CS 388R: Randomized Algorithms, Fall 2021 September 21

Lecture 8: Bloom Filters

Prof. Eric Price *Scribe: Tongrui Li, Yihan Zhou*

NOTE: THESE NOTES HAVE NOT BEEN EDITED OR CHECKED FOR CORRECTNESS

1 Overview

In this lecture we talked about Bloom filters. The problem we want to solve is stated as the following:

Problem: We want to store a set S of n items, where $S \subseteq [U]$ (assuming $U \geq n^2$).

We also want to support the following operations:

Operations:
`insert`(x) : $S \leftarrow S \cup \{x\}$. This operation inserts a new item x to the stored set S.
`query`(x) : is $x \in S$? This operation checks whether an item x is stored.

2 Ways to store the set S

We have many ways to store the set S. To be more specific, we can use a **list**, a **membership array**, a **hash table** or a **Bloom filter** (we will define it later). The table below gives an overview of how these methods compare to each other in terms of space complexity (number of bits needed for storage), insertion time and query time.

Method	Space Complexity	Insertion Time	Query Time
List	$n \log U$ bits	$\Theta(1)$ (if each item is fresh)	$\Theta(n)$
Membership Array	U bits	$\Theta(1)$	$\Theta(1)$
Hash Table	$O(n \log U)$ bits	$O(1)$ worst case	$O(1)$ in expectation
Bloom Filters	$O(n)$ bits ($+O(U \log U)$) if we consider hash functions	$O(1)$	$O(1)$ but with small constant failure probability

2.1 Lower space complexity bound

Note that for deterministic methods, $\Omega(n \log U)$ bits space is necessary. The number of different set S is $\binom{U}{n} \geq \left(\frac{U}{n}\right)^n = 2^{n \log \frac{U}{n}} \geq 2^{\frac{1}{2} n \log U}$. This means we need at least $2^{\frac{1}{2} n \log U}$ different representations. Therefore, $\frac{1}{2} n \log U$ bits are necessary.

3 Bloom filters

The bloom filter will output `YES` with probability 1 if $x \in S$ and will output `NO` with probability $1-\delta$ if $x \notin S$. In other words, Bloom filters approximate set membership queries. There are a few examples of real-life applications of Bloom filters.

Examples:
- Google Chrome uses Bloom filter to store the set of malicious websites.
- Database asymmetric joins.
- Bitcoin light wallets use Bloom filter to store wallet IDs of interests.

A way to construct a Bloom filter is to create a hash table with size m, each location in the hash table has 1 bit, either 1 or 0. Then we pick $h_1, h_2, \cdots, h_k$ **independent** hash functions from U to $[m]$. Now we define the operations to be:

`insert`(x) : $y_{h_i(x)} = 1,\ \forall i \in [k]$. We set the k locations in the hash table given by k hash values to be 1 (there could be less than k locations due to repetitive hash values).

`query`(x) : return $\min_i y_{h_i}(x)$. Only when all locations corresponds to all k hash values in the hash table has value 1, we return 1. Otherwise, we return 0.

This Bloom filter will take $O(m)$ bits for space and each insert or query takes $O(k)$ time.

Now we try to pick k to minimize the failure rate δ. Intuitively, we want to maximize the information each hash table bits contain, which means we want each hash table bit has $\frac{1}{2}$ probability to be 1. This gives the maximum entropy and the maximum mutual information. Formally, we have for any hash table bit y_j,

$$\mathbb{P}(y_j = 0) = \left(1 - \frac{1}{m}\right)^{kn} \approx e^{\frac{-kn}{m}} (= \frac{1}{2} \text{ by intuition.})$$

The probability above comes from that all kn hash values miss the location j. Let $Y = \sum_{j=1}^{m} y_j$. The random variable Y denote the number of hash bits that are 0. The expectation value of Y is

$$\mathbb{E}(Y) = m\left(1 - e^{-\frac{kn}{m}}\right).$$

With high probability, $Y = \mathbb{E}(Y) \pm O(\sqrt{m \log m})$. For any $x \notin S$,

$$\mathbb{P}(\texttt{query}(x) = 1) = \left(\frac{Y}{m}\right)^k \approx \left(1 - e^{-\frac{kn}{m}}\right)^k = \delta.$$

To minimize δ, we need to get

$$\arg\min_k \left(1 - e^{-\frac{kn}{m}}\right) = \frac{m}{n} \arg\min_z \left(1 - e^{-z}\right)^z.$$

Solving for the above we get $z = \ln 2$ and $k = \frac{m}{n} \ln 2$. Then $\delta = 2^{-k} = 2^{-\frac{m}{n} \ln 2} \approx 0.618^{\frac{m}{n}}$.

For example, if we choose $t = 16$, we have $\mathbb{P}(y_j \geq 16) \leq 1.4 \times 10^{-15}$. By union bound, $\mathbb{P}(\text{Any } y_j \geq 16) \leq m \cdot 1.4 \times 10^{-15}$ and is small for $m < 10^9$.

3.1 Bloom Filter With Deletion

Ideally, we want to introduce delete to the set of operations:

`delete`(x) : $S \leftarrow S - \{x\}$. This operation takes out item x from the stored set S.

Note that the algorithm introduced in the above section does not accommodate deletion as any of space that the hash function occupies can also be concurrently occupied by another hash.

We can make a simple modification by replacing each entry in the hash table from a binary switch to a **counter**. This will lead to the below modification:

`insert`(x) : $y_{h_i(x)} += 1,\ \forall i \in [k]$. We **increment** the k locations in the hash table given by 1.

`query`(x) : return $\min_i y_{h_i}(x) >= 1$. Only when all locations corresponds to all k hash values in the hash table has value 1 **or above**, we return 1. Otherwise, we return 0.

`delete`(x) : $y_{h_i(x)} -= 1,\ \forall i \in [k]$. We **decrement** the k locations in the hash table given by k hash values by 1.

Duplicate Item Note although the above formulation allows for deletion, it will not accommodate duplicate item due to repeating value of the counter.

Analysis Note that we will need to assume $\log_2(t)$ bits for the storage of the counter instead of the previous 1 bit requirement. We can compute, for given n, m, k, the probability that a single space in hash table y_i, obtain a value that exceed t.

$$\begin{aligned} P(y_i \geq t) &\leq \binom{nk}{t} \frac{1}{m^t} \\ &\leq (\frac{enk}{t})^t \frac{1}{m^t} \\ &= (\frac{enk}{mt})^t \\ &= (\frac{e \ln(2)}{t})^t \triangleright \text{ substitute k's value from previous derivation} \end{aligned}$$

Lets assume that $t = 16$, this means that we will need 4 bits in storage. With this in mind, the probability that a cell exceed 16 is

$$P(y_i \geq 16) \leq 1.4 \times 10^{-15}$$

With m entries in the hash table, by the union bound we have:

$$P(y_i \geq 16 \forall i \in m) \leq m \cdot 1.4 \times 10^{-15}$$

If we set m to be at most 10^9, the above equation evaluates to a very small failure rate.

4 Independence

In class, we also started talking about how to choose hash functions from a family. This topic will proceed into the next lecture - hence for detail please refer to the note of next class.

We have two options when it comes to picking a hash function

1. **Full Independence** hashes are random oracle model. Hashing computation is essentially free. (i.e. flipping a coin)
2. **Limited Independence** we have to pay in space for hash function (e.x. coccho hashing)

These two options offers different advantages:

4.1 Full Independence

1. Sometimes totally fine (i.e. in load balancing situation)
2. Sometimes you can approximate it. (i.e. entropy in input or cryptographic hash functions)
3. This method often provide better bounds

4.2 Limited Independence

1. Honest about cost of hash (which means that we can get gurantess over the inputs)
2. Per hash function, $O(k)$ words of space can yield k-wise independence. Sets of size $\leq$ k behaves as if independent.
3. E.x cocco hash yields $\log(n)$ independence in standard implementation (Professor also mentioned that some more complicated implementation may yield better result).