CS 31: Algorithms (Spring 2019): Lecture 9

Date: 23rd April, 2019

Topic: Randomized Algorithms 2: Hashing

Disclaimer: These notes have not gone through scrutiny and in all probability contain errors. Please notify errors on Piazza/by email to deeparnab@dartmouth.edu.

1 Maintaining and Searching in a Database

Suppose you want to maintain a database D of all names in your city. Assume that there are n people and everyone has distinct names. These names however come from a huge universe U of *possible* names; indeed, a name is just a string of length ≤ 50 where each character has say 100 possibilities (allowing for accents, umlauts, etc). Thus, $|U| \approx 100^{50} \approx 2^{300}$. In other words, $|U|$ is *huge*.

Your objective is to maintain this database such that given any possible name x, you want to implement the $\textsc{Search}(x)$ operation in the database. This operation should say YES if $x \in D$ and NO otherwise. Furthermore, you also want the runtime of $\textsc{Search}(x)$ to be fast.

Two Naive extremes. Let's look at two naive extremes and see how they are each lacking. One idea is to store the database as an array $D[1:n]$ and when a query $\textsc{Search}(x)$ arises, we search in the array. The storage space[1] is $O(n)$. However, the query time is also $O(n)$. The latter is not fast.

The second idea is to store a *huge* bit-array $A[1:U]$ initialized to 0's, and for each $d \in D$, we set[2] $A[d] = 1$. On the query $\textsc{Search}(x)$, we simply access $A[x]$ and say YES if $A[x] = 1$ and NO if $A[x] = 0$. Thus, it takes $O(1)$ time to search. However, the space requirement is $O(|U|)$ which is infeasible if U is a large set.

We would like the best of both worlds – we want the total space used by our database to be $O(n)$ *and* the queries to be answered in $O(1)$ time. Amazingly, this is possible using *hash functions* which will be the object of our study today.

Hash Functions. A hash function is simply a function whose domain is a (huge) set U and whose range/co-domain is usually a set $\{0, 1, 2, \ldots, m-1\}$. Thus, a hash function looks like

$$h : U \to \{0, 1, \ldots, m-1\}$$

[1]We are assuming that the names take $O(1)$ space to be stored. Technically, this should be $O(\log |U|)$ since there $|U|$ possible names. For this lecture we will assume $\log |U|$ to be "small" but $|U|$ to be large. Imagine $|U| = 1000$ (small) but 2^{1000} (humongous).

[2]I am thinking of names as integers. In general, we would need a map from names to natural numbers. But this is OK because the set of strings is a *countable* set.

The size of the co-domain, m, is $\ll |U|$ much, much smaller than the size of the domain. We will use the notation $\mathbb{Z}_m$ to denote the set $\{0, 1, \ldots, m-1\}$.

How are hash functions useful for the dictionary application? Well, imagine we do have a function $h : U \to \mathbb{Z}_m$ such that for any two names y, z in the dictionary we have $h(y) \neq h(z)$. Then, we can maintain a table (called a hash table) $T[0 : m-1]$ where we store the name y in the cell $T[h(y)]$. Now when a search query arises $\textsc{Search}(x)$, we simply look in the cell $T[h(x)]$; if that is filled up, $x \in D$, otherwise not. The space requirement is $O(m)$ (the hash table). The query time is $O(1)$ *plus* the time required to compute the hash function.

Thus, for our database application, we just need to find the hash function for $m = O(n)$, and which can be computed fast. How does one find such a desirable hash function? This is where *randomization* plays a big role. To describe this, we visit an important notion: that of a *family* of hash functions.

Definition 1 (Universal Hash Family (UHF).)**.** A collection H of hash functions from U to $\mathbb{Z}_m$ if for any two ***distinct*** $x, y \in U$ we have

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

where the probability is over $h \in H$ *uniformly* drawn at random.

We will use UHFs to find the desirable hash function for our database. But do these functions even exist? At the end of this lecture, we will show a construction. However, let's just describe one UHF which is *not* desirable. Indeed, let H be the collection of **all** functions from U to $\mathbb{Z}_m$. Convince yourself : for any two $x \neq y$ in U, the probability a random $h \in H$ has $h(x) = h(y)$ is in fact exactly $1/m$.

Why is this undesirable? Well, what's the time taken to compute $h(x)$? Since h doesn't have a "closed form" formula, the only way I can think of storing these h's is to actually noting down the value $h(x)$ for all $x \in U$. That's like one of the naive solutions we considered. But this sort of shows how the desrirable UHFs (they do exist) are an extrapolation of that naive idea.

1.1 Database Searching using Universal Hash Functions

We now show how UHFs can be used to store data so that lookups are faster in expectation. Let h be a hash function drawn uniformly at random from a UHF H. The range of these functions is $[m]$.

Given $D \subseteq U$ of size $|D| = n$, we store the hashes $h(y)$ for all $y \in D$ as indicated above. We initialize a *hash table* $T[0 : m-1]$ywhere each $T[i]$ is an empty *linked list*. For all $y \in D$, we append y to the end of $T[h(y)]$. The total space used after we are done processing is $O(m + n)$. When a search query $\textsc{Search}(x)$ arises, our algorithm goes to the cell $T[h(x)]$ and searches for x in the list stored at that location. An illustrative example is shown below.

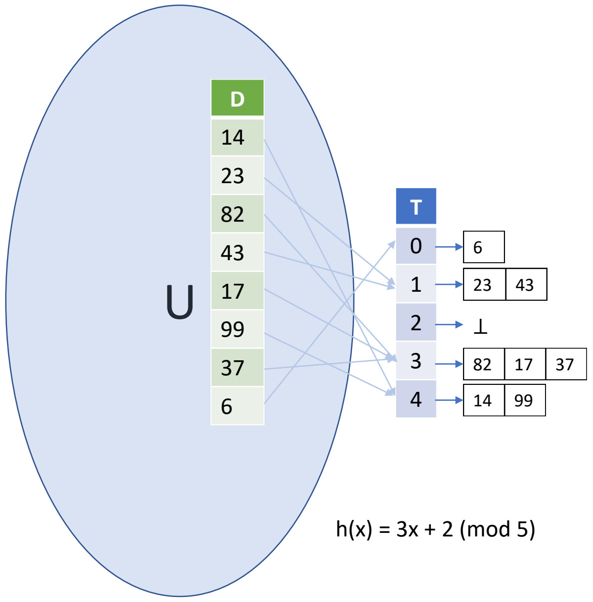

Figure 1: The universe U is the set of all integers. The hash function is $h(x) = 3x + 2(\mod 5)$. Thus the hash table has 5 entries, and each $T[i]$ is a linked list. For the data set D shown, we get the hash table as shown. Given a candidate $x \in U$, say 80, the search process first calculates $h(80)$ which in this case is 2, and the searches for 80 in $T[2]$. As it is, $T[2] = \perp$, and thus we answer NO.

It is clear that the running time is governed by how long the lists $T[i]$ are for each $i \in \mathbb{Z}_m$. Given h and data set D, we say that two distinct $x, y \in D$ *collide* if $h(x) = h(y)$. Intuitively, fewer the collisions, shorter the lists. Universal Hash Families allow us to formally argue about collisions and search time.

Theorem 1. If H is a universal hash family, then for any data set D of size n, if a hash function h is chosen uniformly at random from H, then the expected number of collisions is $\leq \binom{n}{2} \cdot \frac{1}{m}$.

Proof. Let's pick two arbitrary, distinct elements x, y in U. Let C_{xy} be the *indicator* random variable of whether x and y collide, that is, whether $h(x) = h(y)$ for the function h which has been uniformly at random chosen from H.

Since H is a UHF, we know that $\mathbf{Pr}_{h\in_R H}[h(x) = h(y)] \leq \frac{1}{m}$. That is, $\mathbf{Pr}[C_{xy} = 1] \leq \frac{1}{m}$. Which in turn implies $\mathbf{Exp}[C_{xy}] \leq \frac{1}{m}$.

The total number of collisions is $C = \sum_{x,y\in D: x\neq y} C_{xy} \leq \binom{n}{2} \cdot \frac{1}{m}$ since the number of such pairs is precisely $\binom{n}{2}$. □

Let's now get back to the time taken to SEARCH(x). The following theorem shows that this time, *in expectation*, is $\leq 1 + \frac{n}{m}$.

Theorem 2. Let H be a UHF of hash functions mapping from U to $\mathbb{Z}_m$. Suppose we use a random $h \in H$ to store a database D of size n into a hash-table as described above. Then the *expected* time taken by $\textsc{Search}(x)$ is $= O(1 + \frac{n}{m})$ look-ups plus the time required to compute $h(\cdot)$.

Proof. The time taken to $\textsc{Search}(x)$ is precisely the length of the list at $T[h(x)]$. Let this length be L_x. We see that this length is precisely the number of $y \in D$ with $h(x) = h(y)$. Using the random variable from the previous theorem, we get

$$L_x = \sum_{y \in D} C_{xy}$$

Thus,

$$\mathbf{Exp}[L_x] = \sum_{y \in D} \mathbf{Exp}[C_{xy}] = \begin{cases} 1 + \sum_{y \in D : y \neq x} \mathbf{Exp}[C_{xy}] & \leq 1 + \frac{n-1}{m} & \text{if } x \in D \\ \sum_{y \in D} \mathbf{Exp}[C_{xy}] & \leq \frac{n}{m} & \text{if } x \notin D \end{cases}$$

In both cases, the answer is $\leq 1 + \frac{n}{m}$. □

1.2 Searching with $O(1)$ worst case time.

The above method of hashing used $O(n)$ space. If H consisted of hash functions which were fast to compute, then the search time is $O(1)$ in *expectation*. It is important to understand what this means. The above method is *randomized* since it uses a random hash function. Some functions in the family could be good, some could be bad ... on average, the search time is $O(1)$. This is still undesirable. We still would like a single desirable hash function since we don't want to get stuck with a bad one. We now see how to find the desirable hash function at the expense of larger space. The key is Theorem 1.

Let's recall what the theorem said. Let H be a universal family of hash functions mapping the universe U to $\mathbb{Z}_m$. The theorem stated that if we selected a random $h \in H$ and used it to map D of size n into a hash-table, the *expected* number of collisions is $< \frac{n^2}{2m}$. What this means, that if $m = n^2$, the *expected* number of collisions is $< 1/2$. That is, if $m = n^2$ and C is the total number of collisions, then

$$\mathbf{Exp}[C] < \frac{1}{2} \quad \Rightarrow \quad \mathbf{Pr}[C \geq 1] < \frac{1}{2}$$

where the latter used ***Markov's inequality*** which says that for any non-negative random variable Z, $\mathbf{Pr}[Z \geq t] < \frac{\mathbf{Exp}[Z]}{t}$.

In English the above is saying: if $m = n^2$ and if we select a hash function $h \in H$ uniformly at random and use it to store the database D into a hash-table as described in the previous section, then with probability $> 1/2$, we will have $C < 1$, that is, ***no collisions***. And that's the desirable hash function (at the expense of larger space).

```
1: procedure FormHashTable($D$):
2:     ▷ $D$ is a data-set of length $n$ where $D \subseteq U$ comes from a universe $U$.
3:     ▷ Assumes access to a UHF $H$ of functions from $U$ to $\mathbb{Z}_m$ where $m = n^2$.
4:     ▷ Output: a single hash-function $h \in H$, a hash table $T[0 : m-1]$ with no collisions
5:     while true do:
6:         Sample $h$ from $H$ uniformly at random.
7:         Use $h$ on $D$ to construct hash table $T$.
8:         if no collisions in $T$ then:
9:             return $(h, T)$.
```

The probability that Line 8 succeeds, is $\geq 1/2$. Thus, the expected time of FormHashTable is $O(1)$ while loops. Each while loop takes $O(n)$ time (to form the table and check if there are collisions). The space required is $O(m) = O(n^2)$. The time for Search is $O(1)$ since there are no collisions.

Remark: *Indeed, we are almost where we wanted – the space unfortunately is $O(n^2)$ instead of $O(n)$. One more idea takes us to the promised land of "perfect hash functions". The main idea is this: when there are collisions, not to store the thing as lists, but use a second set of hash functions instead. This is sometimes called the "double hashing trick". This was done by Fredman, Kömlos, and Szemeredi and is called the FKS hash as well.*

1.3 Construction of Universal Hash Families

We now show one way to construct UHFs using *modular* arithmetic. Pick a prime number $p > N$. For a fixed $a \in \{1, 2, \ldots, p-1\}$ and $b \in \{0, 1, \ldots, p-1\}$, define the hash function

$$h_{a,b}(x) = ((a \cdot x + b) \mod p) \mod m$$

We remark here that if the integers N, m fit in the word size in the word-RAM model, then computing $h_{a,b}(x)$ takes $\Theta(1)$ time. Thus for modest sized integers, the above is faster to implement. This (partly) explains the reason why we assume evaluating hash functions take $\Theta(1)$ time. ✍

Exercise: *One may wonder why we are taking $\mod p$ and then $\mod m$? Why not just define $h_{a,b}(x) = (ax + b) \mod m$ where a, b are integers between 1 and $m-1$ and 0 and $m-1$, respectively. Argue that this family is* **not** *a UHF even when m was a prime.*

Theorem 3. The family

$$H := \{h_{a,b} : a \in \{1, 2, \ldots, p-1\} \text{ and } b \in \{0, 1, \ldots, p-1\}\}$$

is a universal hash family.

Proof. Fix $x \neq y$ in U. Note that for $h_{a,b}(x) = h_{a,b}(y)$, we must have that $(ax+b) \mod p = (ay+b) \mod p + k \cdot m$ for some integer k. Which in turn implies,

$$a \cdot (x-y) \mod p = k \cdot m$$

Since the LHS is in $\{0, 1, \ldots, p-1\}$, we get $k \leq (p-1)/m$.

Now, we use one number theoretic fact.

Fact 1. If p is a prime and r is a non-zero number, then there exists a unique number r^{-1} in $\{1, 2, \ldots, p-1\}$ such that $r \cdot r^{-1} \equiv 1 \mod p$.

Proof. Consider the products $r_i := r \cdot i \mod p$ for $i \in \{1, \ldots, p-1\}$. If $r_i = r_j$ for any $i \neq j$, we would get $r \cdot (i-j) \equiv 0 \mod p$. That is, p divides $r \cdot (i-j)$. But since p is prime, this means p divides one of the two: that's impossible since they are both strictly less than p. Thus, all the r_i's are distinct and take values in $\{1, \ldots, p-1\}$ ($r_i \neq 0$ for the same reason as above). That implies *exactly* one $r_i = 1$ and $r^{-1} = i$. $\square$

The above fact implies that for $a \cdot (x-y) \equiv k \cdot m \mod p$, we must have $a = (x-y)^{-1} \cdot km \mod p$. No other value would work. Since k has $\leq \frac{p-1}{m}$ distinct values possible, at most $(p-1)/m$ values of a could lead to $h_{a,b}(x) = h_{a,b}(y)$. That is, the probability this occurs is $\leq 1/m$, since a is picked randomly from $\{1, 2, \ldots, p-1\}$. $\square$