CS51 Final Project: Photographing the John Harvard Statue

Draft 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 10, 2015

I. Brief 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 through median filtering and use that mean to produce the pixel in that position within the final image. 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. Feature List:

We have divided our goals into two sections: core functionality and extensions. The features listed under core functionality are our own minimum requirements. Time allowing, we would like to extend our project to those features listed under extensions.

Core functionality:

- 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
- Find the median RGB value of this array using the following algorithms:
  - Randomized algorithm based on QuickSelect
  - Deterministic algorithm to find a good pivot point
  - Counting Sort
- 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

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
- 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
- Other implementations to check image similarity
- Blurring
  - Use mean-finding (or median-finding) algorithm to find the average RGB value within a neighborhood of pixels
  - Create an image replacing the RGB value of each pixel within the neighborhood with the mean value (or median-finding)
- Panorama

Anna Liu 5/1/15 4:17 PM
Comment [1]: We ended up not converting images to PPM format because Python's Image library allowed us to access RGB values without conversion

Anna Liu 5/1/15 4:19 PM
Comment [2]: This is the Median of Fives Algorithm

Anna Liu 5/1/15 4:19 PM
Comment [3]: We generated a PNG image directly, instead of converting to an PPM image

Anna Liu 5/1/15 4:20 PM
Comment [4]: We were successful in turning a video into still photo frames on which we could run our algorithm

Anna Liu 5/1/15 4:21 PM
Comment [5]: w-shingling is only effective in text based documents but would not work for images, so we decided not to implement it

Anna Liu 5/1/15 4:24 PM
Comment [6]: We didn't have enough time to implement additional extensions like blurring or looking at panoramas

III. Technical Specification

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. The following points represent our process flow in order of operation. Each section also loosely represents a single interface.

Image Processing

Images will be manipulated using existing code libraries, as we would rather work with an array of RGB values than with the original image file. For example, in Python, this will be done by using the load() method from the Python Image Library, which returns an array giving RGBA values for a given image. In general, we will ignore the alpha channel portion of the RGBA values, which specifies opacity, focusing on the RGB values instead.

Median Finding

We plan to implement three different algorithms to find medians. Since we have to process each pixel of the photos, we want our median finding algorithm to be as efficient as possible.

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 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.

Converting to Viewable Format

Now that we have iterated over all of the pixels in the image and found the medians, we want to convert the array of RGB values that we have calculated to a viewable format. This will be done using existing libraries to deal with images, most likely the same one we use to convert the image to an array of RGB values in the first place. In Python, this would be the Python Image Library.

Image Similarity

Now we want to compare the images generated by the median finding algorithms with a "clean" image that is free of tourists, or "noise". We can check this by eye but want to be more rigorous, so we will do this using an image similarity algorithm using random permutations. We will use shingling to hash pieces of the image (ie every 16 or 32 pixels), starting every 8 or 16 pixels, and store an array of these hashes. We then take the min of a random permutation applied to each shingle. Repeat this 100 times, and compare these mins with the mins generated from the other image. If more than 90 of these are the same, then we consider the two images similar, otherwise we consider them different.

Further Image Processing

We can apply our median finding algorithms to blur an image. We find the median RGB value within a neighborhood of pixels and then create an image by replacing the RGB value of each pixel within the neighborhood with the median value.

We can form panoramas by applying our median finding algorithms on the borders of images only, and appending them to each other.

Potential Difficulties

We might have difficulty dynamically reading in images, since we are not familiar with the Python Imaging Library yet, and it might not have the functionality that we want.

It might take a long time to iterate over every pixel in the image, so we might need to find a more efficient way to process the images.

IV. Next Steps

Developing a Framework

We have yet to decide which language and tools are best suited to our needs. We are currently considering using Python along with the Python Imaging Library (PIL). Before submitting our final technical specification next week, we intend to investigate other libraries and frameworks, including NumPy, a matrix manipulation library for Python. In order to decide which framework to use, we will go into Harvard Yard and Harvard Square to take the photos that we'll use to test our implementation. We will use these photos to test the ease and efficiency with which we can import and process images. Issues that we may encounter include those related to environment set up and library installation. We also intend to investigate different image file types as well as methods of converting from JPG, BMP, GIF, and PNG to PPM (Portable Pixel Map) types.