CS51 Final Project: Photographing the John Harvard Statue

Final Specification

Amy Kang, Anna Liu, Haoqing Wang, Cecilia Zhou

amykang@college.harvard.edu, annaliu@college.harvard.edu
haoqingwang@college.harvard.edu, zilingzhou@college.harvard.edu

April 17, 2015

I. Overview

Motivation for our project arose from the difficulty of acquiring quality photographs of popular tourist attractions. Heavy traffic, whether composed of tourists, locals, or pigeons, seem to ruin every picture taken. Our solution to this problem is to take multiple images of the same location, each with its own noise. Then, given this input of images with tourists, or "noise," we will remove the "noise" to produce the desired image, sans traffic.

We will accomplish this first by reading the RGB color values at the same position within each image. We will then take the median of these RGB values through median filtering and use that mean to produce the pixel in that position within the final image. Given that there are more images in which there is no noise covering this pixel than those in which there is, the median should return the RGB values of the target subject, not the noise. We will find the median through a deterministic algorithm for finding a good pivot point, and the runtime would be O(n). We would also implement median filtering through counting sort, which is especially effective for fixed maximum values. Because RGB values are limited to the range 0 to 255, our data is a good candidate for counting sort. We will also implement a randomized median finding algorithm using the idea of QuickSelect, which also has an expected runtime of O(n).

Given the case where median fails, we will move to a mode-finding algorithm to find the background color value.

II. Modularization and Signatures

Our project can be modularized into the following parts. The use of different algorithms to find the median also lends itself well to task division as each team member can implement a different algorithm. Furthermore, our project requires preliminary and following image processing, which can also be divided. We intend to divide our code into the following parts in order to both reduce the complexity of our main.py file and to improve the flexibility of our code. The implementation of functions within these classes and modules will remain hidden. Because formulized interfaces and signatures for classes and modules are not commonly used in Python, we have instead included a description of the functionality that each class or module should provide.

Classes

Image:
We will implement this class using the Python Image Library (PIL). Using this library, we'll be able to create image objects with functionality including conversion to RGB image format, retrieving the RGB values of a given pixel, retrieving RGB values of the next pixel, conversion between other image file types, writing image files, and various tester functions. Each image object will have instance variables .img_rgb, which will return the image object of type 'RGB,' height and width of image, and the methods get_RGB_value and get_next_RGB, each of which returns a tuple containing the red, green, and blue values of a pixel at a given position.

Anna Liu 5/1/15 4:38 PM
Comment [1]: Looking back, this was a good idea because by wrapping Image functions inside another class, we can change the way we implement getting images and reading and writing their RGB values without changing our main.py

Anna Liu 5/1/15 4:38 PM
Comment [2]: We also decided to call this picture.py to distinguish it from Python PIL's Image class

Modules

Median:
We'll be writing a module called Median that has several functions that each implement an algorithm that we want to test. We'll have functions for a randomized algorithm based on QuickSort, a deterministic algorithm to find a good pivot point, and a counting sort algorithm. This will all be writing in one python file, and we'll import the median.py into our main file. We will hide the implementations of median so that we can easily switch between our different methods. Further specification on each for these algorithms can be found in section III.

Mode:
For the case in which median fails to produce a sufficiently similar image to the target image, we will write a module called Mode that contains a function to find the mode of an array of RGB values. The implementation of mode will also be hidden. It will be written in a single file, mode.py, which will be imported into our main control file that can easily switch between using mode and median to find the desired pixels.

Anna Liu 5/1/15 4:40 PM
Comment [3]: After running our median algorithms on our pictures, we realized that we didn't need the mode. Mode would actually not work well for this purpose at all because there could be slight variations in lighting and color in each of the photos, which would result in either not being able to find a mode or finding a mode not close to the actual reference image's value.

Similarity:

We'll write a module to implement a way to gauge the similarities of two documents, and in our case, the two images, one produced by our method and a control image of a noiseless target image. We will also hide our implementation of similarity in order to be able to explore multiple methods of comparing images and test each of them. We're still investigating different algorithms to gauge image similarity, but we believe that the post linked below will be helpful once we start implementing this algorithm. http://stackoverflow.com/questions/843972/image-comparison-fast-algorithm

Preconditions

Images can be in any format, but all images in a single set should be of the same format. This is so that no changes to resolution or image size occur across conversions between different image file formats. Each image in a set should have the same resolution, so that comparison can be done pixel by pixel. That each pixel corresponds to the correct pixel in each image is critical in producing the correct image.

II. Modules and Code

Image:
Below is the partial implementation of our image class, found in image.py:

```
from PIL import Image

class image(object):

        #initialize image object
```

```
def __init__(self, filename):
        self.img = Image.open(filename)
        self.rgb_img = img.convert('RGB')
        self.width, self.height = self.img.size

#pos should always be a tuple of two integers
def get_RGB_value(self, pos):
        return self.rgb_img.getpixel(pos)

def get_next_RGB(self, pos):
        (x, y) = pos
        if x >= self.width:
                return self.get_RGB_value(self, (1, y+1))
        else:
                return self.get_RGB_value(self, (x+1, y))
```

Median:

The first algorithm is a randomized recursive algorithm based on QuickSelect. We randomly choose a pivot in the array of size n, and then compare all of the elements in the array with the pivot to create two arrays - one of all the elements larger than the pivot, and one of all the elements smaller than the pivot. If there are more than n/2 elements that are less than the pivot, then we recursively call our algorithm on the those elements smaller than the pivot. If there are more than n/2 elements that are more than the pivot, we recursively call our algorithm on the the elements larger than the pivot. Otherwise, we have found our median. The probability that the pivot decreases the size of the array we are calling our algorithm on by a factor of ¾ is ½, so the expected number of times we call our algorithm before the array decreases by a factor of ¾ is 2. Thus the expected runtime is linear. The implementation of this algorithm is below:

```
import random

lst = list(map(int, raw_input().split()))
#median, or if even number of items in left,
#the item to the left of the median
def quick_select(A, k):
  lesser = []
  greater = []
  pivot = random.choice(A)
  for i in A:
    if i < pivot:
      lesser.append(i)
    else:
      greater.append(i)
  (a, a1, a2) = (len(A), len(lesser), len(greater))
  if k < a1:
```

```
        return quick_select(lesser, k)
    elif k > a - a2:
        return quick_select(greater, k - (a - a2))
    else:
        return pivot

def median_1 (C):
    print quick_select (C, len(C)/2)

median_1 (lst)
```

The second algorithm is a deterministic recursive algorithm that finds a pivot that decreases the size of our array by a constant factor each time we call it. We break up our array into arrays of size 5. We use Insertion Sort on each of these arrays to find the median of each, and then recursively call our median finding algorithm on this array of medians to eventually find m which we will use as our pivot. We then do the same thing as in our first algorithm and split up the array based on our pivot. We similarly decrease the size of the array by a constant factor, so the runtime is linear.

The third algorithm is counting sort. Counting sort is a sorting algorithm that works in linear time, breaking the linearithmic lower bound that exists for comparison sorts. This is possible as counting sort works with the assumption that the numbers are small integers; in this case, we have RGB tuples in the form (a, b, c) with each of a, b, c bounded by 256. Thus we have a clear mapping from these tuples to integers of reasonable size. Heuristically, counting sort works by creating a histogram of the array to be sorted and then using the histogram to find the correct positions of each of the elements in the array. As our goal is to find the median, we do not need to continue sorting the array after the histogram has been created, as we can find the median of the array from the histogram.

III. Timeline

This timeline consists of our weekly goals over the next two weeks. The order of our tasks for each week roughly represent the process flow of our project.

4/17 - 4/24

- Conversion of JPG images to PPM format
- Read and parse RGB values from the PPM on a per pixel basis
- Collect RGB values of the same pixel location from each image into a single array

Anna Liu 5/1/15 4:26 PM
Comment [4]: We didn't need to convert the images into PPM format because Python's PIL Image class allows us to read images in JPG and PNG format directly

Anna Liu 5/1/15 4:29 PM
Comment [5]: Instead, we created a Picture class that wrapped functions from the Image class. We read our .png files as instances of the Picture class and got RGB values from each of the picture instances.

- Implement key algorithms:
  - Find the median RGB value of this array using one of our three algorithms:
    - Randomized algorithm based on QuickSelect
- Store each median RGB value into an array
- Generate a new, "noiseless" PPM file from the median RGB values
- Convert PPM to viewable image file

4/24 - 5/1

- Implement the two other median-finding algorithms:
  - Deterministic algorithm to find a good pivot point
  - Counting sort
- Extensions:
  - Improve convenience and accuracy by generating the noiseless photo from a video of the subject rather than multiple photos.
    - Develop a method to collect and store frames (still images) from a video
    - Implement our image cleaning algorithm above on these frames to generate a single "noiseless" image
  - Implement image similarity algorithms (shingling, keypoint matching, histogram)
  - Image similarity using random permutations
    - Use shingling to hash pieces of the image (ie every 16 or 32 pixels), (overlapping by 8 or 16 pixels)
    - Store array of these hashes and take the minimum of a random permutation applied to each shingle.
    - Repeat this 100 times, and compare with the mins for the other image
    - If more than 90 of these are the same, then we consider the two images sufficiently similar
  - Figure out front-end so we can show product images to the user and develop a user-friendly interface to upload a set of images

Anna Liu 5/1/15 4:30 PM
Comment [6]: We didn't need to convert the image file to PPM. Instead, we just saved it as a .png file directly from the picture class instance.

Anna Liu 5/1/15 4:31 PM
Comment [7]: Done using OpenCV's VideoCapture functionality

Anna Liu 5/1/15 4:33 PM
Comment [8]: Keypoint matching seemed like a good idea, but we realized that it only detected key objects in the image. Most of the pictures we generated would have the same key features, but it wouldn't be a good comparison of how blurry our output file is or how close it was to the reference frame.

Anna Liu 5/1/15 4:23 PM
Comment [9]: In doing more research, we decided to use OpenCV's functionalities that creates histograms and checks for similarity using correlation and chi-squared functions

Anna Liu 5/1/15 4:25 PM
Comment [10]: We realized that w-shingling only works for text-based documents and would not work for our images, so we didn't implement it

Anna Liu 5/1/15 4:25 PM
Comment [11]: We did the user interface in the shell instead because it was easier to get user input directly. If we had more time, we would have implemented a front-end for the users to submit their own images.

Anna Liu 5/1/15 4:45 PM
Comment [12]: After trying out iPython Notebook, we realized that it didn't serve our purposes as well as directly running our scripts through our shell. We stopped using it altogether and just compiled it on our own.

IV. Progress Report

We've created a git repository on GitHub and added all of our group members as collaborators. We've also set up iPython Notebook and we plan to use that to run our code and check our image outputs. We intend to work in separate .py files in order to better modularize our code, but we will manage these modules within cells as we test in iPython Notebook. We've created an image class (image.py) and a median module (median.py) and are in the process of adding implementation to both. We've completed preliminary image

processing, such as creating an initializer for an image object, retrieving rgb values, and getting the next rgb value. We've also completed the implementation of the QuickSelect algorithm for finding the median of an array. Our preliminary work on these implementations can be found in section III, Modules and Code.