PRINCETON COS 521: ADVANCED ALGORITHM DESIGN

Hashing: Preliminaries

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 whizing 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. Furthermore, this renaming is done probabilistically, and the renaming scheme is decided in advance before we have seen the actual addresses. In other words, the scheme is *oblivious* to the actual addresses.

Formally, we want to store a subset S of a large universe U (where $|U| = 2^{128}$ in the above example). And $|S| = m$ is a relatively small subset. For each $x \in U$, we want to support 3 operations:

- $insert(x)$. Insert x into S.
- $delete(x)$. Delete x from S.
- $query(x)$. Check whether $x \in S$.

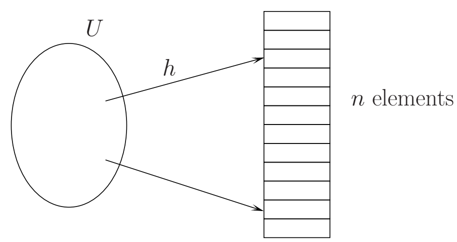

Figure 1: Hash table. x is placed in $T[h(x)]$.

A hash table can support all these 3 operations. We design a hash function

$$h : U \longrightarrow \{0, 1, \ldots, n-1\} \quad (1)$$

such that $x \in U$ is placed in $T[h(x)]$, where T is a table of size n. Typically, we can assume that $m \leq n \ll |U|$, although we will talk about some applications where we hash to a set with size $n < m$.

Since $|U| \gg n$, multiple elements can be mapped into the same location in T, and we deal with these collisions by constructing a linked list at each location in the table.

One natural question to ask is: how long is the linked list at each location?

This can be analysed under two kinds of assumptions:

1. Assume the input is the random.

2. Assume the input is arbitrary, but the hash function is random.

Assumption 1 may not be valid for many applications.

Hashing is a concrete method towards Assumption 2. We designate a set of hash functions $\mathcal{H}$, and when it is time to hash S, we choose a random function $h \in \mathcal{H}$ and hope that on average we will achieve good performance for S. This is a frequent benefit of a randomized approach: no single hash function works well for every input, but the average hash function may be good enough.

Hash Functions

What do we want out of a random hash function? Ideally, we would hope that h "evenly" distributes the elements of S across the hash table. One option would be to map every element in U to a random value in $[n]$. However, constructing such a "fully random" hash function is very expensive: we would need to build a lookup table with $|U|$ rows, each storing $\log_2(n)$ bits to specify the value of $h(x) \in [n]$ for one $x \in U$. At this cost, we might as well have just stored our original data in a $|U|$ length array – it's often simply impossible.

The goal in hashing is to find a *cheaper* function (fast and space efficient) that's still *random enough* to evenly distribute elements of S into our table. For a family of hash functions $\mathcal{H}$, and for each $h \in \mathcal{H}$, $h : U \longrightarrow [n]$[1], what we mean by "random enough".

[1] We use $[n]$ to denote the set $\{0, 1, \ldots, n-1\}$

For any $x_1, x_2, \ldots, x_m \in S$ ($x_i \neq x_j$ when $i \neq j$), and any $a_1, a_2, \ldots, a_m \in [n]$, ideally a random $\mathcal{H}$ should satisfy:

- $\Pr_{h \in \mathcal{H}}[h(x_1) = a_1] = \frac{1}{n}$.
- $\Pr_{h \in \mathcal{H}}[h(x_1) = a_1 \wedge h(x_2) = a_2] = \frac{1}{n^2}$. Pairwise independence.
- $\Pr_{h \in \mathcal{H}}[h(x_1) = a_1 \wedge h(x_2) = a_2 \wedge \cdots \wedge h(x_k) = a_k] = \frac{1}{n^k}$. k-wise independence.
- $\Pr_{h \in \mathcal{H}}[h(x_1) = a_1 \wedge h(x_2) = a_2 \wedge \cdots \wedge h(x_m) = a_m] = \frac{1}{n^m}$. Full independence (note that $|U| = m$).

Generally speaking, we encounter a tradeoff. The more random $\mathcal{H}$ is, the greater the number of random bits needed to generate a function h from this class, and the higher the cost of computing h. The challenge is to prove that, even when we use few random bits, the hash stable still performs well in terms of insert/delete/query time.

Goal One: Bound expected number of collisions

As a first step, we want to understand the expected length of a single linked list. Note that this is just the first step towards understanding the runtime of our desired operations. Assume that $\mathcal{H}$ is a pairwise-independent hash family.

Let L_x be the length of the linked list containing x; this is just the number of elements with the same hash value as x. Let random variable

$$I_y = \begin{cases} 1 & \text{if } h(y) = h(x), \\ 0 & \text{otherwise.} \end{cases} \tag{2}$$

So $L_x = 1 + \sum_{y \in S; y \neq x} I_y$, and

$$\mathbb{E}[L_x] = 1 + \sum_{y \in S; y \neq x} \mathbb{E}[I_y] = 1 + \frac{m-1}{n} \tag{3}$$

Usually we choose $n > m$, so this expected length is less than 2. Later we will analyse this in more detail, asking how likely is L_x to exceed say 100.

The expectation calculation above doesn't need full independence; pairwise independence would actually suffice. In fact, we don't even need pairwise independence, we just need the probability of a collision to be small. This motivates the next idea.

2-*Universal Hash Families*

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

$$\Pr_{h \in \mathcal{H}}[h(x) = h(y)] \leq \frac{1}{n} \tag{4}$$

Exercise: Convince yourself that this property is weaker than pairwise independence – i.e. that every pairwise independent hash function also satisfies (4).

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

$$f_{a,b}(x) = ax + b \mod p \qquad (a, b \in [p], a \neq 0) \tag{5}$$

Then let

$$h_{a,b}(x) = f_{a,b}(x) \mod n \tag{6}$$

One this to note about $f_{a,b}$ is that $f_{a,b}(x_1) \neq f_{a,b}(x_2)$ if $x_1 \neq x_2$. Why? If $f_{a,b}(x_1) = f_{a,b}(x_2) = s$, then

$$a(x_1 - x_2) = 0 \mod p,$$

which can't be the case if $a \in 1, \dots p-1$ and $(x_1 - x_2) \neq 0$. Can someone tell me why?

This isn't immediately useful, because we will still have a hash collision when $f_{a,b}(x_1) = f_{a,b}(x_2) \mod n$, but it's helpful to note.

[2] How do we know that such a prime exists? This is due to Bertrand's Postulate, which exactly states that such a prime exists. Second, how do we find such a prime? One option is to guess random numbers between $|U|$ and $2|U|$, check if they're prime, and continue until we find one. The Prime Number Theorem states that each guess is likely to be prime with probability roughly $1/\log(|U|)$. Also, the AKS primality test lets us test whether a number is in fact prime in time $\text{poly}(\log(|U|))$. Alternatively, one could imagine an online pre-computed database of primes that lie in the correct range.

Lemma 1. *For any $x_1 \neq x_2$ and $s \neq t$, the following system*

$$ax_1 + b = s \mod p \quad (7)$$
$$ax_2 + b = t \mod p \quad (8)$$

has exactly one solution (i.e. one set of possible values for a, b). In that solution, $a \neq 0$.

Proof. If you're familiar with modular arithmetic, this is clear. Since p is a prime, the integers $\mod p$ constitute a finite field. This implies that any element in $[p]$ has a multiplicative inverse $\mod p$, so we know that $a = (x_1 - x_2)^{-1}(s - t)$ and $b = s - ax_1$.

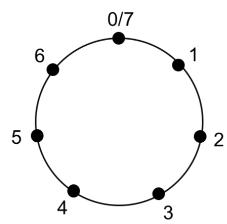

Figure 2: Modular arithmetic for prime $p = 7$.

It's not to hard to see this directly with a little thought. We want to claim that

$$a(x_1 - x_2) = (s - t) \mod p$$

has a unqiue solution a. Without loss of generality, assume that $x_1 > x_2$. When we multiply $(x_1 - x_2)$ by an integer, we're moving around the circle pictured in Figure 2 in increments of $(x_1 - x_2)$. Since p is prime, at each step before the p^{th} step, it better be that we hit a new element of $[p]$ on the circle. Otherwise, we would have found that $(x_1 - x_2)$ (which is $< p$) multiplies by some other number $< p$ to equal a multiple of p. This of course can't be true when p is prime.

So, as we multiply $(x_1 - x_2)$ by integers in $[p]$, we hit $(s - t) \mod p$ exactly once. □

By Lemma 1, since there are $p(p-1)$ different possible choices of a, b:

$$\Pr_{a,b \leftarrow U(\{1,\ldots,p-1\} \times \{0,\ldots,p-1\})}[f_{ab}(x_1) = s \wedge f_{ab}(x_2) = t] = \frac{1}{p(p-1)} \quad (9)$$

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

Proof. For any $x_1 \neq x_2$,

$$\begin{aligned}
&\Pr[h_{a,b}(x_1) = h_{a,b}(x_2)] && (10)\\
=& \sum_{s,t\in[p],s\neq t} \mathbf{1}[s = t \mod n)] \cdot \Pr[f_{a,b}(x_1) = s \wedge f_{a,b}(x_2) = t] && (11)\\
=& \frac{1}{p(p-1)} \sum_{s,t\in[p],s\neq t} \mathbf{1}[s = t \mod n] && (12)\\
\leq& \frac{1}{p(p-1)} \frac{p(p-1)}{n} && (13)\\
=& \frac{1}{n} && (14)
\end{aligned}$$

where $\mathbf{1}$ is an indicator function (that is, $\mathbf{1}[x] = 1$ if statement x is true, and $\mathbf{1}[x] = 0$ otherwise). Equation (13) follows because for each $s \in [p]$, we have at most $(p-1)/n$ different t such that $s \neq t$ and $s = t \mod n$. □

Can we design a collision free hash table then?

Solution 1: Collision-free hash table in $O(m^2)$ space.

Say we have m elements, and the hash table is of size n. Since for any $x_1 \neq x_2$, $\Pr_h[h(x_1) = h(x_2)] \leq \frac{1}{n}$, the expected number of total collisions is just

$$\mathbb{E}[\sum_{x_1 \neq x_2} h(x_1) = h(x_2)] = \sum_{x_1 \neq x_2} \mathbb{E}[h(x_1) = h(x_2)] \leq \binom{m}{2} \frac{1}{n} \quad (15)$$

Let's pick $n \geq m^2$, then

$$\mathbb{E}[\text{number of collisions}] \leq \frac{1}{2} \quad (16)$$

and so

$$\Pr_{h\in H}[\exists \text{ a collision}] \leq \frac{1}{2} \quad (17)$$

So if the size the hash table is large enough, we can easily find a collision free hash function. In particular, if we try a random hash function it will succeed with probability 1/2. If we see a collision when inserting elements of S into the table, we simply draw a new random hash function and try again. The expected function of this proceedure is:

$$\mathbb{E}[\text{time to insert } m \text{ items}] = m + \frac{1}{2}m + \frac{1}{4}m + \ldots = 2m.$$

Solution 2: Collision-free hash table in $O(m)$ space.

At this point, we have designed a hash table that has no collisions. The drawback is that it is that our table must be large: m^2 to store only m elements. But in reality, such a large table is often unrealistic. We may use a two-layer hash table to avoid this problem.

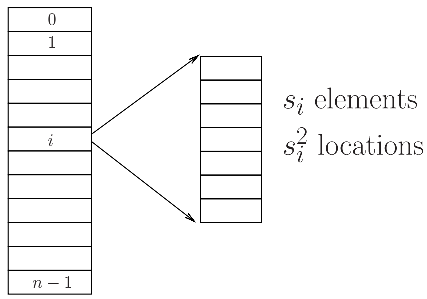

Figure 3: Two layer hash tables.

Specifically, let s_i denote the number of collisions at location i. If we can construct a second layer table of size 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 $\sum_{i=0}^{m-1} s_i^2$.

To bound the expectation size of $\sum_{i=0}^{m-1} s_i^2$, we note that this sum is nearly equal to the total number of hash collisions, which we bound in Equation (15)! Specifically,

$$\mathbb{E}[\sum_i s_i^2] = \mathbb{E}[\sum_i s_i(s_i - 1)] + \mathbb{E}[\sum_i s_i] = \frac{m(m-1)}{n} + m \leq 2m \tag{18}$$

Including the first layer, we have now designed a hash table of expected size $3m$ to store m elements (so some overhead, but much less than before).

Load Balancing

In our 2-level construction, we cared about limiting the *total size* of the hash table, and we were able to do so by bounding $\sum_{i=1}^{m} s_i^2$. However, we did not bound each s_i individually – it could be that some buckets of the first hash table are much larger than others. In some applications of hashing, this is something you want to avoid.

A simple example is when your hash table is distributed and each bucket (or a small set of buckets) is stored on a separate machine. The is a common architecture in large "no-SQL" databases like Amazon's DynamoDB or Apache Cassandra. In the distributed case, memory isn't a shareable resource across machines, so we care about showing that no s_i is too large (i.e no machine is overloaded).

Another example arises when hashing is used to distribute workload across multiple machines. As a toy example, suppose I look up directions from Princeton, NJ to Boston, MA on Google maps. Google has many different serves computing efficient driving routes and one potential strategy is to use a hash function to choose what server to send your request (i.e hash the start and end locations).

Question: Why is *hashing* a good strategy? Why not just send the request to an arbitrary or even randomly chosen server?

Load Balancing for Fully Random Hash Functions

Suppose we have n values, $a_1, \ldots, a_n$, from some universe $|U|$, and we want to hash these values to a table of size n. This is often called the "balls-into-bins" problem because we can think about hashing as randomly throwing balls into bins and seeing how many balls each bin has. It's convenient to analyze the case when the number of balls equals the number of bins, although this isn't always the setup.

In the first lecture, we weren't able to obtain bounds on the maximum load of a particular bin – we just showed that, on average, the bins weren't too overloaded. This could be done using Markov's inequality.

It turns out that we *can* get a bound on the maximum load using Chebyshev's inequality. Let's just consider the first bin and how many balls fall into it. Let $X_i = \mathbf{1}[\text{ball } i \text{ falls into bin 1}]$. Assume that we are using a 2-independent hash function, so:

$$\mathbb{E}[X_i] = \frac{1}{n}.$$

What's the variance of X_i?

$$\mathbf{Var}[X_i] = \mathbb{E}[X_i^2] - \mathbb{E}[X_i]^2 = \frac{1}{n} - \frac{1}{n^2} \leq \frac{1}{n}.$$

Now, let $X = \sum_{i=1}^{n} X_i$. X is the total number of calls that land in bin 1 and $\mathbb{E}[X] = 1$.

What's the variance of X? Since each X_i, X_j are *pairwise independent*,

$$\mathbf{Var}[X] = \sum_{i=1}^{n} \mathbf{Var}[X_i] \leq 1.$$

From Chebyshev's inequality, we therefore have that:

$$\Pr[|X - 1| \geq \sqrt{2n}] \leq \frac{1}{2n}.$$

So bin 1 has load $\leq \sqrt{2n} + 1$ with probability $1/2n$, and this exact same bound holds for all other bins. Thus, by a **union bound**, every bin has load $\leq \sqrt{2n} + 1$ with probability $1/2$. That's not bad! For $n = 1,000,000$, we can say that the maximally loaded bin has $\lesssim 1400$ elements. Shortly, we will see how to get an even tighter bound than $O(\sqrt{n})$.

With our Chernoff bound in place, let's revisit our "balls-in-bins" analysis. Using a Chebyshev bound, we bound the max load of n bins after inserting n balls by $O(\sqrt{n})$. The Chernoff bound will do exponentially better.

Again, we will analyze things one bin at a time. Let $X_i = \mathbf{1}[\text{ball } i \text{ falls into bin } 1]$ and let $X = \sum_{i=1}^{n} X_i$. $\mu = \mathbb{E}X = 1$. To apply Chernoff we will assume fully random hash functions[3]. Since $\mu = 1$, from Theorem **??**, we have that:

$$\Pr[X \geq 1 + 6\log n] \leq e^{-6\log n/3} \leq \frac{1}{n^2}.$$

So bin 1 gets $\leq 1 + 6\log n$ balls with probability at least $(1 - 1/n^2)$. By a union bound, we conclude that *all bins* have $\leq 1 + 6\log n$ with probability $1 - 1/n$.

This bound of $O(\log n)$ on the maximum load of any bin improves exponentially on our bound of $O(\sqrt{n})$ from Chebyshev. Moreover, it holds with much higher probability. We could have succeeded with probability $(1 - 1/n^c)$ for any constant c if we slightly increase the constant factor on $6\log n$.

[3] There was a question about this in class. It's actually possible to prove Chernoff bounds using $O(\log n)$-wise independence, which is much better than full independence, but not as simple as the 2-wise independence we assume for our Chebyshev bound. See recent work on improving over $O(\log n)$ independence in or even more recent work considering "power of two choices" like methods . ; and

Power of Two Choices

The above $O(\log n)$ bound is very good, but it turns out that a simple alternative hashing scheme can do *even better*. Consider 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 line. In the hashing setting this is computationally too expensive: one has to check all n queues. A much simpler version is to pick 2 random bins and place the ball in the one with fewer balls when the ball comes in. It turns out that this modified rule ensures that the maximal load drops to $O(\log\log n)$, which is a huge improvement. The *power of two choices* was first proven in the conference version of [4].

[4]

How about three choices? 4? d? Surprisingly, there's not much to be gained after 2. The bound only improves to $O(\log\log n/\log d)$ for d choices.

Other Considerations: Dealing with Adversaries

Other issues arise when hashing is used in a distributed way instead of to build data structures on a single machine. An interesting one is the issue of *resistance to adversarial attacks*. In particular, a user submitting requests to a centralized server may be able to *learn* the server's hash function, even if that function is chosen randomly.

Question: How easy is it to learn our 2-universal hash function described in (6)?

This can open the door for denial-of-service (DoS) attacks that intentionally issue "colliding" requests. Even if an adversary does not have the resources to take down a large web service, they may have enough to take down one or several servers underlying that service.

Graph Sparsification

This is not quite related to hashing but we will show another application of Chernoff bounds. This will be our excuse to introduce two new and important ideas: uniform sampling and stratified union bound.

A *sparsifier* for a graph is another graph that is smaller in some respect (number of edges/vertices usually) while preserving some properties. The most well-studied notion of graph sparsifiers is a *cut* sparsifier – informally speaking, such a sparsifier preserves the size of all cuts up to an error of a multiplicative $(1 \pm \epsilon)$ factor.

Definition 2. *For $1 > \epsilon > 0$, an ϵ-cut sparsifier for a graph a graph $G(V, E)$ is a (edge weighted) graph $H(V, E')$ on the same vertex set V such that for every $S \subseteq V$, we have:*

$$(1-\epsilon)|E_G(S,\overline{S})| \leq |E_H(S,\overline{S})| \leq (1+\epsilon)|E_G(S,\overline{S})|$$

Here, $E_G(S,\overline{S})$ is the set of all edges of G with exactly one end point in S. And $E_H(S,\overline{S})$ is the sum of the weights of all edges with exactly one end point in S.

What might we do with such a sparsifier? For one, even the existence of such a sparsifier is somewhat surprising as a combinatorial statement regardless of the algorithmic applications. And naturally, if we are interested in computing various notions of optimal cuts (e.g., min cuts, max cuts, sparsest cuts, min/max bisections, among several others – if you are curious, google search these terms), we can simply work with the sparsifier thereby reducing the runtime and still get an almost optimal guarantee.

A consequence of a powerful line of work begun by Karger and then later by Benczur Karger has led to a remarkable theorem that we will see more details on in a later class: *every graph admits a spectral sparsifier – a strengthening of cut sparsifiers – with $O(n)$ edges!*. In a later class, we will prove using the power *matrix Chernoff* bound, this theorem with $O(n \log n)$ edges. Right now, we will settle for something weaker but still quite powerful:

For every graph G with min cut of size k, there is an ϵ-sparsifier with $O(\frac{m}{k\epsilon^2}\log n)$ edges.

We will prove this result by a very simple *sampling* scheme as summarized in the next theorem.

Theorem 1. *For any graph $G(V,E)$, let $H(V,E')$ be a weighted graph constructed as follows: 1) Choose a parameter $0 \le p = c\frac{\log n}{k\epsilon^2}$ where k is the size of a minimum cut in G, 2) for each edge $e \in E$, include it in H independently with probability p with weight $1/p$. Then, with probability at least $1 - 1/n$, H is an ϵ-sparsifier of G wth $O(\frac{|E|}{k\epsilon^2}\log n)$ edges.*

We will use the following consequence of analyzing Karger's contraction algorithm for min-cut (also a HW problem).

Fact 1. *In any graph G with min cut of size k and any $\alpha \ge 1$, the number of cuts of size at most αk is at most $n^{2\alpha}$.*

Proof. First, let's understand what happens in H to the size of a fixed cut, say $S \subseteq V$. The number of edges in H that fall into the cut $(S,\overline{S})$ is a random variable and equals $\sum_{e \in cut(S)} \mathbf{1}(e \in H)$. Because we sampled the edges independently, $\mathbf{1}(e \in H)$ are independent 0-1-valued random variables with mean p. By linearity of expectation, the expected number of edges sampled from $cut(S)$ in H is exactly $p|E_G(S,\overline{S})|$ and thus their weight in expectation is exactly $|E_G(S,\overline{S})|$. Next, by Chernoff bounds, the chance that the fraction of edges from $cut(S)$ in H deviates more than a multiplicative $(1 \pm \epsilon)$ factor is at most $O(e^{-O(p|E_G(S,\overline{S})|\epsilon^2)})$.

How large can this probability be? If we were to calculate it for S being a minimum cut (in which case, this probability is the largest possible over all cuts), then the expression evaluates to $e^{-O(clogn)} \le 1/n^{O(c)}$. This is an inverse polynomial bound.

If we wanted to use union bound to argue that we all the cuts are simultaneously well approximated in H, then, we are in tough luck since the tail probability above is only inverse exponential but there are 2^{n-1} many cuts! Thus, a *naive* union bound fails!

This brings us to the key idea: *stratified* union bound. We will bucket the cuts in G into $B_1, B_2, \ldots,$ such that B_i contains all cuts of size in $[2^{i-1}k, 2^i k)$. Then, by Fact 1, the number of cuts in B_i is at most $n^{2^{i+1}}$. On the other hand, for every cut in B_i, $|E_G(S,\overline{S})| \ge 2^{i-1}k$ and thus, by the same argument as above, the probability that the number of edges from $cut(S)$ sampled in H is $\in (1 \pm \epsilon)|E_G(S,\overline{S})|$ is $1 - O(e^{-c2^{i-1}logn})$. By union bound, the same holds simultaneously for every cut in B_i with probability at least $1 - O(n^{2^{i+1}}e^{-c2^{i-1}logn}) = 1 - O(e^{-(c-4)2^{i-1}logn})$.

By a union bound over all the buckets, we now obtain that the event above happens simultaneously for all the buckets with probability at least $1 - \sum_{i=1}^{\infty} O(e^{-(c-4)2^{i-1}logn}) = 1 - O(e^{-\log n}) = 1 - 1/n^{O(1)}$ if $c \ge 5$, say.

Next, the expected number of edges sampled in H is exactly pm where m is the number of edges in G. By applying the Chernoff bound again, we conclude that with probability at least $1 - 1/n^{O(1)}$, the number of edges sampled in H is at most $O(pm)$. By a union bound, all the cuts are $1 \pm \epsilon$ approximated and the number of edges sampled is $O(pm)$ with probability at least $1 - 1/n^{O(1)}$. $\square$

Bibliography