Bloom Filter Implementation

INFO-F413

Professor : Jean Cardinal

Alex Bataille

November 6, 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++, along with 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 (B)

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 (Adam Kirsch and Michael Mitzenmacher. "Less hashing, same performance: Building a better Bloom filter". In: *Random Structures & Algorithms* 33.2 (2008), pp. 187–218. DOI: 10.1002/rsa.20208. URL: https://www.eecs.harvard.edu/~michaelm/postscripts/rsa2008.pdf):

```
int getIndex(T input) const {
    size_t h1 = std::hash<T>{}(input);
    size_t h2 = std::hash<std::string>{}(std::to_string(this->d));
    return (h1 + d * h2 + d*d) % m;
}
```

2.2 Hash function factory (A)

The hash function factory class has a method createHashFunction(size_t m) that returns new hash functions mapping in the range $\{0, \ldots, m-1\}$. It makes each function unique thanks to a counter that is incremented each time a function is created.

2.3 Bloom filter (C)

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.

3.2.1 Operating procedure

For each k and with fixed m and n, we first insert n elements in the Bloom Filter and then count the number of false positives for a certain amount of iterations (D). We then plot those results using a python script (E) and compare it with the theoretical value of k (x value of the green point).

3.2.2 Plots

The results are shown in the following figures.

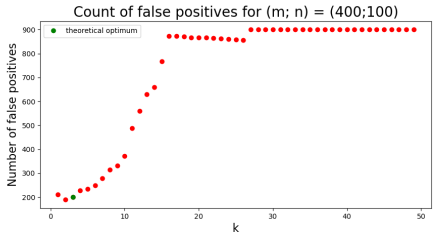

Figure 1: Number of false positives depending on k for $(m, n) = (400, 100)$

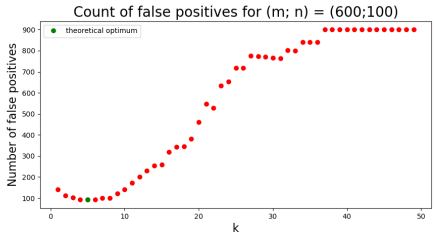

Figure 2: Number of false positives depending on k for $(m, n) = (600, 100)$

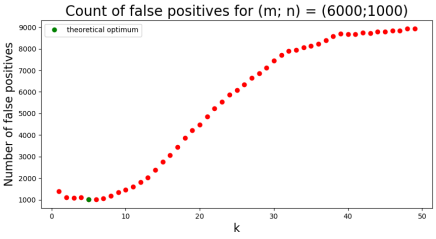

Figure 3: Number of false positives depending on k for $(m, n) = (6000, 1000)$

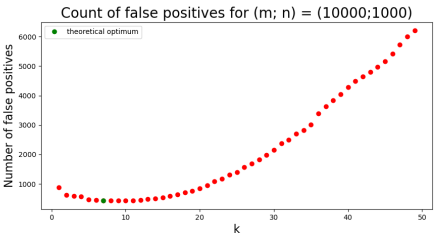

Figure 4: Number of false positives depending on k for $(m, n) = (10000, 1000)$

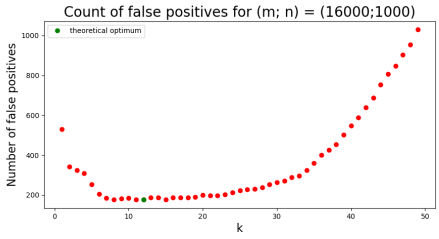

Figure 5: Number of false positives depending on k for $(m, n) = (16000, 1000)$

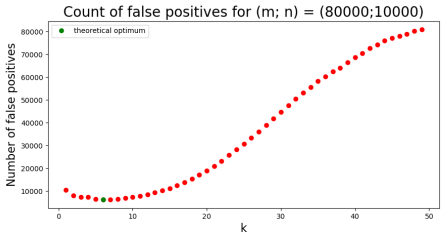

Figure 6: Number of false positives depending on k for $(m, n) = (80000, 10000)$

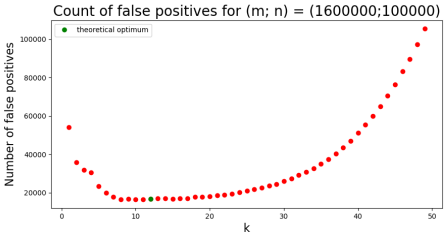

Figure 7: Number of false positives depending on k for $(m, n) = (1600000, 100000)$

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.

The code used in the project can be found in this appendix.

A HashFunctionFactory.hpp

```cpp
#pragma once
#include <memory>
#include "HashFunction.hpp"

class HashFunctionFactory {
    protected:
        size_t it = 0;
    public:
        HashFunctionFactory() = default;
        template<typename T>
        std::unique_ptr<HashFunction<T>> createHashFunction(size_t m) {
            return std::make_unique<HashFunction<T>>(m, this->it++);
        }
};
```

B HashFunction.hpp

```cpp
#pragma once
#include <functional>
#include <string>

template<typename T>
class HashFunction {
    protected:
        size_t m;
        int d;
    public:
        HashFunction(size_t m, int d) : m(m), d(d) {}
        ~HashFunction() = default;

        int getIndex(T input) const {
            size_t h1 = std::hash<T>{}(input);
            size_t h2 = std::hash<std::string>{}(std::to_string(this->d));
            return (h1 + d * h2 + d*d) % m;
        }

};
```

C BloomFilter.hpp

```cpp
#pragma once
#include "HashFunctionFactory.hpp"
#include <vector>

template<typename T>
class BloomFilter {
    protected:
        size_t m;
        size_t k;
        std::vector<std::unique_ptr<HashFunction<T>>> hash_fcts;
        HashFunctionFactory hash_fcts_fact;
        std::vector<bool> bit_vector;
    public:
        BloomFilter(size_t m, size_t k) : m(m), k(k) {
            this->hash_fcts_fact = HashFunctionFactory();
            for (size_t i = 0; i < k; i++) {
                hash_fcts.push_back(this->hash_fcts_fact.
    createHashFunction<T>(m));
            }
            for (size_t j = 0; j < m; j++) bit_vector.push_back(
    false);
        }
        ~BloomFilter() = default;

        bool insert(T element) {
            bool already_ins = isInserted(element);
            for (size_t i = 0; i < this->k; i++) {
                this->bit_vector.at(this->hash_fcts.at(i)->getIndex
    (element)) = true;
            }
            return !already_ins;
        }

        bool isInserted(T element) {
            for (size_t i = 0; i < this->k; i++) {
                if (!this->bit_vector.at(this->hash_fcts.at(i)->
    getIndex(element))) return false;
            }
            return true;
        }
};
```

D main.cpp

```cpp
#include <iostream>
#include <cmath>
#include <fstream>
#include "BloomFilter.hpp"

#define N_K_TEST 50
int count_coll(float m, float n, int k = 0, int it = 0);
void test_coll(std::string path);

int main() {
    test_coll("false_pos.txt");
    return 0;
}

int count_coll(float m, float n, int k, int it) {
    BloomFilter<std::string> *string_bloom;
    if (k == 0) k = std::ceil(m/n*std::log(2));
    if (it == 0) it = 10*n;
    string_bloom = new BloomFilter<std::string>(m, k);
    int n_false_pos = 0;

    for (int i = 0; i < n; i++) {
        string_bloom->insert(std::to_string(i));
    }

    for (int i = n; i < it; i++) {
        if (string_bloom->isInserted(std::to_string(i))) {
            n_false_pos++;
        }
    }
    delete string_bloom;
    return n_false_pos;
}

void iteration_coll(int m, int n, std::ofstream *output_file) {
    *output_file << "(" << m << ";" << n << ")";
    for (int j = 1; j < N_K_TEST; j++) {
        *output_file << "[" << j << "," << count_coll(m, n, j) << "
    ]" << ",";
    }
    *output_file << std::endl;
}

void test_coll(std::string path) {
    std::ofstream output_file (path);
    if (output_file.is_open()) {
        std::vector<int> ns = {100, 1000, 10000, 100000};
        std::vector<int> bs = {4, 6, 8, 10, 12, 16, 20};
        for (int n : ns) {
            for (int b : bs) {
                int m = b * n;
                iteration_coll(m, n, &output_file);
            }
        }
    } else {
        std::cerr << "Error opening output file";
    }
    output_file.close();
}
```

E plot.py

```
import matplotlib.pyplot as plt
from numpy import ceil, log

def extract_points(line):
    x_arr = []
    y_arr = []
    pos = 0
    while True:
        if line[pos] == "[":
            c = line.index(",", pos)
            end_i = line.index("]", pos)
            x_arr.append(int(line[pos+1:c]))
            y_arr.append(int(line[c+1:end_i]))
            pos = end_i
        else:
            pos += 1
        if pos == len(line):
            break

    return x_arr, y_arr

def make_graph(line):
    i = line.index("(")
    j = line.index(")")
    separator = line.index(";")
    title = "Count of false positive for (m, n) = " + line[i:j+1]
    m = int(line[i+1:separator])
    n = int(line[separator+1:j])
    k = int(ceil(m/n*log(2)))
    x_values, y_values = extract_points(line)
    min_th = y_values[k-1]
    plt.figure().set_figwidth(10)
    plt.plot(x_values, y_values, 'ro')
    plt.xlabel("k", fontsize=14)
    plt.ylabel("Number of false positives", fontsize=14)
    plt.plot(k, min_th, 'go', label="theoretical optimum")
    plt.legend()
    plt.title(title, fontsize=20)
    plt.show()

if __name__ == "__main__":
    output_file_path = "false_pos.txt"
    with open(output_file_path, "r") as output_file:
        lines = output_file.read().splitlines()
        for line in lines:
            make_graph(line)
```

References

Kirsch, Adam and Michael Mitzenmacher. "Less hashing, same performance: Building a better Bloom filter". In: *Random Structures & Algorithms* 33.2 (2008), pp. 187–218. DOI: 10.1002/rsa.20208. URL: https://www.eecs.harvard.edu/~michaelm/postscripts/rsa2008.pdf.