pushes an element into the queue.

Input

- item The item to be pushed into queue. Since this is a template queue, any data type can be pushed into the queue as long as it is homogeneous data.

Output

None

Assumptions

Assumes that all of the data being pushed into the queue are all the same data types. You cannot mix and match data types.

POP

```
void pop( );
```

Purpose

Deletes (pops) the front of the queue off.

Input

None

Output

None

Assumptions

None

FRONT

```
T& front() const;
```

Purpose

Returns the value at the front of the queue. The user is also allowed to change the value of the item.

Input

None

Output

The item is returned by reference so that the queue is able to be changed.

Assumptions

None

BACK

```
T& back( );
```

Purpose

Returns the value at the rear of the queue. The user is also allowed to change the value of the item.

Input

None

Output

The item is returned by reference so that the queue is able to be changed.

Assumptions

None

3.4 stack.h

CONSTRUCTOR

```
stack();
```

Purpose

Constructs the stack by setting its pointer to `NULL`

Input

None

Output

None

Assumptions

None

DESTRUCTOR

```
~stack();
```

Purpose

Calls makeEmpty in order to empty and destroy the entire list.

Input

None

Output

None

Assumptions

None

MAKEEMPTY

```
void makeEmpty();
```

Purpose

Deletes the nodes of the list until it reaches the end of the stack.

Input

None

Output

None

Assumptions

None

EMPTY

```
bool empty( )const;
```

Purpose

Returns a bool to show if the list has any elements in it.

Input

None

Output

Bool showing if the list is empty. False for not empty, true for empty.

Assumptions

None

PUSH

```
void push(const T&);
```

Purpose

Pushes an element into the stack.

Input

- item The item to be pushed into stack. Since this is a template stack, any data type can be pushed into the stack as long as it is homogeneous data.

Output

None

Assumptions

Assumes that all of the data being pushed into the stack are all the same data types. You cannot mix and match data types.

POP

```
void pop( );
```

Purpose

Deletes (pops) the top of the stack off.

Input

None

Output

None

Assumptions

None

TOP

```
T& top() const;
```

Purpose

Returns the value at the top of the stack. The user is also allowed to change the value of the item.

Input

None

Output

The item is returned by reference so that the stack is able to be changed.

Assumptions

None

4 Bugs and Errors

During the creation of this program there were a multitude of bugs we ran into, both simple and complex. The first bug that we ran into was when we initially was actually in our old code and had to do with resizing a ImageType that was already created. There was a bug that if the width or height of the old image was equal to the new image the but the other dimension was different, the program would crash by not resizing the pixelValue array. This was fixed by simply changing an OR to an AND.

Another bug we ran into was when building the erode function we forgot to first save a temporary copy of the image which caused very strange results. Luckily the problem was quickly identified and fixed.

5 What was Learned

The most important thing learned in this assignment was the use of templates. This was taught briefly in the previous semester of CS, so the concept of it was fairly new to the students. This challenge did not sway them though. The students quickly grasped the concepts of templates and easily implemented it into the image class to use both ints and RGB values. The students were then able to design the functions that manipulated the 2D array of ints and RGB both, that in turn manipulated the image. This was an important concept to grasp because without understanding the concepts of a template, the students would not be able to correctly develop algorithms that correctly manipulate any type of image format.

Another concept that the students learned was the knowledge of how images are stored. The students had first hand experience with an image format and this knowledge allowed the students to have a greater understanding of how computers and the data structures associated with file formats intertwine with the software developer. The students have more of an appreciation for the necessity of data structures and the implications of designing easy to use and compact data structures for the end user. The students also learned how color images can be represented in computers and gained excellent hands on experience with how they work and how to manipulate them.

One of the other concepts that the students learned was the ability to use ncurses in their program. This knowledge allowed the students to create a very easy-to-use menu that is capable of multiple floating and scroll-able menus. While this part of the program was not needed, the students had a desire to learn more about C++ than what is being taught in the class and brought it upon themselves to learn about ncurses. This knowledge has greatly improved their knowledge of pointers and correct program structure because ncurses utilizes many pointers. The implementation of ncurses also becomes very complex if there is not a clean program structure implemented. There are many facets of ncurses that are intertwined, so the students made a clean and modular ncurses driver that is able to be easily read.

The students were also able to see how image processing can be very powerful in implementation. The students learned that it is possible to count objects in an image file very easily. This will be a powerful tool in their careers as they move on to other things. The students also got to see the difference of how the searches can be implemented and the various cost differences between them.

Since this was a group project, we felt that it would be fairly difficult to merge and maintain our code, so we decided to learn and use git. Git is a revision control program that focuses on software development. It was developed by Linus Torvalds for the development of the Linux kernel.

Git has proved to be extremely useful in the management of the source code. It did have a small learning curve, but once that was crossed, it has become one of the most valuable tools that I know of.

6 Division of Labor

For the requirements that were asked for, each student did an equal amount of work. The students met very early and divided up the required functions equally. Each student worked equally on the implementation of the region counting. For the documentation, Josiah wanted to go above the normal implementation of a standard documentation and implemented a LaTeXdocument that made the writing of the documentation very easy and structured. Overall, the students contributed equally for their level of programming skill.

7 Extra Credit

7.1 Template stack and queue

Our program originally used the standard template library version of the stack and queue, so when we implemented the data structures we used the same member function names so that we wouldn't have to change our driver program. The templating went very smoothly and was similar to the previous programming assignment.

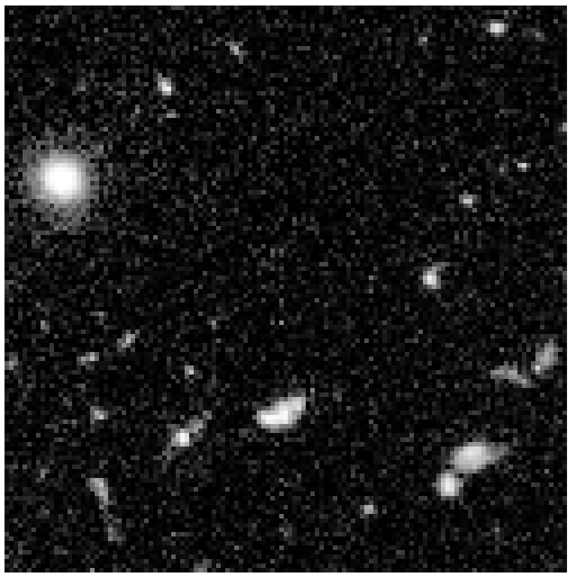
7.2 Auto Threshold

The algorithm we used for auto threshold first finds the mean value of the pixels using the mean-Value function written in the previous programming assignment. After this the average value of pixels greater than the average we calculated. The final step adds two fifths the difference of that average and the maximum pixel values. he last step was determined using trial and error but the first two steps make sense because by taking the average of the values above average all the black and nearly black pixels aren't calculated in the next average which dramatically increases the value. This seems to work very well giving us counts we would expect to receive from all the test images.

7.3 Recursive find component

The recursive function for find components was implemented by first checking the bounds of the location in question to make sure it is not out of bounds. If this is the case the value of the pixel is checked and if it does not already have the label value and it is white, then it recursively calls itself for all the surrounding pixels. This method has a base case of the pixel value is already set in which case it does nothing, and the problem gets smaller because eventually the entire region is filled.

Timing the function we actually found that with a large test image with many small regions the recursive function is actually faster than the BFS and DFS versions. The recursive function ran in approximately 0.24 seconds while the BFS/DFS searches took approximately 2.3 seconds. The only explanation we could come up with for this behavior is that the BFS and DFS need to constantly push and pop onto node based stacks requiring constant memory allocation and de-allocation. If we used an array based stack/queue then perhaps the time for BFS and DFS would be much faster.


(a) Original

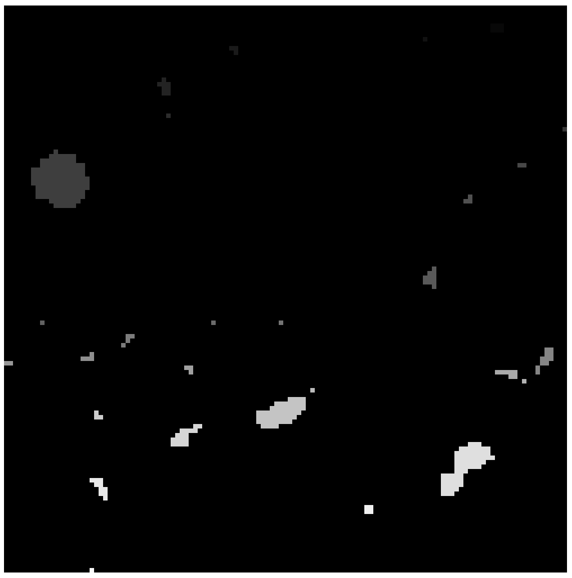

(b) Regions counted

Figure 1: Hubble 1

8 Examples

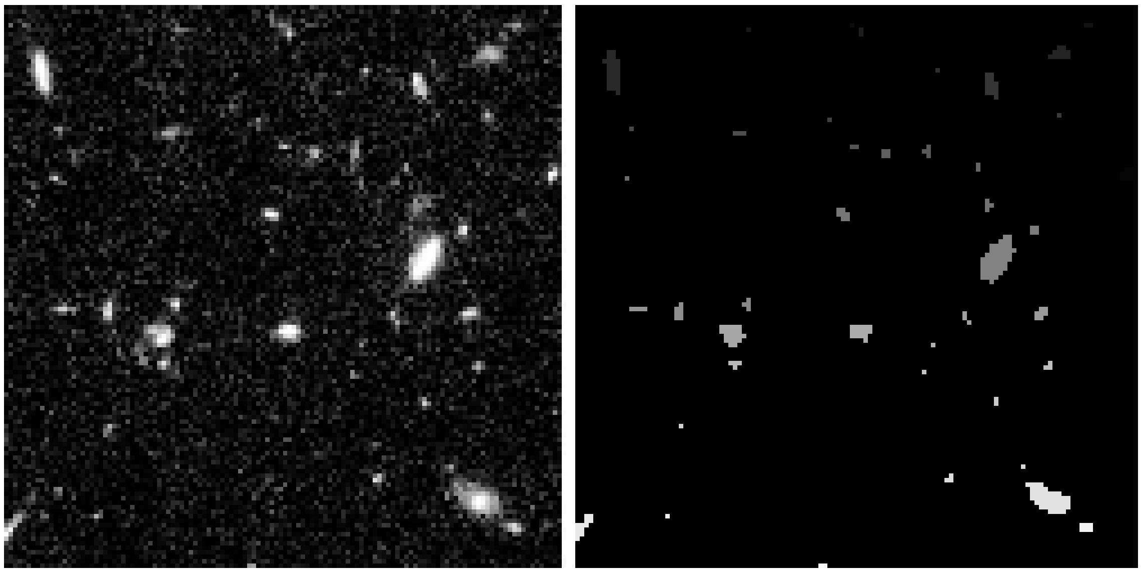

(a) Original (b) Regions counted

Figure 2: Hubble 2

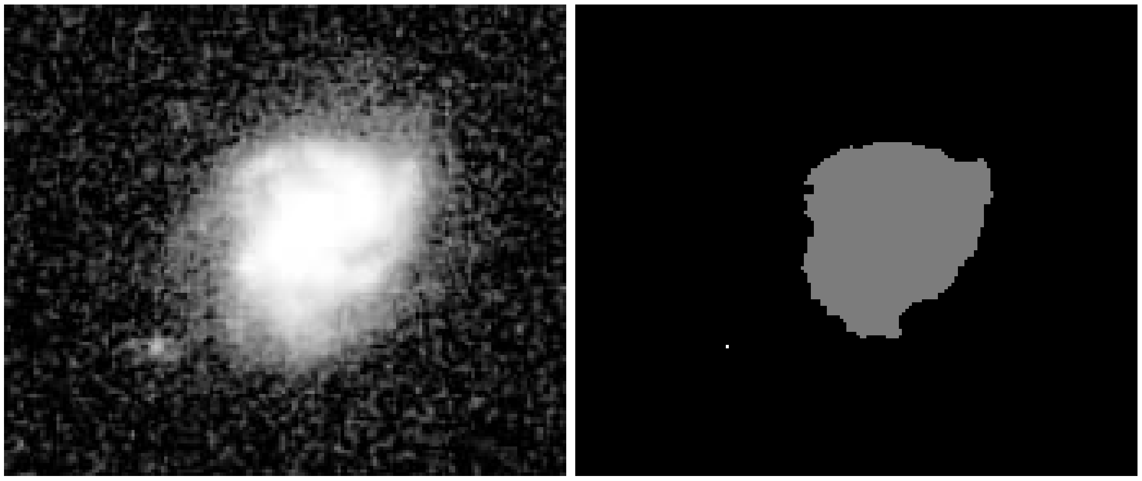

(a) Original (b) Regions counted

Figure 3: Hubble 6