Department of Computer Science and Engineering
National Institute of Technology Srinagar
Hazratbal, Srinagar, Jammu and Kashmir - 190006, India.

ASSIGNMENT REPORT

Data Structures

Submitted by

Mohammad Rabi-ul-Hadi (Group 1)
2022BCSE007
II B.Tech. CSE (4th Semester – Section A)

Submitted to

Dr. VENINGSTON K
Assistant Professor
Department of Computer Science and Engineering
National Institute of Technology Srinagar Hazratbal,
Srinagar, Jammu and Kashmir - 190006, India.

SPRING 2024

Abstract

Bloom filter is a probabilistic data structure which is used to check if an element is present in a set or not. Though not totally accurate, it is super-efficient in terms of speed and space.

It was invented by Howard Bloom in the 1970s. Main motivation behind this data structure was other error-free hashing methods being too space hungry. Bloom-filter may give false positives but never false negatives, which means it tells either element is possibly in set or not in set.

Bloom filter doesn't save the element itself but the information about the element which is relatively small and uses that information to query it. How it works is, it passes a single element through the "k" number of hash-functions and values returned by those hash functions will be positions in the bitset of size "m" (bloom-filter) to be turned on. Here hash functions should be uniformly distributed and independent. Here, 'm' is directly proportional to 'k'. Now, the same values might be returned by these hash functions for two elements, leading to false positives.

Here we will design a bloom filter to check whether a given URL is malicious or not by checking if it is present in a dataset of malicious URLs.

Table of Contents

Introduction

A Bloom filter offers a trade-off, while it is capable of producing false positives, it guarantees no false negatives. This means that if the filter indicates an element is not present, it is definitely not in the set; however, if it indicates the element is present, there is a small probability that this is incorrect. This probabilistic nature is acceptable in many scenarios where the slight chance of a false positive is outweighed by the significant increase in speed and memory efficiency.

Hashing: A hash of an object s is an object H(s) (usually an integer) that satisfies the following conditions: $s \equiv t \Longrightarrow H(s)=H(t)$

$H(s)=H(t) \Longrightarrow s \equiv t$ with high probability

The Bloom filter works by hashing an element through multiple hash functions, each yielding a position in a bit array of fixed size. The corresponding bits are then set to 1. When querying, the same hash functions are applied to the element, and the bits at the resultant positions are checked. If all are set to 1, the element is assumed to be in the set; otherwise, it is not.

Given the growing prevalence of malicious URLs and the critical need for their swift identification, this project leverages the efficiency of the Bloom filter to design a system for detecting such URLs. By comparing URLs against a dataset of known malicious URLs, our Bloom filter-based approach aims to provide a quick and memory-efficient method for identifying potential threats.

In the following sections, we will delve deeper into the design and implementation of our Bloom filter, exploring its performance, limitations, and practical applications in cybersecurity. This project not only underscores the Bloom filter's utility in handling large datasets but also highlights its relevance in contemporary challenges such as cybersecurity Now accuracy of bloom filter can be calculated by simple formula:

$(1 - e^{-(kn/m)})^k$.

k- number of hash functions n-
number of elements m- size of
bitset
We will use this formula in later parts

Software Requirements

1. Functional Requirements:

1.1. **Check URLs:** Filter should quickly check if the given URL is malicious or not.

1.2. **Minimal Storage:** Filter shouldn't take too much space; it would beat the purpose.

1.3. **Hash Functions:** There should be more than one of them. Moreover, they should be uniform and independent i.e they should be efficient enough.

2. Performance Requirements:

2.1. **Speed:** The filter should check URLs with minimum latency

2.2. **False Positives:** There should be minimum amount of false positive

3. Non-Functional Requirements:

3.1. **Ease of use:** Filter should be easy to use and integrate

3.2. **Robustness:** Filter should handle edge cases and unusual inputs gracefully.

Methodology

There are many parts in designing a bloom filter, we need to decide the “k” which will affect our error i.e. false positives. Although false positives are possible, they can be reduced to 1% by using 10 bits per element independent of size or number of elements in the set (*"An Improved Construction for Counting Bloom Filters" (2006)*).
Using 10 bits per element means using 10 hash functions which will give 10 bits in bloom filter to be set on. But along with this, these hash functions should be efficient enough to actually map different strings with the least amount of overlap.

We decided to use three hash functions for simplicity without compromising much accuracy and also for experimental purposes to test limits of bloom-filter. These hash functions had to be uniform i.e. the number of keys mapped to each slot has to be equal and independent i.e.
output of one key doesn’t depend upon other key

We decided to implement this in C++ for simplicity and speed. We just have a basic Bloom filter class which has member variables:

- **p:** a constant; small prime number - *(31) [integer]*
- **m:** another constant, a large prime number - *(1e9 + 9) [integer]*
- **size:** to keep track of how many websites have been read - *[integer]*
- **filename:** to save name of file of DB of malicious URLs - *[string]*
- **bitset:** basically our bloom-filter to keep track of bits - *[bitset]*

We have a constructor which will initialize size with zero and with read file (which contains the list of URLs) with respective name provided using standard fstreams, add the URL string using add function to bloom filter and increment size with 1 with every new string read.

Add function begins by passing the string through three hash functions, which are:

1. **Polynomial Hash:**
   The polynomial hash of a string is defined as:

   $H(s) = (S_0 + S_1 \times p + S_2 \times p^2 + \ldots + S_{n-1} \times p^{n-1}) \; mod \; M$ where

   S_i : is the i-th character of the string,

   p : is an arbitrary integer larger to or equal than the order of the alphabet

   M: is an arbitrary modulus.

```cpp
unsigned long long polynomial_hash(const string &s, long long p, long long m)
{
    unsigned long long hash = 0;
    long long p_pow = 1;
    for (char c : s)
    {
        hash = (hash + (c - 'a' + 1) * p_pow) % m;
        p_pow = (p_pow * p) % m;
    }
    return hash;
}
```

2. **DJB2:**
   The DJB2 hash function was created by Dan Bernstein and is known for its simplicity and effectiveness.

$$H(S) = H'(5381, S_0) + H'(H'(0), S_1) + \ldots$$

and

$$H'(x, y) = ((x \times 33) + y)$$

```cpp
unsigned long long djb2(const string &s)
{
    unsigned long long hash = 5381;
    for (char c : s)
    {
        hash = ((hash << 5) + hash) + c;  // hash * 33 + c
    }
    return hash;
}
```

Function starts with the hash variable set to the number 5381. It then iterates the given array of characters str and performs the following operations for each character:

1. Multiply the hash variable by 33.

2. Add the ASCII value of the current character to it.

After iterating through the whole array, it returns the value held by hash. We use "<<" shift + hash instead of multiplying with 33 because it is faster this way

3. **SDBM:**

This algorithm was created for sdbm (a public-domain reimplementation of ndbm) database library.

$H(S) = H'(0, S_0) + H'(H'(0), S_1) + \ldots$

and

$H'(x, y) = ((x \times 65599) + y)$

It is similar to DJB2 hash but uses 65599 as constant.

```
unsigned long long sdbm(const string &s)
{
    unsigned long long hash = 0;
    for (char c : s)
    {
        hash = c + (hash << 6) + (hash << 16) - hash;
    }
    return hash;
}
```

I made an experimental test with all hash functions to test how scattered they are:

```
G1-bloomfilter./hashtest
String 1: BloomFilter
String 2: BloomFilter2
Polynomial Hash parameters: p=31, m=1000000007
Polynomial Hash String 1: 564586218
Polynomial Hash String 2: 93837496
DJB2 Hash String 1: 13818915757414958212
DJB2 Hash String 2: 13302362225664382262
SDBM Hash String 1: 6087045120976596475
SDBM Hash String 2: 5850671426797883639
```

Once all the values from these functions have been returned, we mod them with the size of the bitset to ensure size safety and avoid runtime errors and set corresponding bits. We use a similar method to test the given URL too using contains which utilizes the "test" method of STL bitset. Then there is a test method which can read from the file and check if URLs contained in it are malicious or not.

Our dataset of malicious URLs had 20,752 URLs and benign one had 57,311 URLs of which it gave 10 false positives

```
G1-bloomfilter./main

--- Bloom Filter Menu ---
Bitset size: 1000001
Hash functions used: Polynomial Rolling, DJB2, SDBM
1. Test a file
2. Test a website string
3. Exit
Enter your choice: 1
Enter the file name to test: benign.csv
```

```
Total Positives: 10
Total Negatives: 57301

Malicious URLs:
https://www.cultures.mtycms.dev.inovestor.com/en/locations/quebec/place-du-centre.aspx
https://www.dictionary.reference.com/browse/duchesses
https://www.en.wikipedia.org/wiki/Washington_State_Route_169
https://www.ericsdrivetotri.blogspot.com/
https://www.espn.go.com/mens-college-basketball/team/stats/_/id/2724/wichita-state-shockers
https://www.forums.redflagdeals.com/switching-bell-cogeco-807633/
https://www.freepages.genealogy.rootsweb.ancestry.com/~knight57/direct/knight/findex3.htm
https://www.gardacares.com/
https://www.gocanada.about.com/
https://www.gomontreal.about.com/od/montrealattractions/p/cca.htm
```

Discussion

The bloom filter we implemented was pretty efficient and performed well in terms of size, accuracy, it only occupied 1000001 bits of memory and gave just 10 false positives out of 57,311 test cases (0.00017 false positive rate) which is less than what bloom filter accuracy formula tells us: around
0.00025 false positive rate.

Advantages:

- It was very fast, space efficient and accurate
- Used multiple Hash functions for proper distribution

Disadvantages:

- We could have used a vector of boolean array to have dynamic sizing but it would compromise speed.
- False positives rate can still be reduced by using more hash functions.

Overall, we got to know how good of a data structure bloom filter is for querying whether an element exists in a dataset or not. However false positives and static nature (i.e. element can't be removed once added) still remain a difficult limitation to be removed.

These can be efficiently used in Cyber Security to analyze URLs.