Bloom Filter Implementation

INFO-F413

Professor : Jean Cardinal

Alex Bataille

October 30, 2025

Contents

1 Introduction

Bloom filters are space-efficient probabilistic data structures for set membership tests that allows false positives but no false negatives. A Bloom filter stores a compact bitset and a small number of hash functions to probabilistically record membership of items. This report goes through the implementation of such a data structure using the programming language C++ and the study of the rate of false positive.

2 Implementation

This implementation uses 3 main C++ classes :

- HashFunction
- HashFunctionFactory
- BloomFilter

Note that this implementation is extensible to other data types than std::string thanks to the use of C++ templates.

2.1 Hash functions

A hash function has 2 attributes being size_t m and int d. These are, respectively, the size of the bitset to which the hash function must return the index and a "unique" identifier that will differentiate the hash function from the others in the Bloom filter. This class implements a method getIndex(input) that returns the digest of input.

2.1.1 Hashing technique

The standard hash function from STL is used. A first way to use it would be :

```
int getIndex(T input) const {
    return std::hash<T>{}(input) % m;
}
```

However, one may see that this does not fit well with our implementation of the hash function factory. Indeed, different instances of hash functions would always return the same digest, therefore the idea behind the k hash functions is lost.
This justifies the second attribute d of the HashFunction class. Another way to code getIndex is thus the following :

```
int getIndex(T input) const {
    return (
        std::hash<T>{}(input) +
        std::hash<std::string>{}(std::to_string(this->d))
        ) % m;
}
```

2.2 Hash function factory

The hash function factory class has a method createHashFunction(size_t m) that returns new hash functions mapping in the range $\{0, \ldots, m-1\}$.

2.3 Bloom filter

The Bloom filter class has the following attributes :

- size_t m
- size_t k
- vector of hash functions
- a hash function factory
- a vector of booleans (bitset)

It fills the vector of hash functions at initialization using the hash function factory, and fills the bitset with 0s. The methods insert(element) and isInserted(element) follows the Bloom filter logic seen in the lectures.

3 False positive rate

3.1 Theoretical result

As said earlier, this data structure allows for false positive to occur. The theoretical number of hash functions k for a Bloom filter that minimizes this false positive rate is (cf. proof in lecture notes)

$$k = \left\lceil \frac{m}{n} \cdot \ln(2) \right\rceil$$

3.2 Test of theoretical result

To validate the theoretical formula for the optimal number of hash functions, we conducted experiments by varying the parameters m, n, k. The range of test for k goes from 1 to 50 since the rate of false positives seems to grow quickly afterwards. The results are shown in the following figures.

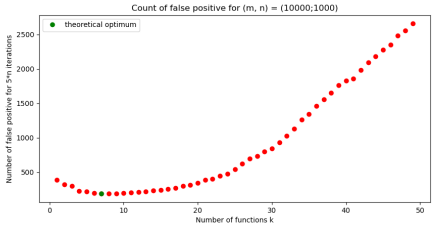

Figure 1: Number of false positives in function of k for $(m, n) = (10000, 1000)$

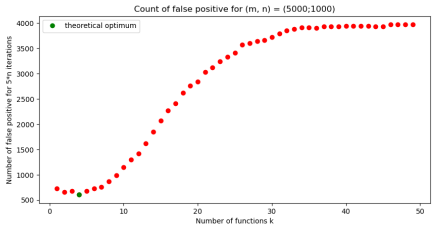

Figure 2: Number of false positives in function of k for $(m, n) = (5000, 1000)$

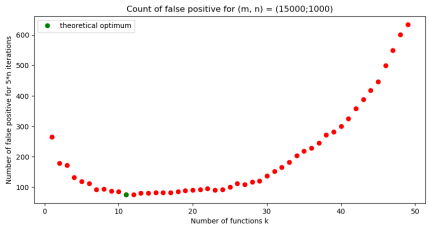

Figure 3: Number of false positives in function of k for $(m, n) = (15000, 1000)$

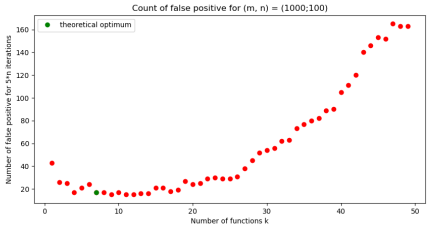

Figure 4: Number of false positives in function of k for $(m, n) = (1000, 100)$

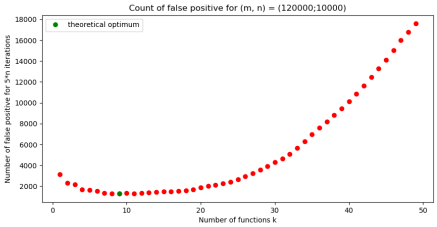

Figure 5: Number of false positives in function of k for $(m, n) = (120000, 10000)$

4 Conclusion

The experimental results confirm the theoretical prediction regarding the optimal number of hash functions in a Bloom filter. The observed false positive rates closely align with the expected values, validating the implementation and the underlying mathematical model. Future work could explore the impact of alternative hash functions or dynamic resizing strategies to further optimize performance.