Implementation of SimHash

Junzhe Zheng A20254389

Yuchen Hong A20283450

Abstract

Most hash functions are used to separate and obscure data, so that similar data hashes to very different keys. We use hash functions to detected similarities between data. As storage capacities become larger it is increasingly difficult to organize and manage growing file systems. Consolidating or removing multiple versions of a file becomes desirable. However, de-duplication technologies do not extend well to the case where files could also be useful for classification purposes and as an aid to search. A standard technique in similarity detection is to map features of a file into some high-dimensional space, and then use distance within this space as a measure of similarity. Unfortunately, this typically involves computing the distance between all pairs of files, which leads to O(n^2) similarity detection. Our goal was to create a similarity hash function, which maps similar data to a very similar hash key.

1. Introduction

Detecting similar files and classifying documents is a well-studied problem, but typically involves complex heuristics and/or $O(n^2)$ pair-wise comparisons. Using a hash function that hashed similar file to similar values, file similarity could be determined simply by comparing pre-sorted hash key values. We purpose to use hash functions to detected similarities between data.

Our goal is to create a "similarity hash function." Typically, hash functions are designed to minimize collisions. With cryptographic hash functions, collision is nearly impossible. And identical data should map to very different keys. The SimHash function has different characteristics: very similar files should map to very similar, or even the same hash key, and distance between keys should be some measure of the difference between files. While similar files are expected to have similar sizes, there is no expectation that two files that are close in size have similar content.

SimHash does produce integer-valued hash keys; we ended up relying on auxiliary data to refine out similarity tests. The key values are based on counting the occurrences of certain binary strings within a file, and combining these sums. The auxiliary data provides an easy and efficient means of refining our similarity detection. A refined version of the keys based on file extension gives a much wider spread of key values, and alleviates some of the aforementioned problems.

We take the view that in order for two files to be similar they must share content. For example, what is the content of a file? Content could refer to the entire file, just the text portion of the file, or the semantic meaning of the text portion of the file.

Many previous attempts at file similarity detection have focused on detecting similarity on the text level. We use binary similarity as our metric. Two files are similar if only a small percentage of their raw bit patterns are different. This often fails to detect other types of similarity. For example, adding a line to the source code file might shift all line numbers within the compiled code. The two source files would be detected as similar under our metric; the compiled results would not.

In order for files to be similar under our type of metric, they must contain a large number of common pieces. SimHash operates at a very fine granularity, specifically byte or word level. We only care about the portions of the file which match out set of bit patterns.

2. Implementation

The hash key is based on counting the occurrence of certain binary strings within a file.

Since the notion of similarity is based on binary similarity, we start by counting the occurrences of various binary strings within the under processing. We preselect a set of strings, called Tags, to search for. We only use a subset of all

possible strings of a given length, as summing matches over all strings would blur file differences. SimHash program opens each file in a directory, scans through it, and looks for matches with each of the tags. When a match is found, skip counter is set and then decremented in the following bits. This prevents overlaps of matches on the same tag. The hash key is then computed as a function of the sum table entries. Once this has all been computed, the file name, path, and size, along with its key and all entries in the sum table, are stored in a database.

Mostly, two files are deemed similar if they contain a very similar number of each of out selected tags in their bit patterns. This method has several strengths and drawbacks. Because the ordering of the tag matches within a file is not accounted for, rearranging the contents of a file will, up to a point, have little impact on key values and sum tables. Consequently, small changes to a file should not throw off the similarity measure. The order of strings within files is not measured, very different files can be detected as similar if they happen to share too many bit patterns. Since key similarity is the comparison that could be theoretically performed with an $O(logn)$ search through a sorted list, an excess of false positive here means a larger number of pair-wise comparisons that mush be performed on sum table, even if those comparisons are computationally minimal.

Simhash algorithm:

1. We choose the length of simhash values, in this implementation, we use 128bit.
2. We initialize each bits of Simhash value to 0.
3. We tokenize the input document into words. Eg, This is a test. Tokens: ‘This’, ‘is’, ‘a’, ‘test’.
4. For each token, we generate a hash value with a 128bit.
5. For each bit of token hash values, if it is 1, we add 1 to the same

position of Simhash, otherwise, we subtract 1.

6. After we iterate all the tokens, we set all bits of Simhash to 1 if the bit is greater than 1, otherwise, to 0.

After computing SimHash value of SimHash function, then we proof that SimHash function is an efficient algorithm by comparing SimHash and hash function to compare the performance between SimHash and hash function. We use SimHash and hash function to two similar strings and calculate the hamming distance. Then we can see the difference between the performances.

The hamming distance between two strings of equal length is the number of positions at which the corresponding symbols are different. In another way, it measures the minimum number of substitutions required to change one string into the order, or the minimum number of errors that could have transformed one string into the other.

3. Result

As a comparison of two algorithms, we use SimHash and hash function on two similar strings and calculate the hamming distance. Here we use two strings, one is “This is google test” and another one is “For example: This is google test”.

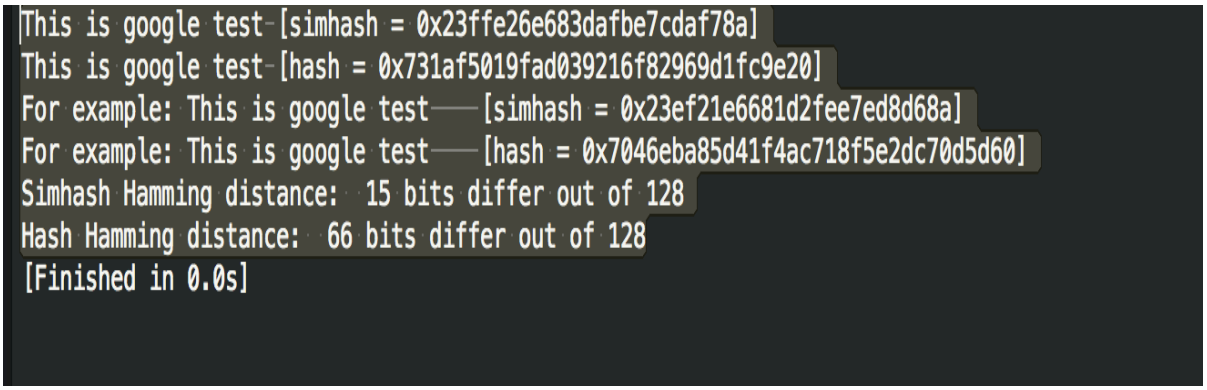

Figure 3.1 The result for test.

Figure 3.1 shows that SimHash hamming distance is much less than hash hamming distance. It shows that our SimHash function can detect similarity better.