1 Perfect Hashing

1.1 Overview

In the previous lecture, we analyzed Cuckoo Hashing, which still has a very low chance of collision. Also, Cuckoo Hashing requires fully randomize functions. Today we will try to find a perfect hashing scheme with no collisions, and which only requires pairwise randomized.

1.2 Goal

Definition 1. A hash function $h$ is perfect for $S \in [U]$ if it has no collisions, i.e. $h(x) \neq h(y)$ for all $x \neq y$ and $x, y \in S$.

Goal: For a given set $S$ of size $k$, find a perfect hash function $h : [U] \rightarrow [m]$, we want $m$ as small as possible.

1.3 Intuition and Easy solutions

- Identity function: $h(x) = x$, but with $m = U$.
- Giant lookup table via hash table, actually depends on which hash table you pick.
- Pairwise independent hash function with $m = O(k)$, but not perfect with $O\left(\frac{\log k}{\log \log k}\right)$ worst case lookup.

Now we still use a pairwise independent hash function $h$ but with more space than $O(k)$, we want to find an upper bound for $m$ such that no collisions will occur.

Lemma 2. With probability more than $\frac{1}{2}$ we can find a perfect random pairwise independent hash function $h$ with $m = k^2$ for $S$.

Proof. Using Markov, we have,

$$\Pr[h \text{ is not perfect for } S] = \Pr[h \text{ has at least 1 collision for } D] \leq \mathbb{E}[\text{number of collisions for } h].$$
By expanding $E[\text{number of collisions for } h]$ as pairs of collision indicators we get:

$$E = \sum_{x_1 < x_2, x_1, x_2 \in S} Pr[h(x_1) = h(x_2)] \leq \binom{k}{2} \max_{x_1, x_2} Pr[h(x_1) = h(x_2)] \leq \frac{k^2}{2m} \quad (1)$$

Therefore we know if we let $m = k^2$ and we randomly choose a hash function that is pairwise independent, we will have failure probability at most a half.

If we design a Las Vegas algorithm to repeatedly find a pairwise independent hash function, after a constant number of tries, we will get a perfect hash function with high probability.

### 1.4 Perfect Hashing

Now we have a way to find a perfect hash function with $m = k^2$. However, we want $m = O(k)$.

Suppose we first get a random hash function $h^* : [U] \rightarrow [m]$, with $m = O(k)$. This hash function may have collisions. Create a linked list for each collision.

Now suppose we map each linked list with size $k'$ with a perfect hash function with size $k'^2$, we can then flatten that link list out and store with some extra space to make it a perfect hash function.

More formally, we have $h^* : [U] \rightarrow [m]$, and $h_i : [U] \rightarrow [Z_i^2]$ to be a perfect hashing, where $Z_i$ is the number of elements that hash to cell $i$, or mathematically denoted as $|(h^*)^{-1} \cap S|$.

Record $Y_i = \sum_{j<i} Z_i^2$. We set our final perfect mapping as

$$h(u) = Y_i + h_i(u) \text{, where } i = h^*(u)$$

It is then easy to see that $h$ is perfect with range $\sum_{i=1}^m Z_i^2$, we need to estimate $\sum_{i=1}^m Z_i^2$.

Since total number of collisions equals to

$$\sum_{i=1}^m \frac{(Z_i^2)}{2} = \frac{1}{2} \left( \sum_{i=1}^m Z_i^2 \right) - \frac{k}{2},$$
By taking expected value of both sides we have:

\[ E\left[ \sum_{i=1}^{m} Z_i^2 \right] = 2E[\text{total number of collisions}] + k \leq \frac{2k^2}{2m} + k \quad (2) \]

If we let \( m = k \), we will get \( 2k \) to be the expected size of our hash function \( h \).

Now we get a Las Vegas algorithm to rebuild \( h \) until size of \( h \) is less or equal to \( 4k \). By Markov, each round our success probability is greater than \( \frac{2k}{4k} = \frac{1}{2} \), thus with \( O(1) \) rounds or \( O(k) \) total times we will success with high probability.

If we want to further achieve \( m = k \), we can use a lookup table for the perfect hashing, where we index each non-empty element in the mapping of perfect hashing \( h \) as \( I \), then we take \( h'(x) = I_{h(x)} \) which still runs in \( O(k) \) times.

\[ h(x) \quad I \quad [n] \quad [4n] \]

## 2 Lower bound on hashing

To hash a set \( S \) of size \( k \) in \([U]\), lots of scheme give \( O(k) \) word of space, where 1 word = \( \log U \) bits. A natural question is to ask, can we do better?

Suppose the hash table was stored using \( b \) bits, then the total number of possible representations you can have is at most \( 2^b \). Since your representations must include all possible subsets of size \( k \) of \( U \) we see that \( 2^b \geq \binom{|U|}{k} \geq \left( \frac{|U|}{k} \right)^k = 2^{k \log(|U|/k)} \). If \( k < \sqrt{|U|} \) then we see \( b \geq \frac{1}{2} k \log(|U|) \). However, \( \log(|U|) \) is the size of any word, and so we need \( \Omega(k) \) words. This means if we need to be able to has ALL POSSIBLE sets, then we cannot do better than a regular hash function.

## 3 Bloom Filters

This is a set membership data structure with some chance of false positives. In particular, for a particular set \( S \) you can get queries of the kind \( x \in S? \), if the answer is ‘yes’ you would like to be
always right, however if the answer is ‘no’, then you are allowed to fail with probability $1 - \delta$. It is possible to do this with $O(k \log(\frac{1}{\delta}))$ bits.

Applications of this structure:

- Use the filter before a slow operation (for example, chrome uses this to maintain a list of malicious urls).
- Database joins (‘Does this key have a different entry in the corresponding table?’)
- Bitcoin (to speed up wallet synchronization).

Let $n$ be the number of items, $m$ be the number of buckets. The datastructure picks up $k$ uniform random hash functions $h_1, \ldots, h_k$ where $k$ is a parameter to be decided later. You then store $\vec{y} \in \{0, 1\}^m$ where $y_j = 1$ iff $\exists x \in S, i \in [k], h_i(x) = j$. Respond with ‘yes’ to a query on $x$ iff $x \in \cap_{i \in [k]} Y_{h_i(x)}$.

We now analyze the failure probability of this. Let $p$ = the fraction of 0’s in an array. $E[p] = Pr[\text{any single entry is 0}] = (1 - \frac{1}{m})^n k \approx e^{-nk/m}$. The variables negatively associate and hence concentrate, which means $p$ is most probably going to be the expectation, upto a constant. The probability that one of these was 1 is $(1 - p)^k \approx (1 - e^{-nk/m})^k$. We will try to find the $k$ that minimizes this value. To do this, observe that $(1 - e^{-nk/m})^k = [(1 - e^{-z})^z]^n/m$ where $z = \frac{nk}{m}$. It sufficies to minimize with respect to $z$ which can be done by differentiating the log and setting it to 0. It turns out that at the minimum $k = \frac{m}{n} \ln(2)$ and $\delta < \frac{1}{e^k} = 0.618 \frac{m}{n}$. Setting $m = O(n \log(1/\delta))$ does the job.