Lecture Notes: Bloom Filter

Yufei Tao
Chinese University of Hong Kong
taoyf@cse.cuhk.edu.hk

7 Feb, 2012

1 Membership problem

Let $\mathcal{U}$ be the set of integers $\{1, ..., U\}$, and S of a subset of $\mathcal{U}$ with size n ($\leq U$). We want to preprocess S into a structure so that, given any integer $q \in \mathcal{U}$, we can determine quickly whether $q \in S$. We will refer to this problem as the *membership problem*.

We represent each integer in $\mathcal{U}$ with $w = \log U$ bits (all logarithms have base 2 by default). A simple solution is to build a hash table on S which answers a query in constant time (which can be made worst-case using perfect hashing). The space of the hash table is $O(nw)$ bits.

In this lecture, we ask the question: is it possible to solve the problem using a structure of $o(nw)$ bits? The answer is negative if a precise answer is always desired – it is easy to show that any structure for this purpose requires at least $\log \binom{U}{n}$ bits, which is $\Omega(nw)$ when $U \gg n$. Thus, we turn attention to trading precision away by allowing *false positives* but not *false negatives*. Specifically, we want to guarantee:

- If $q \in S$, the query algorithm should always report *yes* (i.e., the algorithm correctly tells us q is indeed in the set).
- If $q \notin S$, then the algorithm should return *no* with probability at least $1 - \delta$, where δ is an arbitrarily small positive constant. In other words, the algorithm is permitted to err by saying *yes* with probability at most δ.

2 The first solution

We first give a simple structure with n/δ bits. Note that, regardless of the constant δ, the space consumption is always $O(n)$ bits, thus beating the $O(nw)$ space bound of the hash table when U is large. Let $\mathcal{H}$ be a universal family of hash functions from $\mathcal{U}$ to $\{1, ..., m\}$, where $m = n/\delta$. Randomly picking a function $h \in \mathcal{H}$, we create an array A of size m, such that $A[i] = 1$ ($1 \leq i \leq m$) if some integer $k \in S$ satisfies $h(k) = i$; otherwise, $A[i] = 0$. Since each $A[i]$ requires only 1 bit to store, A occupies n/δ bits. The query algorithm is simple: given a query key q, we report *yes* if $A[h(q)] = 1$, or *no*, otherwise. The query time is constant.

It is easy to see that the algorithm always returns *yes* if $q \in S$. Next, we bound the probability that it returns *yes* when $q \notin S$. Observe that this happens if and only if there is a key $k \in S$ such that $h(k) = h(q)$. Since $\mathcal{H}$ is universal, $\mathbf{Pr}[h(k) = h(q)]$ is at most $1/m$. Thus, the probability that *at least* one such key exists is at most

$$\sum_{\forall k \in S} [\mathbf{Pr}[h(k) = h(q)]] \leq n/m = \delta. \tag{1}$$

Remark. In deriving (1), we used an inequality commonly referred to as the *union bound*. For any events A, B, the bound says that $\mathbf{Pr}[A \cup B] \leq \mathbf{Pr}[A] + \mathbf{Pr}[B]$. Note that this is true no matter

whether A and B are disjoint – in fact, the equality holds if and only if they are. Loose as it may look, the union bound is surprisingly useful in analyzing probabilistic algorithms.

3 Bloom filter

The bloom filter [1] improves our earlier structure by reducing the space to $O(n \log \frac{1}{\delta})$ bits. The analysis of the bloom filter assumes a hash function that is (much) stronger than a function from the universal family, as explained next:

Definition 1. *A hash function $h : \mathcal{U} \to \{1, ..., m\}$ (where m is an integer) is **ideal** if, for every integer $k \in \mathcal{U}$, $h(k)$ independently takes a value uniformly distributed in $[1, m]$.*

In other words, for any k, $h(k)$ takes any number of $\{1, ..., m\}$ with the same probability, and which one it takes does not rely on the hash values of any other keys. Ideal hash functions do not exist yet (to my knowledge). However, algorithm analysis based on such functions is common practice in computer science. One possible reason is that, many functions (like those from a universal hash family) behave in a manner that is fairly close to being ideal.

3.1 Structure and algorithm

Again, let A be an array of size m, where m is to be decided later. Let $h_1, ..., h_l$ be l independent ideal hash functions from $\mathcal{U}$ to $\{1, ..., m\}$. In preprocessing, for each $k \in S$, we set (up to) l bits of A to 1, i.e., $A[h_i[k]] = 1$ for each $i \in [1, l]$. Note that $A[h_i[k]]$ may have already been 1 before the processing of k, in which case setting the bit to 1 does not change it. Furthermore, it is possible that less than l bits are set, because some hash functions may happen to hash k to the same value. The resulting A at the end of preprocessing is the bloom filter.

To answer a query with key q, simply check whether $A[h_i(q)]$ is 1 for *all* $i \in [1, l]$. If so, answer *yes*, otherwise, answer *no*.

3.2 A non-rigorous analysis

We focus on analyzing the probability that the bloom filter errs, i.e., given a $q \notin S$, what is the probability that the query algorithm returns *yes*. Our analysis in this section is exactly what appeared in the paper [1] where the bloom filter was invented. While it is easy to understand and (surprisingly) has been repeated in many subsequent papers, this analysis has been proven wrong [2]. Nevertheless, the error is minor such that it is hardly noticeable in practice, and even in theory, can be fixed with a small correction, as explained in the next subsection.

The crux of the analysis lies in deriving the expected number of bits in A that equal 1, after constructing the bloom filter on S. Recall that, during the construction, the processing of a key $k \in S$ repeats the following l times: set a random bit of A to 1. Hence, a bit of A is not affected by k with a probability $(1 - 1/m)^l$. A bit in the final A remains 0 if and only if it is unaffected by all n keys in S. It follows that the probability that a bit remains 0 equals $(1 - 1/m)^{nl}$. The expected number of ones in the bloom filter equals np where $p = 1 - (1 - 1/m)^{nl}$.

Now consider answering a query with a $q \notin S$. The algorithm repeats the following l times: probe a random bit to see if it is 1. Each probe hits a one with probability p. Hence, the probability that all l probes hit 1 is p^l. In other words, the bloom filter errs with probability

$$(1 - (1 - 1/m)^{nl})^l.$$

A common choice of l is

$$l = (\ln 2)\frac{m}{n}$$

with which the error probability becomes

$$(1 - (1 - 1/m)^{nl})^l \approx (1 - e^{-\frac{nl}{m}}) = (1/2)^{(\ln 2)\frac{m}{n}}.$$

The above used the fact that $1 + x \approx e^x$. Hence, to make sure that the above is at most δ, we need

$$m = n\frac{\ln(1/\delta)}{(\ln 2)^2}. \tag{2}$$

3.3 Fixing the error

The previous analysis neglected the fact that the number of ones in the bloom filter is a random variable itself. Recently, a simple fix has become available. It is shown in [2, 3] that the correct value of m to make the error probability at most δ is only slightly greater than as given in (2) (differing by no more than a small constant factor).

References

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

[2] P. Bose, H. Guo, E. Kranakis, A. Maheshwari, P. Morin, J. Morrison, M. H. M. Smid, and Y. Tang. On the false-positive rate of bloom filters. *Information Processing Letters (IPL)*, 108(4):210–213, 2008.

[3] A. Goel and P. Gupta. Small subset queries and bloom filters using ternary associative memories, with applications. In *Proceedings of International Conference on Measurements and Modeling of Computer Systems (SIGMETRICS)*, pages 143–154, 2010.