Perceptual Hashing Image Similarity Tool (p.h.i.s.t.)

Dennis Devey
m171458@usna.edu

February 28 2016

1 Introduction

Enclosed is a brief description of the data structure and hashing mechanism p.h.i.s.t. uses to provide constant time lookups as outlined in the slide deck. The tool utilizes sets of both average hashes and discrete cosine transform hashes to provide a robust ability to identify matching images. I only demonstrate the average hash functions in this paper for simplicity's sake, though the fundamental concept operates the same way for both.

2 Staggered Hash Table

The below function contains pseudocode for constant time average hash lookup. The DCT lookup is not shown, but it is identical. My current implementation runs the two in succession, and is very accurate.

```
def checkHashes(imgHashes):  # Takes in array of hashes [64_Byte, 4_Byte, 16_Byte]
    if 64_Byte in hashTable64:  # Catches all resizes, compressions, filters, and
↪   minor edits
        return (dataAssociatedWithMatchingHash)

    else if 4_Byte in hashTable4:
        bucket = 4_Byte  # Set bucket to check for matches
        for Bucket_16 in bucket:  # Order n operation, check each 16 byte hash in
↪   bucket
```

I have put little work into optimizing this portion of the code because it just simply works. With 4 byte hashes in the leading hash table there are 65,536 corresponding buckets, and with 6 byte hashes there are 16,777,216. This means that n in the $O(n)$ is small enough that there should never be a bottleneck. I am sure that whatever PhotoDNA currently uses is more than suited for the task.

```
            hammingDistance = hammingFunction(16_Byte, Bucket_16)
            if hammingDistance < 3: # Quick and dirty way to check for similarity
        return (dataAssociatedWithMatchingHash)
            else:  # Does not match any hash found in bucket
                return False
    else:  # Does not match any 4 byte buckets
        return False
```

With all of that said, I am investigating some sort of K-means clustering as I feel that would greatly increase the ability of the tool to correlate images that do not necessarily share the same 4 byte hashes, but are still very similar.

3 Compression Hash Function

You will have noticed that my tool requires 6 hashes to operate, which is admittedly more than I would like. However, I feel that the space/speed and speed/accuracy tradeoff always favors a fast and reliable lookup. These 6 hashes are by far the most computationally expensive operation, so I looked for ways to increase the speed of the operation.

After several false starts, I figured out the correct method of doing this, nearly cutting the time it took to get an images hashes in third. I take a very long, very accurate hash of the image, which can be used as a pseudo-unique identifier, and then apply a series of transforms at the incredibly efficient binary array level to generate two shorter length, lower resolution copies of the initial hash.

The below function demonstrates the steps my tool takes to turn an image into three hashes of varying length for input into the data structure above.

```
def average_hash(imagePath): # Takes in an image
  # First half is forked from https://github.com/JohannesBuchner/imagehash which is
 ↪  forked from https://github.com/bunchesofdonald/photohash
  # Standard python implementation for perceptual hashing
    image = Image.open(imagePath) # Open Image w/ Pillow

    hash_size=16 # Set array size

    image = image.convert("L") # Greyscale image
    image = image.resize((hash_size, hash_size), Image.ANTIALIAS) # Resize image with
 ↪  high-quality downsampling filter
    pixels = numpy.array(image.getdata()) # Convert image to binary array
    pixels = pixels.reshape((hash_size, hash_size))  # Reshape array to correct
 ↪  dimensions
    avg = pixels.mean() # Get average of all pixels
    diff = pixels > avg # Compare average against each pixel, set each as 1 or 0
```

At this point the program has one 16x16 binary array, which we are now going to use to make an 8x8 and 4x4 array. By working off an existing array instead of processing the image two more times we greatly reduce the time it takes to hash an image.

```
    big = 16 # Size of original array
    small = 8 # Size of secondary array
    small2 = 4 # Size of tertiary array
```

This next part is the important part: the function takes the original array and then averages and resizes the array such that the resultant array is a lower resolution. For example, with a big of 4 and a small of 2:

```
[[ 0, 0, 1, 1],                          [[0, 1],
 [ 0, 1, 1, 0],      converts to          [1, 0]]
 [ 1, 1, 0, 1],
 [ 0, 1, 0, 0]]
```

As a quick note, this does not work on gradient hashes, which PhotoDNA uses. I reverted back to using average hashes because it gave me significantly better accuracy post-compression.

```
    diff2 = diff.reshape([small, big/small, small, big/small]).mean(3).mean(1)
    diff3 = diff.reshape([small2, big/small2, small2, big/small2]).mean(3).mean(1)

    diff2 = np.around(diff2) # Rounds averages to nearest whole number
    diff3 = np.around(diff3) # Rounds .5 to nearest even number

    return toHashes(diff, diff3, diff2) # toHashes takes in arrays and returns  64,
↪   4, and 16 byte hexadecimal strings
```