Antiplagiarism system - report

Giuseppe Esposito
s302179

I. Preliminary study

Given a reference text, the aim of the Antiplagiarism system is to detect plagiarism within another document. There are multiple ways to design such a mechanism, and we will analyze its effectiveness. The preprocessing of the reference document consists of text cleaning operations like removing punctuation, making the words in lower case, and then tokenizing the text in sentences of fixed length ($S = 6$). The antiplagiarism approaches differ from the used data structure and these are:

- set of sentences
- set of sentences fingerprint
- bit string array
- bloom filter

But they have a common aim: store the sentences and check from the text suspect of plagiarism whether there are some common sentences and count them. The document we take is a collection of some Canti from "Divina Commedia".

The input of the software are:

- Reference text: in our case, it is from "Divina Commedia".
- Sentences size (S): length of the sentences in terms of the number of words.

By means of this, we were asked to compare the efficiency of the different approaches in terms of the probability of False Positive with respect to the memory occupancy of the whole data structure under analysis. As soon as all the sentences are stored in the chosen data structure, let us consider an element outside the selected structure, the **probability of false positive** is defined as the probability that that element results inside the data structure.

From the preliminary study, it came out that, given S, the poem is composed of 96229 sentences and for each sentence, the average size is ≈ 91.38 bytes as shown in the code.

$$L = \frac{\sum_{i=0}^{m-1} asized.asizeof(< sentence >)}{m} \tag{1}$$

where L is the average size of each sentence and m is the number of sentences. Let us define the list that stores all the sentences as **universe**(sentences).

II. Set of sentences

The set of sentences (sentence_set) is a **subset** of the universe because it contains only the unique values of the universe. So that the size of the set of sentences is 12985624 Bytes which contains both the sizes of the referents of the set and the size of the structure itself. Despite the memory consumption, the probability of false positive in this case is 0 because in the set all the possible sentences of fixed length are stored without any kind of encoding.

III. Fingerprint set

For the fingerprint set, I evaluated the minimum number of bits such that no internal collision [1] occurs in the set. To accomplish this task, for each sentence I used the **md5** hash function as suggested by the professor, within the range [0,n-1] where n is the number of available bits in which the fingerprint of the sentence can be stored (2^k). Starting with k = 0 I added all the encoded hashes in the fingerprint_set. If a hash collision occurs, increase k and store the new fingerprint encoding of the sentences in an empty fingeprint_set. The cycle will stop when all the fingerprints are stored without collision, so that, when $len(fingerprint\_set) = len(sentences\_set)$. From this experiment, we obtained a number of bits (k, which we call Bexp to be compliant with the task) equal to 33. The corresponding probability of false positive is computed as:

$$p = 1 - e^{-\frac{-m}{n}} = 2.24 \times 10^{-5} \tag{2}$$

With probability (p) of internal collision = 0.5, $E[m] = 1.25\sqrt{n}$ holds, where E[m] is the expected number of sentences to observe at least one (hash) collision and n is the total number of available bits. Then the obtained empirical number of bits is then compared with the theoretical value **Bteo** that is computed as follows:

$$Bteo = \log_2((\frac{m}{1.25})^2) = 32 \tag{3}$$

So that we can compute the theoretical probability of false positive:

$$p = 1 - e^{-\frac{-m}{n}} = 1.12 \times 10^{-5} \tag{4}$$

We can observe that, although the probability of collision is 0.5 for the theoretical and 0 in the empirical, the number of bits of the actual simulation is very close and this is due by the fact that, a sensible increase in the number of bits follows high decrease in the probability of false positve.

IV. Bit string array

The bit string array is defined as an array of length equal to the number of bits, which collects ones in the position of the array corresponding to the hash of the encoded sentence. Then to "populate" the array, for each value of n, I initialized an array of zeros, and then, for each sentence, I applied the hash function, eventually, the value of the array in position h is set to one (where h is the result of the hash function).

[1] The **hash collision event** occurs when the same hash function applied on 2 different elements returns the same value

In order to compute the empirical probability of false positive, I just summed the ones in the array and divided by the total available storage (n):

$$P = \frac{\sum_{i=0}^{n}(bit\_string\_array)}{n} \tag{5}$$

On the other hand, the theoretical probability of false positive, is computed as

$$Pr_{fp} = 1 - (1 - \frac{1}{n})^m \tag{6}$$

For each value of the available metric, I plotted the corresponding theoretical and empirical values of the probability of false positive. In Fig 1, I compared the result of the experiment

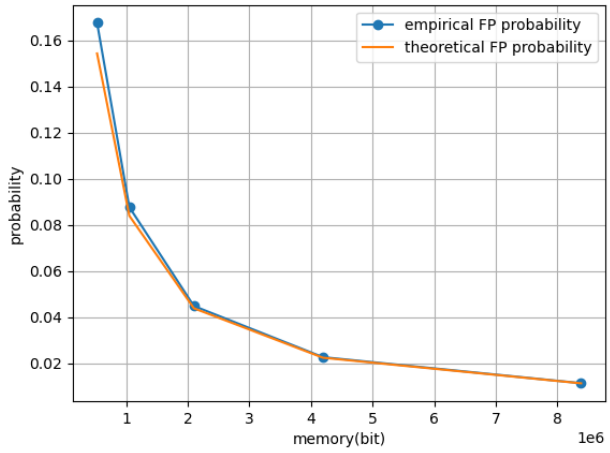

Fig. 1. Probability of false positive VS available memory in **bit stringa rray**. Points are the experiments' results with varying memory.

with the theoretical result. As we can notice the two curves almost overlaps so the result of the simulation is quite accurate. Moreover, we can notice that both trends are decreasing trends, and this is reasonable because, if n increases, it means that more sentences can be stored in the bit string array, then the probability of taking an element outside of the set and checking whether it is encoded in the way as another value is, increases.

Let us recall that Bexp is equal to 32 and now I want to compare Bexp with the values that I have just obtained. As python has problems in initializing an array o zeroes of size 2^{32}, I decided to make an inference on the decreasing trend of the empirical values, so I computed the ratio between the successive values to find a reasonable pattern between them and i obtained the following result:

- $\frac{P_{fp}(n \leftarrow 2^{20})}{P_{fp}(n \leftarrow 2^{19})} \approx 0.52$
- $\frac{P_{fp}(n \leftarrow 2^{21})}{P_{fp}(n \leftarrow 2^{20})} \approx 0.51$
- $\frac{P_{fp}(n \leftarrow 2^{22})}{P_{fp}(n \leftarrow 2^{21})} \approx 0.51$
- $\frac{P_{fp}(n \leftarrow 2^{23})}{P_{fp}(n \leftarrow 2^{22})} \approx 0.50$

As we can notice the probability of false positive halves at each increasing value of the memory size. Assuming that the ratio keeps resulting the same, the probability of false positive in case of memory = 2^{32} should be:

$$0.5^{32-23} \times 0.011 = 2.13 \times 10^{-5} \tag{7}$$

This predicted value is slightly lower than the one evaluated for the Fingerprint set but they have the same order of magnitude that is very low. This sounds reasonable because the probabilities are computed almost in the same way, and the experiment is carried out on the same set of sentences.

V. Bloom Filter

The bloom filter represents an improvement of the bit string array. The structure is basically the same: an array of zeroes is initialized, and for each sentence of the set I apply k different hash functions and then I set to 1 the bit of the array in position h where h is the encoded sentence.

Let us consider a bloom filter with k hash function and size n, the probability to hit a bit that is already set to 1 is:

$$P = (1 - (1 - \frac{1}{n})^{mk})^k \approx (1 - e^{-\frac{km}{n}}) \tag{8}$$

For each value of n, we can find the optimal number of hash functions by deriving the probability with respect to k and find the number of hash functions that minimize the probability of false positive.

$$\frac{\delta Pr(FP)}{\delta k} = 0 \rightarrow k_{opt} = \frac{n}{m} log(2) \tag{9}$$

In Fig 2 I report the optimal number of hash functions with respect to the available size in the bloom filter. As we can

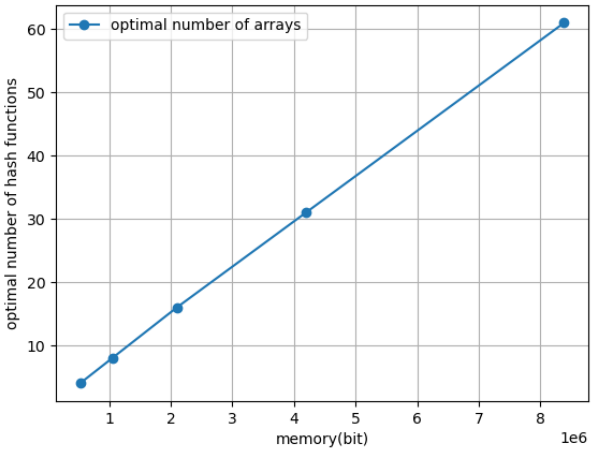

Fig. 2.

notice the optimal number of hash functions increases as the memory in bit increases and this trend is clarified by the equation 8. Once computed the number of hash function we can build the **bloom filter**. To do that I applied the algorithm 1:

Algorithm 1 Bloom filter population

1. $n \leftarrow memory\_size$
2. $k \leftarrow \frac{n}{m} \log 2$ where m is the total number of sentences
3. compute the "suffixes" for each hash function
4. $arr \leftarrow$ an array of zeros of size n
5. **For**: Sentence in sentences
6. indexes $\leftarrow empty\_list$
7. h $\leftarrow$ hash value for each hash function and store in indexes
8. arr[h] = 1 for h in indexes

Note that in lines 6 and 7 I am creating k different hash encoding for the same sentence, this means that each sentence will be encoded k times from k hash functions that are independent of each other. Of course, in the implementation of a search algorithm in the bloom filter we should compute again the md5 encoding for the same sentences with the same hash functions applied in the "population" method.

In the end, I evaluated the theoretical probability of false positive for each corresponding number of bits as

$$Pr_{fp} = (1 - e^{\frac{mk}{n}})^k \tag{10}$$

so that the two curves representing the theoretical and the empirical one actually overlap. The results are reported in the plot 3

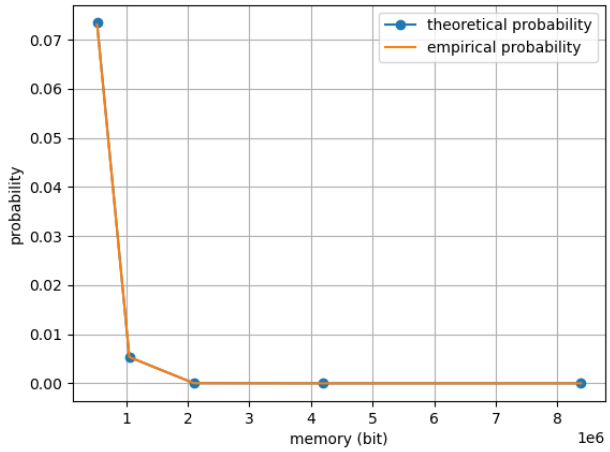

Fig. 3. Probability of false positive VS available memory in **bloom filter**. Points are experiments' results with varying memory.

I have also evaluated the empirical number of distinct sentences as:

$$distinct\_sentences = sentences - collision\_counter \tag{11}$$

and I have seen that it has values very close to the theoretical ones that are computed as:

$$distinct\_sentences = -\frac{n}{k} \log(1 - \frac{N}{n}) \tag{12}$$

where N is the actual number of ones in the array.

VI. Results

Now that we have an overall view of the experiments we can compare the result whose target is to find the best data structure with a trade-off between the occupied memory and the probability of false positive.

TABLE I
EXPERIMENTS RESULTS

	Memory [kB]	Prob. false positive
Set of sentences	12985.62	0
Fingeprint set(X=Bexp)	1073.74	2.24×10^{-5}
Bit string array(X varying)	1048.57 see fig 1	1.14×10^{-2} see fig 1
Bloom filter(X varying)	262.14 see fig 3	2.85×10^{-5} see fig 3 height

Under this assumption, we can compare the results of the experiments shown in table I (notice that in the record of the bit string array and of the bloom filter are reported only the relevant values corresponding to low values of probability and the references to the figures that pictures the probabilities' trend).

We can see that the data structure that guarantees the lowest probability of false positive among the reported ones, is the set of sentences, even though it occupies the highest amount of memory. For what concerns the fingerprint set it has good performances in terms of P_{fp} and of memory occupancy and, as stated in the section describing the bit string array, the two behaviors are almost the same for both data structures, even though the bit string array shows results that are slightly better. Eventually, we have the bloom filter that, with memory consumption of 262.14 kB (which corresponds to 2^{21} bits) it has a P_{fp} comparable with the one of the other data structures, this means that bloom filter is built such that the trade-off between the memory occupancy and the probability of false positive is met very efficiently, in fact if we keep increasing the amount of available memory, the P_{fp} reaches the value of 6.38×10^{-19}. In the end, I would say that the best trade-off is found in the bloom filter with a memory occupancy of 262.14 kB and a corresponding P_{fp} equal to 2.85×10^{-5}.