

EECS 498 W26 System Design of a Search Engine

11 Bloom filter

Due Sunday, March 22, 2026, 11:59 pm

This assignment was written by Alexander Jalkanen, alexjalk@umich.edu. It is designed to introduce you to a probabilistic data structure called a Bloom filter, invented by Burton Howard Bloom in 1970, that can be used to test very quickly whether an element is a member of a set.

Starter files

You are given these starter files in BloomFilterStarter.zip.

BloomFilter.h	Starter file for the BloomFilter class.
main.cpp	A simple driver for testing the Bloom filter.
seen.txt	Sample data consisting of a very long list of URLs.
Makefile	A makefile to build a bf utility using the Bloom filter..

The assignment

This data structure can be thought of as like an unordered set, however once an object is added to the data structure it cannot be removed or iterated over. For this data structure, we can be certain an object is NOT contained, but we can never be certain an object is contained.

While it may not be intuitive why this data structure is a better choice than an unordered set, you will most like understand the benefits while trying to crawl websites. A Bloom filter is much more memory efficient, allowing you to easily have a seen set of 1 billion links, something which would not be possible with an unordered set.

By nature of this data structure, there can be false positives but there can never be false negatives. In our case, the false positive works in our favor. When we are crawling, we do not care about going to EVERY link, but we do want to avoid duplicates as that is a waste of time and it would be nice to not have to worry about duplicates later. This way, when we check to see if a link is in our Bloom filter, if our Bloom filter says it has not seen the link before, we know for certain that is accurate - which is good! However, if the Bloom filter says we have seen a link before, we know most likely this is true, but not for certain - which is good enough for us!

Here is a preview of what the finished Bloom filter will look like. bf will not take in any arguments.

```
$ ./bf
Running bloom filter...
Enter 'd' for default init, or 'c' for custom:
...
```

Setup

First download and unzip the starter files.

Make sure that the files compile as is. If there is a linking error, make sure that OpenSSL is installed on your system and in your path.

```
$ pwd
/Users/nham/eecs440/supplemental/bloomfilter
$ make
g++ -std=c++17 -Wall -Werror -pedantic -O3 main.cpp -ldl -lssl -lcrypto -o bf
```

Main.cpp

This is the driver for your program. It will initialize a Bloom filter to default values or values taken from standard in. Feel free to adjust this file for testing purposes or use as is.

Seen.txt

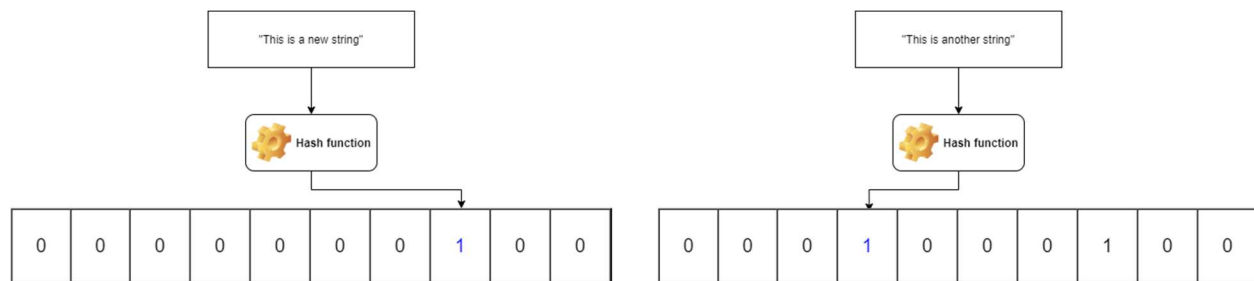
This text file consists of 500,000 unique URLs, which your seen list may look like. (Note: These are NOT quality links, so it is recommended to NOT use these links directly in your search engine). This file is good to use for testing the Bloom filter with a larger number of URLs.

How a Bloom Filter Works

Before we can implement a Bloom filter, we need to understand how it works. We will break down the main components of how this probabilistic data structure functions.

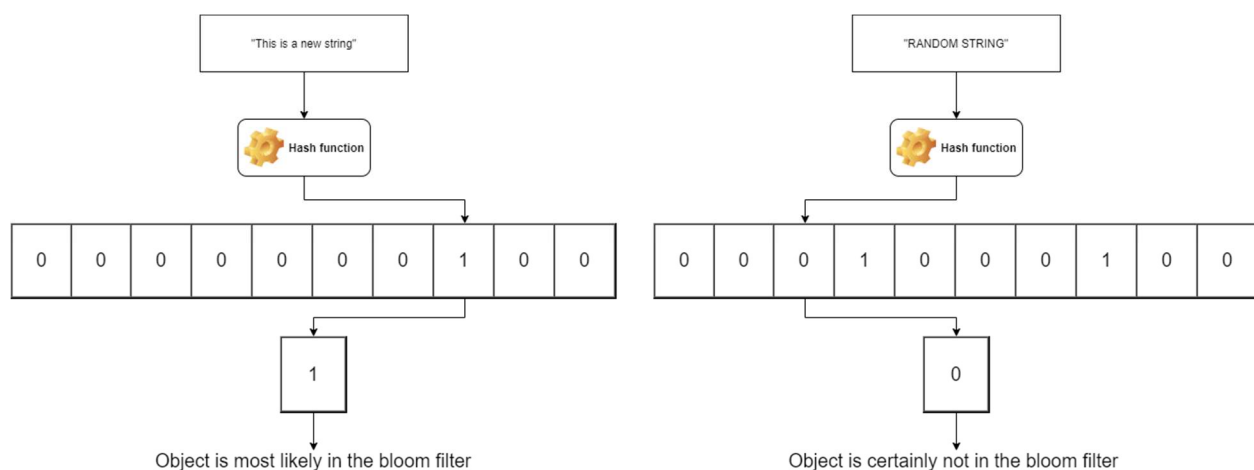
Insertion

A Bloom filter contains a set of bits, which are either true or false, depending on what is inserted. When an object is inserted into the Bloom filter, it is run through a hashing algorithm to get a value. That value then becomes a true bit in the set of bits.



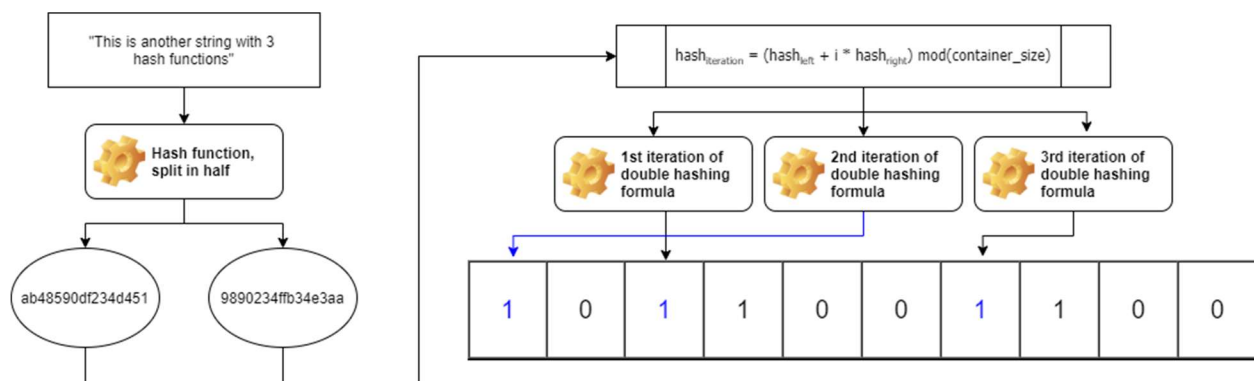
Contains

When we go to see if an object is already in the Bloom filter, we check to see if all the bits of the desired object are true, or if any of them are false.



Double Hashing

However, using only one bit could cause collisions and false positives, which is not a strong indicator of if the object is in the Bloom filter. To solve this, we can use multiple hash functions, with a technique called double hashing. Double hashing works by using a second hash as an offset to avoid collisions. To make sure the two hashes are unique to each other, we can split the original hash in half. The double hashing formula will be discussed further in the implementation portion.



False Positives

Now we can see that since objects are represented by random bits, it is possible for collisions to occur, hence what we call false positives. For example, say we insert many strings into our Bloom filter, which causes many of the bits to be true. It is possible that when we look for a completely new string that our Bloom filter has never seen before, all the bits are true! Perhaps one bit from our first inserted string matched, one bit from our second inserted string matched, etc... and all our new string matched. This would be a false positive from our Bloom filter. We can avoid this by making our Bloom filter larger and using a higher number hashing functions. We will discuss below on how to calculate the most optimal size and number of hash functions.

Implementing Bloomfilter.h

Constructor

There are three tasks we need to accomplish here:

1. Determine what to use as our container.
2. Determine the optimal size for our container.
3. Determine how many hashing functions to use.

Beginning with determining our containing, we need a container that is as small as possible and can store an ordered set of bits. One possible solution is to use `std::vector<bool>` as the STL optimizes this specific type of vector to be a bit set. Another possible solution is to use a dynamically allocated `std::bitset`. (Note that an array of bools is too large as bools require one byte each instead of one bit, so the array would be 8 times larger!).

Next, we need to determine the optimal size for our container. The formula for calculating this is:

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

In this formula, **m** is our size in bits, **n** is our estimated number of objects, and **ε** is our acceptable false positive error rate. The math behind calculating this formula can be found on Wikipedia at https://en.wikipedia.org/wiki/Bloom_filter. We can then resize our vector or dynamically allocate our bit set to be this size.

Finally, we need to determine how many hashing functions to use. The formula for calculating this is:

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

In this formula, **k** is our integer of hashing functions, **m** is the size of our container in bits (from above), and **n** is our estimated number of objects. The math behind calculating this is also found on the [Wikipedia page](#). We will want to save this value for use in our insert and contains functions.

Hash

For both our insert and contains function, we will need to hash our object. The hashing algorithm we are going to use is MD5, since it produces a 128-bit hash value. While you are welcome to choose a different hashing function, this function is already included in our OpenSSL library needed for other parts of the search engine and returns a convenient size to work with.

From <https://docs.openssl.org/3.4/man3/MD5/>, the function definition for using MD5 hashing is:

```
unsigned char *MD5(const unsigned char *d, unsigned long n, unsigned char *md);
```

where **d** is our string, **n** is the length of the string, and **md** is a buffer for the hash value. Our buffer should be 128 bits, or equivalently 16 bytes. OpenSSL provides a constant **MD5_DIGEST_LENGTH** for initializing an unsigned char array buffer. In our case, we can ignore the return value.

Notice how the function signature returns `std::pair<uint64_t, uint64_t>`. Due to our double hashing, we want to return two hash values. Currently, our buffer contains 16 bytes, but we want to split that into two 8-byte hash values (two `uint64_t` values). How should we convert one unsigned char array to two `uint64_t`'s? A possible first thought could be using a string to unsigned long long function, such as `std::strtoull`, however this won't work properly as each byte is attempted to be converted to a number, but our hash is already a number encoded in bits. Since we already have our hash encoded as a number in bits, we can use `memcpy` to copy two sets of 8 bytes to two `uint64_t`'s.

Insert

To insert into our Bloom filter, we first will want to hash our object into two hash values. Once we have those hash values, we can use [double hashing](#) to find an index of our bit in our container. The formula we want to use is:

$$h(i, k) = (h_1(k) + i \cdot h_2(k)) \bmod |T|.$$

where **h** is our new hash value at iteration **i** for string **k**. Here, **h1** and **h2** are our two hash values from our hash function, and **T** is the size of our container.

We will want to loop through the number of hashing functions we determined earlier that we want to use. For each iteration, we will want to calculate a new hash value using the above formula. That new hash value is the index of the bit that we want to set to be true in our container of bits.

Contains

Our contains function is almost the same idea as our insert function, except we do not want to modify the values of any bits. Like above, we will want to compute two hash functions, and iterate num_hash_function times. We will want to use the same double hashing formula as above to determine the index of the bits.

As we iterate to each bit location, we have two options: the bit is true or false.

1. If (and only if) all the bits are true: we know that probably the string is in our Bloom filter.
2. If at least one bit is false: we are certain the string is not in our Bloom filter.

The explanation of Bloom filters and diagrams above will help reinforce why the above statement is true.

Autograder

Submit your Bloomfilter.h file to the Autograder at <https://autograder.io/web/project/3495>.

FAQ

Frequently asked questions on why the program is not working:

There are linker errors with the starter files, what is wrong?

The hash function we are going to use is MD5 from OpenSSL, so we link to the ssl and crypto libraries provided. OpenSSL needs to be installed and in your path for the Bloom filter to compile.