University of Zagreb
Faculty of Electrical Engineering and Computing
Consumer Computing Lab

Big data analysis

Authors: Marin Šilić, Klemo Vladimir, Adrian Kurdija
Ac. year. 2017/2018

1. Lab exercise

The goal of this exercise is to implement text hashing using **Simhash** algorithm. Unlike cryptographic hashing algorithms (which are extremely sensitive to minimal changes of the input text), the Simhash algorithm preserves the similarity of the input texts: if the input texts are similar (i.e. they differ in several words), then their hashes generated by Simhash differ in a small number of bits (by Hamming distance). The generated hashes will be used to identify similar texts.

In the exercise, there are two tasks of identification similar texts. In the first task (task A), identification of similar texts should be done by sequential search of hashes of all texts. In the second task (task B) you should use *Locality Sensitive Hashing* (LSH) techinque. The rest of the document is organized as follows: section 1.1 describes the Simhash algorithm, section 1.2 describes the format of the input file for tasks A and B, and section 1.3 describes the LSH hashing algorithm.

1.1 SimHash algoritm

The input to the simhash algorithm is a sequence of characters (text or a document). The output is a sequence of characters which is a **hexadecimal representation of the 128-bit** hash. The Simhash algorithm is described in detail in the lecture "Detection of near-duplicate documents".

The Simhash internally uses one of the traditional hashing algorithms. In this exercise, the 128-bit cryptographic hashing algorithm **md5** will be used. In addition, the input text will be decomposed into a sequence of space-separated units. For example, if the input text to the simhash is `"fakultet elektrotehnike i racunarstva"`, then the simhash will split the text into 4 units: `["fakultet", "elektrotehnike", "i", "racunarstva"]` and calculate the md5 hash for each unit. The created md5 hashes are analysed and the final output hash of the simhash algorithm is generated according to the following pseudocode:

```
function simhash(text):
  var sh = []
  units = generate_units(text)
  for each unit in units:
      hash = md5(unit)
      if i-th bit in hash equals 1:
        sh[i] += 1
      else:
        sh[i] -= 1
  if i-ti element in sh >= 0:
    sh[i] = 1
  else:
    sh[i] = 0
  return hexadecimal(sh)
```

Note: this pseudocode is one of many possible illustrations of the algorithm and it is not a mandatory template of the solution.

The example of the expected hash generated by the simhash algorithm for input "**`fakultet elektrotehnike i racunarstva`**" is "**f27c6b49c8fcec47ebeef2de783eaf57**".

We recommend to use Java or Python for this exercises. The assistants have implemented the exercises in these languages and are able to provide you help if needed. You can send questions to avsp@zemris.fer.hr.

If you use Python, the md5 algorithm is available in the **hashlib** library. In Java, please use only the following package:

- **`org.apache.commons.codec.digest.DigestUtils`** (http://commons.apache.org/proper/commons-codec/apidocs/org/apache/commons/codec/digest/DigestUtils.html)

1.2 Task A - sequential search of the hashes

Note: *Task A is worth 50% of the points for the 1. lab*.

Apart from calculating the simhash, in this exercise you should implement the identification of the similar texts. The format of the input file for the program in this lab exercise is the following:

```
N
```

```
text_0
...
text_N-1
Q
query_0
...
query_Q-1
```

In the input file, each line ends with a newline character (**\n**). Line 1. contains an integer (**N <= 1000**) representing the number of texts for which simhash is to be calculated. After this line, N lines with N texts follows. Each text is given in a separate line and contains the lowercase English characters. After N input texts, the following line contains an integer representing the number of queries (**Q <= 1000**). The last Q lines are queries of similarity calculation.

Each query contains two space-separated integers (**I, K; 0 <= I <= N-1 i 0 <= K <= 31**). For **each query** the program should generate an integer representing the total number of texts whose hashes differ by at most K bits from the hash of the I-th text (by Hamming distance). For example, if I=16 and K=3, the output for this query is the number of texts whose hashes differ from the hash of the 17th text (we use 0-based indexing of texts) by at most 3 bits..

Important notes:

- The time limit for the program execution for any input file equals 20 seconds
- The input point for solutions in Java should be in the **SimHash class**. The input point for the solutions in Python should be in the **SimHash.py** file.

1.2.1 Sample test case

On the course website you can find the sample input file along with the expected output (*lab1A_primjer.zip*). Please check the correctness of your solution on the given example before submitting it to the *sprut* system. The evaluation of this task will be performed on 5 input files, including the sample one.

1.3 Task B - LSH

Note: *Task B is worth 50% of the points for the 1. lab*.

In task B, the input file format is exactly the same as for the previous task A, but the following parameter contraints will apply:

N <= 10^5
Q <= 10^5
0 <= I <= N-1 and 0 <= K <= 31

From these constraints, sequential search for similar texts is clearly not efficient enough. Therefore, you should use the LSH hashing algorithm, described in the lecture AVSP_03a (*slide* 30-31).

In the first step you should, as in the task A, create **128-bit hashes** for all input texts using the **SimHash** algorithm described in section 1.1. In lectures we used the the MinHash algorithm, but here we use the faster SimHash algorithm, where the hash size is **k = 128**.

In the second step you should apply the LSH algorithm to the obtained hashes. The basic idea is to divide the hashes into **b bands**. In this exercise we will set **b = 8**, which means that each band will contain **r = k/b = 128/8 = 16 bits**. The algorithm for each band hashes the integer interpretation of the part of the hash belonging to the band. All pairs of hashes which map to the same bucket in at least one band become the candidates for similarity. “Mapping to the same bucket in a band” means that the hashes in this band are identical. For example, in case of two hashes of size k = 4, namely S_1 = “01**01**” and S_2 = “10**01**”, and b = 2 bands (r = k/b = 2), the hashes S_1 and S_2 map to the same bucket in the second band because the hashes in that band are identical. After the hashing procedure in all bands is done, for each text we get the corresponding set of similar text candidates.

The pseudocode for LSH algorithm:

```
list simHash = makeSimHash(texts);
dictionary candidates = {} ;
for band = 1 to b {
      buckets = {};
      for current_id = 0 to N - 1 {
            hash = simHash[current_id];
            // hash contains 128 bits
            // Take r = 16 bits in the current band
            // starting from less significant bits
            // e.g. for band = 1, take bits 0:15,
            // for band = 2, take bits 16:31, etc.
            // Using hash2int function, convert 16 bits to integer
            val = hash2int(band, hash);
            texts_in_buckets = {};
            if (buckets[val] != {}) {
                  texts_in_buckets = buckets[val];
                  for each text_id in texts_in_buckets {
                        candidates[current_id].add(text_id);
                        candidates[text_id].add(current_id);
                  }
            } else {
```

```
                texts_in_buckets = {};
            }
            texts_in_buckets.add(current_id);
            buckets[val] = texts_in_buckets;
        }
}
```

After LSH is done, the `candidates` map for each input text contains a set of candidate similar texts (namely, the indices of those texts based on their ordering in the input file).

In the third step, the program should, as in task A, generate an integer representing the total number of documents whose hashes differ by at most K bits from the I-th document hash (by Hamming distance). While generating the number, the program should use the `candidates` map, i.e., similar texts to the I-th text should be found only according to the `candidates` map built using the LSH algorithm.

For example, if I=16 and K=3, the program for the given query searches in `candidates` the set of candidate texts mapped to the 16th text, and prints the number of texts whose hashes differ from the hash of the 17th text (we use 0-based indexing of texts) by at most 3 bits..

Important notes:

- The time limit for the program execution for any input file equals 100 seconds
- The input point for solutions in Java should be in the **SimHashBuckets class**. The input point for the solutions in Python should be in the **SimHashBuckets.py** file.

1.3.1 Sample test case

On the course website you can find the sample input file along with the expected output (*lab1B_primjer.zip*). Please check the correctness of your solution on the given example before submitting it to the *sprut* system. The evaluation of this task will be performed on 5 input files, including the sample one.