CS 6550: Randomized Algorithms **Spring 2019**

Lecture 18: Bloom Filters and Cuckoo Hashing

March 12th, 2019

Lecturer: Eric Vigoda *Scribes: Satchit Sivakumar*

18.1 Introduction

To motivate the formal setting we are going to work in, consider the following problem: You've been whisked away in a time machine to 1996, when the first big public email service has been setup. Security is not a thing yet and everybody is entering passwords that are very easy to break like '1234'. Yahoo hires you to solve this problem and hands you a list of unacceptable passwords that they have created. The list could grow, a bunch of smartasses thought that 'password' was a great password and it was just added to the list yesterday. Yahoo anticipates that this list is going to grow remarkably large, and so tasks you with finding a space-efficient way of storing the list. You also need to be able to query the storage structure to check if a password is acceptable or not.

This can be formalized to the following setting: we have a huge universe of elements $U = \{0, 1, ..., N-1\}$ and we want to maintain a subset $S \subset U$, where $|S| = m$. We want to store the subset in some structure H such that $|H| = n = cm$ where c is a constant $>= 1$. The following operations have to be accommodated: insertion of an element into the subset S and for any element x being able to check whether $x \in S$. Deletion isn't considered here. We are willing to accommodate a small number of false positive responses to our queries i.e. if $x \notin S$ then we want to answer no, but we are okay with rarely answering the query wrongly.

In the motivating example, the universe would correspond to the set of all passwords, and the subset would correspond to the set of unacceptable passwords. To complete the analogy, if a password is unacceptable, we want to GUARANTEE that it isn't used- so given a password x, if $x \in S$, then we always want to reject the password but if $x \notin S$ then we are okay with occasionally making the user pick a new password because the cost is only mild annoyance. However, we want to keep the probability of such a false positive low.

Through this lecture, we discuss 2 solutions to the problem framed above: one is called Bloom Filters and the other is called Cuckoo Hashing (the setting is slightly different in the Cuckoo Hashing setting and the changes are described in the corresponding section). Note that they are both generalizations of the well known Hash-Table structure. Hash-Tables are discussed briefly in the textbook by Mitzenmacher and Upfal [MU] for any reader that wants some more background.

18.2 Bloom Filters

The representation structure is a simple $0-1$ array H of size n, initially all 0. One of the major assumptions we make is that we are given k hash functions, $h_1, h_2, ..., h_k$ that map elements from the universe U to $\{0, 1, ..., n-1\}$, that is they map elements from the universe to indices of H.

Also, we assume that the hash functions we are given are random and independent of each other. To be very clear, random in this context means that an element when hashed under a particular function has equal chance of being mapped to any location in H. Independent means that this probability does not change even when conditioned on what indices this or any other element is hashed to by any subset of the $k-1$ other hash functions.

Now we describe the algorithms for insertion and querying.

18.2.1 Bloom Filter Algorithms for Operations

Algorithm 1 inserts an element x into H.

Algorithm 1: Insertion

```
input  : $x$
output: The updated $H$ structure, with $x$ inserted
1 for $i \leftarrow 1$ to $k$ do
2     compute $h_i(x)$
3     Set $H[h_i(x)]$ to 1
```

Algorithm 2 checks if $x \in S$

Algorithm 2: Query

```
input  : $x$
output: YES or NO
1 Set count to 0
2 for $i \leftarrow 1$ to $k$ do
3     compute $h_i(x)$
4     if $H[h_i(x)] = 1$ then
5         Increment count by 1
6 if $count$=$k$ then
7     return YES
8 return NO
```

To briefly describe the algorithms in simple words: insertion simply computes the hash values for the input element under the different hash functions and makes all the corresponding bits 1. Querying simply computes the hash values for the input element and checks if all the corresponding indices are set to 1. If yes, then it assumes that the bits were set when the element was inserted earlier and says YES, else it says NO.

18.2.2 Bloom Filter Querying Analysis

The first important note is that clearly, when x has previously been inserted, all of the corresponding hash bits are set to 1 and that does not change with any additional insertion or query operations. Hence, if $x \in S$, then the algorithm will always correctly output YES. The analysis hence reduces to considering the probability of a false positive. Clearly, false positives are possible; If the insertion of some other elements caused all the bits corresponding to x to be set to 1, then the algorithm will output the wrong result.

Now, consider the addition of one element x. Consider the probability that for a hash, that x is indexed to some index i.

$$\forall k: \ \Pr(h_k(x)) = i) = 1/n \implies \Pr(h_k(x)) \neq i) = 1 - 1/n \tag{18.1}$$

The above equation follows from the fact that we assume that the hash functions are completely random. Now, since the hash functions are completely independent of each other, the probability that NONE of the hash functions is indexed to i can be easily calculated. Note that this is also the probability that the index $H(i)$ is set to 0 after this insertion.

$$\Pr(H[i] = 0 \text{ after the first insertion}) = (1 - 1/n)^k \tag{18.2}$$

Now, we assume that the elements are randomly drawn and hence the insertions are independent of each other in terms of the bits they set. Consider that all m insertions are done. Then, the probability that $H[i]$ is 0, is the probability that H[i] is 0 after the first insertion multiplied by the probability that H[i] is 0 after the second insertion and so on. Hence:

$$\Pr(H[i] = 0 \text{ after all insertions}) = (1 - 1/n)^{mk} \tag{18.3}$$

$$\Pr(H[i] = 1 \text{ after all insertions}) = 1 - (1 - 1/n)^{mk} = 1 - (1 - 1/n)^{nmk/n} \approx 1 - e^{-mk/n} \tag{18.4}$$

Now, for a false positive, all the k hashed bits that correspond to x have to be set to 1. Since the hash functions are independent of each other, the probability that the hashed bits are set to 1 is simply a product of the individual probabilities which can be bounded by:

$$\Pr(\text{False Positive}) = (1 - e^{-mk/n})^k = (1 - e^{-k/c})^k \tag{18.5}$$

where the last equality follows because $n = cm$.

Now we can find the value of k that minimizes the probability of a false positive by simply differentiating the probability with respect to k and setting it to 0, and then taking the second derivative to check that it is indeed a global minimum.

Set $f = (1 - e^{-k/c})^k$. Now set $g = log_e(f) = klog_e(1 - e^{-k/c})$.

$$\frac{d}{dk}(g) = ln(1 - e^{-k/c}) + \frac{k}{c}\frac{e^{-k/c}}{(1 - e^{-k/c})} = 0 \tag{18.6}$$

Solving the above equation, we get the optimal value for k as $k = cln2$. Taking the second derivative and substituting this value for k, we see that the value is ≥ 0 indicating that this point is indeed a global minimum. Now, going back to the equation for k and substituting:

$$\Pr(\text{False Positive}) = (1 - e^{-k/c})^k = (1 - e^{-ln2})^{cln2} = ((1/2)^{ln2})^c = 0.61^c \tag{18.7}$$

Note that for c=100, Pr(False Positive) = 1.3×10^{-21} which is remarkably low.

18.2.3 Bloom Filter Time Complexities

Now let us compute the complexity of the operations. Assuming that hashing takes constant time, for both insertion and querying, k hashes are required and hence insertion and querying both are $O(k)$, which for k fixed to $cln2$ becomes an $O(1)$ operation.

Hence, the Bloom filter implements the structure that we desired- with linear space and efficient operations. However, one drawback of Bloom filtering is that deletion is not possible easily with this representation- alternate representations have been described called counting Bloom filters that accommodate this but in that case we have to allow for false negatives as well as false positives. Next, we describe an alternative approach called Cuckoo Hashing that uses the Power of 2 ideas to obtain no false positives and an expected insertion and querying complexity of $O(1)$.

18.3 Cuckoo Hashing

The setting we are considering is similar to the setting motivated and described in the introduction with a couple of changes- one addition is that we want an operation to delete an element x from the data structure. Another change is that to simplify the approach presented, we do not ask for linear space.

Our representation is in the form of an array of elements H indexed from 0 to $n-1$ where not more than 1 element is stored per index. Note that a difference from Bloom Filters is that the representation is NOT a $0-1$ array. We have 2 independent and random hash functions that map from the universe to the indices 0 to $n-1$. We shall call them h_1 and h_2. We now give the algorithms for insertion, querying and deletion.

18.3.1 Cuckoo Filter Algorithms for Operations

Algorithm 3 inserts an element x into H.

Algorithm 3: Cuckoo Filter Insertion

```
input  : $x$
output: The updated $H$ structure, with $x$ inserted
1  compute $h_1(x)$
2  if $H[h_1(x)]$ is empty then
3      $H[h_1(x)] = x$
4      return $H$
5  compute $h_2(x)$
6  if $H[h_2(x)]$ is empty then
7      $H[h_2(x)] = x$
8      return $H$
9  store temp=$H[h_2(x)]$
10 Set $H[h_2(x)] = x$
11 Send temp to its other location i.e. if $h_1(temp) = h_2(x)$ set $H(h_2(temp)) = temp$ or if
     $h_2(temp) = h_2(x)$ set $H[h_1(temp)] = temp$ .
12 Repeat the above process until there is no element displacement.
13 if cycle then
14     Randomly re-initialize $h_1(x), h_2(x)$
15     Re-insert every element.
```

To simply explain the above algorithm, we insert an element; if there is another element there we displace it and move it to its other hashed location; if there is an element there, we move it to its other hashed location and so on until we reach an empty spot or we get a cycle i.e. we reach a position that we have already reached before (meaning that the process would continue indefinitely). At this point, we draw 2 new independent and random hash functions, reset the data structure H and re-insert every element.

Algorithm 4 checks if $x \in S$
Query simply checks the 2 spots that x could be indexed to and returns YES if either of them contains x and returns NO otherwise.

Algorithm 5 deletes x if x is in H, else it just returns the old H.
It simply checks the 2 spots that x could be present and deletes the corresponding location of x.

Algorithm 4: Cuckoo Hashing Querying

```
input  : $x$
output: YES or NO
1 Compute $h_1(x)$ and $h_2(x)$
2 if $H[h_1(x)] = x$ or $H[h_2(x)] = x$ then
3     return YES
4 return NO
```

Algorithm 5: Cuckoo Hashing Delete

```
input  : $x$
output: $H$ with $x$ deleted
1 Compute $h_1(x)$ and $h_2(x)$
2 if $H[h_1(x)] = x$ then
3     set $H[h_1(x)]$ to empty
4     return H
5 if $H[h_2(x)] = x$ then
6     set $H[h_2(x)]$ to empty
7     return $H$
8 return H
```

18.3.2 Cuckoo Hashing Analysis

Firstly, note that there are no false positives or false negatives, querying is always correct which is obvious from the description above. Similarly, deletion works correctly as well. Both are clearly also constant time operations. Hence, the analysis reduces to analyzing the complexity of the insertion operation.

To perform the analysis, we will assume that the constant c is ≥ 6. We will see why this requirement is necessary later in the analysis. Now, we construct a *Cuckoo Graph* as follows:

The graph consists of n vertices (one for every index). Let us assume the data is fixed. Then, there is an edge from i to j if for some x, $h_1(x) = i$ and $h_2(x) = j$ or vice-versa. Clearly, an edge represents a potential movement due to an insertion. Now, we analyze the insertion time. First, we state and prove a useful lemma.

Lemma 18.1 $\Pr$*(Shortest path from i to j is of length l)* $\leq 3^{-l}/n$.

Proof: We prove this by induction.

First, consider the base case, l=1. In this case, the shortest path will be of length 1 when there is a direct edge from i to j. This happens when either $h_1(x) = i$ and $h_2(x) = j$ or vice-versa for any x. Since the hash functions are random, the probability that $h_1(x) = i$ is $1/n$. Hence, since the hash functions are independent, the probability that the shortest path is of length 1 due to x is $2/n^2$. Considering all possible x, we get $2/n^2 \times |S| = 2/n^2 \times m = 2/6n = 1/3n = 3^{-1}/n$. It should now be clear why we needed $c >= 6$. Hence, the base case is satisfied.

Now, we assume the hypothesis holds for all r from 2 to $l-1$ and prove it holds for l. Now consider the shortest path of length l from i to j. Then, there is a k such that the shortest path from i to k is of length $l-1$ and there is an edge from k to j. Now, the probability for a fixed k that the shortest path from

i to k is $l-1$ is $\leq 3^{-(l-1)}/n$ from the induction hypothesis and the probability that there is an edge from k to j is $1/3n$ from the base case. Hence, the probability that the shortest path through k is of size l is $\leq 3^{-(l-1)}/n \times 1/3n$. Summing over the n different values of k possible, we get that the probability that the shortest path from i to j is of size l is $\leq 3^{-l}/n$ which proves the lemma. ■

Now, based on this lemma: we state the following theorem

Theorem 18.2 *Expected insertion time of an element is $O(1)$ given there are no rehashes.*

Proof: Now, we use the lemma 18.1 to prove the theorem. First, we say that x and y collide if there is a path from x to y, meaning that there is a path from $h_1(x)$ or $h_2(x)$ to $h_1(y)$ or $h_2(y)$. Then, the expected insertion time (given no cycles) will be equal to the expected number of collisions on inserting x. (Since that is equal to the number of movements on inserting x). The probability that x and y collide is $\leq 4\sum_{l=1}^{n} \Pr(\text{path of length } l \text{ between i and j})$ (where i and j can be arbitrary since by our lemma, the bound we are going to use is the same). The 4 comes from the fact that we have 4 different combinations of indices. The sum is because the shortest path length if 2 elements collide can be anything from 1 to n. Now applying lemma 18.1, we get

$$\Pr(x \text{ and } y \text{ collide}) \leq 4\sum_{l=1}^{n} 3^{-l}/n \leq 4/n \sum_{l=1}^{\infty} 3^{-l} = 4/2n = 2/n \tag{18.8}$$

Now, when I insert an x, it could collide with all the previously inserted elements. Now, if I want to bound the expected number of collisions for a particular insertion, I can assume that all the other elements have been inserted for every x that I insert (this will give a worst case bound). For every, element m_i if I have a $0-1$ random variable y_i representing whether x collides with m_i or not, then clearly, the random variable representing number of collisions is just the sum of the y_is. Since y_i is just a $0-1$ random variable, the expectation of y_i is simply the probability that x and m_i collide, which is given by equation 18.8 . Hence, the expected number of collisions is just the sum of the expectations which is $2/n \times cn = O(1)$. Hence, an insertion is a constant expected time operation if it doesn't cause a rehash. Hence, the time to insert all m elements into H is simply $O(m)$ given none of them cause a rehashing. ■

Now, we will consider rehashing and then bound the estimated insertion time including rehashing. First, note that if there is a rehash caused by a set of insertions, then there is correspondingly a cycle in the cuckoo graph. Thus, the probability that a rehashing is caused by a set of insertions is $\leq$ the probability of a cycle in the cuckoo graph. We bound this probability by proving the following theorem:

Theorem 18.3 $\Pr($*Cycle in cuckoo graph*$) \leq 1/2$

Proof: If there is a cycle, then there is a path from some position i to itself. Now, the probability of a cycle involving position i can be bounded using lemma 18.1 as follows:

$$\Pr(\text{path from } i \text{ to } i) \leq \sum_{l=1}^{n} 3^{-l}/n \leq 1/n \sum_{l=1}^{\infty} 3^{-l} = 1/2n \tag{18.9}$$

Hence, the probability of a cycle can be obtained by summing equation 18.9 over all possible vertices which gives us:

$$\Pr(\text{cycle in graph}) \leq \sum_{i=1}^{n} 1/2n = n \times 1/2n = 1/2 \tag{18.10}$$

Hence, the required statement has been proved. ■

This also proves that the probability that a rehashing is caused by a set of m insertions is $\leq 1/2$.

Now, if we rehash once, then in the worst case we have to insert all the elements twice. If we rehash twice, then in the worst case we have to insert all the elements thrice. Let the insertion time given no rehashes be I. Hence, the total insertion time is going to be given by:

$$\text{Insertion Time} \leq (1/2 \times I) + (1/4 \times 2I) + (1/8 \times 3I) + (1/16 \times 4I) + \ldots\ldots\ldots\ldots \tag{18.11}$$

Taking expectation on both sides, we get:

$$\text{E(Insertion Time)} \leq (1/2 \times E(I)) + (1/4 \times 2E(I)) + (1/8 \times 3E(I)) + (1/16 \times 4E(I)) + \ldots\ldots\ldots\ldots \tag{18.12}$$

Now, substituting for expected insertion time which we get from theorem 18.2, we get:

$$\text{E(Insertion Time)} \leq (1/2 \times m) + (1/4 \times 2m) + (1/8 \times 3m) + (1/16 \times 4m) + \ldots\ldots\ldots\ldots \tag{18.13}$$

This is an infinite arithmetico-geometric series (of the form ab, $(a+d)br$, $(a+2d)br^2$ and so on) where $a = m, b = 1/2, d = m, r = 1/2$. Now, sum of an infinite arithmetico-geometric series is given by $\frac{ab}{1-r} + \frac{dbr}{(1-r)^2}$. Substituting, we clearly see that:

$$\text{E(Insertion Time of all elements)} = O(m) \tag{18.14}$$

Hence, the expected insertion time for a single element is going to be $1/m \times O(m) = O(1)$. Hence we have shown that cuckoo hashing involves constant expected time insertion.

18.4 Conclusion

In this lecture, we explored data structures to store data more efficiently in terms of space. In this context, we studied Bloom Filters, data structures which give constant time insertion and querying guarantees, with linear space but with a small probability for false positives. The representation studied here does not allow for deletion. We then considered Cuckoo Hashing, another approach which allows for deletions and gives constant time querying and deletion guarantees as well as a constant expected time insertion guarantee. This approach also involves no false positives but when naively implemented as described above consumes more space than a Bloom Filter since the entire element has to be stored as opposed to just 0s and 1s.

References

[1] M Mitzenmacher and E Upfal. *Probability and Computing: Randomized Algorithms and Probabilistic Analysis*. Cambridge University Press, 2005, Ch. 5, pages 106-110.