Lecture 7

Universal and Perfect Hashing

7.1 Overview

Hashing is a great practical tool, with an interesting and subtle theory too. In addition to its use as a dictionary data structure, hashing also comes up in many different areas, including cryptography and complexity theory. In this lecture we describe two important notions: *universal hashing* (also known as *universal hash function families*) and *perfect hashing.*

Material covered in this lecture includes:

- The formal setting and general idea of hashing.
- Universal hashing.
- Perfect hashing.

7.2 Introduction

We will be looking at the static and dynamic "dictionary" problems defined as follows:

- Static: Given a set S of items, each with an associated key, we want to store them in such a way that we can do lookups quickly using the keys. E.g., think of a fixed dictionary..
- Dynamic: here we have a sequence of insert, lookup, and perhaps delete requests. We want to do these all efficiently.

For the first problem we could use a sorted array with binary search for lookups. For the second we could use a balanced search tree. However, hashing gives an alternative approach that is often the fastest and most convenient way to solve these problems. For example, suppose you are writing an AI-search program, and you want to store situations that you've already solved (board positions or elements of state-space) so that you don't redo the same computation when you encounter them again. Hashing provides a simple way of storing such information. There are also many other uses in cryptography, networks, complexity theory.

7.3 Hashing basics

The formal setup for hashing is as follows.

- Keys come from some large universe U. (E.g, think of U as the set of all strings of at most 80 ascii characters.)

- There is some set S in U of keys we actually care about (which may be static or dynamic). Let $N = |S|$. Think of N as much smaller than the size of U. For instance, perhaps S is the set of names of students in this class, which is much smaller than 128^{80}.

- We will perform inserts and lookups by having an array A of some size M, and a **hash function** $h : U \to \{0, \ldots, M-1\}$. Given an element x, the idea of hashing is we want to store it in $A[h(x)]$. Note that if U was small (like 2-character strings) then you could just store x in $A[x]$ like in bucketsort. The problem is that U is big: that is why we need the hash function.

- We need a method for resolving collisions. A *collision* is when $h(x) = h(y)$ for two different keys x and y. For this lecture, we will handle collisions by having each entry in A be a linked list. There are a number of other methods, but for the issues we will be focusing on here, this is the cleanest. This method is called *separate chaining.* To insert an element, we just put it at the top of the list. If h is a good hash function, then our hope is that the lists will be small.

One great property of hashing is that all the dictionary operations are incredibly easy to implement. To perform a lookup of a key x, simply compute the index $i = h(x)$ and then walk down the list at $A[i]$ until you find it (or walk off the list). To insert, just place the new element at the top of its list. To delete, one simply has to perform a delete operation on the associated linked list. The question we now turn to is: what do we need for a hashing scheme to achieve good performance?

Desired properties: The main desired properties for a good hashing scheme are:

1. The keys are nicely spread out so that we do not have too many collisions, since collisions affect the time to perform lookups and deletes.

2. $M = O(N)$: in particular, we would like our scheme to achieve property (1) without needing the table size M to be much larger than the number of elements N.

3. The function h is fast to compute. In our analysis today we will be viewing the time to compute $h(x)$ as a constant. However, it is worth remembering in the back of our heads that h shouldn't be too complicated, because that affects the overall runtime.

Given this, the time to lookup an item x is $O(\text{length of list } A[h(x)])$. The same is true for deletes. Inserts take time $O(1)$ no matter what the lengths of the lists. So, we want to be able to analyze how big these lists get.

Basic intuition: One way to spread elements out nicely is to spread them *randomly*. Unfortunately, we can't just use a random number generator to decide where the next element goes because then we would never be able to find it again. So, we want h to be something "pseudorandom" in some formal sense.

We now present some bad news, and then some good news.

Claim 7.1 (Bad news) *For any hash function h, if $|U| \geq (N-1)M+1$, there exists a set S of N elements that all hash to the same location.*

Proof: by the pigeon-hole principle. In particular, to consider the contrapositive, if every location had at most $N-1$ elements of U hashing to it, then U could have size at most $M(N-1)$. ■

So, this is partly why hashing seems so mysterious — how can one claim hashing is good if for any hash function you can come up with ways of foiling it? One answer is that there are a lot of simple hash functions that work well in practice for typical sets S. But what if we want to have a good *worst-case* guarantee?

Here is a key idea: let's use randomization in our *construction* of h, in analogy to randomized quicksort. (h itself will be a deterministic function, of course). What we will show is that for *any* sequence of insert and lookup operations (we won't need to assume the set S of elements inserted is random), if we pick h in this probabilistic way, the performance of h on this sequence will be good in expectation. So, this is the same kind of guarantee as in randomized quicksort or in 2-player zero-sum games. In particular, this is idea of **universal hashing**.

Once we develop this idea, we will use it for an especially nice application called "perfect hashing".

7.4 Universal Hashing

Definition 7.1 *A randomized algorithm H for constructing hash functions $h : U \rightarrow \{1, \ldots, M\}$ is* **universal** *if for all $x \neq y$ in U, we have*

$$\Pr_{h \leftarrow H}[h(x) = h(y)] \leq 1/M.$$

We also say that a set H of hash functions is a **universal hash function family** *if the procedure "choose $h \in H$ at random" is universal. (Here we are identifying the set of functions with the uniform distribution over the set.)*

Theorem 7.2 *If H is universal, then for any set $S \subseteq U$ of size N, for any $x \in U$ (e.g., that we might want to lookup), if we construct h at random according to H, the* **expected** *number of collisions between x and other elements in S is at most N/M.*

Proof: Each $y \in S$ ($y \neq x$) has at most a $1/M$ chance of colliding with x by the definition of "universal". So,

- Let $C_{xy} = 1$ if x and y collide and 0 otherwise.
- Let C_x denote the total number of collisions for x. So, $C_x = \sum_{y \in S, y \neq x} C_{xy}$.

- We know $\mathbf{E}[C_{xy}] = \Pr(x \text{ and } y \text{ collide}) \leq 1/M$.
- So, by linearity of expectation, $\mathbf{E}[C_x] = \sum_y \mathbf{E}[C_{xy}] < N/M$. ■

We now immediately get the following corollary.

Corollary 7.3 *If H is universal then for any* **sequence** *of L insert, lookup, and delete operations in which there are at most M elements in the system at any one time, the expected total cost of the L operations for a random $h \in H$ is only $O(L)$ (viewing the time to compute h as constant).*

Proof: For any given operation in the sequence, its expected cost is constant by Theorem 7.2, so the expected total cost of the L operations is $O(L)$ by linearity of expectation. ■

Question: can we actually construct a universal H? If not, this this is all pretty vacuous. Luckily, the answer is yes.

7.4.1 Constructing a universal hash family: the matrix method

Let's say keys are u-bits long. Say the table size M is power of 2, so an index is b-bits long with $M = 2^b$.

What we will do is pick h to be a random b-by-u 0/1 matrix, and define $h(x) = hx$, where we do addition mod 2. These matrices are short and fat. For instance:

$$\begin{array}{ccccc} \text{h} & \text{x} & & \text{h(x)} \\ \boxed{\begin{matrix}1&0&0&0\\0&1&1&1\\1&1&1&0\end{matrix}} & \boxed{\begin{matrix}1\\0\\1\\0\end{matrix}} & = & \boxed{\begin{matrix}1\\1\\0\end{matrix}} \end{array}$$

Claim 7.4 *For $x \neq y$, $\Pr_h[h(x) = h(y)] = 1/M = 1/2^b$.*

Proof: First of all, what does it mean to multiply h by x? We can think of it as adding some of the columns of h (doing vector addition mod 2) where the 1 bits in x indicate which ones to add. (e.g., we added the 1st and 3rd columns of h above)

Now, take an arbitrary pair of keys x, y such that $x \neq y$. They must differ someplace, so say they differ in the ith coordinate and for concreteness say $x_i = 0$ and $y_i = 1$. Imagine we first choose all of h but the ith column. Over the remaining choices of ith column, $h(x)$ is fixed. However, each of the 2^b different settings of the ith column gives a different value of $h(y)$ (in particular, every time we flip a bit in that column, we flip the corresponding bit in $h(y)$). So there is exactly a $1/2^b$ chance that $h(x) = h(y)$. ■

There are other methods to construct universal hash families based on multiplication modulo primes as well (see Section 7.6.1).

The next question we consider is: if we fix the set S, can we find a hash function h such that *all* lookups are constant-time? The answer is *yes*, and this leads to the topic of *perfect hashing*.

7.5 Perfect Hashing

We say a hash function is **perfect** for S if all lookups involve $O(1)$ work. Here are now two methods for constructing perfect hash functions for a given set S.

7.5.1 Method 1: an $O(N^2)$-space solution

Say we are willing to have a table whose size is quadratic in the size N of our dictionary S. Then, here is an easy method for constructing a perfect hash function. Let H be universal and $M = N^2$. Then just pick a random h from H and try it out! The claim is there is at least a 50% chance it will have no collisions.

Claim 7.5 *If H is universal and $M = N^2$, then* $\Pr_{h\sim H}(\textit{no collisions in } S) \geq 1/2$.

Proof:

- How many pairs (x, y) in S are there? **Answer:** $\binom{N}{2}$
- For each pair, the chance they collide is $\leq 1/M$ by definition of "universal".
- So, Pr(exists a collision) $\leq \binom{N}{2}/M < 1/2$. ■

This is like the other side to the "birthday paradox". If the number of days is a lot *more* than the number of people squared, then there is a reasonable chance *no* pair has the same birthday.

So, we just try a random h from H, and if we got any collisions, we just pick a new h. On average, we will only need to do this twice. Now, what if we want to use just $O(N)$ space?

7.5.2 Method 2: an $O(N)$-space solution

The question of whether one could achieve perfect hashing in $O(N)$ space was a big open question for some time, posed as "should tables be sorted?" That is, for a fixed set, can you get constant lookup time with only linear space? There was a series of more and more complicated attempts, until finally it was solved using the nice idea of universal hash functions in 2-level scheme.

The method is as follows. We will first hash into a table of size N using universal hashing. This will produce some collisions (unless we are extraordinarily lucky). However, we will then rehash each bin using Method 1, squaring the size of the bin to get zero collisions. So, the way to think of this scheme is that we have a first-level hash function h and first-level table A, and then N second-level hash functions $h_1, \ldots, h_N$ and N second-level tables $A_1, \ldots, A_N$. To lookup an element x, we first compute $i = h(x)$ and then find the element in $A_i[h_i(x)]$. (If you were doing this in practice, you might set a flag so that you only do the second step if there actually were collisions at index i, and otherwise just put x itself into $A[i]$, but let's not worry about that here.)

Say hash function h hashes n_i elements of S to location i. We already argued (in analyzing Method 1) that we can find $h_1, \ldots, h_N$ so that the total space used in the secondary tables is $\sum_i (n_i)^2$. What remains is to show that we can find a first-level function h such that $\sum_i (n_i)^2 = O(N)$. In fact, we will show the following:

Theorem 7.6 *If we pick the initial h from a universal set H, then*

$$\Pr[\sum_i (n_i)^2 > 4N] < 1/2.$$

Proof: We will prove this by showing that $\mathbf{E}[\sum_i (n_i)^2] < 2N$. This implies what we want by Markov's inequality. (If there was even a 1/2 chance that the sum could be larger than $4N$ then that fact by itself would imply that the expectation had to be larger than $2N$. So, if the expectation is less than $2N$, the failure probability must be less than 1/2.)

Now, the neat trick is that one way to count this quantity is to count the number of ordered pairs that collide, including an element colliding with itself. E.g, if a bucket has {`d`,`e`,`f`}, then `d` collides with each of {`d`,`e`,`f`}, `e` collides with each of {`d`,`e`,`f`}, and `f` collides with each of {`d`,`e`,`f`}, so we get 9. So, we have:

$$\begin{aligned}
\mathbf{E}[\sum_i (n_i)^2] &= \mathbf{E}[\sum_x \sum_y C_{xy}] \quad (C_{xy} = 1 \text{ if } x \text{ and } y \text{ collide, else } C_{xy} = 0)\\
&= N + \sum_x \sum_{y \neq x} \mathbf{E}[C_{xy}]\\
&\leq N + N(N-1)/M \quad \text{(where the } 1/M \text{ comes from the definition of universal)}\\
&< 2N. \quad \text{(since } M = N\text{)} \quad \blacksquare
\end{aligned}$$

So, we simply try random h from H until we find one such that $\sum_i n_i^2 < 4N$, and then fixing that function h we find the N secondary hash functions $h_1, \ldots, h_N$ as in method 1.

7.6 Further discussion

7.6.1 Another method for universal hashing

Here is another method for constructing universal hash functions that is a bit more efficient than the matrix method given earlier.

In the matrix method, we viewed the key as a vector of bits. In this method, we will instead view the key x as a vector of integers $[x_1, x_2, \ldots, x_k]$ with the only requirement being that each x_i is in the range $\{0, 1, \ldots, M-1\}$. For example, if we are hashing strings of length k, then x_i could be the ith character (assuming our table size is at least 256) or the ith pair of characters (assuming our table size is at least 65536). Furthermore, we will require our table size M to be a prime number. To select a hash function h we choose k random numbers $r_1, r_2, \ldots, r_k$ from $\{0, 1, \ldots, M-1\}$ and define:

$$h(x) = r_1 x_1 + r_2 x_2 + \ldots + r_k x_k \bmod M.$$

The proof that this method is universal follows the exact same lines as the proof for the matrix method. Let x and y be two distinct keys. We want to show that $\Pr_h(h(x) = h(y)) \leq 1/M$. Since $x \neq y$, it must be the case that there exists some index i such that $x_i \neq y_i$. Now imagine choosing all the random numbers r_j for $j \neq i$ first. Let $h'(x) = \sum_{j \neq i} r_j x_j$. So, once we pick r_i we will have $h(x) = h'(x) + r_i x_i$. This means that we have a collision between x and y exactly when $h'(x) + r_i x_i = h'(y) + r_i y_i \bmod M$, or equivalently when

$$r_i(x_i - y_i) = h'(y) - h'(x) \bmod M.$$

Since M is prime, division by a non-zero value mod M is legal (every integer between 1 and $M-1$ has a multiplicative inverse modulo M), which means there is exactly one value of r_i modulo M for which the above equation holds true, namely $r_i = (h'(y) - h'(x))/(x_i - y_i) \bmod M$. So, the probability of this occuring is exactly $1/M$.

7.6.2 Other uses of hashing

Suppose we have a long sequence of items and we want to see how many *different* items are in the list. What is a good way of doing that?

One way is we can create a hash table, and then make a single pass through our sequence, for each element doing a lookup and then inserting if it is not in the table already. The number of distinct elements is just the number of inserts.

Now what if the list is really huge, so we don't have space to store them all, but we are OK with just an approximate answer. E.g., imagine we are a router and watching a lot of packets go by, and we want to see (roughly) how many different source IP addresses there are.

Here is a neat idea: say we have a hash function h that behaves like a random function, and let's think of $h(x)$ as a real number between 0 and 1. One thing we can do is just keep track of the *minimum* hash value produced so far (so we won't have a table at all). E.g., if keys are 3,10,3,3,12,10,12 and $h(3) = 0.4, h(10) = 0.2, h(12) = 0.7$, then we get 0.2.

The point is: if we pick N random numbers in $[0,1]$, the expected value of the minimum is $1/(N+1)$. Furthermore, there's a good chance it is fairly close (we can improve our estimate by running several hash functions and taking the median of the minimums).

Question: why use a hash function rather than just picking a random number each time? That is because we care about the number of *different* items, not just the total number of items (that problem is a lot easier: just keep a counter...).