# **Lab 17: Bit string arrays and bloom filters**, Aurora Gensale s303535

s303535@studenti.polito.it

## I. Problem statement

In this lab, we had to create an anti-plagiarism poem system using several data structures with the objective of comparing performances and memory occupancy. *La Divina Commedia* by Dante Alighieri was the poem under consideration. The software has to operate with phrases of length $s$, where s in this case is equal to 6.
The following data structures will be investigated:

- a set storing the sentences
- a set storing the fingerprints of the sentences
- a bit string array
- a optimally-designed bloom filter

### *A. Input parameters*

- **s = 6** : number of words in the sentences that will be created;
- a **.txt** file containing the poem, this is going to be cleaned and preprocessed in order to create sentences of length **s**.

### *B. Output metrics*

This software provides us with *false positive probability, memory occupancy*, and other specified metrics as outputs in order to compare data structures.

### *C. Preliminary study*

After reading the **.txt** file, we must preprocess it, which entails cleaning the text by removing all punctuation and titles. Furthermore, all of the words are converted to lower case, and we may ultimately divide the text into sentences of length **s**, with a stride of one word. Now we have useful data for our purpose.
Some of the first metrics requested are included below, along with their results.

- *Number of sentences stored for $s = 6$ :* **101707**.
- *Average size of each sentences in bytes is* **88.74**.

## II. Data structures involved

We will answer all of the requests concerning to one data structure at a time in this section.

### *A. Set of sentences*

*Sets* in Python are collections of elements that are built on hashing, do not allow duplicates, and offer rapid memory access due to their nature. In our scenario, we wanted to save the sentences in a set and measure the memory occupancy.

- *Memory occupancy for the set of sentences is* **13207448 bytes**

### *B. Fingerprints set*

Instead of saving sentences, we want to save their *fingerprints* in a set. When we refer to fingerprints, we mean the results of applying a hashing function to each sentence individually. Since we must set a parameter $n = 2^b$, that represents the hash range, the memory dimension of the fingerprint is up to us.
More precisely, $n = 2^b$, $b$ is an integer number $\geq 0$, means that the element considered will be transformed to a $b-bit$ value and its fingerprint can range from 0 to $n-1$.
After creating this fingerprints set, we can compute, by simulation or analytically , the following metrics:

1) **Bexp** : the minimum value of bits such that no collisions are experienced when storing all the sentences in a fingerprint set, *by simulation.*
2) **Bteo** : the number of bits necessary to get a probability, $p = 0.5$, of fingerprint collision when storing all the sentences with a fingerprint set, *analytically.*
3) Moreover, the probability of false positive for a fingerprint set in which the number of bits is **Bexp**, as computed in 1).

*1) Bexp:* The essential concept is that, starting with $n = 2^{b=1}$, we fingerprint each sentence and save the results in a list.
We compare the lengths of the list and the set produced from it to see whether there was any conflict. This works because lists allow data duplication whereas sets do not, therefore the presence of differing lengths indicates the existence of at least one conflict. We repeat all of the preceding steps with a higher exponent, so $n = 2^{b=b+1}$, until no conflicts are found.
*Simulated results showed that* $\boldsymbol{Bexp = 35}$.

*2) Bteo:* Computations are simplified here because we have a formula to use.
Supposing that we have stored all elements in a fingerprints sets we want to store $m$ elements, to observe at least one collision with probability $p$, it must hold:

$$Bteo = \sqrt{2n \ln\left(\frac{1}{1-p}\right)}. \tag{1}$$

If we put in 1 $p = 0.5$ the formula reduces to

$$Bteo = 1.17\sqrt{n}. \tag{2}$$

With our parameters we obtained $\boldsymbol{Bteo = 32}$.

*3) False positive probability using b = bexp = 35:* We also have a mathematical formula for calculating the false positive probability, indicated as $P(FP)$, of a given fingerprint set. Given:

- $n = 2^b$: as mentioned before, $n$ is the hash range and $b$ is the amount of bits used to compute hashing;

- $m$: the number of elements needed to be stored in the fingerprints set.

We compute $P(FP)$ as follows:

$$P(FP) = 1 - (1 - \frac{1}{n})^m \quad (3)$$

Using out parameters we obtained:

$$\boldsymbol{P(FP)_{b=35} \simeq 2.954 \times 10^{-6}}$$

After our experiments, we can state that $Bexp$ could be a good approximation of $Bteo$ since we are computing the former in order to have zero collisions, while the latter accepts a probability of collision of 0.5. Also, even they differ for only 3 bits, when working with fingerprints, these 3 bits can really change the total probability of false positives, since from $n = 2^{32}$ to $n = 2^{35}$ we have $2^3$ extra possible values for fingerprints.

## C. Bit string array

The fingerprints of elements can also be stored using bit string arrays. Here, an array $BA$ of length $n$, the same meaning as before, with values 0 or 1 is used in place of the set. Given an hash map $h$, an element $y$, after computing $h(y)$ instead of storing it, we set to 1 the corresponding position of the array, *i.e.* $BA[h(y)] = 1$. Doing this we are creating the bit string array.
Using this data structure we want to evaluate, by *simulation* or *theory*:

1) the probability of false positive in function of memory $n = x$, for $x$ in the set $\boldsymbol{X} = [2^{19}, 2^{20}, 2^{21}, 2^{22}, 2^{23}]$ bits, *by simulation*
2) the probability of false positive in function of memory $x \in X$, *by theory*

*1) By simulation:* To compute false positive probability by simulation, first, we have to populate the $BA$ as indicated above. Then, imagining to add a never seen element, the probability of the fingerprint of this new element is already encoded in the $BA$ is equal to the total number of ones in the $BA$ divided by the length of $BA$, *i.e.* $n = x$. Of course, we compute this for all the different values $x \in X$.

*2) By theory:* To compute the same quantity but in a theoretical way we need to use the same formula as before, hence (3) where $n = x \in X$.
Figure 1 displays a comparison of theoretical and simulated false positive probability in a bit string array as a function of memory $x \in X$, as we can see the simulated method is a very goos approximation of the theoretical one. Because we have more potential values for each fingerprint when memory rises, as expected, the probability of having a false positive reduces. Even with the highest value of x, the probability does not reach zero, but it is extremely close in both computational methods.

When comparing this solution to the one in II-B, we can conclude that both are compliant, but the bit string array appears to perform better. This is due to the fact that, as shown in Figure 1, in the current situation, we achieved a false positive probability very close to zero with $x = 2^{23}$, corresponding in $b = 23$, but in II-B, we found similar results for $b = 35$.

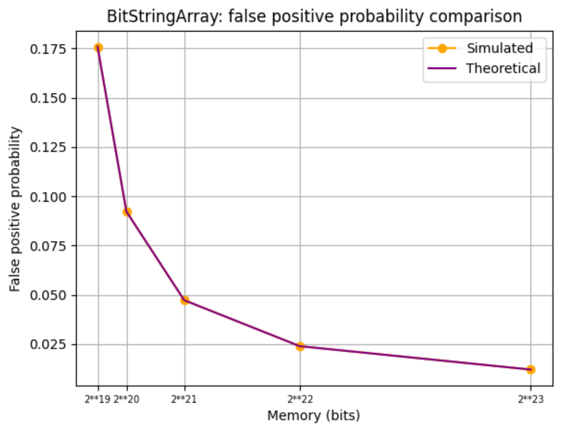

Fig. 1: This plot compares the theoretical and simulated false positive probability in relation to the memory taken into account while using a Bit String Array.

## D. Bloom filter

Bloom filter, $BF$, is a bit string array extension; it still implements an array to keep ones in elements fingerprints positions, but the novelty is that it employs multiple, $k$, hash functions. The consequence of having $k$ fingerprints is that for each element we set to 1 $k$ positions of the $BF$. Still referring to $\boldsymbol{X}$ as in II-B we had to:

1) Compute *analytically* the optimal number, $k_{opt}$, of hash functions in function of memory $X$;
2) by *theory*, evaluate the probability of false positive in function of $X$, using $k_{opt}$.
3) by *simulation*, evaluate the probability of false positive in function of X, $k_{opt}$.

*1) optimal k:* : in order to find, for each $x \in X$, the optimal number of hash functions we have to calculate the following formula. Given:

- $n = x$: the fingerprint range;
- $m$: the number of elements we want to store

optimal $k$ is compute as follows:

$$k_{opt} = \frac{n}{m} \log_2 . \quad (4)$$

Moreover, since it happens that $k_{opt}$ is not an integer number,hence $k_opt \in (k_L, k_U)$, contrary to what we want. To choose the best integer we compute theoretical false positive probability with $k_L, k_U$, using 5, and take the number that gives us the lower probability. Figure 2 shows obtained results.

*2) by theory:* to compute theoretical false positive probability we have an *ad-hoc* formula for bloom filters. Given:

- $n = x$: the fingerprinting range
- $m$: the number of elements to store
- $k$: the number of hash functions considered

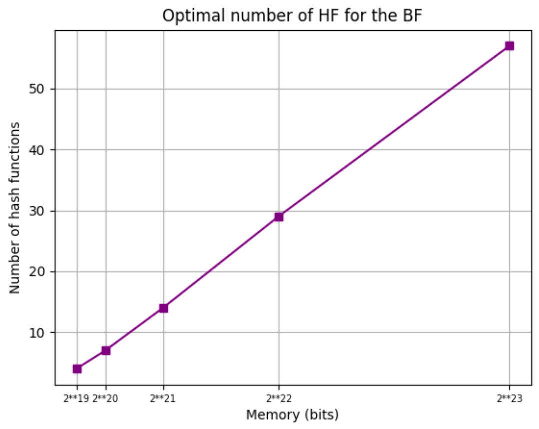

Fig. 2: This plot shows the optimal number of hash functions in function of memory $x \in X$.

the requested quantity is computed as follows:

$$P(FP)_{bf} = (1 - e^{\frac{-km}{n}})^k \quad (5)$$

in our case, for each $x \in X$, $k$ in the formula is equal to $k_{opt}$.

*3) by simulation:* in this case, like in bit string array, the main idea behind the computations is: first of all we populate the $BF$, then supposing to taken a never seen element we want to compute the probability of that it is already encoded in the $BF$. In this case, since we are using $k$ different hash functions, we compute the simulated false positive probability as the total ones in the $BF$ over the $BF$ length, *i.e.* $n = x$, all elevated to the power of $k$. Another thing to be outlined is that, in simulation, we do not use really $k$ different hash functions, but we modify each element $k - 1$ times and use the same hash function. The element is modifies only $k - 1$ times because the first fingerprint will come from the original element.

Figure 3 shows the results obtained, more specifically it compares the theoretical and simulated false positive probability in relation to $x \in X$ and its $k_{opt}$ computed before. As we can state from the image, like for the previous section, the simulated approximate very well the theoretical one.

## III. Optional Part

The optional part is related to *Bloom filters*. It can be shown that the *number of distinct elements stored* in a bloom filter with $n$ bits and $k$ hash functions can be estimated as

$$-\frac{n}{k} \ln(1 - \frac{N}{n}) \quad (6)$$

where $N$ is the actual number of bits equal to 1 in the bloom filter. We want to show how accurate is the formula when inserting one single sentence at the time in the bloom filter, in function of the inserted sentences. We arbitrary set $n = 2^{19}$. In our simulation the number of distinct elements stored in the $bf$ is the number of sentences on which we will perform $bf$, so we will compare it with the output of the formula. Figure

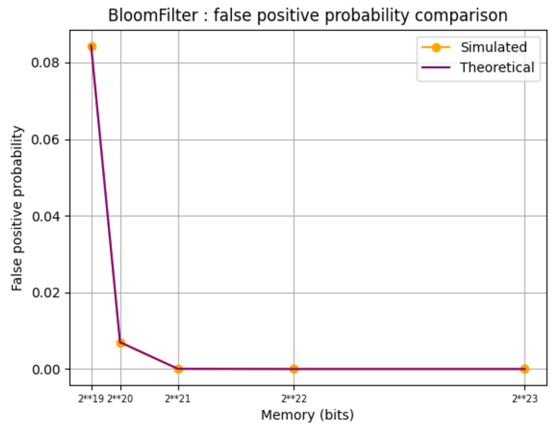

Fig. 3: This plot compares the theoretical and simulated false positive probability in relation to the memory taken into account while using a Bloom Filter and $k = k_opt$.

4 shows the results obtained, as we can see the estimation is pretty accurate.

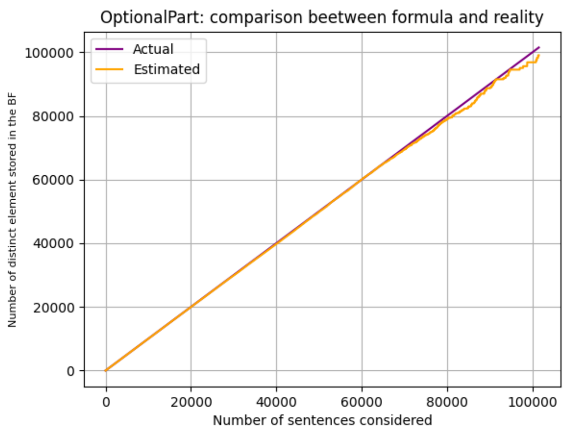

Fig. 4: This plot compares the actual number of distinct elements stored in the $bf$ and the estimated one, $n = 2^{19}$

## IV. Results

Probabilities reported in the following table are taken from the simulation, since are shown while code is running.

| Data structure | Memory kB | P(FP) |
|---|---|---|
| Set of senteces | 13207.448 | 0 |
| Fingerprints set $n = 2^{b\_exp}$ | $2^{35} = 4294967.296$ | $2.954 \times 10^{-6}$ |
| Bit string array $x = n = 2^{20}$ | $2^{20} = 131.072$ | 0.09226 |
| Bloom filter $x = n = 2^{20} = best$ | $2^{20} = 131.072$ | 0.00699 |

We can state that bit string array and bloom filter solutions are the best one since they exploit theoretically the same memory, which is the lowest one in the table, and ensure false positive probability very close to zero for just $n = 2^{20}$ bits.