Université Libre de Bruxelles

INFO-F413 - Data structures & algorithms

Bloom filters

DUDZIAK Thomas : 000542286

December 23, 2022

Contents

1 Introduction

In some applications, it is sometimes useful to only retain the information of the membership of an element to a set and not the complete information about the set. In those cases, we need fast and efficient algorithms to tell us whether an element is contained in a set or not. In this report, we explain and analyze the Bloom Filter probabilistic data structure. From explaining the base concepts, passing by the complexity analysis, to the implementation and the testing of the properties, this report covers the topic of the Bloom Filter data structure.

2 Hashing

Firstly, we need to understand the concept of (non-cryptographic) hashing. Indeed, hashing is a crucial topic in computer science which is used in the bloom filter data structure. A hash function $h(x)$ takes as input a message x, of variable size. As output, the function will provide a message $h(x)$ of constant size (which will often be smaller). In other words, a hash function maps some input of any size to a fixed-size (and generally smaller) universe [2, p. 2]. For example, we could have a hash function $h(x)$ that maps any integer to a smaller range of m integers, say $[0,\ m-1]$ where $m \in \mathbb{N}$. The latter example is what will be used in the implementation of the bloom filter but hashing is not limited from integers to integers.

2.1 Collisions

For some hash function $h(x)$, we could compute $h(x_1)$ and $h(x_2)$ (with $x_1 \neq x_2$) and obtain the same result. This is called a collision. We will see later in this report that this notion of collision has an important role in the bloom filter data structure.

3 A probabilistic data structure

Let X be a set of n elements from a universe $\mathcal{U}$. Bloom filter allows to query whether some element $a \in \mathcal{U}$ is an element that is in X [1, p. 1].

To implement this structure, we create an array B of bits, of size m (where indexes start at 0 and ends at $m-1$). We also need k hash functions which maps an element of the universe to an integer in the following way : $h_0, h_1, ..., h_{k-1} : \mathcal{U} \rightarrow \{0, ..., m-1\}$. For each element $x_i \in X, \forall i \in \{0, ..., |X|-1\}$, the algorithm evaluates every hash function $h_j(x) \forall j \in \{0, ..., k-1\}$ and puts the corresponding bit $B[h_j(x)]$ at 1. This means that, for every element of X, we will put at most (because two hash functions could give the same integer) k bits of B to 1. The final algorithm to initialize the structure with a given set X is the following [1, p. 2] :

Algorithm 1 Create bloom filter

Input: A set X of elements where $\forall x_i \in X, x_i \in \mathcal{U}$
Output: The bloom filter B as an array of bits

```
for h ← 0 to m − 1 do
  B[h] ← 0
end for

for i ← 0 to n − 1 do
  for j ← 0 to k − 1 do
    B[h_j(x_i)] ← 1
  end for
end for

return B
```

If we want to check the membership of a given item, we simply need to check the bits associated to each hash function in B. If every bit that has been checked, then the element is in X, otherwise it is not. The last sentence is true only if we do not face a false positive (see section 4). The algorithm is the following [1, p. 2] :

Algorithm 2 Check membership of y in X

Input: The array of bits B and an element y.
Output: Whether the element is in X or not.

```
for j ← 0 to k − 1 do
  if B[h_j(y)] = 0 then
    return False
  end if
end for

return True
```

4 False positives

In this section, we cover the collisions happening in the hash functions as well as different hash functions returning the same index of the bit array B of the data structure. Indeed, we could have a situation where adding an element to the set modifies a bit of the array which could create an incorrect result for another element. This is called a false positive. Of course, the goal is to reduce as much as possible the amount of false positives. The optimal amount k of hash functions can be computed for given values of m (size of bit array) and n (number of elements of the set X) with the following formula [1, p. 2] :

$$k = \frac{m}{n} \cdot ln(2) \tag{1}$$

The theoretical expectation for the rate of false positive values is the following [1, p. 2]:

$$(1 - e^{\frac{-k \cdot n}{m}})^k \tag{2}$$

However, this false positive rate cannot be reduced to zero in a practical case. Therefore, this data structure is not suitable for exact membership applications due to this false positive problem. Anyhow, if we reduce that rate to a low value, bloom filters are really quick and can have other usages such as sanity checks, filters, ... [1, p. 1].

5 Complexity

In this section, we analyze the time and space complexities of the bloom filter data structure.

5.1 Time complexity

It is easy to deduce that, for any element, we need to check at most k places of the bit array to check the membership of an element in the structure according to the algorithm 2. This algorithm could be executed in less than k checks if an element that is checked is 0. Hence, we can deduce that the time complexity for the membership check is $\mathcal{O}(k)$. To insert an element in the structure, we need to go through the k hash functions and thus, we know the exact time which is bound by k operations. However, k being really small in practice, it is common to say that the complexity both operations are $\mathcal{O}(1)$, which is not really true.

Algorithm	Time complexity
Search (membership check)	$\mathcal{O}(k)$
Insertion	$\mathcal{O}(k)$

5.2 Space complexity

The space complexity of the bloom filter depends on the number of hash functions k as well as the size of the bit array m. We can easily deduce that the space complexity of the bloom filter is $\mathcal{O}(m+k)$. As said in section 5.1 about time complexity, k remains very low and in practice. This is why the space complexity of the bloom filter is usually approximated to :

$$\mathcal{O}(m+k) \approx \mathcal{O}(m+1) \approx \mathcal{O}(m)$$

6 Implementation

In this project, it was required to implement the Bloom Filter data structure. In this section, we cover the technologies used and how to use the program as well as some implementation details. Note that the program is available as appendix to this report as well as on GitHub.

6.1 Technology

The programming language used for the implementation is C#.NET. Indeed, C# is a powerful, high-level, cross-platform programming language. Furthermore, its great readability makes it a great choice for such a project.

6.2 Implementation details

In order to implement the bloom filter data structure, there is the need for a hash function factory. Indeed, if we have a bit array of size m, we need k hash functions that maps any element to an integer in the range $\{0, ..., m-1\}$. To achieve such a task, a library implementing the MurmurHash3 function in .NET has been used. The library is available on NuGet (package manager for .NET technologies).

In the previous paragraph we discussed about mapping any element to some integer in the range $\{0, ..., m-1\}$. The testing program makes use of integers mapped to other integers in the range $\{0, ..., m-1\}$. However, the data structure has been implemented in its own class and supports genericity. This means you can extract the class apart of the probability analysis code easily and reuse this class with any element. By "any element", we mean any C# object as long as this object is marked with the "serializable" tag (note that C# primitive types are objects).

6.3 Program usage

The program given in appendix has the goal to test the probabilities of false positives for the bloom filter data structure. The provided implementation simply requires a traditional C# compiler. In case you do not have one, an executable of the program in release mode is available under *Assignment2/bin/Release/net6.0/*. The program requires **three** arguments. The first argument is the number of hash functions k, the second is the size of the bit array (m) and the last one is the number of elements to put in the set X (n). A last, optional, argument is "–verbose" which gives more logs about the executed tests while executing without that argument only outputs the strict minimum. The following example would run the program with $k = 10, m = 1000, n = 100$ and all logs enabled :

```
./Assignment2.exe 10 1000 100 --verbose
```

Note that a Powershell script is available under *Assignment2/Tests* which allows to run the tests of section 7.

7 Tests & Probabilistic analysis

In this section, we cover the methodology used to test and analyze the false positive rate in practice.

7.1 Methodology

Firstly, let us start by describing the testing methodology used in this project. A "sub-test" consists a random set X of n elements and a set Y of 150 random elements which are not in X. A bloom filter object is instanciated with the given k and m parameters. For each element of the set Y, we check the membership to the bloom filter. If it returns true, then it is a false positive. A complete test consists of 500 of these subsets. This allows to be as accurate as possible. After a test, if f is the number of false positives, we can compute the rate with the following formula :

$$rate = \frac{f}{150 \cdot 500} = \frac{f}{75000} \tag{3}$$

7.2 Results

By fixing two of the three values (k, m and n) and making the third one vary, we can observe how the false positive rate behaves and evolves in practice. In this section, we will analyze the results of all the three different test cases by using the equations 1 and 2.

7.2.1 Results for a variable k

In this test where $m = 1000$ and $n = 100$, the optimal theoretical value of k should be 7, according to the formula in section 4. Thus, we will make k vary from 1 to 15 and see what happens with the false positive rate :

Value of k	Real rate	Expected rate
1	0.0968	0.0961
2	0.0327	0.0335
3	0.0173	0.0179
4	0.0125	0.0122
5	0.0093	0.0098
6	0.0085	0.0088
7	0.0082	0.0086
8	0.0090	0.0089
9	0.0094	0.0096
10	0.0106	0.0108
15	0.0227	0.0242
50	0.7197	0.7251
100	0.9954	0.9959

In this situation, it is clear that increasing k until 7 (which is the expected optimum) gives us the practical optimal solution that minimizes the rate of false positives. Furthermore, we observe on figure 1 that increasing k to use more hash functions is a bad idea ! This is due to the fact that if we increase the number of hash functions, more places of the bit array will be updated and we increases the odds of having a bit array full of 1's. We can easily deduce that, if we do not increase the space (m) and/or the number of elements (n), then, increasing k would be an oversight.

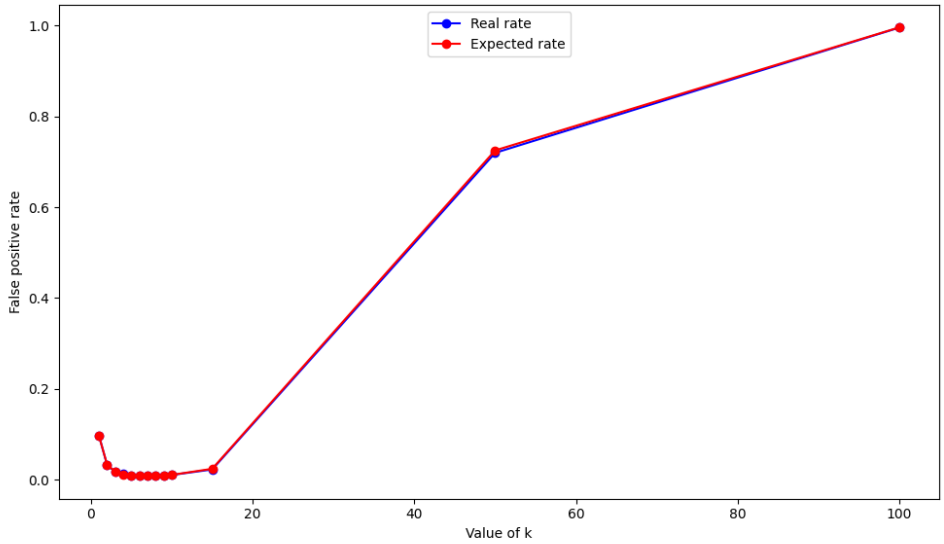

Figure 1: Evolution of the expected and real false positive rate when k varies.

7.2.2 Results for a variable m

For this test, we fix $k = 10$ and $n = 100$. The value of m should be 1443, according to the formula in section 4. We made m vary from a very low value and increased it to a large value of 4000 :

Value of m	Real rate	Expected rate
10	1	1
20	1	1
50	1	0.9999
100	0.9991	0.9996
200	0.9367	0.9377
500	0.2352	0.2409
1000	0.0103	0.0108
1443	0.0012	0.0010
2000	0.000067	0.000096
4000	0	0.000000306

In this case, we simply increase the space of the bit array and, thus, we clearly reduce the odds to have a biased bit array ! We can conclude that increasing the space of the bit array (m) without modifying neither k nor n will make our rate tend to 0. This is clearly visible on figure 2. Unfortunately, this is not something which is good since the goal of the bloom filter is to reduce the used space to the minimum.

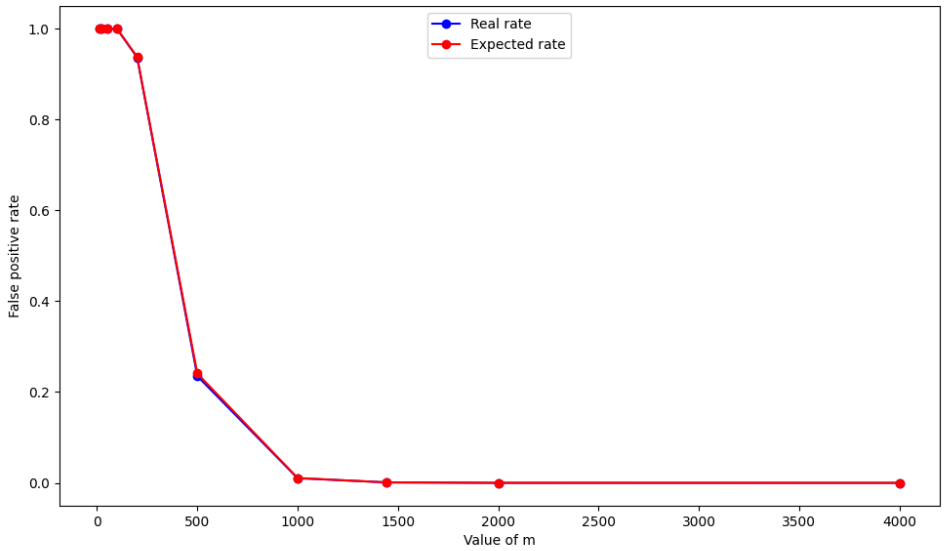

Figure 2: Evolution of the expected and real false positive rate when m varies.

7.2.3 Results for a variable n

For this last test, we fix $k = 10$ and $m = 1000$. The optimal value of n should be 69, according to the formula in section 4. We make n vary with the following values :

Value of n	Real rate	Expected rate
20	0	5.945^{-8}
40	2.666^{-5}	1.853^{-5}
69	0.0009	0.0011
80	0.0027	0.0028
100	0.0105	0.0108
200	0.2356	0.2373
500	0.9339	0.9353

In this scenario, we have an increasing rate when we add new values. Indeed, when adding new values, we increase the odds to have collisions and the odds for a bit to be set to 1 when it shouldn't. We clearly see on figure 3 that, at some point (actually the "optimal" point), adding values has a big impact on the false positive rate. Nevertheless, the idea of such a structure is to store elements and we need to find a trade-off between the rate of false positives and the number of values that are stored. Indeed, storing only a few values would make the structure more robust with very few errors but we would waste space for only a small number of elements.

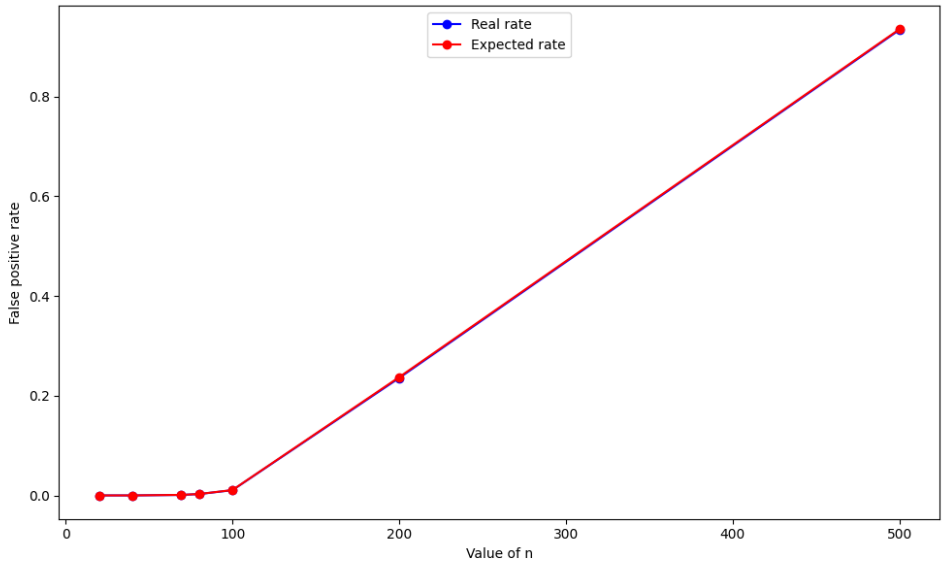

Figure 3: Evolution of the expected and real false positive rate when n varies.

8 Conclusions

In this project, we covered the bloom filter data structure. Firstly, we showed that it is a data structure which is very efficient in terms of time and space complexity. Indeed, the time complexity of insertions and searches are bound by $\mathcal{O}(k) \approx \mathcal{O}(1)$ and the space complexity is bound by $\mathcal{O}(m+k) \approx \mathcal{O}(m)$. However, bloom filters have a certain error rate (false positive rate). After analysis of the behaviour and evolution of that rate in function of different parameters k, m and n, we showed that we need to find a trade-off between the rate of false positives and the used space. We showed that the most interesting and optimal values would be calculated with $k = \frac{m}{n} \cdot ln(2)$.

Such a data structure is not suitable for exact membership checks due to its non-zero rate of false positives. Anyhow, its very low space consumption as well as its low time complexity allows for other real-life applications such as sanity checks, filters, ...

9 Bibliography

[1] J. Erickson. Foundations for category theory. `https://jeffe.cs.illinois.edu/teaching/algorithms/notes/06-bloom.pdf`. Accessed: 19-12-2022.

[2] C. Estébanez, Y. Saez, G. Recio, and P. Isasi. Performance of the most common non-cryptographic hashfunctions. *Software: Practice and Experience*, 44(6):681–698, 2014.