princeton univ. F'18 cos 521: Advanced Algorithm Design

Lecture 4: Hashing with real numbers and their big-data applications

Lecturer: *Pravesh Kothari* Scribe:

Using only memory equivalent to 5 lines of printed text, you can estimate with a typical accuracy of 5 per cent and in a single pass the total vocabulary of Shakespeare. This wonderfully simple algorithm has applications in data mining, estimating characteristics of huge data flows in routers, etc. It can be implemented by a novice, can be fully parallelized with optimal speed-up and only need minimal hardware requirements. There's even a bit of math in the middle!

Opening lines of a paper by Durand and Flajolet, 2003.

As we saw in Lecture 1, hashing can be thought of as a way to *rename* an address space. For instance, a router at the internet backbone may wish to have a searchable database of destination IP addresses of packets that are whizzing by. An IP address is 128 bits, so the number of possible IP addresses is 2^{128}, which is too large to let us have a table indexed by IP addresses. Hashing allows us to rename each IP address by fewer bits. In Lecture 1 this hash was a number in a finite field (integers modulo a prime p). In recent years large data algorithms have used hashing in interesting ways where the hash is viewed as a *real number*. For instance, we may hash IP addresses to real numbers in the unit interval $[0, 1]$.

Example 1 (Dartthrowing method of estimating areas) Suppose gives you a piece of paper of irregular shape and you wish to determine its area. You can do so by pinning it on a piece of graph paper. Say, it lies completely inside the unit square. Then throw a dart n times on the unit square and observe the fraction of times it falls on the irregularly shaped paper. This fraction is an estimator for the area of the paper (assuming that each dart lands uniformly in the square).

Of course, the digital analog of throwing a dart n times on the unit square is to take a random hash function from $\{1, \ldots, n\}$ to $[0, 1] \times [0, 1]$.

Strictly speaking, one cannot hash to a real number since computers lack infinite precision. Instead, one hashes to *rational* numbers in $[0, 1]$. For instance, hash IP addresses to the set $[p]$ as before, and then think of number "i mod p"as the rational number i/p. This works OK so long as our method doesn't use too many bits of precision in the real-valued hash. This lecture we won't stress about the error between a uniform random number in $[0, 1]$ and the closest rational number of the form i/p - this is very small as long as p is large.

Thumb Rules About Concentration of Random Variables

1. As pointed out in Lecture 3 using the random variable "Number of ears," the expectation of a random variable may never be attained at any point in the probability space.

2. **Chebyshev's Inequality and Mean Estimation** If X is a random variable with mean μ and variance σ^2, then, by Chebyshev's inequality, the probability that X lies in the interval $[\mu - k\sigma, \mu + k\sigma]$ is $\geq 1 - 1/k^2$. For example, if you choose $k = 10$, then, this means that you can get an estimate of the mean within an additive error of at most 10σ with probability 99 percent from a single sample. If your goal is to get a $(1 \pm \epsilon)$ accurate estimate of the mean, this is meaningful only if $\sigma \ll \mu$.

3. **Variance Reduction by Averaging Independent Copies** We can use the simple trick of averaging independent copies to fix this issue of high variance giving us bad estimates. Given X as above, we can get a random variable with same mean but much smaller variance by averaging independent copies of X. More precisely, if $\bar{X} = 1/k \sum_{i=1}^{k} X_i$ where $X_1, X_2, \ldots, X_k$ are independent copies of X, then, by a direct computation $\mathbb{E}\bar{X} = \mu$ and $\mathbb{E}(\bar{X} - \mu)^2 = \sigma^2/k$. The proof is a direct computation:

$$\begin{aligned}
\mathbb{E}(\bar{X} - \mu)^2 &= \mathbb{E}[\left(\frac{1}{k}\sum_{i=1}^{k}(X_i - \mu)\right)^2] \\
&= \frac{1}{k^2} \sum_{1 \leq i,j \leq k} \mathbb{E}[(X_i - \mu)(X_j - \mu)] \\
&= \frac{1}{k^2} \sum_{1 \leq i \leq k} \mathbb{E}[(X_i - \mu)^2] \\
&= \sigma^2/k
\end{aligned}$$

Here, in the third line, we use the fact that since X_i and X_j are independent, $\mathbb{E}(X_i - \mu)(X_j - \mu) = \mathbb{E}(X_i - \mu)\mathbb{E}(X_j - \mu) = 0$.

Observe that this "Variance reduction" fact does not require X_is to be independent, just pairwise independence is enough!

0.1 Estimating the cardinality of a set that's too large to store

Continuing with the router example, suppose the router wishes to maintain a count of the number of *distinct* IP addresses seen in the past hour. Formally, we see a stream of bitstrings $x_1, \ldots, x_n$, among which there are N distinct strings. We wish to estimate N using as little memory as possible.

Option one: we could just hash every string as it arrives, and store everything we see. Using some of the clever hashing ideas from lecture one, we could use $\Theta(N)$ memory. Once the entire stream has passed, we just output the number of filled entries in the hash table.

Option two: we could subsample the stream, taking every string independently with probability p, then using the previous idea to estimate the number of distinct elements from the subsampled stream. Unfortunately, if $n >> N$, successfully distinguishing between streams that look like $a_1, \ldots, a_1, a_2, \ldots, a_N$ (where a_1 is repeated $n-N$ times) and $a_1, \ldots, a_1$ (repeated n times) would require $p > 1/N$. Therefore, any such idea either runs the risk of storing n/N (which is still very large) different elemnts (if p is big), or runs the risk of doing a horrible job estimating N (if p is small).

So both of these options are conceptually simple, but use tons of memory. Here's a better idea: pick a hash function h that maps to $[0,1]$ (assume that $h(x)$ is uniformly sampled in $[0,1]$, and independent for all x for the following analysis. Obviously this isn't exactly true for any $h(\cdot)$, but it's a good approximation if we do a good job building our hash function).

Now, keep a string x initialized to $\perp$, and use the following update rule: when x_i streams past, compute $h(x_i)$. If $h(x_i) < h(x)$, update $x := x_i$. Otherwise, let x_i pass and do nothing. What is this accomplishing? Well, each $h(x_i)$ is independent and uniformly random in $[0,1]$. At the end of the stream, $Y = h(x)$ is the minimum of N uniformly random draws from $[0,1]$ (Y is a random variable because our hash function is random, even though the stream is fixed). So as N goes up, we should expect Y to go down - and looking at Y gives us some idea of what N might have been.

Lemma 1
$\mathbf{E}[Y] = \frac{1}{N+1}$ *and* $\sigma^2(Y) \leq 1/(N+1)^2$.

The expectation looks intuitively about right: the minimum of N random elements in $[0,1]$ should be around $1/N$.

Let's do the expectation calculation. The probability that $Y > x$ is the probability that N uniformly random draws from $[0,1]$ are all $> x$. So $Pr[Y > x] = (1-x)^N$. Therefore, we can write:

$$\mathbf{E}[Y] = \int_{x=0}^{1} Pr[Y > x]dx = \int_{x=0}^{1} (1-x)^N = -(1-x)^{N+1}/(N+1)|_{x=0}^{1} = 1/(N+1).$$

(Here's a slick alternative proof of the $1/(N+1)$ calculation that we also discussed in the class. Imagine picking $N+1$ random numbers in $[0,1]$ and consider the chance that the $N+1$th element is the smallest. By symmetry this chance is $1/(N+1)$. Now, consider evaluating this probability in the following alternate way: given the first N numbers, the $N+1$th random number is the smallest only if it is smaller than the minimum of the first N random numbers. This happens with probability equal the the minimum of the first N random numbers. Thus, the probability that the $(N+1)$th number is the smallest is precisely the expected value of the minimum of the first N random numbers.)

The variance calculation is similar. Observe that the probability that $Y^2 > x$ is just the probability that $Y > \sqrt{x}$, which happens with probability $1-\sqrt{x}$. So again $Pr[Y^2 > x] = (1-\sqrt{x})^N$. We get:

$$\mathbf{E}[Y^2] = \int_{x=0}^{1} Pr[Y^2 > x]dx = \int_{x=0}^{1} (1-\sqrt{x})^N = -\frac{2(1-\sqrt{x})^{N+1}((N+1)\sqrt{x}+1)}{(N+1)(N+2)}|_{x=0}^{1} = \frac{2}{(N+1)(N+2)}.$$

So $\sigma^2(Y) = \frac{2}{(N+1)(N+2)} - \frac{1}{(N+1)^2} = \frac{N}{(N+1)^2(N+2)} < 1/(N+1)^2$.

Now, we could potentially hope to estimate the number of distinct elements by looking at Y. Indeed, if Y behaved exactly as its expectation suggests, then, $\hat{N} = \frac{1}{Y} - 1$ would give a good estimate of the number of distinct elements. Quantitatively, if $Y \in (1 \pm \epsilon/2)\mathbb{E}[Y]$, then, $|\hat{N} - N| \leq \epsilon N + 1$. This, however, requires that $\sigma^2(Y) \ll \frac{1}{N+1}$ which is not quite the case right now.

We can now use the third thumb rule though - we can get a random variable that has the same expectation but much smaller variance by just averaging independent copies! The analysis just amounts to repeating the experiment k times independently (simultaneously, storing k numbers instead of just one), and taking the average. The average clearly has the same expectation, although now the variance goes down by a factor of k (and therefore the standard deviation by a factor of $\sqrt{k}$). Therefore, by Chebyshev's inequality we get:

$$Pr[|\bar{Y} - 1/(N+1)| > \varepsilon\sqrt{k} \cdot \frac{1}{\sqrt{k}(N+1)}] < \frac{1}{\varepsilon^2 k}.$$

So if we set $k = t/\varepsilon^2$, we get a multiplicative $(1 \pm \varepsilon)$-approximation for $N+1$ with probability $1/t^2$. Note that this only requires storing $k << N$ numbers, so the savings compared to the simple initial proposals is huge.

0.1.1 Pairwise Independent Hash Functions

All this assumed that the hash functions are random functions from 128-bit numbers to $[0, 1]$. Let's now show that it suffices to pick hash functions from a pairwise independent family, albeit now yielding an estimate that is only correct up to some constant factor. Specifically, we'll take the *median* (instead of the mean) of the k trials, and show that this estimate lies inside the window $[\frac{1}{5N}, \frac{5}{N}]$ with high probability.

For a particular h, we'll bound the probability that we hash N different inputs and the smallest hash is not in $[\frac{1}{5N}, \frac{5}{N}]$. We'll do this by separately bounding the probability of falling below $1/5N$ and above $5/N$.

First, for any particular h, and any x, the probability that $h(x) < 1/5N$ is exactly $1/5N$. Therefore, by a union bound, the probability that $h(x) < 1/5N$ for *any* x is at most $1/5$.

Now we want to bound the probability that $h(x) > 5/N$ for all x. To do this is a little more involved. Define Z_i to be the random variable which is 1 if $h(x_i) < 5/N$ and 0 otherwise, and $Z = \sum_i Z_i$. Then $h(x) > 5/N$ for all x if and only if $Z = 0$. So we want to upper bound the probability that $Z = 0$. Observe first that each Z_i is 1 with probability exactly $5/N$, so $\mathbf{E}[Z] = 5$. Also, because each of our hash functions are pairwise independent, we get that $\sigma^2(Z) = \sum_i \sigma^2(Z_i) \leq 5$. Therefore, $\sigma(Z) < \sqrt{5}$, and we can conclude by Chebyshev's inequality that $Pr[Z = 0] \leq 1/5$.

Taking a union bound over both events, we see that the probability that for a given $h(\cdot)$, $\min_i h(x_i)$ lies in $[1/5N, 5/N]$ with probability at least $3/5$. Finally, observe that as long as $> k/2$ of the trials have $\min_i h(x_i) \in [1/5N, 5/N]$, then the median will definitely lie in $[1/5N, 5/N]$ as well. As each trial succeeds independently with probability $3/5 > 1/2$, we can apply a Chernoff bound to see that if we take $k = \Omega(1/\ln(\delta))$ trials, at least half succeed with probability $1 - \delta$ (and therefore the median is within a factor of 5).

0.2 Estimating document similarity

One of the aspects of the data deluge on the web is that often one finds duplicate copies of the same thing. Sometimes the copies may not be exactly identical: for example mirrored copies of the same page but some are out of date. The same news article or blog post may be reposted many times, sometimes with editorial comments. By detecting duplicates and near-duplicates internet companies can often save on storage by an order of magnitude.

Note that detecting *exact* duplicates would be easy: just hash every document and mark two documents with the same hash as duplicates. Also, detecting near-duplicates among n documents in time n^2 would also be easy: just compare all pairs of documents. We want to get a near-linear runtime but still be able to detect near duplicates.

We present a technique called *locality-sensitive hashing* that allows this approximately. It is a hashing method such that the hash preserves some "sketch" of the document. Two documents' similarity can be estimated by comparing their hashes. This is an example of a burgeoning research area of hashing while preserving some *semantic* information. In general finding similar items in databases is a big part of data mining (find customers with similar purchasing habits, similar tastes, etc.). Today's simple hash is merely a way to dip our toes in these waters.

The formal definition of a locality-sensitive hash is the following: for a given similarity measure $sim(A, B)$, defined on the set of documents, a locality-sensitive hash family $\mathcal{H}$ is a hash family satisfying $Pr_{h \leftarrow U(\mathcal{H})}[h(A) = h(B)] = sim(A, B)$. That is, documents that are more similar are more likely to have the same hash.

With respect to documents, one common measure of similarity between two documents is to use the "bag of words" approach. That is, every document is viewed as nothing more than an unordered set of words, and two documents are similar if they contain many similar words. Formally, the *Jaccard similarity* of two documents/sets A, B is defined to be $|A \cap B| / |A \cup B|$. This is 1 iff $A = B$ and 0 iff the sets are disjoint.

Basic idea: Pick a random hash function mapping the underlying universe of elements to $[0, 1]$. Define the hash of a set A to be the *minimum* of $h(x)$ over all $x \in A$. Then by symmetry, $\Pr[\text{hash}(A) = \text{hash}(B)]$ is exactly the Jaccard similarity. (Note that if two elements x, y are different then $\Pr[h(x) = h(y)]$ is 0 when the hash is real-valued. Thus the only possibility of a collision arises from elements in the intersection of A, B.) Thus one could pick k random hash functions and take the fraction of instances of $\text{hash}(A) = \text{hash}(B)$ as an estimate of the Jaccard similarity. This has the right expectation but we need to repeat with k different hash functions to get a better estimate.

The analysis goes as follows. Suppose we are interested in flagging pairs of documents whose Jaccard-similarity is at least 0.9. Then we compute k hashes and flag the pair if at least $0.9 - \epsilon$ fraction of the hashes collide. Chernoff bounds imply that if $k = \Omega(1/\epsilon^2)$ this flags all document pairs that have similarity at least 0.9 and does not flag any pairs with similarity less than $0.9 - 3\epsilon$.

To make this method more realistic we need to replace the idealized random hash function with a real one and analyse it. That is beyond the scope of this lecture. Indyk showed that it suffices to use a k-wise independent hash function for $k = \Omega(\log(1/\epsilon)$ to let us estimate Jaccard-similarity up to error ϵ. Thorup recently showed how to do the estimation with pairwise independent functions. This analysis seems rather sophisticated; let me know if you happen to figure it out.

Bibliography

1. Broder, Andrei Z. (1997), *On the resemblance and containment of documents*, Compression and Complexity of Sequences: Proceedings, Positano, Amalfitan Coast, Salerno, Italy, June 11-13, 1997.

2. Broder, Andrei Z.; Charikar, Moses; Frieze, Alan M.; Mitzenmacher, Michael (1998), *Min-wise independent permutations*, Proc. 30th ACM Symposium on Theory of Computing (STOC '98).

3. Gurmeet Singh, Manku; Das Sarma, Anish (2007), *Detecting near-duplicates for web crawling*, Proceedings of the 16th international conference on World Wide Web, ACM.

4. Indyk, P (1999). A small approximately min-wise independent family of hash functions. Proc. ACM SIAM SODA.

5. Thorup, M. (2013). `http://arxiv.org/abs/1303.5479`.