CS5330: Optional problems on Hashing

April 11, 2020

Here are solution sketches to the optional problems on hashing. If anything is unclear, please talk to me or your TA.

1. Construct a 2-universal family of hash functions $\mathcal{H}$ from[1] $[n]$ to $[n]$ such that:

$$\mathop{\mathbb{E}}_{h\in\mathcal{H}} \max_{j\in[n]} |\{i \in [n] : h(i) = j\}| \geq \Omega(\sqrt{n}).$$

Extend your construction to be strongly 2-universal, not just 2-universal.

Let a random hash function be defined as follows. For any $1 \leq i < n$, let $h(i) = 1$ with probability $1/\sqrt{n}$ and otherwise, $h(i) = i + 1$; let $h(n)$ be a uniformly random element of $[n]$. Clearly, the expected number of elements which hash to 1 is $\frac{n-1}{\sqrt{n}} + \frac{1}{n} \geq \Omega(\sqrt{n})$. The hash family is also 2-universal. If i, j are distinct elements and neither is equal to n, then $\Pr[h(i) = h(j)] = \frac{1}{n}$ because the only way they collide is if both are hashed to 1. Moreover, $h(n)$ is independent of any of the other hashes. The family is not strongly 2-universal though because for example, $\Pr[h(1) = 3] = 0$.

To get strong 2-universality, we have to introduce "more randomness" in defining the hash function. To be precise, choose a random $j^* \in [n]$ uniformly and choose a random permutation $\pi : [n] \to [n]$. For any $i \in [n]$, let $h(i) = j^*$ with probability $1/\sqrt{n}$ and otherwise, $h(i) = \pi(i)$. The expected load on j^* is again $\Omega(\sqrt{n})$. Let's try to argue strong 2-universality. For any $i_1 \neq i_2$ and j,

$$\begin{aligned}
&\Pr[h(i_1) = j, h(i_2) = j] \\
&= \Pr[j^* = j] \cdot \Pr[h(i_1) = j^*, h(i_2) = j^*] + \Pr[j^* \neq j] \cdot \Pr[h(i_1) = j, h(i_2) = j \mid j \neq j^*] \\
&\leq \frac{1}{n} \cdot O\left(\frac{1}{\sqrt{n}}\right) \cdot O\left(\frac{1}{\sqrt{n}}\right) + 0 = O\left(\frac{1}{n^2}\right)
\end{aligned}$$

[1] $[n]$ denotes the set $\{1, \ldots, n\}$.

The hidden constants in the $O(\cdot)$ notation reflect the fact that $h(i_1)$ could equal j^* in two ways: either with $1/\sqrt{n}$ probability, it is directly set to j^*, or otherwise, the random permutation π defines $\pi(i) = j^*$. Now suppose $j \neq k$.

$$\begin{aligned}
&\Pr[h(i_1) = j, h(i_2) = k] \\
&= \frac{2}{n} \cdot \Pr[h(i_1) = j^*, h(i_2) = k] + \left(1 - \frac{2}{n}\right) \Pr[h(i_1) = j, h(i_2) = k \mid j, k \neq j^*] \\
&\leq \frac{2}{n} \cdot O\left(\frac{1}{\sqrt{n}}\right) \cdot O\left(\frac{1}{n}\right) + O\left(\frac{1}{n^2}\right) \\
&\leq O\left(\frac{1}{n^2}\right)
\end{aligned}$$

As mentioned in class, the presence of a constant factor multiplying $1/n^2$ does not usually affect applications. One can also get perfect strong 2-universality with the factor being 1; see https://cstheory.stackexchange.com/a/31094.

2. For any family $\mathcal{H}$ of hash functions from a finite set U to a finite set V, show that there exist $x, y \in U$ such that:

$$\Pr_{h \sim \mathcal{H}}[h(x) = h(y)] \geq \frac{1}{|V|} - \frac{1}{|U|}$$

where h is drawn uniformly from $\mathcal{H}$.

First, fix a hash function $h \in \mathcal{H}$. The total number of collisions under h is exactly: $\sum_{v \in V} \binom{|h^{-1}(v)|}{2}$. This is minimized when each $|h^{-1}(v)| = |U|/|V|$. Hence:

$$\Pr_{x,y}[h(x) = h(y)] = \frac{1}{\binom{|U|}{2}} |V| \cdot \binom{|U|/|V|}{2} \geq \frac{1}{|V|} - \frac{1}{|U|}$$

where x, y are sampled uniformly from U conditioned on $x \neq y$. Since this is true for any hash function, we can take the expectation over $h \in \mathcal{H}$:

$$\mathop{\mathbb{E}}_{h \in \mathcal{H}}\left[\mathop{\mathbb{E}}_{x,y}[1[h(x) = h(y)]]\right] \geq \frac{1}{|V|} - \frac{1}{|U|}$$

Interchanging the expectations, we get that:

$$\mathop{\mathbb{E}}_{x,y} \Pr_{h \in \mathcal{H}}[h(x) = h(y)] \geq \frac{1}{|V|} - \frac{1}{|U|}$$

Hence, there exists $x \neq y$ such that $\Pr_{h \in \mathcal{H}}[h(x) = h(y)] \geq \frac{1}{|V|} - \frac{1}{|U|}$.

3. (Adapted from Exercise 15.11 of MU) In a multi-set, each element can appear multiple times. Suppose that we have two multi-sets, S_1 and S_2, consisting of positive integers. We want to test identity between S_1 and S_2, meaning whether each item occurs the same number of times in S_1 and S_2. One way of doing this is to sort both sets and then compare the sets in sorted order. This takes $O(n \log n)$ time if each multi-set contains n elements.

   Consider the following Monte Carlo algorithm. Hash each element of S_1 into a hash table with cn counters; the counters are initially 0 and the ith counter is incremented each time the hash of an element is i. Using another table of the same size and the same hash function, do the same for S_2. If all the counters in the first table and the second table are the same, then report that the sets are the same. Otherwise, report that the sets are different.

   Assuming that the hash function is drawn from a 2-universal family, analyze the running time and error probability of the algorithm.

   When $S_1 = S_2$, clearly the algorithm is always correct. If $S_1 \neq S_2$, consider some element j such that its multiplicity in S_1 and S_2 are different. We claim that with probability at least $1 - 1/c$, no other element has the same hash value as j. Let $X = |\{i : i \neq j, h(i) = h(j)\}|$. Then, $\Pr[X > 1] \leq \mathbb{E}[X] \leq \sum_{i \neq j} \Pr[h(i) = h(j)] \leq n \cdot \frac{1}{cn} = \frac{1}{c}$ using 2-universality. In the event that j is hashed to its unique cell, the algorithm reports S_1 and S_2 to be different. The running time of the algorithm is $O(n)$ if the hash function can be evaluated in constant time.

4. We discussed the count-min sketch on an *insertion stream*, that is, each occurrence of an item x in the stream contributes +1 to the count of x. In *turnstile streams*, the data stream consists of pairs (i_t, c_t) where i_t is an item and c_t is an integer (possibly negative). Moreover assume that at any time T and for any item x:

$$\sum_{t : i_t = x, 1 \leq t \leq T} c_t > 0$$

   Explain how you could modify or otherwise use count-min filters to find heavy hitters in this situation.

   Let $\hat{f}_j(u) = \sum_{v : h_j(v) = h_j(u)} f(v)$, where $f(x) = \sum_{t : i_t = x} c_t$ is the cumulative count of x. By the same analysis of the count-min sketch done in class, for any item u, $\mathbb{E}[|\hat{f}_j(u) - f(u)|] \leq \|f\|_1 / w$, where $\|f\|_1 = \sum_x |f(x)|$. Hence, $\Pr[|\hat{f}_j(u) - f(u)| > 3\|f\|_1 / w] \leq 1/3$.

   Now the minimum of $\hat{f}_j(u)$ does not make sense as it may be significantly less than $f(u)$. We replace the min with median. Applying the Chernoff bound, $\Pr[|\mathrm{med}_j(\hat{f}_j(u)) - f(u)| > 3\|f\|_1 / w] \leq e^{-h/108}$. Hence, setting $w = \varepsilon/3$ and $h = O(\log 1/\delta)$, we get that $\Pr[|\mathrm{med}_j(\hat{f}_j(u)) - f(u) > \varepsilon \|f\|_1] \leq \delta$.

5. (Thanks to Eric Price for this problem.)

   The Python programming language uses hash tables (or "dictionaries") internally in many places. Until 2012, however, the hash function was not randomized: keys that collided in one

Python program would do so for every other program. To avoid denial of service attacks, Python implemented hash randomization—but there was an issue with the initial implementation. Also, in python 2, hash randomization is still not the default: one must enable it with the `-R` flag.

Find a 64-bit machine with both Python 2.7 and Python 3.5 or later.

(a) First, let's look at the behavior of `hash`("a") − `hash`("b") over $n = 2000$ different initializations. If `hash` were pairwise independent over the range (64-bit integers, on a 64-bit machine), how many times should we see the same value appear?

(b) How many times do we see the same value appear, for three different instantiations of python: (I) no randomization (`python2`), (II) python 2's hash randomization (`python2 -R`), and (III) python 3's hash randomization (`python3`)?

(c) What might be going on here? Roughly how many "different" hash functions does this suggest that each version has?

(d) The above suggests that Python 2's hash randomization is broken, but does not yet demonstrate a practical issue. Let's show that large collision probabilities happen. Observe that the strings "8177111679642921702" and "6826764379386829346" hash to the same value in non-randomized python 2.

Check how often those two keys hash to the same value under `python2 -R`. What fraction of runs do they collide? Run it enough times to estimate the fraction to within 20% multiplicative error, with good probability. How could an attacker use this behavior to denial of service attack a website?

Have fun!