

CSE 203A: Randomized Algorithms

Spring 2026

Lecture 9: Cuckoo hashing and bloom filters

Date: April 17, 2026

Instructor: Daniel M. Kane

Scribe: Shishir Iyer

Lecture Overview

In this lecture, we introduced cuckoo hashing. By proving bounds on various collision probabilities, we established that cuckoo hashing provides amortized $O(1)$ inserts, with deterministically $O(1)$ lookups and deletes. We also discussed bloom filters, a memory-efficient probabilistic data structure. Bloom filters always correctly determine when an element is in a set, but have some error rate determining whether an element is not in the set. We looked into how large the data structure should be to get the desired error bound.

1 Introduction

This lecture continues our discussion of hashing — a technique widely used to store a subset of a larger “universe” and efficiently check for membership of said subset. Because of the pigeonhole principle, any function mapping the universe to a subset will inevitably have collisions. We discussed some techniques to mitigate this issue, such as a two-level hash table. However, this data structure requires a lot of preprocessing and only really works for static data structures.

Cuckoo hashing is a hashing technique that guarantees $O(1)$ performance for lookups and deletes. It also provides $O(1)$ average-case performance for inserts, allowing a mutable data structure and resolving the problem with the two-level hash table. However, in a worst-case scenario the time to insert can be as slow as $O(n)$, so we proved bounds on the probability this scenario happens and showed that it does not impact the expected $O(1)$ insert time.

Some applications also do not require the entire set to be stored in memory as a hash table would, as we only care about whether an element is in the set (rather than whether it isn't). A bloom filter is a probabilistic data structure that achieves this; it will always return true if an element is in the set, and it will return false with some small probability of error if an element is not in the set. We discussed how the desired error bound impacts the size of the data structure.

2 Main Definitions

Definition 2.1 (Cuckoo hashing). Cuckoo hashing is a hashing technique that provides perfect hashing dynamically — $O(1)$ lookups and deletes deterministically, and $O(1)$ inserts in the average case. This is done via two arrays A and B of size $m = O(n)$ (where n is the number of elements in the data structure) and two hash functions f and g . When inserting some $x \in U$, we can store it at either $A[f(x)]$ or $B[g(x)]$.

Definition 2.2 (Bloom filter). A bloom filter is a probabilistic data structure that checks for set

membership. It does so using k hash functions h_1, \dots, h_k and an array A of m bits. Inserting some $x \in U$ involves setting $A[h_i(x)]$ to 1 for all $i \in \{1, \dots, k\}$. Then, if $x \in S$, $A[h_i(x)]$ must be 1 for all $i \in \{1, \dots, k\}$.

3 Main Results

Theorem 3.1 (Cuckoo hashing expected collision chain length). *The expected length of a cuckoo hash collision chain is $O(1)$.*

Proof. The probability of a cuckoo hashing collision chain having k steps can be expressed as

$$\sum_{x_1, \dots, x_k} \Pr[f(x_1) = f(x_2), g(x_2) = g(x_3), f(x_3) = f(x_4), \dots]$$

If we assume all x_i 's are distinct (i.e. the hash collision chain can't loop back to an earlier value), these events are all independent. Therefore, assuming f and g are fully random functions, for a single chain x_1, \dots, x_k ,

$$\Pr[f(x_1) = f(x_2), g(x_2) = g(x_3), f(x_3) = f(x_4), \dots] = \left(\frac{1}{m}\right)^k$$

There are less than n^k such sequences in which all k elements are distinct (since you have at most n choices for each element). Assuming $m \geq 2n$,

$$\sum_{x_1, \dots, x_k} \Pr[f(x_1) = f(x_2), g(x_2) = g(x_3), f(x_3) = f(x_4), \dots] \leq \left(\frac{n}{m}\right)^k \leq \left(\frac{1}{2}\right)^k$$

As a result, the expected length of a collision chain is given by

$$\mathbb{E}[\text{chain length}] \leq \sum_k \left(\frac{1}{2}\right)^k = O(1)$$

□

Note that the result above assumes all x_i 's are distinct! Otherwise, the independence argument we used breaks down.

Theorem 3.2 (Probability of two or more cycles in cuckoo graph). *The probability of two or more cycles in the “cuckoo graph” (the graph formed by connecting $f(x_i)$ and $g(x_i)$ for each x_i in the collision chain) is at most $O(1/n)$.*

Proof. We can use a very loose union bound as follows:

$$\Pr[\text{two cycles}] \leq \sum_{a,b,c} \Pr[a, b\text{-cycle connected by path length } c]$$

Because $a + b + c$ distinct elements are contained in this connected cycle and the number of edges must be one more than the number of vertices, $a + b + c + 1$ collisions must happen in total. Assuming $m \geq 2n$,

$$\begin{aligned} \Pr[\text{two cycles}] &\leq \sum_{a,b,c} \Pr[a-, b\text{-cycle connected by path length } c] \\ &\leq \frac{n^{a+b+c}}{m^{a+b+c+1}} \leq \frac{(1/2)^{a+b+c+1}}{n} \\ &= O\left(\frac{1}{n}\right) \end{aligned}$$

□

Theorem 3.3 (Cuckoo hash function independence). *For cuckoo hashing to satisfy the assumptions provided by fully random functions f and g , $O(\log n)$ -wise independent hash functions are sufficient.*

Proof. Provided as a proof sketch in lecture. We would like to compute the expected number of chains of length k given k -wise independence. Since k elements in S could make up the chain and $k - 1$ pairs of elements are colliding, the expected number of chains of length k is

$$\mathbb{E}[\text{chains length } k] \leq \frac{n^k}{m^{k-1}} \leq O\left(\frac{n}{2^k}\right)$$

If $k > \log n$, this becomes $O(1)$. Since having too many elements in chains means we need to rehash, having $O(\log n)$ independence is thus sufficient. □

Theorem 3.4 (Bounds on bits in Bloom filter). *If ε is the upper bound on the false positive rate of a Bloom filter, the filter must contain at least $n \log_2(1/\varepsilon)$ bits, where $n = |S|$ is the number of elements stored.*

Proof. Let T be the set of values accepted by the Bloom filter. Generally, $|T| \leq O(\varepsilon|U|)$. If $|T| = O(\varepsilon N)$, where $N = |U|$, the number of size n subsets is

$$\binom{O(\varepsilon N)}{n} = O\left(\frac{\varepsilon N}{n}\right)^n$$

Meanwhile, the number of possible size n subsets of the entire “universe” is

$$\binom{N}{n} = \Omega\left(\frac{N}{n}\right)^n$$

The number of possible T 's is obtained by dividing the number of subsets in the universe by the number of subsets for each possible T , which leads to $\geq \Omega(1/\varepsilon)^n$ options for T . To encode this in bits, at least $n \log_2(1/\varepsilon)$ bits are needed. □

Theorem 3.5. *To guarantee a false positive probability of at most ε , we require $k \approx \log_2(1/\varepsilon)$ hash functions.*

Proof. The probability of a false positive is the probability that, for some $x \notin S$,

$$\Pr(A[h_1(x)] = \dots = A[h_k(x)] = 1)$$

The number of 1 entries in A must be at most nk , so we can bound this probability as

$$\Pr(A[h_1(x)] = \dots = A[h_k(x)] = 1) \leq \left(\frac{nk}{m}\right)^k$$

Taking $m = 2k$,

$$\Pr(A[h_1(x)] = \dots = A[h_k(x)] = 1) \leq \left(\frac{nk}{m}\right)^k = \left(\frac{1}{2}\right)^k \leq \varepsilon$$

$$k \geq \log_{1/2}(\varepsilon) = \log_2\left(\frac{1}{\varepsilon}\right)$$

□

4 Proof Ideas and Intuition

Theorem 3.1 demonstrates that in cuckoo hashing, for fully random functions f and g , the collision chains never get too long. This makes sense as longer chains would require more elements colliding exactly as shown in the probability equation above (i.e. $f(x_1) = f(x_2)$, $g(x_2) = g(x_3)$, etc.) Therefore, the probability of these longer chains decreases exponentially. Any insertion would potentially have to deal with one of these collision chains, so having the expected collision chain length be $O(1)$ means the expected insert time will be $O(1)$. This result does, however, depend on the fact that all x_i 's are distinct, which may not necessarily be the case. This is when we need to consider cycles in the cuckoo graph.

Similar reasoning applies for Theorem 3.2, which shows that the probability that the cuckoo graph contains two connected cycles decreases as $O(1/n)$. This is especially important because having two connected cycles forces us to rehash (which takes $O(n)$ time); any such graph will have more edges than vertices. Then by the pigeonhole principle, there simply won't be a way to assign all x_i 's to distinct locations. So having the probability of this happen decrease as $O(1/n)$ means we can preserve our $O(1)$ insertion time. Note that having a single cycle in the graph (even one connected to a longer chain) is still fine. This is because such a graph can have at most as many edges as vertices, so the issue above doesn't apply.

While Theorems 3.1 and 3.2 assume a fully random f and g , this is not a very practical assumption. So we instead look at k -wise independent hash functions. Recall that k -wise independence refers to a family of hash functions such that applying a random hash function to any k values results in those values being uniformly distributed. In similar reasoning to the previous theorems, we can then see that the probability of any given chain being of length k exponentially decreases if $k > \log n$. Therefore, $\log n$ independence is necessary and sufficient.

Theorem 3.4 shows how Bloom filter size increases as our desired false positive rate decreases. This proof works by treating the Bloom filter as encoding all possible accepting sets T in bits. Because

the error rate is across all elements in the universe, we can assume $|T| = O(\varepsilon N)$. Then, we simply determine how many size n subsets (representing the set we want to store) each possible T will have, followed by the number of possible size n subsets in total. Dividing these two values gives us the number of possible T 's, and our Bloom filter must encode all of these in order to achieve the desired error probability. This means that even though the Bloom filter also uses $O(n)$ memory just like a hash table does, the actual memory footprint is significantly smaller unless ε is very small.

Theorem 3.5 shows how the number of hash functions required to achieve the desired error probability increases with the logarithm of the error probability. If, for some $x \notin S$, all k of the corresponding entries for x 's hashes in the Bloom filter happen to be set to 1 by other elements of S , then we will see a false positive. Since there are at most nk entries that can be initialized to 1, since we set up to k entries as 1 for each of the n elements in S , and n entries in total, the probability that a single hash of x was already set to 1 is nk/m . We assume $m = 2nk$ because otherwise it is possible that every single entry in the Bloom filter is set to 1. The factor of 2 ensures a load factor of $1/2$, similar to our analyses with cuckoo hashing.

5 Examples

Example 5.1. For a cuckoo hash table storing $n = 6$ elements, the probability of having a 7-length chain is given by

$$\sum_{x_1, \dots, x_7} \Pr[f(x_1) = f(x_2), \dots, g(x_6) = g(x_7)] \leq \left(\frac{1}{2}\right)^7 = \frac{1}{128}$$

Note that we don't actually use n here, and we only use the fact that n/m (i.e. the load factor) is at most $1/2$.

Example 5.2. Suppose we want to store 100 8-bit integers in a Bloom filter with a false positive rate of 0.01. Our Bloom filter then requires

$$n \log_2 \left(\frac{1}{\varepsilon}\right) = 100 \log_2 \left(\frac{1}{0.01}\right) = 664 \text{ bits}$$

If we were to store these integers in a cuckoo hash table, we would need two arrays of length $m \geq 2n$ for best performance, which would require 3200 bits.

6 Further Remarks

Remark 6.1. It is still an open problem as to whether 6-wise (or any constant) independence is sufficient; most people seem to think so, but it hasn't been proven yet. 5-wise independence has been proven to be insufficient by constructing a super weird hash function family that gives you a lot of collisions.

Remark 6.2. The paper that Kane et. al. wrote on proving the bounds on k -independence for cuckoo hashing can be found here: <https://cseweb.ucsd.edu/~dakane/cuckoohashing.pdf>.

Remark 6.3. The proof of Theorem 3.5 assumes fully random hash functions, but it will also work with k -independent hash functions. In fact, it is possible to implement a Bloom filter with only two hash functions, generating our function family as follows:

$$\begin{aligned}h_1(x) &= f(x) \\h_2(x) &= f(x) + g(x) \\&\vdots \\h_k(x) &= f(x) + (k-1)g(x)\end{aligned}$$

7 Summary

In this lecture, we discussed cuckoo hashing and Bloom filters, and proved bounds on their probabilistic performance.

References

- [1] R. Motwani and P. Raghavan, *Randomized Algorithms*, Cambridge University Press, 1995.