MIT 6.854/18.415: Advanced Algorithms Spring 2016

Lecture 1 – February 3, 2016

Prof. Ankur Moitra *Scribe: Ray Hua Wu and Zachary Funke*

1 Hashing and Load Balancing

1.1 Motivation and Background

Hashing and load balancing are ideas in computer science that can be very powerful. A correct implementation of hashing in an algorithm can yield drastically superior asymptotic efficiency. However, when a hash table is used to store items, there is always the chance that two items are stored in the same address, causing a collision; there are several ways to mitigate collisions while also optimizing the memory required by the hash table and the run time of functions operating on the data structure.

1.2 Setup

Simply put, hashing is just a way to rename an address space.

Let's consider a universe $\mathbf{U}$ (say, the set of all possible IP addresses). The size of this set would be enormous: 2^{128}! We want a better way to manage this large set so that we can perform desirable operations on it.

Imagine that we want to store a subset $\mathbf{S}$ of the universe $\mathbf{U}$ such that $|\mathbf{S}| = m \ll |\mathbf{U}|$. We would like to support three operations of interest:

- Insert(x): Add x to $\mathbf{S}$
- Delete(x): Remove x from $\mathbf{S}$
- Query(x): Check if x is in $\mathbf{S}$

The usual approach to this is through hashing, wherein we map objects from our universe to a hash table with n slots, where n is much smaller than the size of $\mathbf{U}$:

$$h : \mathbf{U} \to [n]$$

$[n]$ just means the numbers $1, 2, \ldots, n$. If $|\mathbf{U}| > n$, there can be collisions where, for two distinct objects x and y, it just so happens that $h(x) = h(y)$. There will be many collisions when $|\mathbf{U}| \gg n$.

A standard strategy for dealing with collisions is called 'hashing with chaining.' If two items hash to the same address (that is, the hash function yields the same value for both items), you may store a linked list at that address containing all such conflicting items. The issue is how long the linked list becomes; in fact, the entire strategy relies on the fact that the linked lists do not become too

long, since we are only able to search them linearly. You can imagine the worst case being given by finding the entire set residing in the linked list of a single address, and performing a linear search along that linked list - in which case we'd have made no improvement at all by using a hash table over a simple list.

2 Universal Hashing

To reduce collisions, we want to ensure that the elements of $\mathbf{S}$ get "evenly distributed" amongst our n hash table buckets. One way to make sure this happens with good probability for any set $\mathbf{S}$ is to choose h completely randomly – i.e. each element of $\mathbf{U}$ maps to a random bucket.

Unfortunately, if we choose a hash function h that happens to be a completely random function from $\mathbf{U}$ to $[n]$, we would actually need $|\mathbf{U}| \log n$ bits to describe it. The problem is that the size of $|\mathbf{U}|$ is still huge; for our IP address example, it is 2^{128}. We would be better to just store an array with $|\mathbf{U}|$ buckets and avoid collisions entirely.

Instead, what we want is something "almost random" that can be represented efficiently. Consider this definition from Carter and Wegman [1]: a family of hash functions $\mathcal{H}$ is 2-universal if, for any pair $x \neq y$,

$$\mathbf{Pr}_{h \in \mathcal{H}}[h(x) = h(y)] \leq \frac{1}{n}$$

A completely random function certainly satisfies this property. However, we will see that much more efficient hash functions do as well.

Here we're thinking about choosing h uniformly at random from a set of hash functions $\mathcal{H}$. When we were choosing a completely random function, $\mathcal{H}$ was the set of all possible mappings from $\mathbf{U}$ to n. For our more efficient hash functions, it will be a much smaller set.

2.1 Basic Bounds

First, let's see what a universal hash function is good for with some easy calculations. By the linearity of expectation,

$$\mathbb{E}[C] \leq \binom{m}{2}\left(\frac{1}{n}\right)$$

where C is the number of collisions, which we want to minimize. Recall that m is the size of the set $\mathbf{S}$ that we're hashing. We can also find the expected value of the size of the list at $h(x)$, denoted by L_x:

$$\mathbb{E}[L_x] \leq 1 + \frac{m-1}{n}$$

We claim that if we take $n > m^2$, then $\mathbb{E}[C] \leq \frac{1}{2}$. By Markov's inequality, this means that the probability of having *any* collisions is bounded by:

$$\mathbf{Pr}_{h \in \mathcal{H}}[C \neq 0] \leq \frac{1}{2}$$

Accordingly, we can keep choosing random hash functions and will quickly find one with no collisions for set $\mathbf{S}$. Notice that this property of requiring $n > m^2$ to have no collisions with decent probability is reminiscent of the Birthday Paradox.

2.2 Two-Level Hashing

What if we want us use less than m^2 space? We can actually achieve an expected number of collisions below 1 with just $n = O(m)$ space if we implement what's called two level hashing. Two-level hashing schemes involve creating a secondary hash function for each bucket in our primary hash table. According, if there are collisions, we can find an element much faster than if all elements in a bucket were stored in a linked list. Specifically, if S_i is the number of items mapped to the ith address in our main hash table, we allocate a secondary hash table of size S_i^2 for that address. By our previous analysis, this choice ensures that the expected number of collisions on the second level < 1 and accordingly there is at least some choice of hash functions that guarantee no collisions overall. The expected value of space required by this technique is given by:

$$\mathbb{E}\left[\sum S_i^2\right] = \mathbb{E}\left[\sum_i S_i(S_i - 1)\right] + \mathbb{E}\left[\sum_i S_i\right] \leq \frac{m(m-1)}{n} + m \leq 2m$$

as long as we choose $n > m$. Recall that $\mathbb{E}\left[\sum_i S_i(S_i - 1)\right]$ is just the expected number of collisions seen, times 2, a value we already calculated. Recall that m is the total number of items stored in the hash table, and n is the number of addresses in the hash table. So $n > m$ but it just needs to be $O(m)$ instead of $O(m^2)$ as before.

2.3 Las Vegas Algorithm

Here's a simple scheme for finding a two-level hash function that actually has no collisions for the elements in $\mathbf{S}$.

1. Choose 2-universal $h : \mathbf{U} \to [m]$ until:

$$\sum_i S_i^2 \leq 4m$$

By Markov bound you'll find such an h with probability $> 1/2$ on each try.

2. For each $i \in 1, \ldots, m$, choose $h_i : \mathbf{U} \to [S_i^2]$. Re-sample until there are no collisions in the hash table.

Again, for each h_i we'll succeed with probability $1/2$ on each trial so with good probability we'll only require $O(m)$ tries over all buckets.

In general, for hash functions we prove that the 'odds' of getting a sufficiently random hash function that avoids collisions are rather good. You can then use a Las Vegas Algorithm like the one described to 're-shuffle the deck' until you actually find a hash function with no collisions.

3 Recipe for 2-Universal Hash Families

Now that we know how 2-universal hash families can be used, we'll describe a method for constructing them:

Choose a large prime p such that $|\mathbf{U}| \leq p \leq 2|\mathbf{U}|$. The hash function is:

$$h_{a,b}(x) = (ax + b \mod p) \mod n, \ a \neq 0$$

Note that because p is so large, we want to map down to a smaller value using mod n. The hash family is given by:

$$\mathcal{H} = \{h_{a,b} | a \in \{1, 2, ..., p-1\} \text{ and } b \in \{0, 1, 2, ..., p-1\}\}$$

To prove that the hash function is 2-universal, let's begin by defining two variables for convenience: let $s = ax + b \mod p$, and let $t = ay + b \mod p$.

Claim 1. *For $x \neq y$, if $h_{a,b}(x) = h_{a,b}(y)$, then $s \neq t$ but $s = t \mod n$.*

Proof. (Without loss of generality: $x > y$) If $s = t$ then p divides $(ax + b) - (ay + b) = a(x - y)$. But both a and $(x - y)$ are $< p$ and nonzero, so this isn't possible since p is prime. □

Now, we are ready to prove the 2-universality of $\mathcal{H}$. Since $s \neq t$, there is a unique pair (a, b) such that:

$$\begin{aligned} a &= (s - t)(x - y)^{-1} \mod p \\ b &= s - ax \mod p. \end{aligned}$$

So, for any x, y there's just a single hash function $h_{a,b}$ with $h_{a,b}(x) = s$ and $h_{a,b}(y) = t$.

Now, since we choose $a \in \{1, 2, ..., p-1\}$ and $b \in \{0, 1, 2, ..., p-1\}$, there are $p(p-1)$ total hash functions in $\mathcal{H}$. How many satisfy $h(x) = h(y)$ for a given x, y?

$h(x) = h(y)$ if and only if $s = t \mod n$. For a given s, there are at most $(p-1)/n$ other values of t such that $s = t \mod n$. Since there are p possible values of s, this means there are $p(p-1)/n$ hash functions mapping $h(x) = h(y)$.

$p(p-1)/n$ is a $1/n$ fraction of the total number of hash functions in $\mathcal{H}$. Accordingly, 2-universality is satisfied.

4 Prelude to Next Lecture: Consistent Hashing

One of the things that goes wrong with the formulation for hashing outlined above is the application of it to new technologies, specifically where the number of items being stored is constantly changing. That is, for a hash function given by $h_{a,b}(x) = (ax + b \mod p) \mod n$, what happens if n isn't a static value? For example, consider the n representing the number of servers handling website URL queries. Is there a way to formulate a hash function that is resilient to changes in the number n of servers/machines in use?
The idea of 'consistent hashing' grows out of this conundrum. It works with the idea that no two users may agree on the active servers visible to them, but the data structure remains intact.

References

[1] Carter, Larry and Wegman, Mark N. (1979). Universal Classes of Hash Function. *Journal of Computer and System Sciences* 18 (2): 143–154.