Project III: Bloom filters

CSE 107 Probability and Statistics for Engineers
Textbook: Probability & Statistics with Applications to Computing, Alex Tsun
http://www.alextsun.com/files/Prob_Stat_for_CS_Book.pdf

Instructor: Chen Qian

1 Motivation

Google Chrome has a huge database of malicious URLs. When you visit a webpage, it needs to check if the page is malicious. But it takes a long time or large space to do a database lookup. Think about 10 million malicious URLs. They could cost hundreds of MegaBytes (MB) memory to store the set and you do not want Google Chrome to eat you hundreds of MB just to check malicious URLs – remember a browser has many other functions that cost memory space. Another solution is to store the malicious URLs in Google's server and every time you visit a webpage, you need to contact Google's server first to check if the page is malicious. That takes you a long time (hundreds of milliseconds versus < 1 milliseconds for checking locally). If we want to have a quick check in the web browser itself (on your computer), a space-efficient data structure must be used.

That is, we want to save both time and space. But what will we trade for it? It turns out we will have limited operations (fewer than a Set), and some probability of error which turns out to be fine.

2 Definition

(For detailed explanation of Bloom filters, please refer to the lecture content.)

A Bloom filter is a probabilistic data structure which only supports the following two operations:

- add(x): Add an element x to the structure.
- contains(x): Check if an element x is in the structure. If either returns "definitely not in the set" or "could be in the set".

It does NOT support the following two operations:

- Delete an element from the structure.
- Give a collection of elements that are in the structure.

The idea is that we can check our Bloom filter if a URL is in the set. The bloom filter is always correct in saying a URL definitely isn't in the set, but may have false positives (it may say a URL is in the set when it isn't). So most of the time, we get instant time, and only in these rare cases does Chrome have to perform an expensive database lookup to know for sure.

Suppose we have k bit arrays $t_1, .., t_k$ each of length m (all entries are 0 or 1), so the total space required is only km bits or $km/8$ bytes (as a byte is 8 bits). See below for one with $k = 3$ arrays of length $m = 5$:

So regardless of the number of elements n that we want to insert store in our bloom filter, we use the same amount of memory! That being said, the higher n is for a fixed k and m, the higher your error rate will be.

bloom filter t of length m = 5 that uses k = 3 hash functions

```
function INITIALIZE(k,m)
    for $i = 1, \dots, k$: do
        $t_i$ = new bit vector of m 0's
```

Index →	0	1	2	3	4
$\mathbf{t_1}$	0	0	0	0	0
$\mathbf{t_2}$	0	0	0	0	0
$\mathbf{t_3}$	0	0	0	0	0

Suppose the universe of URL's is the set $\mathbb{U}$ (think of this as all strings with less than 100 characters), and we have k independent and uniform hash functions $h_1, ..., h_k$: $\mathbb{U} \to \{0, 1, ..., m-1\}$. That is, for an element x and hash function h_i , pretend $h_i(x)$ is a discrete $Unif(0, m-1)$ random variable. Basically, when we see a new URL, we will add it to one random entry per row of our bloom filter.

See the image below to see how we add the URL "thisisavirus.com" into our bloom filter.

```
function ADD(x)
    for $i = 1, \dots, k$: do
        $t_i[h_i(x)] = 1$
```

add("thisisavirus.com")

```
h₁("thisisavirus.com") → 2
h₂("thisisavirus.com") → 1
h₃("thisisavirus.com") → 4
```

Index →	**0**	**1**	**2**	**3**	**4**
$\mathbf{t_1}$	0	0	**1**	0	0
$\mathbf{t_2}$	0	**1**	0	0	0
$\mathbf{t_3}$	0	0	0	0	**1**

For each of our $k = 3$ hash functions (corresponding to each row), we hash our URL x as $h_i(x)$ to get a random integer from $\{0, 1, .., 4\}$ (0 to $m - 1$). It happened that $h_1(x) = 2$, $h_2(x) = 1$ and $h_3(x) = 4$ in this example: each hash function is independent of the others and chooses a position uniformly at random.

But if we hash the same URL, we will get the same hash. In other words, if I tried to add this URL one more time, nothing would change because all the entries were already set to 1. Notice we never "unset" an entry: once a URL sets an entry to 1, it will stay 1 forever.

Now let's see how the contains function is implemented. When we check whether the URL we just added is contained in the bloom filter, we should definitely return yes.

contains("thisisavirus.com")

```
h₁("thisisavirus.com") → 2
h₂("thisisavirus.com") → 1
h₃("thisisavirus.com") → 4
```

```
function CONTAINS(x)
    return $t_1[h_1(x)] == 1 \land t_2[h_2(x)] == 1 \land \cdots \land t_k[h_k(x)] == 1$
```

True True True

Since all conditions satisfied, returns True (correctly)

Index →	**0**	**1**	**2**	**3**	**4**
$\mathbf{t_1}$	0	0	**1**	0	0
$\mathbf{t_2}$	0	**1**	0	0	0
$\mathbf{t_3}$	0	0	0	0	**1**

We say that a URL x is contained in the bloom filter, if when we apply each hash function $h_i(x)$, the corresponding entries are already set to 1. We added this URL "thisisavirus.com" right before this, so we are guaranteed that $t_1[2] == 1$, $t_2[1] == 1$, and $t_3[4] == 1$, and so we return TRUE overall! You might now see how this could lead to false positives: returning TRUE even though the URL was never added! Don't worry if not, we'll see some examples below.

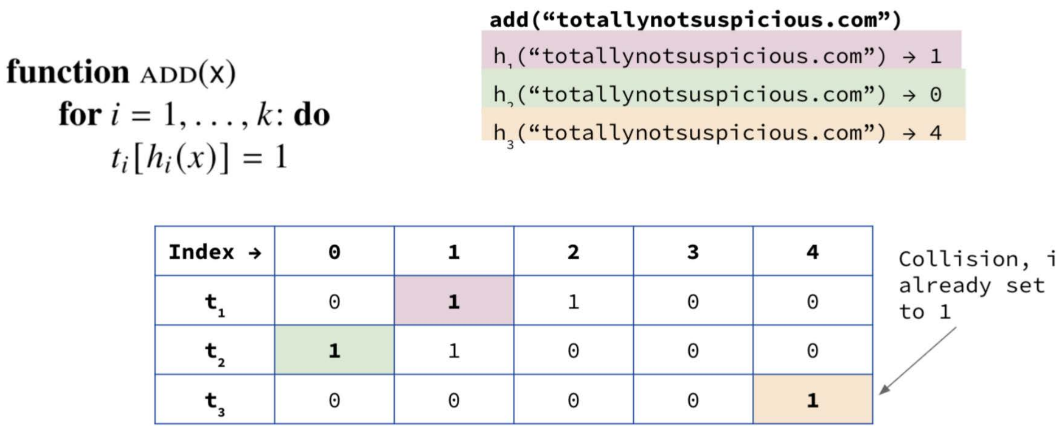
That's all there is for bloom filters!

Example: Starting with the current state of the bloom filter above:

1. Add the URL $x =$"totallynotsuspicious.com" which has $h_1(x) = 1$, $h_2(x) = 0$ and $h_3(x) = 4$. Draw the resulting bloom filter.

2. Check whether or not the URL "verynormalsite.com" is in the bloom filter, which has $h_1(x) = 2$, $h_2(x) = 0$ and $h_3(x) = 4$.

Solution:



```
function ADD(x)
    for $i = 1, \ldots, k$: do
        $t_i[h_i(x)] = 1$
```

add("totallynotsuspicious.com")

h_1("totallynotsuspicious.com") → 1

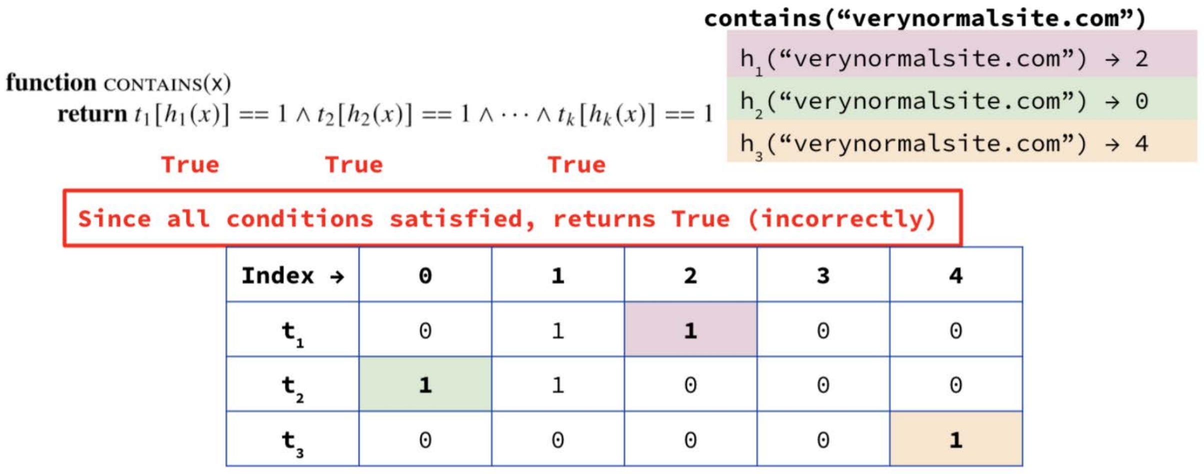
h_2("totallynotsuspicious.com") → 0

h_3("totallynotsuspicious.com") → 4

Index →	0	1	2	3	4
t_1	0	**1**	1	0	0
t_2	**1**	1	0	0	0
t_3	0	0	0	0	**1**

Collision, is already set to 1

Notice that $t_3[4]$ was already set to 1 by the previous entry, and that's okay! We just leave it set to 1.



```
function CONTAINS(x)
    return $t_1[h_1(x)] == 1 \wedge t_2[h_2(x)] == 1 \wedge \cdots \wedge t_k[h_k(x)] == 1$
```

True True True

contains("verynormalsite.com")

h_1("verynormalsite.com") → 2

h_2("verynormalsite.com") → 0

h_3("verynormalsite.com") → 4

Since all conditions satisfied, returns True (incorrectly)

Index →	0	1	2	3	4
t_1	0	1	**1**	0	0
t_2	**1**	1	0	0	0
t_3	0	0	0	0	**1**

Notice here we got a false positive: that means, saying a URL is in the bloom filter when it wasn't. This is a tradeoff we make in exchange for using much less space.

3 Analysis

You might be dying to know, what is the **false positive rate (FPR)** for a bloom filter, and how should I choose k and m? These are great questions, and we actually have the tools to figure this out already.

Theorem 1. *Bloom Filter FPR: After inserting n distinct URLs to a $k \times m$ bloom filter (k hash functions/rows, m columns), suppose we had a new URL and wanted to check whether it was contained in the bloom filter. The false positive rate (probability the bloom filter returns True incorrectly), is*

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

So based on n, the number of malicious URLs Google Chrome would like to store, should definitely play a part in how large they should choose k and m to be. Let's now see (by example) the kind of time and space improvement we can get

Example: 1. Let's compare this approach to using a typical Set data structure. Google wants to store 5 million URLs, with each URL taking (on average) 40 bytes. How much space (in MB, 1 MB = 1 million bytes) is required if we store all the elements in a set? How much space (in MB) is required if we store all the elements in a bloom filter with $k = 30$ hash functions and $m = 900,000$ buckets? Recall that 1 byte = 8 bits.

2. Let's analyze the time improvement as well. Let's say an average Chrome user attempts visit 102,000 URLs in a year, only 2,000 of which are actually malicious. Suppose it takes half a second for Chrome to make a call to the database (the Set), and only 1 millisecond for Chrome to check containment in the bloom filter. Suppose the false positive rate on the bloom filter is 3%; that is, if a website is not malicious, the bloom filter will will incorrectly report it as malicious with probability 0.03. What is the time (in seconds) taken if we only use the database, and what is the expected time taken (in seconds) to check all 102,000 strings if we used the bloom filter + database combination described earlier?

Solution:

1. For the set, we would require 5 million times 40 bytes, for a total of 200 MB.

For the bloom filter, we need just $km/8 = 27/8$ million bytes, or 3.375 MB, wow! Note how this doesn't depend (directly) at all on how many URLs, or the size of each one as we just hash it to a few bits. Of course, k and m should increase with n though :) to keep the FPR low.

2. If we only use the database, it will take $102000 \cdot \frac{1}{2} = 51000$ seconds.

If we use the bloom filter + database combination, we will definitely call the bloom filter 102000 timesat 0.001 seconds each, for a total of 102 seconds. Then for about 3% of the 100, 000 other URLs (3,000 of them), we'll have to do a database lookup, costing $3000 \cdot \frac{1}{2} = 1500$ seconds. For the 2000 actually malicious URLs, we also have to do a database lookup, costing $2000 \cdot \frac{1}{2} = 1000$ seconds . So in total, $102 + 1500 + 1000 = 2602$ seconds.

Just take a second to stare at how much memory savings we had (the first part), and the time savings we had (the second part)!

4 Project Content

Google Chrome has a huge database of malicious URLs, but it takes a long time to do a database lookup (think of this as a typical Set). They want to have a quick check in the web browser itself, so a space-efficient data structure must be used. A bloom filter is a probabilistic data structure which only supports the following two operations:

- I. add(x): Add an element x to the structure.
- II. contains(x): Check if an element x is in the structure. If either returns "definitely not in the set" or "could be in the set".

It does not support the following two operations:

- I. Delete an element from the structure.
- II. Give a collection of elements that are in the structure.

The idea is that we can check our Bloom filter if a URL is in the set. The bloom filter is always correct in saying a URL definitely isn't in the set, but may have false positives (it may say a URL is in the set when it isn't). Only in these rare cases does Chrome have to perform an expensive database lookup to know for sure.

Suppose we have k bit arrays $t_1, ..., t_k$ each of length m (all entries are 0 or 1), so the total space required is only km bits or $km/8$ bytes (as a byte is 8 bits). Suppose the universe of URL's is the set $\mathbb{U}$ (think of this as all strings with less than 100 characters), and we have k independent and uniform hash functions $h_1, .., h_k : U \to 0, 1, .., m-1$. That is, for an element x and hash function h_i , pretend $h_i(x)$ is a discrete $Unif(0, m-1)$ random variable.

- (a) Implement the functions add and contains in the BloomFilter class of bloom_filter.py (or .cpp). (Use the pseudocode provided earlier). What is the sample false positive rate? (This is printed out for you automatically).
- (b) Let's compare this approach to using a typical Set data structure. Google wants to store 1 million URLs, with each URL taking (on average) 25 bytes. How much space (in MB, 1 MB = 1 million bytes) is required if we store all the elements in a set? How much space (in MB) is required if we store all the elements in a bloom filter with $k = 10$ hash functions and m = 800,000 buckets? Recall that 1 byte = 8 bits.

Instructions for running program:

- For students using Python language: If you want to run it, just type:
  python3 bloom_filter.py
  Note that you need to install the package numpy and mmh3 using commands:

  pip3 install numpy

  pip3 install mmh3

- For students using C++ language: If you want to run it:

  g++ bloomfilter.cpp -o main

  ./main

  Note that c++ 11 or above is required.

- For students using C language: If you want to run it:

  gcc bloomfilter.c -o main

  ./main

Please include a pdf report that include

- Your name and cruzid
- The instructions to compile/run your program
- Screenshot of the real output of your program
- Your answer towards problem (b), you can provide more information to explain your solution (as long as reasonable).