**CSE 3353**

Assigned: Wednesday April 18, 2018

Due: Wednesday May 2, 2018 at 11:59pm

**Project 3 - Images and Process Scheduling**

This project will focus on two new concepts we have not discussed in class: image manipulation and process scheduling.

**Part 1: Image Manipulation**

Image and object recognition algorithms are some of the most complex examples of machine learning currently being developed. Future classes may have you dive into some of those algorithms, but for this class we’re interested in some common first steps: some basic manipulation and grouping pixels together by connected components.

An image can be imagined as a 2D grid of RGBA values (red, green, blue, and alpha). The combination of RGB determines the color of a pixel, while the alpha determines the opacity (an alpha of 1 is opaque, while an alpha of 0 is transparent).

**1a) Grayscale**

To get you started with image manipulation problems, you will start by converting a color image to grayscale. First, install and read the documentation for the node package jimp (<https://github.com/oliver-moran/jimp>). You will use this package to load images from disk.

Do some research on how to grayscale an image. In code comments, explain how grayscaling works (this will be short). Then, implement the algorithm manually. **NOTE: You cannot use the built in grayscale function in the jimp library.** You are limited to loading the image from disk, the pixel functions in the Low-Level Manipulation section (<https://github.com/oliver-moran/jimp#low-level-manipulation>) and the functions in the image creation section (<https://github.com/oliver-moran/jimp#creating-new-images>).

The input to this function should be an image of your choosing, and the output should be the same image in grayscale.

Unlike the node.js library, there are no synchronous versions of certain jimp functions. Either callbacks or promises must be used to handle reading from/writing to image files.

**1b) Image blurring and sharpening**

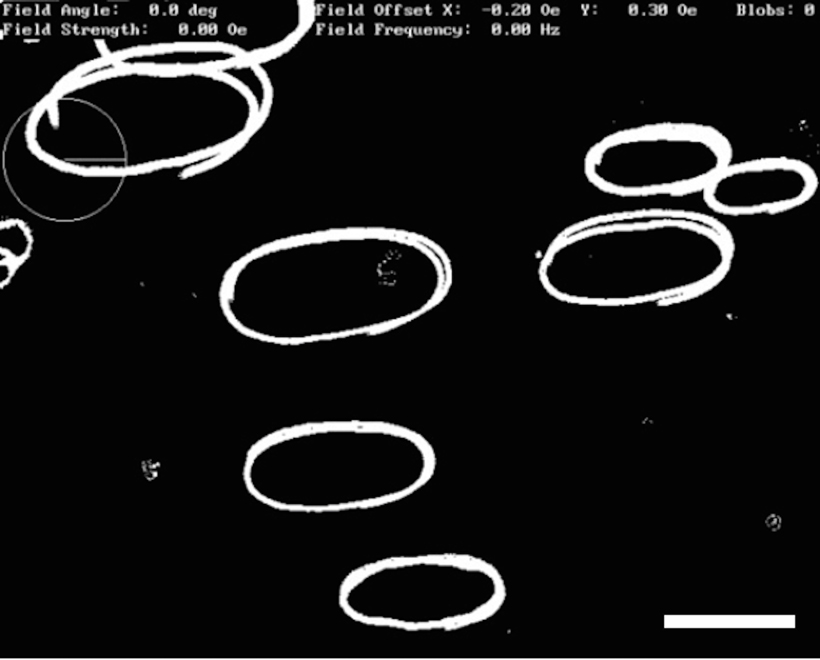
While grayscaling operates on a per-pixel basis, image blurring and sharpening operates on groups of pixels at a time. Do some research on basic blurring and sharpening algorithms. In code comments or in a separate document, discuss how these algorithms work. Your write-up should include descriptions on what “convolution” and “kernels” are, as well as the kernels you use for blurring and sharpening.

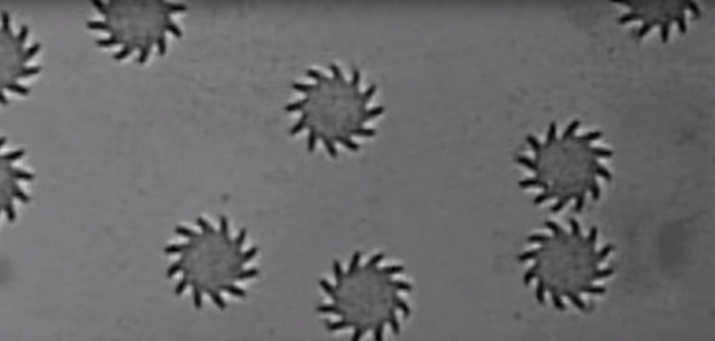
Implement the image blurring and sharpening algorithms. Again, you cannot use the built in image processing functions in jimp. The input to the blurring algorithm should be an image of your choosing, and the output should be a file with the blurred image. The input to the sharpening algorithm should be the blurred image file, and the output should be a file with the sharpened image. Observe how different the sharpened image is from the original.

**1c) Connected components**

The first fundamental step to object recognition in images is to determine what pixels are “related” to one another. This is not a trivial problem to solve, especially with full color images with many shapes. So for this problem I will be giving you some guiding images.

First, observe the following images of bacteria:





You’ll notice that there are distinct shapes, or connected components. As you can tell, components have *similar* RGB values, but not necessarily the same. So for your first step, you will take an image and label each pixel as a *foreground* (white) or *background* (black) pixel. How you determine foreground and background pixels is up to you, but you should explain how you ended up solving the problem. Output this image to a file. Note that in some cases it may help to sharpen the image first before classifying to make the edges clearer.

From there, take your modified image and run an algorithm that determines connected components of foreground pixels. Each unique connected component should be colored a random color, such that the resulting image is a bunch of colorful shapes with a black background. Just like in the previous parts, describe the algorithm in detail. Provide an in-depth Big-O analysis of this algorithm.

The inputs to your connected component functions should be a grayscale image of basic, clear patterns. Don’t go overboard in complexity here: you’ll note that the two sample images are mainly composed of two distinct colors with clearly defined edges and shapes. The images can be of your choosing, but “grayscale bacteria” is a good starting point.

**Part 2) Process Scheduling in an OS**

An operating system manages an arbitrary number of processes. Some of these processes run and terminate, while others live on as long as there is power (I’m looking at you, Google Chrome). In addition, some processes have output that act as input to other processes. Take for example the following set of bash commands, which outputs the number of unique lines of text in a file:

cat file.txt | sort -u | wc -l

The command cat is a process. The output of cat gets “piped” into sort, which sorts the file contents. The -u flag keeps only unique lines. The output of sort gets “piped” into wc, which gives the word count of some text. The -l flag means it will output the line count instead.

As you can see, there are three commands that are connected by one another. With an arbitrarily large number of commands, there can be a lot of interdependencies between process inputs and outputs, and as an OS scheduler you want to be sure to process things in the correct order.

We can think of a set of processes as a directed acyclic graph (DAG). Each process is a vertex Pj, while an edge (Pj*,* Pk) outlines a dependency. Specifically, the output of Pj acts as the input of Pk, and thus Pj must be scheduled first. Suppose we have a dependency graph that looks like this:



Observe that in order to run through all five processes, we have to go through the following order: C → A → D → B → E.

For this problem, run through the following requirements:

1. Research a solution to the problem. The solution should provide a valid ordering of processes in O(|V| + |E|) time. Provide the name of the algorithm, how it works, and an in-depth Big-O analysis that leads to the solution previously mentioned.
2. Define an adjacency list data structure.
3. Manually encode the graph in the previous page to its adjacency list form.
4. Implement the algorithm you outlined in part 1, and output the results.
5. Encode the graph below to its adjacency list form, and re-run the algorithm to give a valid process ordering.



**What to submit:**

As you can see, this project leaves a lot of details out with regards to file names, inputs, outputs, etc. There are several components you need to submit: all code for the various parts, all inputs, all outputs, and a report with algorithmic explanations where appropriate. In your report, you **must** include a description on how to run the various parts. I should be able to copy a set of commands into terminal and run them, and see valid output (you can assume I’ll be in the right folder and that any required dependencies will be installed). Take care that your report is both readable and presentable.

As much as possible, the submission should be something that you could present to a client at a software company. Clear descriptions, clear code, and clear documentation is key. I will be asking you follow up questions about this project during your final technical interview.