Advanced Algorithms November 18, 2024

Lecture 16: Streaming Algorithms

Notes by Ola Svensson[1]

1 Hashing with limited independence

1.1 2-Universal Hash Families

Definition 1 (Carter Wegman 1979) *Family $\mathcal{H}$ of hash functions is 2-universal if for any $x \neq y \in U$,*

$$\mathbb{P}_{h\in\mathcal{H}}\left[h(x) = h(y)\right] \leq \frac{1}{N}\,.$$

Sometimes this definition is relaxed to allow $2/N$ (or $3/N$) instead of $1/N$, since that doesn't greatly affect the bucket sizes need to handle collisions.

We can design 2-universal hash families in the following way. Chose a prime $p \in \{|U|, \ldots, 2|U|\}$, and let

$$f_{a,b}(x) = ax + b \mod p \qquad (a, b \in [p], a \neq 0)\,.$$

And let

$$h_{a,b}(x) = f_{a,b}(x) \mod N\,.$$

The reason this construction works is that the integers modulo p form a field when p is prime, meaning that it is possible to define addition, multiplication, and division (except division by 0, which is undefined) among them.

Lemma 2 *For any $x \neq y$, and $s \neq t$, the following system*

$$\begin{aligned} ax + b &= s \mod p \\ ay + b &= t \mod p \end{aligned}$$

has exactly one solution.

Proof Since $[p]$ constitutes a finite field, we have that $a = (x-y)^{-1}(s-t)$ and $b = s - ax$. ■

Notice that the above proof also says that there is no solution with $a \neq 0$ if $s = t$. Moreover, there are $p \cdot (p-1)$ different choices of $a \in \{1, 2, \ldots, p-1\}$ and $b \in \{0, 1, \ldots, p-1\}$. Therefore, ($a$ and b are selected uniformly at random)

$$\mathbb{P}_{a,b}\left[f_{a,b}(x) = s \wedge f_{a,b}(y) = t\right] = \begin{cases} \frac{1}{p(p-1)} & \text{if } s \neq t, \\ 0 & \text{if } s = t. \end{cases} \tag{1}$$

Lemma 3 *$\mathcal{H} = \{h_{a,b} : a, b \in [p] \wedge a \neq 0\}$ is 2-universal.*

[1] **Disclaimer:** These notes were written as notes for the lecturer. They have not been peer-reviewed and may contain inconsistent notation, typos, and omit citations of relevant works.

Proof For any $x \neq y$,

$$\begin{aligned}
\mathbb{P}\left[h_{a,b}(x) = h_{a,b}(y)\right] &= \sum_{s,t \in [p]} \mathbb{1}_{s=t \bmod N} \cdot \mathbb{P}\left[f_{a,b}(x) = s \wedge f_{a,b}(y) = t\right] \\
&= \frac{1}{p(p-1)} \sum_{s,t \in [p]: s \neq t} \mathbb{1}_{s=t \bmod N} \qquad \text{(by (1))} \\
&\leq \frac{1}{p(p-1)} \frac{p(p-1)}{N} \\
&= \frac{1}{N}\,.
\end{aligned}$$

The inequality follows because for each $s \in [p]$, we have at most $(p-1)/N$ different t such that $s \neq t$ and $s = t \bmod N$. ■

Notice to store a hash function from our 2-universal hash family, we only need to store $a \in [p]$ and $b \in [p]$. This requires $O(\log |U|)$ space which is in stark contrast to the completely random hash function which required $O(|U| \log N)$ bits. Let us now calculate the expected number of collisions:

$$\sum_{x \neq y \in S} \mathbb{P}_{h \in \mathcal{H}}\left[h(x) = h(y)\right] \leq \binom{|S|}{2} / N\,. \tag{2}$$

Hence, if we select N to be greater than $|S|^2$, then we can have no collisions with high probability using 2-universal hashing. But in reality, such a large table is often unrealistic. A more practical method to deal with collisions is to use the aforementioned linked list or to use two layer hash tables.

Two layer hash tables work as follows. Let s_i denote the number of collisions at location i. If we can construct a second layer table of size $\approx s_i^2$, we can easily find a collision-free hash table to store all the s_i elements.Thus the total size of the second-layer hash tables is $\sum_{i=1}^{N} s_i^2$. Note that $\sum s_i(s_i - 1)$ is just the number of collisions calculated in Equation (2) (times 2), so

$$\mathbb{E} \sum_i s_i^2 = \mathbb{E} \sum_i s_i(s_i - 1) + \mathbb{E} \sum_i s_i \leq \frac{|S|(|S|-1)}{N} + |S| \leq 2|S|\,,$$

where we assume that our hash table has capacity $N \geq |S|$.

Do we need to know the size of the set? The above calculation shows that if N, the size of the hash table, is roughly the same as N, the size of the set being hashed, then the expected bucket size at each hash location (and hence the expected lookup time) is at most $2m/n$, which is $O(1)$. This leads to the question: *Is it necessary to know $|S|$ before we pick the size of the hash table?* The answer is no. Instead it suffices to adaptively increase the size of the hash table. Suppose the current hash table has size 2^i. As elements of the set arrive, keep hashing them until the expected bucket size rises above 2. This means that the set must be now of size about 2^i. Now *rebuild* the hash table to one of size 2^{i+1} using a new hash function. The total time spent in rebuilding is only $O(2^{i+1})$, which is still a constant factor times the size of the set we already have, which is 2^i. Thus the entire hashing takes $O(1)$ time per element (this is called amortized analysis).

1.2 Pairwise independence

The proof of 2-universality can also be adapted (allow $a = 0$) to give pairwise independence: We say $\mathcal{H}$ is 2-wise independent if for any $x \neq y$ and any pair of numbers $s, t \in [N]$,

$$\mathbb{P}_{h \in \mathcal{H}}\left[h(x) = s, h(y) = t\right] = \frac{1}{N^2}\,.$$

We also note that 2-wise independence implies 1-wise independence. We say that a family $\mathcal{H}$ is 1-wise independent if for any $x \in U$ and $s \in [N]$,

$$\mathbb{P}_{h\in\mathcal{H}}\left[h(x) = s\right] = \frac{1}{N}\,.$$

More generally, we remark that we can extend the above techniques and obtain a family of hash functions that is k-wise independent. For some prime number p, consider the family of functions

$$f_{a_0,\ldots,a_{k-1}}(x) = a_{k-1}x^{k-1} + \ldots + a_1 x + a_0,$$

where $a_0, \ldots, a_{k-1}$ are uniformly chosen in $\{0, 1, \ldots, p-1\}$. Similar to the invertible argument we used above, the proof that this construction is k-wise independent follows from the fact that the Vandermonde matrix

$$\begin{bmatrix} 1 & 1 & \cdots & 1 \\ x_1 & x_2 & \cdots & x_k \\ x_1^2 & x_2^2 & \cdots & x_k^2 \\ \vdots & \vdots & \ddots & \vdots \\ x_1^{k-1} & x_2^{k-1} & \cdots & x_k^{k-1} \end{bmatrix}$$

is invertible for distinct $x_1, \ldots, x_k$. So, in general we can store a k-wise independent hash function with only $O(k \log |U|)$ amount of memory.

2 Load balancing

Now we think a bit about how large the linked lists (i.e., number of collisions) can get. Let us think for simplicity about hashing n keys in a hash table of size n. Also assume for simplicity, assume that each ball is assigned to a random bin. (In other words, the hash function is completely random instead of just 2-universal as above.) This is the famous balls-and-bins calculation.

Clearly, the expected number of balls in each bin is 1. But the maximum can be a fair bit higher. For a given i,

$$\mathbb{P}\left[\text{bin}_i \text{ gets more than } k \text{ elements}\right] \leq \binom{n}{k} \cdot \frac{1}{n^k} \leq \frac{1}{k!}\,.$$

(This uses the union bound, that the probability that any of the $\binom{n}{k}$ events happen is at most the sum of their individual probabilities.) By Stirling's formula,

$$k! \approx \sqrt{2nk}\left(\frac{k}{e}\right)^k\,.$$

So if we choose $k = O\left(\frac{\log n}{\log\log n}\right)$ then $\frac{1}{k!} \leq \frac{1}{n^2}$. Hence

$$\mathbb{P}\left[\exists \text{ a bin } \geq k \text{ balls}\right] \leq n \cdot \frac{1}{n^2} = \frac{1}{n}\,.$$

So with probability larger than $1 - 1/n$,

$$\text{max load} \leq O\left(\frac{\log n}{\log\log n}\right)\,.$$

By changing the parameters little, this success probability can be improved to $1 - 1/n^c$ for any constant c.

Exercise: Show that with high probability the max load is indeed $\Omega(\log n / \log\log n)$.

2.1 Improved load balancing: Power of two choices

The above load balancing is not bad; no more than $O\left(\frac{\log n}{\log \log n}\right)$ balls in a bin with high probability. Can we modify the method of throwing balls into bins to improve the load balancing? How about the method you use at the supermarket checkout: instead of going to a random checkout counter you try to go to the counter with the shortest queue? In the load balancing case (especially in distributed settings) this is computationally too expensive: one has to check all n queues. A much simpler version is the following: when the ball comes in, pick 2 random bins, and place the ball in the one that has fewer balls. Turns out this modified rule ensures that the maximal load drops to $O(\log \log n)$, which is a huge improvement. This is called the *power of two choices*. The intuition why this helps is that even though the max load is $O(\log n / \log \log n)$, most bins have very few balls. For instance at most 1/10th of the bins will have more than 10 balls. Thus when we pick two bins randomly, the chance is good that the ball goes to a bin with constant number of balls. For a proof of this cool fact, I recommend various lecture notes around the web.

3 Streaming Algorithms

Suppose you have a massive set of data and you wish to calculate a function (think statistics) on the data. Think of the following examples:

- What are the most frequent search query on Google?
- What are the number of distinct cities on Facebook?

What properties of the algorithm would you like to have?

- It needs to be super fast;
- It cannot store the whole date in memory.

Streaming algorithms are algorithms that study this setting. More formally:

- The input is a long stream $\sigma = \langle a_1, a_2, \ldots, a_m \rangle$ consisting of m elements where each element takes a value from the universe $[n] = \{1, \ldots, n\}$.

  (In our previous Facebook example, m would be the number of profiles and n the number of different cities in the world.)

- Our central goal is to process the input stream (going from left to right) using a small amount of *space* s, i.e., to use s bits of random-access memory while calculating (approximately) some interesting function/statistics $\phi(\sigma)$.

How much memory do we need? We think of m and n as huge so we want s to be much smaller than m and n. In symbols, we want

$$s = o(\min\{m, n\})$$

and the holy grail is to achieve

$$s = O(\log m + \log n)$$

which is roughly the number of bits needed to store the length of the stream and to store some of the values.

Can we calculate anything using so little space? The answer is of course yes but we have to allow some error margins, i.e., approximate solutions. Randomization is also very useful. Before seeing some interesting examples, let us also mention that the number p of passes through the data is important. The more passes the easier it gets. Again the holy grail is to do a single pass, i.e., to have $p = 1$.

4 Finding Frequent Items Deterministically

(This is the Google search query problem).

We have a stream $\sigma = \langle a_1, \ldots, a_m \rangle$, with each $a_i \in [n]$. This implicitly defines a frequency vector $\mathbf{f} = (f_1, \ldots, f_n)$ (describing the number of times each query has been searched). Note that $f_1 + f_2 + \cdots + f_n = m$.

MAJORITY problem: If exists j such that $f_j > m/2$, then output j, otherwise, output "$\perp$".

FREQUENT problem with parameter k: Output the set $\{j : f_j > m/k\}$.

Unfortunately, both these problems requires space $\Omega(\min\{m, n\})$ if we limit ourselves to deterministic one-pass algorithms. However, we will see (in the next lecture) the Misra-Gries Algorithm'82 that solves the related problem of estimating the frequencies f_j and can also be used to solve the above problems in two passes.