

CSE 203A: Randomized Algorithms

Spring 2026

Lecture 9: Cuckoo Hashing & Bloom Filters

Date: April 17th

Instructor: Daniel M. Kane

Scribe: Xinwei Mai

1 Introduction

This lecture introduced Cuckoo Hashing and Bloom Filter.

Cuckoo Hashing is dynamical perfect hashing. It uses two random functions to create two arrays instead of one. In this way, it needs $O(n)$ space and can achieve look ups and delete in $O(1)$ deterministically and insert with an expected runtime of $O(1)$.

Bloom Filter is similar to traditional hashing table, but it does not store the element directly in an array. Instead, it uses k hash functions to map $x \in S$, where S is the subset of the universe U , to k different positions in a size m bit array with a value of 1. Then, for any $x \in S$, all $A(h_i(x)) = 1$; for $x \notin S$, we want some $A(h_i(x)) \neq 1$, but there is a small chance of ε such that all $A(h_i(x)) = 1$. By storing information in a bit array with the Bloom Filter, it requires less space than it would for a traditional hashing table.

2 Brief Reminders about Previous Topics

Hashing: We have an array A with a size of m and a hash function $h : U \rightarrow \{1, \dots, m\} = [m]$, where U is the universe of all elements. Then, we can store x at the position of $h(x)$ in A . Therefore, the lookup time is $O(1)$ instead of $O(\log(n))$ for sorting based approaches.

K-wise Independence: A list of hash functions $H = \{h_1, \dots, h_k\}$ is a k -wise independent family, if for any distinct $x_1, \dots, x_k \in U$, $y_1, \dots, y_k \in [m]$, where m is the length of the array: $\forall h \in H, Pr(h(x_1) = y_1, h(x_2) = y_2, \dots, h(x_k) = y_k) = \frac{1}{m^k}$

Perfect Hashing: A hashing table that has no collision with the worst case lookup time of $O(1)$ and a total memory of $O(n)$.

Secondary Hashing: A static perfect hashing. In stead of storing the element directly, the array A stores another array A_k at every position k . The primary hash function $h(x)$ will map x to the corresponding k^{th} position in A . Then, there is an secondary hash function h_k that maps to the corresponding position in the array A_k . The A_k arrays and hash functions are carefully designed such that there is no collision.

3 Main Lecture Content

Algorithm 1: Cuckoo Hashing

Input: $S \subset U$

Output: A hashing table T

Create two arrays A, B of size $m = O(n)$ with corresponding total random hash functions f, g .

To insert $x \in S$, f will first map x to a position in A .

If there is an element x_1 stored in that position, g will map x_1 to a place in B and x replaces x_1 's position in A .

Similarly, if there is an element x_2 stored in $B[g(x_2)]$, f will map x_2 to a place in A and x_1 replaces x_2 's position in B .

Keep doing this until all the elements in S are stored.

The two arrays A and B in Cuckoo hashing are both in $O(n)$, so the hash table takes $O(n)$ of space. The delete and look ups are both $O(1)$ since they are simply looking up elements in the array by computing hash functions and checking values.

To compute the expected runtime of insertion, it is the same of asking the expected number of steps that it will take to insert an element. In other words, the chain $f(x) = f(x_1), g(x_1) = g(x_2), \dots, f(x_{k-1}) = f(x_k), g(x_k)$ takes $k + 1$ steps to properly insert x .

Now the probability of a k steps chain is $Pr(\text{a chain lasts } \geq k \text{ steps}) \leq \sum_{x_1, x_2, \dots, x_k \in S} Pr(f(x) = f(x_1), g(x_1) = g(x_2), \dots, f(x_{k-1}) = f(x_k)) \leq (\frac{n}{m})^k$. This comes from an assumption that each of these x_i are distinct, each step (collision) has a probability of $\frac{1}{m}$, and there is k event for each element in the sum. We want to have $m \in O(n)$, so we set that $m = 2n$ then the probability is bounded by $(\frac{1}{2})^k$. If we want to know the expected length of a chain, that is $\mathbb{E}(\text{length of a chain}) = \sum_k Pr(\text{a chain lasts } \geq k \text{ steps}) = \sum_k \frac{1}{2^k} = O(1)$. So, now we have a perfect hash table that uses $O(n)$ spaces, has an $O(1)$ time for looks up and delete, and an $O(1)$ expected runtime of insertion.

However, that expected runtime is only in a constant time when there's no more than one loop in the chain. Consider a collision is an edge and that will be resolved at a node, then a single loop will have n edges and n nodes and therefore all the collision has been resolved and everything works well. A single loop with a long chain also works, since the long chain can be considered as the first collision that enters the single loop.

If there are at least two loops, then there are two cases for that. First, there is a bridge between the two loops. If there is an edge between the two loops, then there is $n + 1$ edges and n nodes and there is always one collision that can not be resolved. If there is more node on the bridge, by adding one node, one edge is created, so there is still $n + 1$ edges and n nodes. The probability of this case happens is $Pr(\text{two connected cycles with a bridge}) \leq \sum Pr(\text{cycle } a \text{ exists, cycle } b \text{ exists, the bridge } c \text{ exists})$ for all possible cycles a and b and all possible bridges c . This situation means that there are at least $a + b + c$ many distinct elements (suppose a, b, c also denotes the size of each objects), and there are in total of $a + b + c + 1$ collision need to happen. Then, $Pr(\text{this case happens}) \leq \frac{n^{a+b+c}}{m^{a+b+c+1}}$ because we have $a + b + c$ choices for the nodes and $a + b + c + 1$ choices for the edges, and each collision has a probability of $\frac{1}{m}$ to happen. Again, if we set $m \geq 2n$, then the probability of this case happens for each a, b and c , is bounded by $(\frac{1}{2})^{a+b+c+1}/n$. Summing up for all possible a, b and c , the total probability of a such case happens is bounded by $O(\frac{1}{n})$.

There is another case that the two loops can be stacked together without a bridge. That is another loop's starting point and its end point are on the same loop. It means that there exists at least

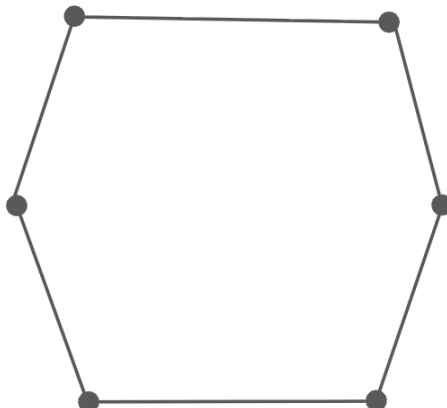


Figure 1: A single loop

one point outside of the loop that has two edges connect to the starting point and the end point correspondingly. Also, when adding one node to the external loop, one more edge will be added to the graph. Then, suppose there are a nodes in the cycle a and b nodes in the cycle b , there are $a + b + 1$ edges in total. By a similar computation as above, we can also show that the probability for this case to happen is also bounded by $O(\frac{1}{n})$. If either of these two cases happened, all elements must be rehashed until there is no collision. This process will take $O(n)$ time to do.

We can also do a further analysis on what is the expected number of chains with length k . Then, $\mathbb{E}[\text{the expected number of chains with length } k] \leq \frac{n^k}{m^{k-1}} \leq O(\frac{n}{2^k})$ for there are at most $k - 1$ collisions for k nodes and $m = 2n$. If $k \gg \log(n)$, then this expectation is small. So, there will not be lots of long chains. Also, in this analysis, we assume that the hashing functions f and g must be a total random functions. However, that is impossible to implement in practice. If there is rarely a chain with a length of $\log(n)$ in the table, then we only needs to have f and g as $O(\log(n))$ -wise independence. That means, as long as there is a chain with more than $O(\log(n))$ elements, with $O(\log(n))$ -wise independent f and g , all the analysis above is still valid.

In general, Cuckoo Hashing hashes an element through a long chain. If the long chain hits a cycle, it will go to the other direction. For example, the forward direction is $f(x) = f(x_1)$ then compute $g(x_1)$ and the loop starts here; the backward direction is $g(x_k) = g(x_1)$, $f(x_1) = f(x)$ then compute $g(x)$ and go to the other direction. If this enters another cycle or loops back to the first cycle, then the whole thing needs to be re-hashed. But as n gets larger and $m = 2n$, the two-loops cases are very unlikely to happen ($O(\frac{1}{n})$), so the algorithm works fine in practice. Moreover, since the looping cases are so rare, when there is a new element adding to S , the probability that we insert this element and get into a two-loops case is also bounded by $O(\frac{1}{n})$. So, overall, the expected time to insert an element is still a constant time as we have analyzed above.

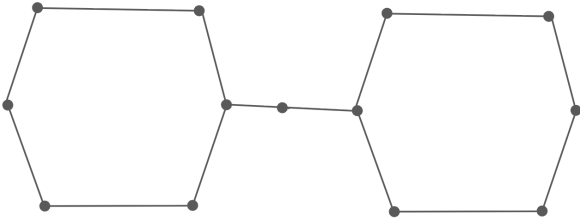


Figure 2: Two loops with a bridge

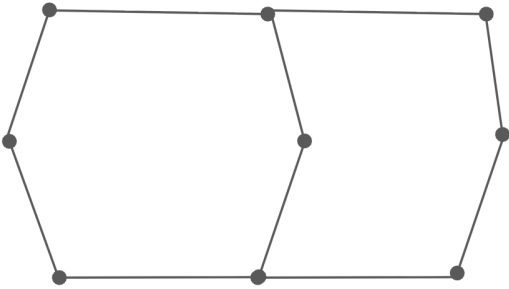


Figure 3: Two loops that without a bridge

Algorithm 2: Bloom Filters**Input:** $S \subset U$ **Output:** A look up table T

Have k hash functions h_1, h_2, \dots, h_k , where $k \approx O(\frac{1}{\epsilon})$ and ϵ is the false positive rate, and a bit array A with a length of $m = O(n)$.

If $x \in S$, then $A(h_i(x)) = 1$, for all $i = \{1, \dots, k\}$

A traditional hash table wants to store every elements in $S \subset U$. Suppose $|S| = n$ and $|U| = N$, Then, the total number of bits that we need store all possible $S \subset U$ with size n is larger than $\log_2(\binom{N}{n}) \approx nO(\log(\frac{N}{n})) \approx nO(\log(N))$ if we assume N is much larger than n as the universe is a much larger space. If we do not want to know what exactly the element is but just want to know if $x \in S$, a Bloom Filter can achieve this goal and save us some space.

The general idea is that we want a way that there is no false negative for all $x \in S$ but there is small probability ϵ that it will give a false positive for all $x \notin S$.

Now, let T be a set of x that are accepted, then $S \subset T$. Then, since there is much more elements in the universe U than in S and we should accept an ϵ fraction of that on average, the total size of $T \leq O(\epsilon|U|) = O(\epsilon N)$ with a probability of 50% by the Markov's inequality. For a such set T , it has $\binom{O(\epsilon N)}{n} = O(\frac{\epsilon N}{n})^n$ number of subset with a size of n , and there are in total of $\binom{N}{n} = \Omega(\frac{N}{n})^n$ number of S that we need to encode for the universe. So, the number of T that we need to do that is the ratio of $\frac{\#Ss}{\#size\ n\ subset \in T} \geq \Omega(\frac{1}{\epsilon})^n$. In other words, the data structure needs $\gg n \log(\frac{1}{\epsilon})$ bits to encode every possible $S \subset U$. Then if ϵ is a constant or $\frac{1}{\epsilon} \ll \frac{N}{n}$, then this regime requires less space to encode all information of S than a traditional has table if we were willing to have this small probability of false positive.

The Bloom Filter is a way of doing this. It uses k hash functions to map each $x \in S$ to a size m bit array A with a value of one. That is $A(h_i(x)) = 1$ for all $i = \{1, \dots, k\}$ and all $x \in S$. In this way, if we want to do a look up, then we just need to look up whether all $A(h_i(x)) = 1$. There is no false negative because we have encoded every $x \in S$ with all $A(h_i(x)) = 1$. However, there is a chance that for $x \notin S$, all $A(h_i(x)) = 1$. Suppose for each element in S , h_i maps it to a different position of A . Then, the number of entries that are one is bounded by kn . Now, suppose all hash functions are random, $Pr(A(h_i(x)) = 1, \text{ for all } i = \{1, \dots, k\}, \text{ and } x \notin S) \leq (\frac{kn}{m})^k$ since each position of A have the same probability to be 1. If we set $k = \log_2(\frac{1}{\epsilon})$ and $m = 2n$, then the false positive rate for an element is bounded by ϵ . In addition to that, the space that it requires is m because A is already a bit array, which is equivalent to $n \log(\frac{1}{\epsilon})$. Then the Bloom Filter fulfills our goal of having a space-optimal information storing algorithm while sacrificing a small fraction of accuracy.

4 Further Remarks

- We say that $O(\log(n))$ -wise independence is enough for Cuckoo Hashing, and Professor Kane's early work shows that 5-wise independence is not enough. Then, in practice 6-wise independence is enough for most of cases, but is 6-wise independence enough in theory?
- There are some improvements we can do to Cuckoo Hashing. For example, there is still a low probability that we have to rehash. For the elements causing the two-loops cases, we can use the secondary hashing to store these elements.
- There is also a de-amortized trick. We can store the elements that causes the insert time

to be $O(n)$ in a secondary hash table to make the loop ups and deletes clean, then slowly amortize this insert over the course of the next operations. Then, unless something that is unlikely to happen, the insertion is also having an almost worst case $O(1)$ runtime.

- We can do better to show that there is a tighter bound on the false positive rate for the Bloom Filter. That is because the hash functions h_i may not map to distinct positions for each $x \in S$, so the actual bound for the number of entries that are 1 is smaller than kn .
- We do not need to have all h_i to be completely independent. A k -wise independent is enough to generate these has functions. Or, in a much simpler way, we can use only $f(x)$ and $g(x)$ to generate a list of $\{f(x), f(x) + g(x), f(x) + 2g(x), \dots, f(x) + (k - 1)g(x)\}$. Then, we only need to store two store two hash functions and do the computation and look up a little faster.

5 Summary

Cuckoo Hashing uses two arrays and two hash functions that is $O(\log(n))$ -wise independence. If there is a collision in one array, it puts the input array in the collided position and uses the other hash function to hash the original element to the other array. The whole process is like a chain bouncing between two arrays. The look up and delete is always in a constant time, and the insert takes a constant on average. If there two loops in the chain, it requires a re-hash, which takes a $O(n)$ time. However, these cases are rare in practice, so it usually works fine.

Bloom filter is not a traditional hashing method. In stead of storing every elements explicitly in the array, it gives a way to check whether the element is not in the set S . It uses k different hash functions h_i from a k -wise independence family, and set $A(h_i(x)) = 1$. If $x \in S$, then all $A(h_i(x)) = 1$ so it will not give any false negative. By setting $k = \log_2(\frac{1}{\epsilon})$ and $m = 2n$, we can have the false positive rate bounded by ϵ . The method requires less space than a traditional hash table if ϵ is a constant or $\frac{1}{\epsilon} \ll \frac{N}{n}$, where $|S| = n$ and $|U| = N$.

References

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