Seminar paper on Probabilistic Near-Duplicate Detection Using Simhash

Arne Beer, MN 6489196, University of Hamburg

08.07.2019

1 Introduction

In the age of the modern Internet, many services depend in large parts on crawlers and proper document detection and duplication elimination, near duplicate document detection becomes a necessity. Real time detection of which website has already been visited and whether a website is new or just has been edited are important tasks during crawling [4].

Standard hashing functions are often inefficient and operate in $O(n^2)$ space requirements for RAM and computing time [3]. At the same time, the size of available documents grow steadily. Google's website index alone has multiple hundreds of billions of web pages and over 100 Petabyte in size, according to their information website [1].

This paper attends to the paper *Seminar paper on Probabilistic Near-Duplicate Detection Using Simhash* [8]. The main challenge tackled by this paper is to find all matching pairs of fingerprints withing a certain Hamming distance *h*. At this time, the fasted implementation for this procedure has been *Block-Permuted Hamming Search* (BPHS), which requires RAM space at least four times the size of the whole dataset. The authors of [8] aim to design a new algorithm that allows significantly faster online and batch document comparison and furthermore reduces RAM requirements, in exchange for a small percentage of recall loss.

In the first chapters, the basics for understanding this topic will be explained. Afterwards the proposed algorithm will be looked at and the authors' findings will be discussed.

1.1 Conventional Hashing

Hashing is a technique, which is used to map data of an arbitrary size to a fingerprint with some fixed size. This procedure could be seen as a function $f(i) \rightarrow j$, which produces a value j from from any value i, where $j \in H$ and H is the set of values of the fixed length s with $s \in \mathbb{N}$. Well-known hash functions are, for instance, *MD5* or *SHA265*. These hashing functions are commonly used to check whether two files are absolutely identical or, for instance, to verify that a file has not been corrupted during transport. This is possible, since these hashing functions are designed to flip half of the output hash bits on average, if an input bit changes [3]. Without this property it would be easier to change the input without the hash signature being modified. This would allow malicious third parties to, for instance, change code in a binary, without users being able to detect the change with the help of this hash and would require a full byte level comparison between the original and the copied file to verify its integrity.

If, on the other hand, one's goal is to find near duplicates, which are identical for the most part, but sometimes only differ by a few bits or bytes, this hashing approach immediately becomes useless, due to this property. Due to the need for a hashing algorithm, that creates a fingerprint based on the features and structure of the input data, *simhash* has been created.

2 Simhash

To compare documents for similarity a methodology to create some uniform value, which should be identical for two similar matching documents.

simhash is a procedure used to create a fingerprint of a any kind of data. This fingerprint can then be used to, for instance, inspect two files for similarity.

The process for creating such a fingerprint can completely differ depending on the features in the hashed data one is interested in. In case one wants to find similar binary files, it would be reasonable to split the data into equal chunks and use these chunks as features. For websites or documents, looking at the composition and structure of text could be a viable approach to select features for hashing The original data can then be refined to a high dimensional vector of features.
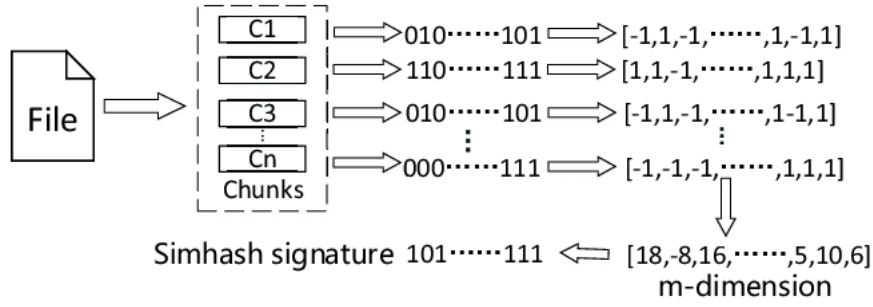
Figure 1: Visual example for calculating a *simhash* fingerprint. A binary file split into chunks, which are then hashed. Combining all hashes results in the desired fingerprint. [9]

The size of available features can vary significantly and is completely in the hand of the designer for each *simhash* implementation. It's important to note, that there exists no clear guideline on which features of a data set are interesting and which features can be ignored. The performance of a *simhash* implementation thereby also depends on the chosen features and the respective properties of the dataset. Such features can be for instance binary chunks, file extension [6], individual words, tags or URLs [8].

After determining in which way feature are extracted from the original data, each feature is hashed and mapped onto a fingerprint, which represents the constellation of features.

Algorithm 1 explains the process of [2] for creating fingerprints. At first one iterates over the features subset $F(u)$ of size b, which is selected from the set F of all unique features with $F(u) \in F$. Each feature is then uniformly hashed to a fix size. This uniform hashing can be seen as a random projection of a arbitrarily large feature to a fixed size. The amount of ones and zeros in this hash are then counted. A one increments the counter, a zero decrements it. The result is then stored in a weight vector W under the respective index of the feature. W has the same length b.

After calculating all values of W, a new binary Vector B will then be created with the same Length as W. Each value v with $v_i \geq 0$ at index i will result in $B[i] = 1$, while each $v_i < 0$ will result in $B[i] = 0$.

2.1 Weights and Hamming Distance

To achieve their goal of decreasing the runtime of fingerprint comparisons, the authors of *inproc:main* investigated the properties and distribution of weights of the temporary weight vector W that was introduced the previous section.

3

Algorithm 1 Simhash (u)

1: $W \leftarrow$ array of b zeros
2: **for** $i \in \mathcal{F}(u)$ **do**   ▷ Examine each feature
3:     $\phi_i \leftarrow \text{UniformHash}(i)$ ▷ Compute b-bit hash
4:     **for** $j = 1$ to b **do**   ▷ Iterate through each bit
5:         **if** $\phi_{ij} = 1$ **then**   ▷ j-th bit of ϕ_i
6:             $W[j] \leftarrow W[j] + w_i$ ▷ Add feature weight
7:         **else**
8:             $W[j] \leftarrow W[j] - w_i$ ▷ Subtract feature weight
9:         **end if**
10:     **end for**
11: **end for**
12: **for** $j = 1$ to b **do**   ▷ Revisit all bits
13:     **if** $W[j] \geq 0$ **then**
14:         $B[j] \leftarrow 1$ ▷ Positive weight, set bit to 1
15:     **else**
16:         $B[j] \leftarrow 0$ ▷ Negative weight, set bit to 0
17:     **end if**
18: **end for**
19: **return** array $B[1 \ldots b]$ ▷ simhash

For their current work, they used normalized term frequency–inverse document frequency (TF-IDF) weights. This kind of weights is used to reflect the importance of a word to a document in a collection and is great for text-based document comparison methods [7]. The corpus of the researchers consisted of 70M web pages.

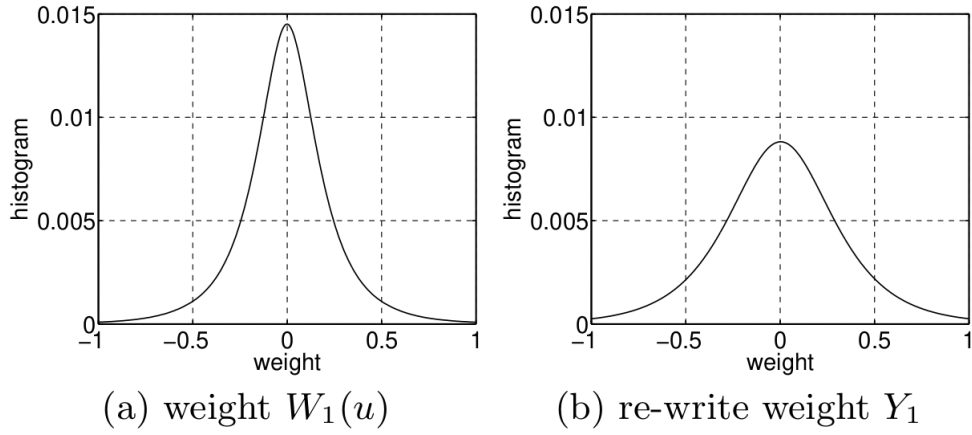(a) weight $W_1(u)$ (b) re-write weight Y_1

Figure 2: Histogram of *simhash* weights. A clear trend towards a Laplace-like distribution can be seen in a.

While using normalized TF-IDF weights, they noticed, that the distribution of weights rather resembles a Laplace-like distribution, than a Gaussian distribution, which has been expected, since the Central Limit theorem suggests it. This can be seen in Figure 2.

Using this knowledge they were able to generalize the formula for determining the specific weight of a feature to non-Bernoulli cases:

$$W_j(u) = \sum_{i \in F(u)} v_{ij} w_i(u)$$

v is a vector, whose elements are drawn from a zero-mean symmetric distribution, which is theoretically equivalent to the Laplace-like distribution mentioned above.

After investigating the various probabilities and necessities for changing a weight to the side, it becomes apparent that certain bits are much more volatile to flip than others. If one looks at the *simhash* weights $(12, 0, -10)$, the second bit is much more likely to flip to the negative side, when small changes occur, than the other weights. Much bigger changes would be needed to flip the weights at position 1 or 3.

3 Bit order

With the insight gained in the previous section, we are now able determine, which bits are more volatile to flip than others. This knowledge can be utilized to determine an order, that should be used to determine the Hamming distance *h* between the fingerprint and all hashes in the collection *D*. While this is straight forward for single

bit flips, where bits are simply ordered by their volatility, it becomes more complex, as soon as one wants to detect two-bit combination flips.

3.1 Ordering

$p_j(u)$ be the probability of another page in D, which has a flipped bit in position j in the simhash of a page u with $p_j(u) = P(Y_j > |W_j(u)|)$. Y is the random weight chance that occurs to changes in a document with $Y = X_j^\alpha - X_j^\beta$, where $X_j^\alpha - X_j^\beta$ are corresponding to the changed features.

Now it's possible to order and limit the Hamming search to the most likely subsets of hash bits. For this let $S \subseteq \{1, 2, \ldots, b\}$ be a subset of non-empty hash bits. The probability for the existence of a page in D with differing bits on S is then:

$$p(u, S) = \prod_{i \in S} p_i(u) \prod_{j \notin S} (1 - p_j(u))$$

Creating all possible subsets of S in decreasing would be too time and space consuming, since it requires $\Theta(l \log l)$ operations and $\Theta(\sum_{i=1}^{h} \binom{b}{i})$ space.

3.2 Volatility Heap

To tackle the computation power and space requirement problems mentioned in the section above, the authors of [8] create a new data structure they call *Volatility Ordered Set Heap* (VOSH).

For defining such a data structure, a specific property of the Sets of considered bits must be ensured.

Lemma 1 *If two sets S_1, S_2 of the same size h differ in exactly 1 bit and $S_1 \prec S_2$, then $u : p(u, S_1) \geq p(u, S_2)$.*

$\prec$ represents the lexicographical comparison operator on S. Lemma 1 describes that two sets fulfill $S_1 \prec S_2$, if there exists an index $i < h$ such that both sets have the same leading elements of length i, but S_2 is still larger in the $(i+1)$th element.

Algorithm 2 ProduceChildren($\mathcal{S}$)

1: **if** $\mathcal{S}[h] \leq b - 1$ **then** $\qquad\qquad\qquad$ ▷ Attempt left child
2: $\qquad \mathcal{S}_L \leftarrow \mathcal{S}; \quad \mathcal{S}_L[h] \leftarrow \mathcal{S}_L[h] + 1$ $\qquad\quad$ ▷ Add 1 to last bit
3: **else**
4: $\qquad \mathcal{S}_L \leftarrow \emptyset$ $\qquad\qquad\qquad\qquad\qquad$ ▷ Left child is empty
5: **end if**
6: **for** $j = h - 1$ **downto** 1 **do** $\qquad\qquad$ ▷ Attempt right child
7: $\qquad \text{gap} = \mathcal{S}[j+1] - \mathcal{S}[j]$
8: $\qquad$ **if** $\text{gap} = 2$ **then** $\qquad\qquad$ ▷ Should increment bit j?
9: $\qquad\qquad \mathcal{S}_R \leftarrow \mathcal{S}; \quad \mathcal{S}_L[j] \leftarrow \mathcal{S}_L[j] + 1$
10: $\qquad$ **else if** $\text{gap} > 2$ **then**
11: $\qquad\qquad \mathcal{S}_R \leftarrow \emptyset; \quad$ **break** $\qquad\qquad$ ▷ Right child is empty
12: $\qquad$ **end if**
13: **end for**
14: **return** $(\mathcal{S}_L, \mathcal{S}_R)$ $\qquad\qquad\qquad$ ▷ Produce both children

---

Figure 3: Pseudocode representation for the generation of a Volatility Ordered Set Heap (VOSH).
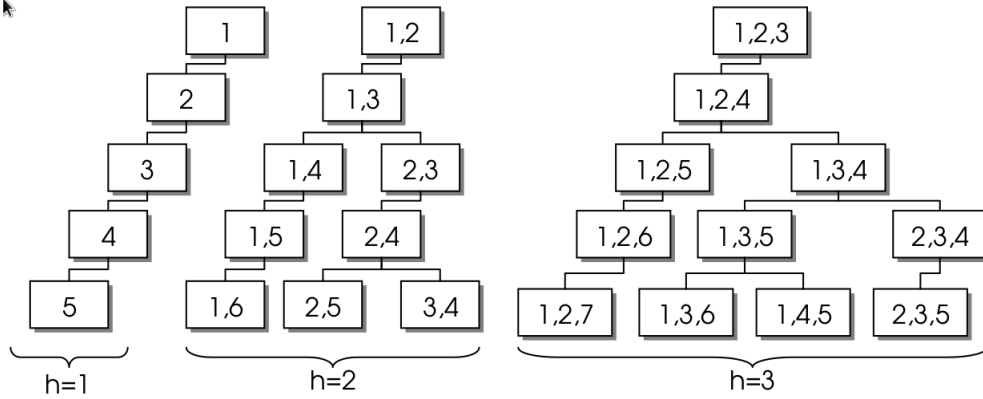

Figure 4: Example for a generated VOSH structure generated by Algorithm 2.

The assertions gained by Lemma 1 allow us to create an algorithm, that creates a space efficient lookup data structure for determining near-duplicates within a certain Hamming distance h. The bit combinations are sorted using the algorithm visualized in figure 3.

The algorithm creates a binary tree structure, which is generated as follows. For each node, check, whether the rightmost bit can be incremented. If that is the case,

7

create a new node with the new bit constellation as the left child node. If the bit cannot be incremented, the left child node will be omitted.

At the same time, check the bits from right to left whether there exists a difference (or gap) between two adjacent bits that equals 2. If there exists such a gap, the left bit of those two bits is incremented. The new bit constellation now forms the right child node. If there is no such gap, the right child node is simply omitted.

Figure 4 shows example results for volatility heaps created by this algorithm. The volatility heaps are created with different Hamming distance thresholds h.

4 Probabilistic Search

5 Impact in the scientific community

This paper has been cited by 36 different publications. Considering this paper has been released 2011, it's still cited by papers and books in 2019, which makes it still valuable contribution to the scientific community.

6 Relevance as of 2019

Near duplicate detection is still a very important research field. The current trend shows, that data will continue to be generated in a exponential manner. Tackling an even larger amount of data and filtering it for duplicates will continue to create new challenges in respect to data throughput and RAM limitations.

Simhash is being used by many companies in various application fields. For instance, Google started using *simhash* in 2007 for their web-crawling services [5]. With Google being one of the biggest Internet companies and the number one search engine, relevance of this topic is obviously very high.

This is especially interesting, since this paper takes the original algorithm proposed for usage in Googles web crawler [5] as comparison for their own research while showing that they are capable of significantly speeding up the lookup.

References

[1] Google. *How Search organizes information*. 2019. URL: https : / / www . google . com / search / howsearchworks / crawling - indexing/ (visited on 07/02/2019).

[2] Monika Henzinger. "Finding near-duplicate web pages: A large-scale evaluation of algorithms". In: Jan. 2006, pp. 284–291. DOI: 10.1145/1148170.1148222.

[3] A.G. Konheim. *Hashing in Computer Science: Fifty Years of Slicing and Dicing*. July 2010. Chap. 8. DOI: 10.1002/9780470630617.

[4] Hsin-Tsang Lee et al. "IRLbot: Scaling to 6 Billion Pages and Beyond". In: *ACM Transactions on the Web (TWEB)* 3 (Jan. 2008), p. 8. DOI: 10.1145/1541822.1541823.

[5] Gurmeet Singh Manku, Arvind Jain, and Anish Das Sarma. "Detecting Near-duplicates for Web Crawling". In: *Proceedings of the 16th International Conference on World Wide Web*. WWW '07. Banff, Alberta, Canada: ACM, 2007, pp. 141–150. ISBN: 978-1-59593-654-7. DOI: 10.1145/1242572.1242592. URL: http://doi.acm.org/10.1145/1242572.1242592.

[6] Caitlin Sadowski and Greg Levin. "SimHash: Hash-based Similarity Detection". In: Dec. 2007. DOI: 10.1.1.473.7179.

[7] Gerard Salton and Michael J. McGill. *Introduction to Modern Information Retrieval*. New York, NY, USA: McGraw-Hill, Inc., 1986. Chap. 6.2. ISBN: 0070544840.

[8] Sadhan Sood and Dmitri Loguinov. "Probabilistic Near-Duplicate Detection Using Simhash". In: Oct. 2011, pp. 1117–1126. DOI: 10.1145/2063576.2063737.

[9] Yongtao Zhou et al. "EPAS: A Sampling Based Similarity Identification Algorithm for the Cloud". In: *IEEE Transactions on Cloud Computing* PP (Feb. 2016), pp. 1–1. DOI: 10.1109/TCC.2016.2527646.