CS 270: Combinatorial Algorithms and Data Structures Spring 2023

Lecture 13 — Feburary 28, 2023

Prof. Jelani Nelson *Scribe: Alec Li, Anna Deza*

1 Overview

In this lecture, we will look at the approximate membership query problem, and how Bloom filters [Blo70] provide an efficient solution using only $O(nw)$ bits of memory. We'll also look at Cuckoo hashing [PR01], a solution to the dynamic dictionary problem that gives $O(1)$ worst-case query time, an improvement over the expected $O(1)$ query time from hashing with chaining or with linear probing.

2 Bloom Filters

Recall the approximate membership problem; we want to maintain a set $S \subseteq [U]$ (where $n \coloneqq |S|$), subject to:

- `insert(x)`: Updates S to include x, i.e. $S \leftarrow S \cup \{x\}$
- `query(x)`: Query the system to ask if $x \in S$; i.e. return true if $x \in S$, false if $x \notin S$.

Here, we want the probability of a wrong answer to be at most ε.

We'll look at one (Monte Carlo) randomized data structure to solve this problem; it'll always be efficient, but it may give the wrong answer with some probability. In particular, we'll look at Bloom filters (from the 1970s) [Blo70].

A natural question is: what's the point of approximate membership? We've already seen dynamic hashing and linear probing, which gives linear space and expected constant time operations, and we can use these solutions to the dynamic dictionary problem to solve approximate membership.

The advantage of Bloom filters is that it uses only $o(nw)$ bits of memory; we use less space than just storing the keys themselves. In particular, it uses approximately $1.44n \ln \frac{1}{\varepsilon}$ bits (the 1.44 comes from $\log_2(e)$).

The lower bound for the space complexity is $\Omega(n \ln \frac{1}{\varepsilon})$ bits of space (i.e. removing the 1.44 constant multiple), which is actually achievable in the static case; it turns out it is impossible to achieve this space for the dynamic case, shown by Lovett and Porat in 2013 [LP13].

With Bloom filters, we first initialize a bit array A of m bits to all 0's. We then pick k independent fully random hash functions mapping $[U] \to [m]$, and implement the operations as follows:

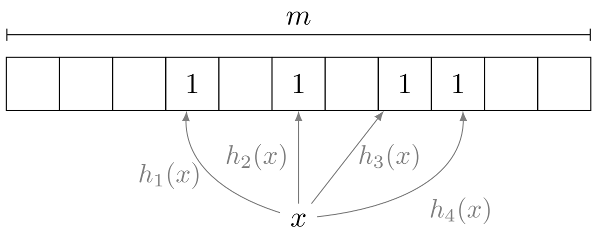
- `insert(x)`: set $A[h_i(x)] = 1$ for each i from 1 to k.
- `query(x)`: query $A[h_i(x)]$ for each i from 1 to k, and take the AND of all of them.

Note that if x was truly in the set, then all of these locations would be set to 1; otherwise, at least one of these location is probably 0, and we'd say no (there is the possibility that other elements have hashed to some of the same locations).

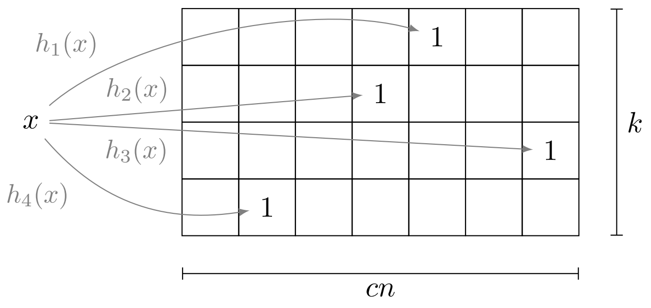
We could also analyze this algorithm to find the optimal values for m and k; the optimal value of m turns out to be $1.44n \ln \frac{1}{\varepsilon}$, and k to be $\Theta(\ln \frac{1}{\varepsilon})$.

We won't show the analysis of traditional Bloom filters here, since we run into independence problems, but we'll analyze a slightly different version, which is easier to explain.

In the traditional Bloom filter, we have an array of length $m = \Theta(n \log \frac{1}{\varepsilon})$, and we set k locations in the array to 1. (Note that we can possibly have two hash functions hashing to the same location.)



Today, we'll analyze a slightly different formulation; we have instead a 2D array with k rows and cn columns. When we insert x, we set the location $h_i(x)$ in row i to 1, for each $i = 1, \ldots, k$. Here, the crucial difference is that each row is independent from one other.



Notice that (in both versions) we never have any false negatives; we only have false positives. That is, we'll always be certain that an item is *not* in the set, but we could incorrectly say that an item *is* in the set.

Claim 2.1. If $x \notin S$, then $\mathbb{P}(\text{say } x \in S) \leq \varepsilon$.

Proof. Suppose we fix a row i. We have a false positive if each $A[i][h_i(x)]$ is already occupied for $i = 1, \ldots, k$. This means the false positive probability is $\mathbb{P}(A[i][h_i(x)] = 1)^k$. To compute this value, suppose we look at the expected number of items in S that collide under x:

$$\mathbb{E}[\#\text{ items in } S \text{ colliding with } x \text{ under } h_i] = \sum_{y \in S} \underbrace{\mathbb{P}(h_i(x) = h_i(y))}_{1/cn} = \frac{1}{c}.$$

Notice that the probability that at least 1 item collides with x is at most $\frac{1}{c}$ by Markov's inequality. This is because the random variable for the number of items that collided with x takes a value c times its expected value. (We can improve this analysis to get a better constant, but we omit it here.)

This probability we computed is the probability we get fooled in row i, so the probability we got fooled in *all* k rows is $\frac{1}{c^k}$, which we want to be at most ε. This gives $k = \Theta(\log \frac{1}{\varepsilon})$.

Notice that although we got a worse constant factor, we only needed 2-wise independence here, to reduce $\mathbb{P}(h_i(x) = h_i(y)) = \frac{1}{cn}$. □

If we wanted to get a better constant multiple for the space, we can reduce to the dictionary problem; we can map to a range where there is a small probability of collisions, and then store the set of $h(x)$'s using a dictionary that uses as little space as possible. Notice that the main reason why we were able to get away with using less memory in bloom filters is because we just set k bits to 1, instead of storing x. Here, we're doing something similar, storing $h(x)$ instead of storing x in the dictionary.

3 Cuckoo Hashing

The goal of Cuckoo hashing [PR01] is to solve the dynamic dictionary problem. In hashing with chaining and linear probing, the query and insertion time are both expected constant time, whereas in cuckoo hashing, query time is *worst case* constant (we only look in two locations), and insertion time is expected constant. It's an open problem to get a solution to the dynamic dictionary problem that is worst case constant in both query and insertion. (It's theoretically possible to get linear space and constant time query and insertion time, deterministically, but the best way we know of is to use a balanced binary search tree.)

To implement Cuckoo hashing, we pick two fully independent hash functions $h, g : [U] \to [m]$, where m is the size of the hash table. We then implement the operations as follows.

To query x, we check both $A[h(x)]$ and $A[g(x)]$. The invariant we'll maintain is that x will always be in either $A[h(x)]$ or $A[g(x)]$. If x is in neither location, then we say that x is not in the database.

For insertion, we put x in $A[g(x)]$. If $A[g(x)]$ was null, then we're done. However, if $A[g(x)]$ is already occupied, let x' be the old key in $A[g(x)]$ (let $j = g(x)$ for ease of notation). If $j = g(x')$, then we move x' into $h(x')$; otherwise, move x' to $A[g(x')]$ (since here $j = h(x')$). We then recurse in these two cases to insert x' if there are any additional conflicts.

Intuitively, we insert x into $A[g(x)]$, and move any existing element into their other hash location. If this element *also* conflicts with another element, we continue moving these elements until there are no conflicts. However, it is possible for this to go on forever (if there's a cycle).

To resolve this, if this process goes on for longer than say $10 \log n$ steps, we rebuild the entire hash table from scratch. That is, we pick new hash functions h and g and re-insert the items. We then repeat this process until it works, i.e. until we successfully insert every element back into the hash table. The main idea is that the probability of this occurring is very tiny, so it's unlikely this process will happen or go on for to long.

(This insertion algorithm is why this algorithm is called "cuckoo hashing"; a cuckoo chick pushes other eggs or young out of the nest when it hatches, much like how this algorithm pushes existing elements out when inserting a new item.)

It's clear that query time is worst case constant, but we'll show that the query time is expected constant in a little bit.

As a little detour, cuckoo hashing can also be used to solve the static approximate dictionary problem in $O(nr)$ bits of memory.

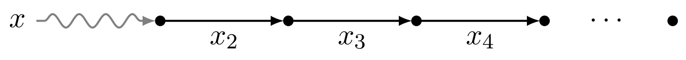
The approximate dictionary problem is similar to approximate membership; the only difference here is that we store a set of n key-value pairs, where all the values are r-bit strings. Notice that here we'll never actually store the keys. (This is in problem set 4; a hint is to use cuckoo hashing when the cuckoo graph has no cycles, where $\mathbb{P}(\text{no cycles}) \geq \frac{1}{2}$.)

3.1 Cuckoo Hashing Analysis

Definition 3.1 (Cuckoo Graph)**.** A *Cuckoo graph* is a multigraph (i.e. there could be multiple edges between the same vertices) illustrating the process of Cuckoo hashing. Vertices are locations in the hash table (so there are m vertices), and edges connect between $h(x)$ and $g(x)$ for each key $x \in S$, where S is the set of keys in the database (so there are n edges).

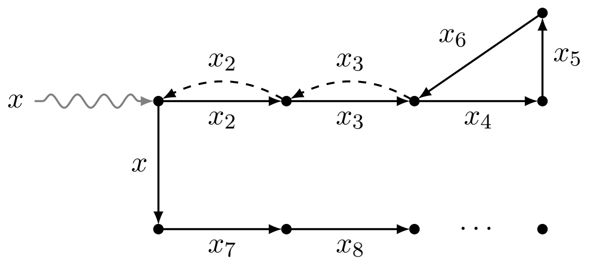
Let's look at the possible cases that can happen during an insertion.

- One case is that we have a **path**. (The squiggly line is the first hash of x, and other arrows denote the movement of elements due to conflicts.)



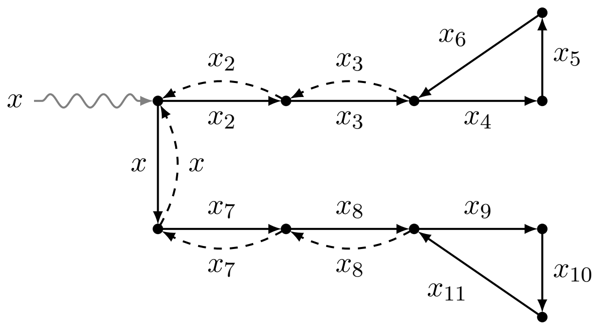
  Here, x hashes to where x_2 is originally, so x_2 moves to where x_3 is originally, and this continues until there are no collisions.

- Another case is that we have a **single cycle**. (Solid arrows denote the first movement of an element due to a collision, and dashed arrows denote a second movement due to another collision after the cycle.)



  Here, the movement of x_6 causes x_3 to get moved once more, which propagates back to x. This means that we now try hashing x using $h(x)$, which displaces x_7, etc., until we have no collisions along this second path.

- A last case is that we have a **double cycle**, which does actually go on infinitely.



Here, after we go through the first cycle, we hit another cycle in the second path; this causes an infinite loop of collisions that never resolves.

For the analysis, suppose we define:

- T: the runtime to do an insert
- P_k: the indicator $\mathbf{1}\{\text{have path of length} \geq k\}$
- C_k: the indicator $\mathbf{1}\{\text{have cycle of length} \geq k\}$
- D: the indicator $\mathbf{1}\{\text{have a double cycle}\}$

Our expected runtime is

$$\begin{aligned}\mathbb{E}[T] \leq \underbrace{\mathbb{E}\left[\sum_{k=1}^{\infty} P_k\right]}_{\text{path case}} + \underbrace{\mathbb{E}\left[\sum_{k=1}^{\infty} C_k\right]}_{\text{cycle case}} + \underbrace{\mathbb{P}(D=1)}_{\text{double cycle}} \cdot \underbrace{(10\log n + n\,\mathbb{E}[T])}_{\text{rebuild table}} \\ + \underbrace{\mathbb{P}(\text{path/cycle of length} \geq 10\log n)}_{\text{took too long}} \cdot \underbrace{(10\log n + n\,\mathbb{E}[T])}_{\text{rebuild table}}\end{aligned}$$

We can show that $\mathbb{P}(P_k = 1) = \exp(-\Omega(k))$ for m sufficiently large (say $m = 4n$), and we'll also show that $\mathbb{P}(C_k = 1) = \exp(-\Omega(k))$. This means the sums in the first two expectations converge to a constant.

We'll also show that $\mathbb{P}(D = 1)$ and $\mathbb{P}(\text{path/cycle of length} \geq 10\log n)$ are both $O(\frac{1}{n^2})$. This simplifies the runtime to

$$\begin{aligned}\mathbb{E}[T] &\leq O(1) + O(1) + O\left(\frac{1}{n^2}\right)(10\log n + n\,\mathbb{E}[T]) + O\left(\frac{1}{n^2}\right)(10\log n + n\,\mathbb{E}[T]) \\ \mathbb{E}[T] &\leq O(1) + O\left(\frac{1}{n}\right)\mathbb{E}[T] \\ \left(1 - O\left(\frac{1}{n}\right)\right)\mathbb{E}[T] &\leq O(1) \\ \mathbb{E}[T] &\leq O(1)\end{aligned}$$

Claim 3.2. $\mathbb{P}(P_k = 1) = \exp(-\Omega(k))$, in particular $\mathbb{P}(P_k = 1) \leq \frac{1}{2^k}$.

Proof. Notice that we can have a lot of possibilities for paths of length k; we have a choice of what elements are involved in the path. (We'll call each of these possibilities a "realization" of a path.)

The probability by union bound gives

$$\mathbb{P}(P_k = 1) \leq \sum_{\substack{\text{all possible paths} \\ P \text{ of length } k}} \mathbb{P}(\text{have } P)$$

Notice that the number of possible realizations is at most $m^{k+1} \cdot n^k$, since there are at most m possibilities for each of the $k+1$ vertices in the path, and n possibilities for each of the k edges in the path.

The probability of a fixed realization is

$$\mathbb{P}(\text{a fixed realization}) \leq \frac{1}{m} \cdot \frac{2^k}{m^{2k}}.$$

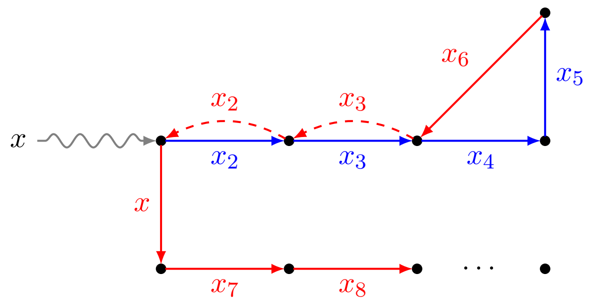
Here, the $\frac{1}{m}$ factor is because x must hash to the start of the path, and for each of the k edges along the path, we have a probability at most $\frac{2}{m^2}$ for the pair of hash functions to hash to the two locations incident to the edge.

If we fix $m = 4n$, then this simplifies to

$$\begin{aligned}\mathbb{P}(P_k = 1) &\leq (\#\text{ possible realizations}) \cdot \mathbb{P}(\text{fixed realization}) \\ &\leq m^{k+1} \cdot n^k \cdot \frac{2^k}{m^{2k+1}} \\ &= \left(\frac{2n}{m}\right)^k = \frac{1}{2^k}\end{aligned}$$

□

To show $\mathbb{P}(C_k = 1) \leq \exp(-\Omega(k))$, we break up the cycle into two parts; edges before the cycle (in blue), and edges after the cycle (in red).



If the entire cycle is of length k, then at least one part must be of length at most $\frac{k}{2}$. We can then reuse the same analysis we did for paths to get a similar bound for cycles.

The probability of a path or cycle of length at most $\geq 10 \log n$ is similar; it's also $\exp(-10 \log n) = O(\frac{1}{n^{10}})$, which is definitely $O(\frac{1}{n^2})$.

Claim 3.3. $\mathbb{P}(D = 1) = O\big(\frac{1}{n^2}\big)$.

Proof. $\square$

4 Preview of Power of 2 Choices

Think about hashing with chaining; the expected length of a linked list is $O(1)$, but the worst case load is on the order of $\frac{\log n}{\log \log n}$. To reduce this load, we can instead pick two random hash functions $h, g : [U] \to [m]$. We'll also augment the hash table to keep track of the length of the linked list.

When we insert x, we look at both $h(x)$ and $g(x)$, and choose the linked list with fewer items. Intuitively, this can only do better than with only one hash function. With high probability, max load turns out to be at most $\frac{\ln \ln n}{\ln 2} + \Theta(1)$. (This essentially goes from a max load of $O(\ln n)$ to $O(\ln \ln n)$.)

References

[Blo70] Burton H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7):422–426, Jul 1970.

[LP13] Shachar Lovett and Ely Porat. A space lower bound for dynamic approximate membership data structures. *SIAM Journal on Computing*, 42(6):2182–2196, Jan 2013.

[PR01] Rasmus Pagh and Flemming Friche Rodler. Cuckoo hashing. In Friedhelm Meyer auf der Heide, editor, *Algorithms — ESA 2001*, Lecture Notes in Computer Science, page 121–133, Berlin, Heidelberg, 2001. Springer.