Project Report

By:

Thierry St-Arnaud-Simard

ID: 27649460

A Project Report

Submitted in partial fulfillment

of the requirements for COEN 352

Presented to Dr. Nawwaf Kharma

Concordia University

Report Due December 5th, 2017

Abstract

For this project, I developed a software using the Java language that can search for images similar to a user-provided image within a user-specified tolerance of deviation. To achieve this in a computationally efficient manner, the software makes use of a perceptual hashing technique to encode as much information as possible into a limited number of bits, which it then uses as a key to store into a search tree. When the user inputs an image into the program, that image is then hashed using the same algorithm as was used to encode the images forming the database. That hash is then used to retrieve all entries in the search tree within the specified tolerance.

Generally speaking, the software contains three main parts, namely, the hashing algorithm, the underlying data structure of the search tree and the image search program, which acts as the interface between the user and the other parts of the program. More specifically, the hashing algorithm makes use of the properties of the Discrete Cosine Transform (DCT) to encode as much information as possible in a limited size. The search tree is based on a bitwise trie implementation, optimized for the purpose of this program. Finally, the main program is used generate the image database by parsing a set of images, calling the other parts of the program as necessary, then take input from the user to perform the operations necessary to find images, as specified. Each parts of the program are detailed in this report.

By windowing a DCT of the image and comparing the coefficient to an average, I created a very discriminatory perceptual hashing method. The chosen hash length lets the program find matches that a human operator might not necessarily find similar, while classifying images that he would find to be similar as not being a match. This is a product of using a perceptual hash instead of more sophisticated methods such as semantics hashing, which require orders of magnitude more computational power. In comparison, the hashing and searching algorithms used in this program are memory and computationally efficient by the use of a fast DCT algorithm to hash and a maximization of cache friendly operations by using a bitwise trie to store and retrieve images. Thanks to that, they can be run on sizeable databases of images on a home computer in a reasonable amount of time.

METHODS

Image Hashing Algorithm

The image hashing algorithm used for this program makes use of the properties of the DCT to achieve a high density of stored information and increase the accuracy of the results. DCT is derived from the Discrete Fourier Transform (DFT) and is used extensively in image and audio encoding, most notably in JPEG image compression and MPEG audio and video compression. When applied to a system that consists only of real values, the DCT can store as much information in half the space compared to a DFT of the same signal. Additionally, the DCT has the property of concentrating the information into the lower frequencies of the transform, which I use in this algorithm by windowing the frequencies to the lowest components and avoiding the coefficient of the constant component.

While calculating the DCT of a signal can be computationally intensive, requiring up to $O(n^2)$ in its naïve implementation, I was able to find a Java implementation that is expected to work in $O(n \log(n))$, where n is the number of values to compute. The algorithm is the work of a freelance software developer and makes use of Fast Fourier Transform (FFT). It was found on the developer's blog[1] and I managed to integrate it in my perceptual hashing algorithm with little effort. The resulting pseudo code is shown below and represents the file named *Perceptual.java*.

```
Receives:
   - Image as an array of values
   - Length of hash
   - Empty array of bits to store the hash
-  Verify validity of inputs
   •  Length of hash must be greater or equal to 4
   •  Length of hash must be perfect square
   •  Size of image must be greater than lengh of hash
   •  Size of image must be perfect square
   •  Size of image must be power of 2
-  Perform DCT on image
   •  Call in-place Fast DCT algorithm
-  Create an array of values with same length as hash
-  Window the elements by choosing "hashlength" lowest frequencies
   •  Select values by choosing top-left elements as a square
      matrix, disregarding first element, but adding a final
      diagonal coefficient.
```

[1] https://www.nayuki.io/page/fast-discrete-cosine-transform-algorithms

- Compute the hash
    - Take the average of windowed DCT coefficients
    - For all coefficients in the windowed selection
        - If it is larger than the average
            - Set the corresponding bit in the hash

Search Tree Datastructure

The suggested type of search tree to use in our program is a Normalized B+ tree (NB+ tree). These types of trees are very efficient at accessing data in a multidimensional space because they filter out the data that is too far or too close from a "center" and compare only what is within a specified range. In our case, we would use the number of set bits in the hash as our key for storage and retrieval. All keys that had the same number of set bits (+/- the given tolerance) would be compared with the given hash to see if they were within the provided tolerance.

After some consideration, these are my conclusions regarding the use of an NB+ tree in this case. Normalizing the data filters an exponentially increasing number of invalid matches as the number of dimensions the data exists in increases. Since we are not dealing with multidimensional data, we can't get as much added value by normalizing the data. Additionally, if we encode based on an average, we would expect the number of coefficients larger (the 1's in the hash code) to be normally distributed around the half. This means that, by basing our keys for the search tree on the number of bits that are set to 1, we are, in effect, artificially constraining all the values in a few nodes around half the length of the hash code.

To avoid these issues. I initially thought of a way to store data in a fixed length tree where all leaves would be at the bottom level and branches would represent the 1's and 0's of the hash code. When we reach the leaf level, we have compared all the bits in the hash code and the leaves we find there correspond exactly to the hash we wanted and we have performed M compares, where M is the length of the hash, regardless of the number N of data stored in the structure. We can also see that this type of tree will not be balanced like an B+ tree, but unlike most trees, this does not be affect the performance negatively.

While this was implemented and worked fine, I ended up reading on the subject and found that my tree was in fact a form of trie, specifically a bitwise trie. My final implementation makes use of a common way to increase space and computing efficiency of a bitwise trie by storing common prefix in each branch to create a form of binary radix tree. The result is described in the pseudo-code on the next pages. Note that, since the image database will not need to delete entries, I haven't implemented that function. It would be reasonably simple to implement it if it was necessary to generalize the structure.

Insertion pseudo code

Receives:

- Value to store
- Hash to compare

- If tree is empty
  - Create a new leaf to store the value with hash as the prefix
- Otherwise
  - Choose the root of the tree as our first node
  - While we're not at leaf level
    - Compare prefix with corresponding part of hash
      - If there's a difference
        - We split the prefix where the difference occurs
        - We put a new branch with the higher prefix as the child of the current node
        - The new branch will have a new leaf to store value, with a prefix of hash-to-level-of-branch
        - The other child of branch will be the previous child of the current node with lower prefix
        - We're done
      - Otherwise
        - We've gone up by length of the prefix
        - If we still haven't reached leaf level, we need to branch left or right, depending on hash
  - When we reach leaf level, we have a leaf with exact same hash, so we just add a child to that leaf to store the value.

Lookup pseudo code

Receives:

- Hash code to look for
- Tolerance as the number of bits that can be different
- List to put the matches
- Node to start from (External lookups start at root)

- While we're not at leaf level
  - Compare prefix with corresponding part of hash
  - Each difference reduces the remaining tolerance by 1
  - If tolerance become negative, we're done
  - Otherwise, we went up by length of prefix
  - If we're not at leaf level, we evaluate a branch

    - If tolerance is greater than 0, we start a new lookup on the "wrong" branch with tolerance – 1
- When we reach leaf level, we add the values of all the leaves we find to the list

Main program

To achieve the specified result, we need an overarching program to use the previous parts in a meaningful manner. To this end, the main program must first create the database by reading the image file, hashing them and inserting them in the tree. Once it has gone through all files in the database, it must interact with the user by asking them for an image or a directory of images. It then uses the same algorithm used to hash the database to create a hash for the file(s) given by the user. Once the user has chosen a value for the tolerance, the program interrogates the search tree with the hash(es) and tolerance and then prints out all the matching images. The program continues to ask the user for images to search for until he or she wants to quit. Below is a general top level pseudo-code for the main program

```
Has:
   - Constant imageSize 65536 (256x256)
   - Constant hashLength = 64

- Generate database
   • For all image files in the database folder
      ○ Attempt to parse image of size imageSize
      ○ Attempt to hash to hashLength
      If any attempt fails, warn user and go to next image
      ○ Insert image file in search tree with hash
- Search for user images
   • Ask user for image location or if he wants to quit
      ○ If location is folder, list images in that folder
         ▪ If at least one image, ask user for tolerance
            ❖ For all images in the folder
               ✓ Attempt to parse image of size imageSize
               ✓ Attempt to hash to hashLength
               If any attempt fails, warn user and go to next image
               ✓ Send hash and tolerance to search tree
               ✓ Print the name of all matches

      ○ If location is a single file
         ▪ Attempt to parse image of size imageSize
         ▪ Attempt to hash to hashLength
```

If any attempt fails, warn user
    - Ask user for tolerance
    - Send hash and tolerance to search tree
    - Print the name of all matches
- If user doesn't want to quit, Search for user images again

Results

Accuracy

To settle on a length for the hash, I tested various values against the timings and number of returned images from a database of 1514 images. There were a few things I had to consider. Because of the nature of my tree and because it made extensive internal use of Java BitSets, there was no point in choosing less than 64 bits for my hash, since it is the minimum value of bits stored in a BitSet. From there, I tested the closest perfect square to the next increment of 64 bits, 121 bits, then 256 bits. Because the hashing algorithm is based on DCT, increases in the length of the hash means an increase in the window size, which means more higher frequencies are considered in the hash. Past a certain point, this means that I could put too much importance on the higher frequencies, which typically don't contribute as much to the visual representation of the image. For this reason, I decided not to go past 256 bits, even though, performance wise, the scaling seemed to not be a problem. I have included the data I accumulated while testing different hash lengths in Table 1, below.

Table 1 – Timings (averages in ns) and average number of matches at different hash length

	Hash Length (bits)	**64**	**121**	**256**
Search Tolerance	Hash time (ns)	22577453	22691516	22737946
	Insert time (ns)	19636	19951	21255
0%	Search time (ns)	15577	13873	15047
	Matches	1.004	1.004	1.004
5%	Search time (ns)	98933	344757	338178
	Matches	1.06	1.02	1.01
10%	Search time (ns)	230144	425490	481682
	Matches	2.5	1.07	1.03
20%	Search time (ns)	304931	410346	520240
	Matches	132	36.1	17.6
100% (traversal)	Search time (ns)	407849	446528	490225
	Matches	1514	1514	1514

From this data, I eventually settled on using 64 bits, because, although it only gave modest performance gains, the other sizes were too discriminatory for my purposes and it was very rare to get the search to return images other than the one that was searched for. We can see that, in the case of the 64 bits hash, there seems to be a friction point at around 10% tolerance, where we start seeing a lot more matches. In the case of 121 and 256 bits, the friction point seems to appear somewhere between 10% and 20%. Regardless, we can see that the algorithm is very efficient at filtering images that are not similar when using 10% or less tolerance.

It is worth noting at this point that, although DCT hashing is generally more accurate than block hashing, it is still a "flat" encoding, in the sense that it cannot tell similar objects or differences in perspectives to classify images as similar like a human might do. This leads to images that return a match even though they might not look similar to a human operator. There are a number of more sophisticated techniques that attempt to do this, like semantics image hashing, but requires a lot more processing power. In figures 1 and 2, below, we have examples of input images and their resulting matches.

Figure 1 – Normalized output image (left) and its normalized output image (right) with a tolerance of 5%

Figure 2 - Normalized output image (left) and its normalized output image (right) with a tolerance of 5%

Below are two examples of images and their computed hash in hexadecimal.

Figure 3 – Image with hash 4F 00 0E 80 18 00 42 00

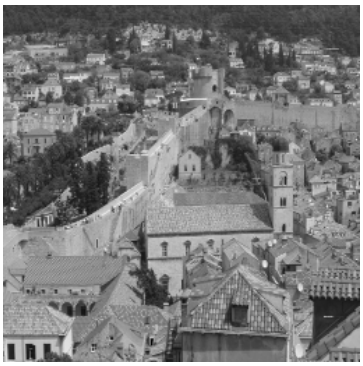

Figure 4 – Image with hash E0 7F A3 FF B9 7F AE FF

Efficiency

Comparing the efficiency of both trees is difficult without fully implementing and testing both, because the complexity of an NB+ is largely dependent on the number of elements, but the binary radix tree is almost entirely dependent on the length of the hash code. Regardless, I believe that my implementation is better suited to our scenario and will perform better in most cases. To verify this, I assembled a large amount of data while varying the size of the database. Because computing the DCT is the most expensive part of the hashing process, hashing time is almost entirely dependent on the size of the image, which we'll consider constant for our purposes. In table 2, I recorded various timings to observe the effect on search time, insertion time and traverse time as the size of the database increased. To do this, averages trim the 5% lowest and highest values to avoid erroneous timings due to the JVM curing and other hiccups like OS calls for example. This final data is compounded in a plot of the result, shown in Figure 5, where trend lines were added. Note that the Search Time shown in Figure 5 is the same as Search Time (Tol <= 20) in Table 2, which is an average of all searches.

Table 2 – Comparison of Search and insertion timing with varying number of images, all times are averages in ns

Number of images	**5**	**10**	**15**	**20**	**100**	**500**	**1000**	**1514**
Insert time	74692	49358	46613	45666	40355	25012	19161	19636
Search time (Tol = 0)	34656	48960	43505	51455	39765	30123	18394	13858
Search time (Tol = 5)	76090	97098	108277	65617	73017	48582	76351	97469
Search time (Tol = 10)	52497	42971	42004	41414	45096	81828	160415	226568
Search time (Tol = 20)	31439	38334	42826	42518	29222	122085	224401	303287
Search time (Tol <= 20)	48670	56841	59153	50251	46775	70655	119890	160296
Traversal Time	32268	38862	60534	44082	30138	147254	289427	402078

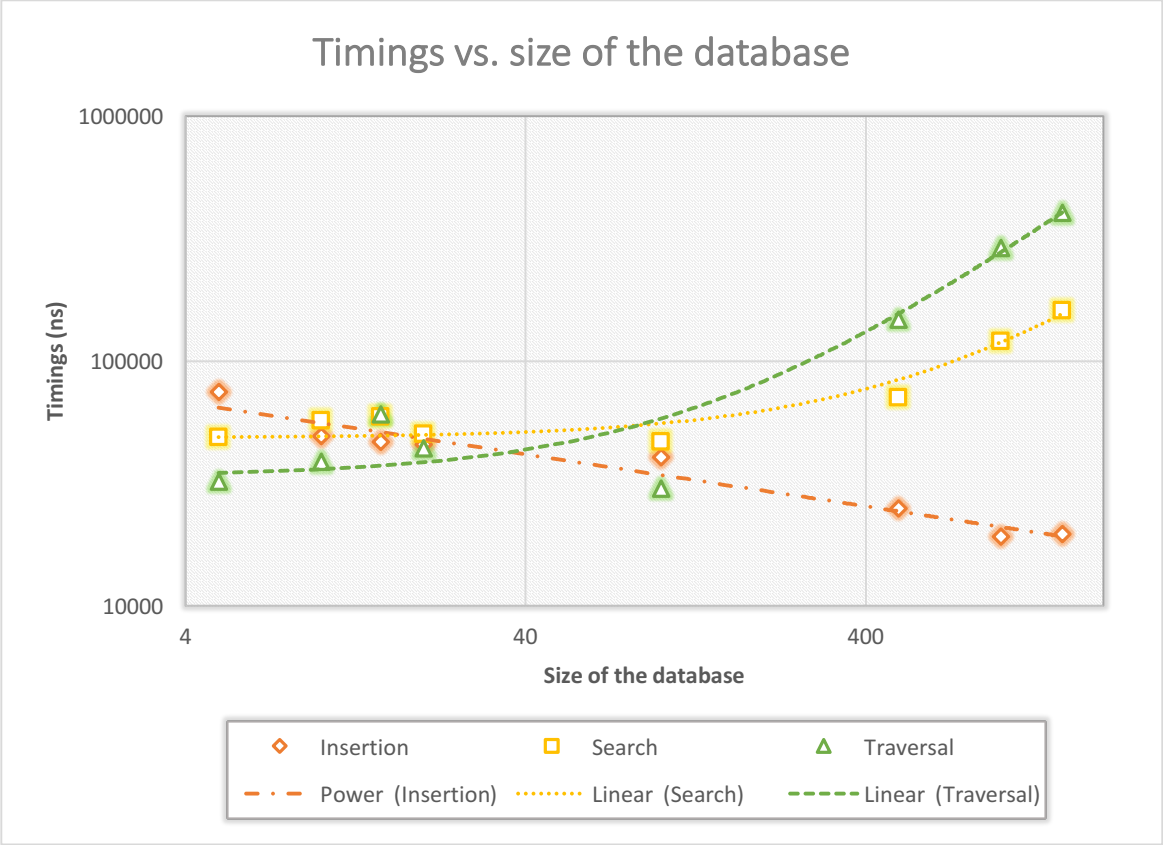

Figure 5 – Graph presenting timings compared to database size with trend lines

Conclusion

By first encoding images using DCT compression before inserting it in an efficient radix tree for storage and retrieval, I created a software that fits the design specifications. There is still much room for improving the software, the most obvious example being multi-threading. Both the hashing algorithm and data structure could easily be parallelized to make better use of current computer hardware, especially the DCT computation and the branching searches. While there exists much more sophisticated algorithm to search for images that make use of neural networks and machine learning, perceptual image hashing is a much more efficient manner of finding exact matches in large databases.