CS103L Connected Component

August 20, 2018

1 Introduction

In this assignment you will implement a program to read an image and identify different objects in the image and label them using a method called Connected-component labeling. After that you will assign a different color to each identified object see figure 1.

(a) Original Image (b) Segmented Image

Figure 1: an image after segments are colored

Connected-component labeling is used in computer vision to detect connected regions in binary images. Labeling can be integrated with a number of computer vision systems such as image recognition systems and human-computer interaction[1].

In order to perform connected-component labeling you have to implement *bread-first search*, a simple search algorithm that after the first pixel of a connected component - an object- is found, searches for all the connected pixels of that connected component and label them. Following the labeling stage, the image may be partitioned into objects.

To make the task easier, we are going to perform the labeling on a binary image - black & white. The input is a color bitmap that uses RGB color space, where each pixel has separate red, green, and blue components that are combined. You will have to convert the RGB image into a gray-scale then to a black/white image.

2 What you will learn

In this assignment you will practice the following skills:

1. Dynamic memory allocation of single, double and triple dimension arrays.
2. Using a struct and a class.
3. Finishing an implementation of a queue (queue).
4. Implementing the Breadth-First-Search algorithm.
5. Connected component labeling.

3 Data Format

You will not know the size of the images until runtime, when you read the image file. Thus we will need to dynamically allocate an array to hold the binary and gray scale images. Remember that a single call to new can only allocate a 1D array. You will need to allocate some 2D and 3D arrays in this assignment.

The way to allocate a 2D array in C++ is: use new[] once to allocate a 1D array of pointers, then using a loop containing new[] to allocate many 1D arrays whose locations are stored in the array of pointers.

The way to allocate a 3D array in C++ is: use new[] once to allocate a 1D array of pointers to pointers, then using a loop containing new[] to allocate many 1D arrays of pointers whose locations are stored in the array of pointers. After that, you need another loop containing new[] to allocate many 1D arrays whose locations are stored in the array of pointers. Remember that you need to deallocate everything that you allocate.

This means that every call to new[] needs a matching call to delete[]. Otherwise your program will have a memory leak.

Your program reads an RGB color bitmap image in a way similar to PA3. However, we modified the BMP library a little bit to make it work with dynamically allocated arrays. This means we can support images with (almost) arbitrary size.

```
unsigned char ***readRGBBMP(const char filename[],int *height,int *width);
```

readRGBBMP takes the filename, a pointer to two integer and returns a pointer to a 3D array. You should provide the function two pointers to integer in which it will place the height and the width of the image.

Once read, the image is stored 3D array with dimensions [height][width][3].

Your program should convert an RGB image into a gray-scale image. Grey-scale images are similar to RGB except that each pixel has only one component. In other word, each pixel is represented by one 8-bit unsigned integer. The image is stored as a 2D array with dimension [height][width].

Further, your program should convert a gray-scale image into a binary one, where each pixel is either black or white. Black pixels are considered background and are ignored. If a pixel is greater than a certain *threshold* then it should be white, otherwise it is black. Similar toe the gray-scale counterpart, the binary image is represented as 2D array of character with dimension [height][width]. Each pixel holds one of the two values 0 or 1.

4 Connected-component labeling

Connected-component labeling aims to label different connected-component - segments or objects - of a binary image. A simple and fast method to implement connected is based on a graph traversal method. Once the first pixel of a connected-component is found, all connected pixels of that connected component are labeled before going to the next pixel. The connected pixels are found after performing a graph search, *breadth-first* search in our case. Before going into the algorithm details we are going to explain necessary stuff.

4.1 Gray-scale

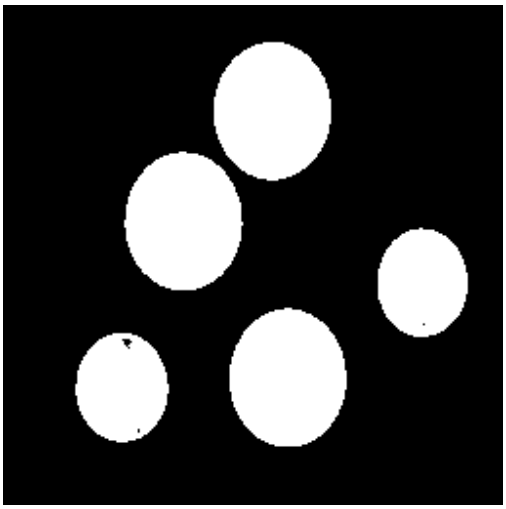
Gray-scale are composed exclusively of shades of gray (S), varying from black at the weakest, see figure 2a. In an RGB image, each pixel is represented by three components red (R), Green (G), Blue (B) ,see figure 2b.

To convert an RGB image to a gray-scale one, you have to transform each pixel independently. The three components of an RGB image Each pixel are transformed into "Shade of Gray" using following formula:

$$S = 0.2989 * R + 0.5870 * G + 0.1140 * B$$

(a) RGB Image

(b) Grayscale Image



(c) Binary Image

Figure 2: Different image representations

4.2 Binary Image

Each pixel in a binary image is represented by a binary component. The binary component takes one value, either zero or one. One corresponds to white, while zero corresponds to black.

To convert a gry-scale image to binary image you have to transform each pixel independently. If the pixel value is greater than a certain threshold - you can use the constant *THRESHOLD* - then set it to one, otherwise to zero, see figure 2c.

4.3 Breadth-First Search

Breadth First Search is a general technique with many uses including flood fill, shortest paths, and meet-in-the-middle search. The idea is to explore every possible valid location, beginning at the start location, using an ordering so that we always explore ALL locations at a shorter distance from the start before exploring any location at a longer distance from the start. In other words, first we "explore" the start, then all locations at distance 1 from the start, then all locations at distance 2 from the start, etc, in that order.

As we search we mark cells that we've explored so that we don't explore them again and so the search does not run forever.

How do we ensure that we are going breadth not depth first? We keep a queue of locations in the binary image. A queue has the property that the first item added is the first to be removed (first-in, first-out, a.k.a. FIFO). We can easily implement this by always adding new values to the end of a list (i.e. tail of the queue) but remove from the front (i.e. head of the queue). A queue mimics a line of people waiting. New arrivals enter at the back and leave from the front when they are served. The longest/oldest item will be at the front of the queue and the newest at the back.

In our pixel search the queue is initially empty and we begin by putting the start location into it. In each iteration, we remove the oldest remaining location from the queue, and we add all of its new neighbors to the end of the list. This simple algorithm successfully implements the BFS. We keep the BFS going as long as possible, until the queue is exhausted. When this happens, we have identified all pixels in a connected-component. Then we perform BFS on the next unlabeled pixel.

4.4 Queue

The binary image in this assignment is a 2D grid. Therefore, each location in the image is identified by two numbers: a row position and a column position. The approach that we will use for finding connected component will involve keeping track of a list of these locations. It will be extremely convenient to be able to talk about a combined structure that contains both a number for the row and a number for the column; we can accomplish this by creating a simple new data type:

Listing 1: Location struct

```
struct Location { // define a new data type
    int row;
    int col;
};
```

A Location object is simply a package with two numbers that can be manipulated using a period . (the member operator); here is an artificial example.

Listing 2: Struct usage example

```
Location start;
start.row = 3;
start.col = 5;
Location one_below = start; // make a copy
one_below.row += 1;
cout << start.row << "," << start.col << endl; // 3 5
cout << one_below.row << "," << one_below.col << endl; // 4 5
```

This will be convenient because we can have an array (or queue) of Locations, have functions that take a Location as input or output, etc.

Moreover, you will be required to complete the definition of another new data type, the Queue class that will be useful in implementing the search. The Queue class will store the list of Locations waiting to be searched.

The Queue class should support the following operations:

1. create an empty Queue (but with a given maximum capacity of how many Locations it can store)
2. add a new Location to the back of the Queue

3. Remove the oldest Location from the front of the Queue

4. check if the Queue is empty

When you are writing your higher level BFS algorithm, having this class should make your life easier. Everytime you find a new, unexplored Location you will call the Queue's push(Location) function. Every time you want to get the next location to explore from you will call the Queue's pop() function. Internally the Queue class should just create an array to hold the maximum number of Locations that could ever be entered into the queue and use integer index variables to remember where it should place new values (i.e. where the back is located) and where it should remove old values (i.e. where the front is located).

In this assignment you are required to complete the implementation of the queue class.

Remember that our Queue class should support the following operations:

Here is an example of how it should behave.

Listing 3: Queue Example

```
// create some locations;
Location three_one, two_two;
three_one.row = 3; three_one.col = 1;
two_two.row = 2; two_two.col = 2;

// create an queue with max capacity 5
Queue q(5);

cout << boolalpha;
cout << q.is_empty() << endl;            // true
q.add_to_back(three_one);
cout << q.is_empty() << endl;            // false
q.push(two_two);

Location loc = q.remove_from_front();
cout << loc.row << "," << loc.col << endl; // 3 1
loc = q.pop();
cout << loc.row << "," << loc.col << endl; // 2 2
cout << q.is_empty() << endl;            // true
```

As you can see, the queue gave us back the oldest location (the pop() added earliest) first.

We will give you an implementation of this class that is almost complete. It is based on the idea of using a long array/pointer called contents that holds Locations, as well as two counters:

- tail counts the number of **push** calls so far. Equivalently, its value is the next unused index in the contents array.
- head counts the number of **pop** calls so far. Equivalently, its value is the oldest index that has not yet been extracted.

For instance, this is what the internal variables of the Queue should look like for the example above, after both locations are added:

```
tail: 2
head: 0
contents[0]: (3, 1)
contents[1]: (2, 2)
contents[2..4]: garbage
```

After that, when we make the first call to **pop**, it should both increase head to 1 and return the Location (3, 1).

After that, when we make the second call to **pop**, it should increase head to 2 and return the Location (2, 2).

After that, because head and tail are now equal, the Queue knows it is empty.

Note: when you delete from the queue, you do NOT move all the other items. You simply move the head counter forwards (leaving the old stuff sitting in its original location). Actually, the expense of moving items would make your program much slower. Later in the course you will learn how to optimize this.

Here are some additional notes about the Queue class.

1. the Queue constructor takes an integer max_size. For our BFS application, you should pass rows*cols as this maximum size, since that it the maximum number of locations that our Queue could be used to explore.
2. the constructor syntax is Queue q(5);
3. C++ calls the destructor automatically. So while you need to write the code for the destructor you do not call it yourself, meaning we will never call the function ∼Queue() anywhere.

4.5 Connected-component labeling algorithm

In this section, we will explain the core of the programming assignment, which is the connect component labeling algorithm. The algorithm works as following:

1. Set current_label to 1.
2. Look at the next pixel in the image. (Start at pixel 0,0, then 0,1... and so on)
3. If the pixel is white - not background - and is not labeled, then give it the label 'current_label', and push it into your BFS queue. If the pixel is black (the background) repeat step (2) with the next pixel.
4. Now perform BFS.
   (a) Pop out an element from the queue and look at it's neighbors. If a neighbor is a white pixel and is not already labeled, give it current_label, and push it onto the queue.
   (b) Repeat until no more locations in the queue.
   (c) Increment current_label by 1 then go to (2) for the next pixel in the image.

After identifying the different components, assign a random color to each component.

5 Implementation

1. We provided skeleton code in Vocareum.
2. The overall program will be called **label** and implement four different operations described above. Your program will take command line arguments to control which operation is performed.
3. You should complete the queue implementation. Once you finished you can test it by running: ./label test_queue This should output

```
true
false
3 1
2 2
true
```

4. You should implement the following function:

```
void rgb2gray(unsigned char ***in, unsigned char **out, int height, int width);
```

The function takes a pointer to 3D array, which contains the image. A pointer to a 2D array to which the gray-scale version of the image will be placed. Also, it take the height and the width of the array. Once you finished you can test it by running

```
./label gray coins.bmp coins_gray.bmp.
```

You should see a gray scale version of the original image.

5. You should implement the following function:

   ```
   void gray2binary(unsigned char **in, unsigned char **out, int height, int width);
   ```

   Once you finished you can test it by running

   ```
   ./label binary coins.bmp coins_binary.bmp
   ```

6. You should implement the following function:

   ```
   int component_labeling(unsigned char **binary_image, int **labeled_image, int height, int width);
   ```

   the function should implement the strong-component labeling algorithm. The input is a binary image, and the output is a 2D array with labels. The function returns the number of segements as well.

7. You should implement the following function:

   ```
   void label2RGB(int **labeled_image, unsigned char ***rgb_image, int num_segment, int height, int width);
   ```

   This function takes a labeled image and generate an RGB image. Each label should be colored differently. You can assign random colors, though they should be distinct from each other so you can tell that your algorithm is working.

   Once you finished you can test it by running

   ```
   ./label segment coins.bmp output.bmp.
   ```

   You should see an image like in figure 1

6 Notes

- You should understand the differences between the data structures we are using in this assignment.
  - We have several images: the original RGB, the grey scale, the binary image and finally the RGB output image. All of these will be size `width x height`.
  - We have a number of Location objects that are instantiated for the purposes of the running the BFS algorithm. When the algorithm asks you to push a location on the queue, you are pushing a Location object. The `.row` and `.col` values correspond to the pixel being examined in the binary image.
  - We have a `width x height` integer array used for labeling. This array should be initialized to 0, meaning the corresponding pixel in the binary image has not been labeled. When you are asked to check if a pixel is labeled, or to label a pixel, you set the value in this array.
- Thinking about the over all structure of the labeling algorithm, you will have a main outer loop that examines all of the pixels in the image. Inside that loop you will run BFS every time you find an unlabeled foreground pixel. It is not required, but you may want to make your BFS algorithm a function.
- The `main()` implementation we give you reuses the input array for the final output image.

References

[1] label-wiki. `https://en.wikipedia.org/wiki/Connected-component_labeling`.