6.854 Advanced Algorithms

Lecture 6: 9/4/2004 Lecturer: David Karger

Scribes: Ali Mohammad

Universal Hashing

6.1 Problem formulation

We all do it, but many of us have never analyzed it. We have n items and we want to associate them to integers $1 \cdots m$ where $m \gg n$. We could store them in an array of size m, whence we could perform insert/delete/query operations in constant time (much faster than, say, a tree data structure) because of indirect addressing. Hash tables don't allow you to do predecessor or successor very easily. The real drawback, though, is space. Hashing algorithms really are just about saving space.

6.2 2-universal Hashing

We will develop 2-universal hashing, first introduced by Carter and Wegman in the 80's.

Goal: Store it in an array of size $O(n)$.

Method: Use a hash function that maps $\{1 \cdots m\}$ to $\{1 \cdots n\}$. Store item with key k in array position $f(k)$.

Problem: Collisions.

First Fix: Linked list from each array position (buckets).

Problem: We've lost $O(1)$ lookups.

Goal: Guarantee few items per bucket and therefore few collisions. This is sort of a load balancing problem– randomization can help with this.

Idea: Use a random function f where $f(k)$ is random but well-defined. Then,

$$\begin{aligned} \mathcal{E}\left[\#\text{ items in } k\text{'s bucket}\right] &= \mathcal{E}\left[\sum c_{ik}\right] \\ &= \sum_n \mathcal{E}\left[c_{ik}\right] = \sum \Pr\left(i \text{ collides with } k\right) \\ &= \sum_{i \neq k} 1/n = 1 - 1/n \\ \Rightarrow \qquad \mathcal{E}\left[\text{time to find } k\right] &= O(1). \end{aligned}$$

Another (avoidable) problem is that a purely random function takes $O(m)$ space, which brings us back to where we started!

Solution: Use pseudo-random number generators, which are "random enough for a given application." The key for us is to reduce the collisions of pairs, so we will use a "pairwise-independent" hash function. That is, given any one output of the hash function, you have no information about any other (weakening the independence condition).

Definition 1 (Pairwise Independence) *Random variables $B_1, \cdots, B_n$ are said to be pairwise independent if B_i and B_j are independent $\forall\ i, j$.*

Here is an algorithm for generating pairwise-independent random variables in $\mathbb{Z}_p$. First, pick a random prime p and pick random values $a, b \in \mathbb{Z}_p$. Then, set $r_i = ai + b \bmod p$. Observe that knowing r_i doesn't tell you anything about r_j for $j \neq i$. However, knowing any two r_i and r_j will reveal a and b, and therefore the remaining r_k. Here a and b correspond to the seed and the r_is correspond to a sequence of random variables based on the seed.

Claim 1 *$\forall i, j < p$, r_i and r_j are pairwise independent and uniformly distributed.*

Proof: We are arguing that $r_i = ai + b$ and $r_j = aj + b$ are pairwise independent. Observe that:

$$\begin{aligned}\Pr\left(ai + b = x,\ aj + b = y\right) &= \Pr\left(\begin{array}{ccccc} ai & + & b & = & x \\ aj & + & b & = & y \end{array}\right) = 1/p^2 = 1/p \cdot 1/p \\ &= \Pr\left(ai + b = x\right) \Pr\left(aj + b = y\right),\end{aligned}$$

as desired. ■

Now how do we use this knowledge for hashing? Pick a prime $p > m$, pick a, b randomly from $\mathbb{Z}_p$, and set $h(k) = [(ak + b) \bmod p] \bmod n$.

Claim 2 *Items are distributed "almost" uniformly. That is,*

$$\Pr\left(h(k) = x\right) \approx 1/n.$$

This is because we know that $(ak + b) \bmod p$ is absolutely uniform; when we mod it down to n, p will cover the n slots many times over plus some negligible remainder, which we will consequently neglect.

Claim 3 *The placements in the array are pairwise independent.*

Clearly the extra mod n doesn't create dependence. r_i and r_j are independent implies that $f(r_i)$ and $f(r_j)$ are independent $\forall f$.

Claim 4 $\mathcal{E}\left[\sum_{i \neq j} c_{ij}\right]$ *is unchanged. c_{ij} depends only on the buckets of i and j.*

Summary: By choosing random a, b and using $h(k) = [(ak + b) \bmod p] \bmod n$, get $1 - 1/n$ collisions in expectation. Therefore, insert and lookup cost $O(1)$ in expectation.

We don't know that there will only be two or fewer items in each bucket. In fact, in the worst case, $\sqrt{n}$ items can be placed in one bucket! (This is possible if the pairwise-independent family is chosen poorly.) It is also important that the adversary is unaware of what function was chosen (i.e., what seed was chosen).

The cost of defining our pseudo-random function is now just two integers of size $O(m)$ (a total of two machine words).

6.3 2-level Hashing

2-universal hashing is nice in expectation, but what about the worst-case?

Let's try to define a hash function with no collisions! To simplify things, we are not going to worry about a dynamic scenario where there is insertion and deletion. Instead, we are going to consider a fixed set of n keys. Now, it is easy to spread them out, but we will have to work to make lookup fast. This seems really tricky because even a fully random function has collisions (recall the birthday paradox).

First Try: Use more space.

How many items can we map into n spaces with collisions? The expected number of total collisions is:

$$\mathcal{E}\left[\sum_{\substack{i,j \\ i \neq j}} c_{ij}\right] = \sum_{\substack{i,j \\ i \neq j}} \mathcal{E}[c_{ij}] = \sum_{\substack{i,j \\ i \neq j}} \frac{1}{n} = \binom{s}{2}\frac{1}{n} < \frac{s^2}{2n}$$

In particular, if $s < \sqrt{n}$, then the expected number of collisions is less than $1/2$. This is nice, but we want a guarantee of no collisions.

Recall Markov's Inequality (one of the few probabilistic gems we will use in the course):

Lemma 1 (Markov's Inequality) *If X is nonnegative, then* $\Pr(X \geq a) \leq \mathcal{E}[X]/a$.

Consequently,

$$\Pr\left(\sum c_{ij} \geq 1\right) < \mathcal{E}\left[\sum c_{ij}\right].$$

Observe that we are again using pairwise independence, by the definition of the c_ij's. Thus, the 2-universal hashing results still apply.

Now, if you fail, try again. That is, one can amplify the probability of success by repetition. The expected number of tries before I succeed is only 2! So, in exepcted linear time, I can find a perfect hash function into quadratic space. This is nice, but it still isn't perfect.

Maybe by using a more random function (using more independence) we can do better? Well, no. The birthday paradox comes into play again. So we can't do better by using more randomness.

Another trick: First, use a 2-universal hash function to divide n tiems into n buckets. Then, build a quadratic space to achieve a perfect hash table in each bucket. How much space does this use?

$$\begin{aligned}
\text{Total space} &= \sum_b [\text{size of bucket } b]^2 \\
s_b = \text{size of bucket } b &= \sum_i s_{ib} \qquad (\text{event that } i \text{ hashes to bucket } b) \\
s_b^2 &= \left(\sum_i s_{ib}\right)^2 = \sum_{i,j} s_{ib} s_{jb} \\
\sum_b s_b^2 &= \sum_{i,j,b} s_{ib} s_{jb} = \sum_{i,j} \sum_b s_{ib} s_{jb} = \sum_{i,j} c_{ij} \\
\mathcal{E}\left[\sum_{i,j} c_{ij}\right] &= \sum_{i,n} 1/n = n - 1 + n = 2n - 1 = O(n).
\end{aligned}$$

Thus, the expected space for extra tables is $O(n)$. If you do this carefully, the space used is $6n$. If yo uwork a lot harder, you can actually trim it down to $n + o(n)$ with this simple technique!

If you want to be certain that you use at most n space, simply use the technique of trying again and again until you succeed. One small problem is that you need m a, b pairs – about m^2 space. This might be ok depending on the application.

One more thing: using more sophisticated techniques, one can achieve similar bounds in the dynamic case.