Jihočeská univerzita
v Českých Budějovicích
University of South Bohemia
in České Budějovice

University of South Bohemia

Faculty of Science

Artificial Intelligence and Data Science, M.Sc.

Designing a Bloom Filter for a Target False Positive Rate

Submitted for the course **Information Theory**

Professor: Kaštovský Jan prof. RNDr. Ph.D.

Submitted by:	María Isabel Sánchez-O'Mullony Martínez
Student ID:	B24763
Date:	December 3, 2025

Supervisor: Kaštovský Jan prof. RNDr. Ph.D.

Declaration

I declare that I have written this report by myself and have only used the sources and aids mentioned, and that I have marked direct and indirect citations as such. This report has not been submitted prior for any other examination.

I agree that the results of this study work / report may be used free of charge for research and lecturing purposes.

Contents

1 Introduction **3**

2 Algorithm Description **3**

2.1 Insertion 4

2.2 Lookup 4

3 Implementation **4**

3.1 Parameter Selection 4

3.2 Hash Functions 4

3.3 Insertion Function 5

3.4 Lookup Function 5

4 Code Structure **5**

4.1 Testing 5

5 Results **6**

A Bloom Filter **7**

B Experiments **8**

References 10

1 Introduction

A Bloom Filter (BF) is a simple, space-efficient, randomized data structure for representing a set in order to support approximate membership queries [1]. Instead of storing the actual elements, the filter stores only the values of multiple hash functions, enabling significant memory savings. Because of this design, Bloom Filters can represent keys with only a small number of bits per element, and their space usage does not depend on the size of the individual keys [2].

Bloom Filters offers several interesting properties. They provide a compact and probabilistic representation of a set that may reasult in false positives, claiming an element to be part of the set when it is not, but never false negatives, reporting an inserted element to be absent from the set any inserted element is always reported as present. A fixed-size Bloom Filter can, in principle, support insertion of an arbitrarily large number of elements; however, each insertion increases the probability of false positives [2, 3]. Insertions never fail, but the accuracy of the filter degrades as it becomes saturated.

The key trade-off is that in exchange for the possibility of returning false positives, Bloom Filters achieve large memory savings and fast operations compared to traditional data structures such as hash tables [4]. Another key point, is that a BF does not support deletion of elemenets once inserted, because clearing bits may revome information about other elements.

The objective of this project is to design and implement a Bloom Filter that automatically selects the optimal bit-array size (m) and number of hash functions (k) required to achieve a target false-positive rate (FPR) (p), given an expected number of inserted elements (n).

2 Algorithm Description

A Bloom Filter supports two primary operations: inserting an element and querying whether an element is in a set or not. It provides space-efficient membership testing at the cost of allowing false positives.

A Bloom Filter representing a set $S = \{x_1, \ldots, x_n\}$ of n items is implemented using a bit array of length m, with all bits initially set to 0. The filter relies on k independent hash functions $h_i(x)$, where $1 \leq i \leq k$, each mapping an element $x \in S$ uniformly to one of the positions in the range $\{1, \ldots, m\}$ [3].

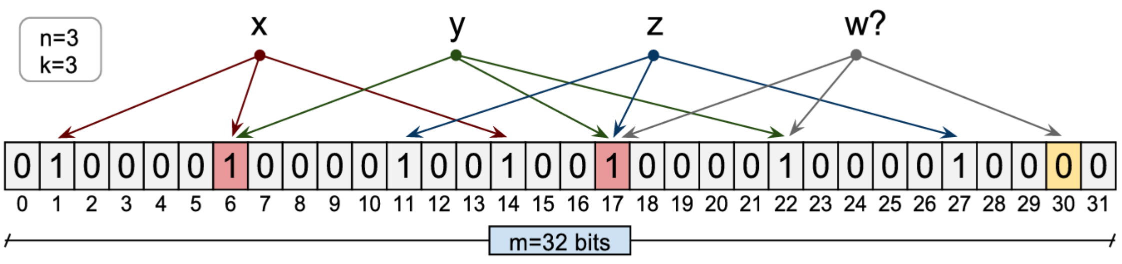


Figure 1: Overview of a Bloom Filter [3].

2.1 Insertion

To insert an element x into the BF, we start out empty with all bits set to 0, then we calculate all k hash functions. Each hash function outputs an index in the bit array, and the corresponding bit is set to 1. If a bit is already 1, it remains unchanged. After processing all k hash outputs, the Bloom Filter reflects the presence of x probabilistically [3, 4].

2.2 Lookup

To check whether an element y is in the set, we again use the same k hash functions for y. If any of the corresponding bits in the array is 0, then y is guaranteed not to be in the set. If all k bits are 1, the Bloom Filter reports that y may be in the set. In this case, the answer is possibly correct, but false positives can occur because the bits may have been set by other elements [3].

3 Implementation

This implementation was written in Python. It consists of three main parts: parameter selection, hashing and the core operations, insertion and lookup.

3.1 Parameter Selection

To achieve the desired false-positive probability, the implementation computes the BF parameters using the standard optimal formulas. The optimal number of hash functions can be computed by the forumla:

$$k = \frac{m}{n} \ln 2,$$

where k is the optimal number of has functions, n is the estimated number of inserted objects and m is the bit- array size. m can also be calculated for an expected number of elemenets n with a desired false positive probability p:

$$m = -\frac{n \ln p}{(\ln 2)^2},$$

The values of m and k are rounded to the nearest integers, with the constraint that $k \geq 1$. The computation is performed at initialization time to ensure that the filter is configured appropriately before any elements are inserted.

A bit array with size m is then created and initialized to zero. The library `bitarray` [5] was used to make it easier, helping with simplicity but at the same time allowing high performance.

3.2 Hash Functions

According to A. Kirsch and M. Mitzenmacher [6], rather than implementing k independent hash functions from scratch, only two hash functions, $h_1(x)$ and $h_2(x)$, are enough in Bloom Filters, without increasing the false positive probability and simplifying the implementation. Therefore, we can use the following standard double-hashing technique:

$$h_i(x) = (h_1(x) + i \cdot h_2(x)) \bmod m, \quad i = 0, \ldots, k-1$$

To choose which hash functions to use we need to take into account that they should be independent hash functions, the values should be uniformly distributed and efficient/fast. In my case, I chose to use MD4 and SHA256, designed for security and good distribution, they are also slower, but their randomness make them difficult to predict and secure.

For this implementation, the performance is not critical, but if it was, we could change these hash functions for non-cryptographyc, faster hash functions like MurmurHash or xxHash.

It is crucial that h_2 is not zero for any item to avoid conflicts with the modulus operation, the reason is that adding zero repeatedly won't produce distinct values. If and item in h_2 is zero, the generated indexes will be the same across iterations.

3.3 Insertion Function

Once we have the parameters and the base has functions, we can define the operations for insertion and lookup. Following the theory in section 2.1, to insert an element x in the BF, we can calculate k indexes or hash locations with the double hash function, then set each corresponding bit in the bit-array to 1.

It is not required to check wether a bit is already set or not. There is no possibility of insertion failure because bits may be set multiple times without conflict.

3.4 Lookup Function

According to section 2.2, to test if an element y may belong to the set, the same k hash locations are computed. For each index generated by the double hashing, we check if the bit at the bit-array is set to 1. If all the bits are 1, the item is possibly in the set, but if any bit is 0, then we know that it definitely does not belong to the set.

4 Code Structure

The implementation consists on two files, `bloom_filter.py` and `run_experiment.py` (the whole code can be found in the appendix). The first one with all the logic for the Bloom Filter and the second one for testing the outputs of different given values.

The code was done by creating a specific class for the Bloom Filter, its parameters are based on the values n and p provided by the user. Then the other required parameters are calculated with these values (m and k) and a bit-array with size m is initialized to 0 as mentioned in section 3.1.

This class contains two functions, `insert(element)` and `lookup(element)`, which do exactly what their names say, insert an element into the filter and test whether the filter might contain the element as expalined in sections 3.3 and 3.4.

4.1 Testing

We are testing two target values for the False Positive Rate, 1% and 0.1%. The testing is implemented in the file `run_experiment.py` (check listing 2). In both cases, $n = 10000$ is used, any other number could have been chosen.

For both experiments n elememts were inserted into the BF, each item is a string formed by concatenating "element" with an index (e.g., "element_0", "element_1", ...). The specific string format does not matter — it could be any unique string.

Once the elements are inserted, we can estimate the actual FPR of the Bloom Filter to check if it aligns with the theoretical calculations.

To perform this estimation, a function `estimate_fpr(bf, trials)` was defined, which takes the Bloom filter to test and the number of trials. In our experiments, we used 20,000 trials.

The function works by generating random strings of length 10, consisting of random letters. These strings are very unlikely to be the same as any of the inserted elements before. Each string is then checked with the Bloom filter.

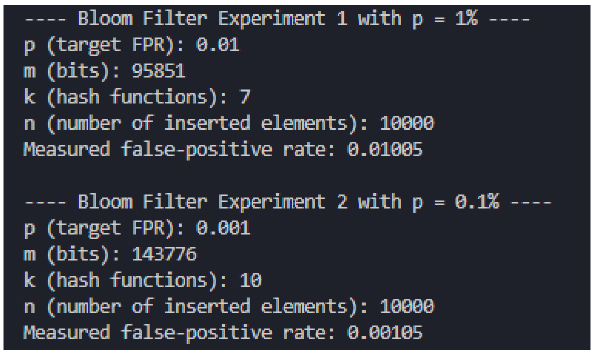
A false positive occurs when the BF indicates that a random string is present, even though it was never inserted. Since the random string generated are very unlikely to match, if we get any positive is most likely to be a false positive. This procedure allows us to empirically measure the Bloom filter's false positive rate and compare it with theoretical expectations. The empirical FPR is computed with the following formula:

$$\hat{p} = \frac{\text{number of false positives}}{\text{total tested}}$$

Note that here the total tested should be equal to the number of trials we specified.

5 Results

As mentioned in section 4.1, experiments were performed for two target false positive rates: 1% and 0.1%. In each case, we inserted $n = 10000$ elements into the Bloom Filter and measured the FPR using 20000 random trials. These were the results we got after running the program:


```
---- Bloom Filter Experiment 1 with p = 1% ----
p (target FPR): 0.01
m (bits): 95851
k (hash functions): 7
n (number of inserted elements): 10000
Measured false-positive rate: 0.01005

---- Bloom Filter Experiment 2 with p = 0.1% ----
p (target FPR): 0.001
m (bits): 143776
k (hash functions): 10
n (number of inserted elements): 10000
Measured false-positive rate: 0.00105
```

Figure 2: Comparison of theoretical and observed false positive rates.

The observed FPR values are very close to the target rates, showing that the Bloom filter is behaving as expected. The results are not exactly the same as the target rates, this was expected due to the random sampling.

A Bloom Filter

The following is the complete code used for the Bloom filter experiments.

Listing 1: Complete code for Bloom filter experiments

```python
import math
import hashlib
import random
import string
from bitarray import bitarray

# optimal Bloom filter parameters m and k for n and p
def compute_parameters(n, p):
    m = int(math.ceil(-n * math.log(p) / (math.log(2) ** 2))) # size of bit array
    k = int(round((m / n) * math.log(2))) # number of hash functions to be used

    return m, k


# h_i(x) = h1(x) + i * h2(x) mod m
def double_hashing(item, m, k):
    indexes = []

    # To be able to use the hash functions we need to convert first to bytes
    if isinstance(item, str):
        item = item.encode("utf-8")

    h1 = int(hashlib.sha256(item).hexdigest(), 16)
    h2 = int(hashlib.md5(item).hexdigest(), 16)
    if h2 == 0: h2 = 1 # make sure that h2 is not 0

    for i in range(k):
        indexes.append((h1 + i * h2) % m)

    return indexes


class BloomFilter:
    def __init__(self, n, p):
        self.n = n
        self.p = p
        self.count = 0

        self.m, self.k = compute_parameters(n, p)
        self.bit_array = bitarray(self.m)
        self.bit_array.setall(0) # initialize all bits to 0

    def insert(self, element):
        for idx in double_hashing(element, self.m, self.k):
            self.bit_array[idx] = 1
        self.count += 1

    def lookup(self, element):
        return all(self.bit_array[idx] for idx in double_hashing(
            element, self.m, self.k))


# Query random elements NOT inserted into the Bloom filter
# and estimate the empirical false-positive rate
def estimate_fpr(bf, trials=10000):
    false_positives = 0
    tested = 0

    for _ in range(trials):
        # random string not likely to be inserted
        x = ''.join(random.choice(string.ascii_letters) for _ in range
            (10))

        if bf.lookup(x):
            false_positives += 1
        tested += 1

    return false_positives / tested


def estimation(bf, n):
    p = (1 - (1 - (1 / bf.m))**(bf.k * n)) ** bf.k
    return p
```

B Experiments

The following is the complete code used of how to run the experiments.

Listing 2: Complete code for running the experiments

```
from bloom_filter import BloomFilter, estimate_fpr, estimation

n1 = 10000   # number of items expected to be insert in filter
p1 = 0.01    # False Positive probability (1%)

bf1 = BloomFilter(n1, p1)

# Insert n items
for i in range(n1):
    bf1.insert(f"elemet_{i}")
```

```python

# Estimate FPR
measured_fpr = estimate_fpr(bf1, trials=20000)

print("---- Bloom Filter Experiment 1 with p = 1% ----")
print(f"p (target FPR): {p1}")
print(f"m (bits): {bf1.m}")
print(f"k (hash functions): {bf1.k}")
print(f"n (number of inserted elements): {bf1.count}") # check that n elements have been inserted
print(f"Measured false-positive rate: {measured_fpr:.5f}\n")

# ============================================

n2 = 10000
p2 = 0.001     # False Positive probability (0.1%)

bf2 = BloomFilter(n2, p2)

for i in range(n2):
    bf2.insert(f"elemet_{i}")

measured_fpr = estimate_fpr(bf2, trials=20000)

print("---- Bloom Filter Experiment 2 with p = 0.1% ----")
print(f"p (target FPR): {p2}")
print(f"m (bits): {bf2.m}")
print(f"k (hash functions): {bf2.k}")
print(f"n (number of inserted elements): {bf2.count}")
print(f"Measured false-positive rate: {measured_fpr:.5f}")
```

References

[1] D. Guo, J. Wu, H. Chen, and X. Luo, "Theory and network applications of dynamic bloom filters," in *Proceedings IEEE INFOCOM 2006. 25TH IEEE International Conference on Computer Communications*. IEEE, 2006, pp. 1–12.

[2] GeeksforGeeks. (2025) Bloom filters - introduction and implementation. [Online]. Available: https://www.geeksforgeeks.org/python/bloom-filters-introduction-and-python-implementation/

[3] S. Tarkoma, C. E. Rothenberg, and E. Lagerspetz, "Theory and practice of bloom filters for distributed systems," *IEEE Communications Surveys & Tutorials*, vol. 14, no. 1, pp. 131–155, 2011.

[4] ByteMonk, "Bloom filters, hashtable, system design," YouTube video, May 2024, accessed: 2025-11-26. [Online]. Available: https://www.youtube.com/watch?v=GT0En1dGntY

[5] I. Schnell, "bitarray: efficient arrays of booleans," https://pypi.org/project/bitarray/, 2025, version 3.8.0, accessed 2025-12-02.

[6] A. Kirsch and M. Mitzenmacher, "Less hashing, same performance: Building a better bloom filter," in *European Symposium on Algorithms*. Springer, 2006, pp. 456–467.