Bloom filters and Swedish nouns

Daniel Chavez dace@kth.se

June 15, 2016

This essay is intended for computer science students who are comfortable with algorithms, data structures and time complexity. The purpose of this essay is to introduce the Bloom filter data structure by tying it together with a dictionary-like application.

1 Introduction

One of the most difficult parts of learning Swedish is determining the correct indefinite article of a noun [1]. There are only two to choose from: *en* or *ett*. For example, the noun *chair* is written *stol* in Swedish and the correct article is one of these:

en stol   *ett stol*

Someone who speaks Swedish will tell you it is definitely the left one: *en stol*, and the right one looks and sounds completely wrong. If you do not speak the language, how would you figure this out? Is there a formula? Are the first and last characters important? There is no formula which you can use to determine that the left one is correct. There are however methods and tricks available which are useful and they work most of the time. The last resort is to find *stol* in a Swedish dictionary which would have some information about the article.

Given all nouns in the Swedish language together with their corresponding article, how would you design a program which returns *en/ett* for any Swedish noun? The naive design is linear searching through all nouns until the queried noun is found and look up the corresponding *en/ett* article. If we have n nouns we would need a $2 \times n$ matrix, see Figure 1.

...	...	"stol"	...	...
...	...	"en"	...	...

Figure 1: *A matrix data structure is one way of storing nouns and their article.*

The avid reader would note that we could find the article even faster if we binary search instead of linear search, if all the nouns are alphabetically sorted. The pragmatic reader would note that the Swedish language consists of approximately 100 000 nouns. Hence, linear search would be fast enough.

Both readers are correct and both solutions are fast enough in practice, but we can do better. In this essay a data structure is presented to help us find the article in constant time.

2 Bloom filter

A Bloom filter is a data structure created by Burton H. Bloom in 1970 [2]. A Bloom filter is different from the traditional data structures, it does not contain the data like the $2 \times n$ matrix did in Figure 1 above. Instead, it is used to represent membership. A Bloom filter is an array of bits and a number of hash functions. Before we formally define a Bloom filter, let us look at how we would insert a noun like *stol* into a Bloom filter.

If $h(x)$ is some hash function, we hash the string *"stol"* and we use the hash as an index, for example:

$$h(\text{“}stol\text{”}) = 4711$$

Now in the array of bits, we set the bit at index 4711 to 1. The new state of the Bloom filter is shown in Figure 2 below. Note that the noun is not explicitly inside the data structure, however we say that *"stol"* has been inserted and is inside the Bloom filter.

index	0	1	...	4711	...	$m-1$
bit	0	0	...	1	...	0

Figure 2: *The state of a Bloom filter after representing the membership of the hash 4711.*

Now, the next question is what if another noun also hashes to 4711? This would result in a hash collision, which is why Bloom filters usually have more than one hash function. More hash functions would require that the nouns are hashed by all the hash functions resulting in more indices to set. However, collisions might still occur regardless of the number of hash functions. This is the reason why Bloom filters are usually presented as a *space-efficient probabilistic data structure for representing membership* [3]. It is space-efficient because an array of bits is cheaper than containing the actual nouns and it is probabilistic because hash collisions can cause problems. The next section gives a more formal mathematical definition of a Bloom filter together with some notation that we will use in the rest of this essay.

2.1 Definition

A Bloom filter represents a set $S = \{s_1, s_2, ..., s_n\}$. This is done with:

- An array of m bits, all initially set to zero
- k hash functions $h_1, h_2, ..., h_k$ with output range of 0 to $m - 1$
- For each element $s \in S$, the bits $h_i(s)$ are set to 1 for all k hash functions.

To query the Bloom filter whether an item y is in the set S:

- Compute $h_i(y)$ for all k hash functions
  - If any $h_i(y) = 0$ then $y \notin S$
  - If all $h_i(y) = 1$ then it is likely that $y \in S$

2.2 Probability

The definition in Section 2.1 stated that a query is not certain that an element is in the Bloom filter. The problem is that the hash functions introduce hash collisions when setting the bits. Some element $y \notin S$ which returns $y \in S$ after a query is known as a false positive.

The probability of getting a false positive on some element y is the probability of k specific bits being set to 1 by other elements $x \neq y$. The calculation follows a traditional probabilistic reasoning [4]:

Some hash function would set a specific bit to 1 with probability

$$Pr[\text{some bit is } 1] = \frac{1}{m}$$

and the same bit is 0 with probability

$$1 - \frac{1}{m}.$$

If we have k hash functions, instead of one, that bit would still be 0 with probability

$$\left(1 - \frac{1}{m}\right)^k$$

After inserting n elements, that bit would still be 0 with probability

$$\left(1 - \frac{1}{m}\right)^{kn}$$

and that bit would be 1 with the inverse probability

$$1 - \left(1 - \frac{1}{m}\right)^{kn}$$

Recall that a false positive was when k specific bits were set to 1, thus the probability of a false positive is:

$$Pr[\text{false positive}] = \left(1 - \left[1 - \frac{1}{m}\right]^{kn}\right)^{k}.$$

We can simplify the last probability with the approximation $\left(1 - \frac{1}{a}\right)^{a} \approx e^{-1}$ and we obtain:

$$Pr[\text{false positive}] \approx (1 - e^{-kn/m})^{k}.$$

This is a useful formula in practice, it gives us some intuition of what impact the number of hash functions k, the number of elements n and the size of the array m has on the false positive probability. Besides intuition, the probability stated as a function of k, n and m gives us full control of the false positive probability. In most cases, the choice of k, n and m is up to the programmer and the false positive probability can be tuned down to an acceptable magnitude.

It is important to note that the formula does not hold for smaller Bloom filters. When the size, m, of the array is less than 512, the formula can show that the probability of a false positive is lower than it is in reality [4]. The relative error between the real probability and the probability given by the formula increases as m gets closer to zero.

2.3 Hashing

In the probability section above we silently made an assumption that all hash functions were independent i.e. no hash function can be derived by any of the other hash functions. Another assumption is that all the hash functions return uniformly distributed hashes. Simply put, we have assumed that a Bloom filter has k perfect hash functions. These are safe assumptions in theory, but they require some thought in practice. Depending on the required false positive probability, the number of hash functions k might increase. This leads to an implementation issue if k is relatively large, where do we find k hash functions? According to a method called Double Hashing [5], only two hash functions are needed to generate $k - 2$ other hash functions.

2.4 Implementation

The simple structure of the Bloom filter makes for an elegant implementation. The Bloom filter supports two methods: `insert(x)` and `query(x)`. The method names are self-explanatory, although the query method is sometimes called `isIn` or `contains`. The pseudocode for both methods is given below.

Given that we have an array of bits `A` and a number of `hash_functions`, inserting an element `x` will set the bits with the indices retrieved by the hashes of `x`:

```
insert(x) {
    for h in hash_functions
        A[h(x)] = 1
}
```

As we mentioned earlier, the query requires all bits to be set:

```
query(x) {
    for h in hash_functions
        if A[h(x)] == 0
            return false
    return true
}
```

Note that besides the short and simple implementation, both methods run in constant time since the number of hash functions is constant.

3 Swedish nouns revisited

Recall that our goal was to design a program which given all the Swedish nouns and their corresponding articles, could answer which *en/ett* article belongs to some noun. With the Bloom filter data structure, we can avoid having any searching algorithms in our design:

1. Build two Bloom filters, one for each *en/ett* article
2. Insert en-nouns to the EN BloomFilter
3. Insert ett-nouns to the ETT BloomFilter

Now, any noun lookup would query both Bloom filters to test the membership of the noun. Figure 3 below shows a query for the noun *stol*. It starts by hashing the noun with two hash functions h and g, and their output is used as indices (highlighted in orange color). If all bits are set to 1 on these indices then the noun is a member of the corresponding Bloom filter.

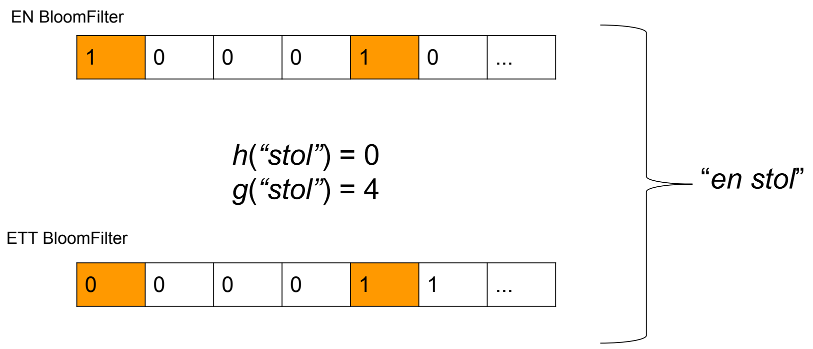

Figure 3: *Determining the article of the noun "stol" by querying two Bloom filters.*

4 Conclusions

The Bloom filter has several advantages in both time complexity and implementation. The disadvantage is the occasional false positive which depending on the application might have different consequences. In the application presented in this essay, the consequences of a false positive is an incorrect article of a noun or garbage input considered as a Swedish noun. In real-life applications, Bloom filters are used in a way where an occasional false positive is acceptable, for example in networking applications and caches. There are variations of the Bloom filter which deal with this problem and provide other functionality, like removing an element [3].

The program discussed in this essay can be found as a web application at `www.enellerett.se`. The input will be hashed 8 times and checked against two Bloom filters.

References

[1] Stockholm School of Economics. Things in general & particular. `http://www2.hhs.se/isa/swedish/chap3.htm`, 1997-1998. [Online; accessed 18-May-2016].

[2] Burton H Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7):422–426, 1970.

[3] Saibal K Pal, Puneet Sardana, et al. Bloom filters & their applications. *International Journal of Computer Applications Technology and Research*, 1(1):25–29, 2012.

[4] Ken Christensen, Allen Roginsky, and Miguel Jimeno. A new analysis of the false positive rate of a bloom filter. *Information Processing Letters*, 110(21):944–949, 2010.

[5] Adam Kirsch and Michael Mitzenmacher. Less hashing, same performance: Building a better bloom filter. In *Algorithms–ESA 2006*, pages 456–467. Springer, 2006.