Lecture 4 (09-12) - Universal Hashing

Let's first motivate the idea of **hashing** - suppose we want to maintain an unordered dictionary mapping some astronomical set of keys U (i.e. all ASCII strings with length $< s$) to values. In actuality, we want to support a relatively small subset $S \subset U$ as keys (such as the set of valid English words with length $< s$) to our dictionary.

We want our dictionary to support three major operations:

1. `add(k, v)` - Adds a key-value pair, (k, v), to our dictionary S.
2. `lookup(k)` - Given a key k, checks if it is in our dictionary. If so, returns the associated value v.
3. `delete(k)` - Removes the key k and associated value v if it is in our dictionary. $k \in U$.

From these, we can think of certain models for our dictionary

- A **static setting** occurs when our dictionary S doesn't or extremely rarely changes and we really only care about optimizing our `lookup` operation.
- An **insertion setting**, on the other hand, we really only bother ourselves with optimizing for `add` and `lookup` queries.
- The last and most expansive case, the **fully-dynamic setting** utilizes all three operations.

The ultimate goal of our dictionary is to show that we can improve upon the operations of other data structures such as search trees or sorted arrays by performing these operations in a constant expected time. More formally, we wish for the following properties:

We can consider $N = \mid S \mid$ and M to be the size of the array A that stores our dictionary. The way our dictionary operates then, is it takes a hash function, H that maps the space $U \to \{0, 1, 2, 3, \dots M-1\}$. In fact, we will store the key $k \in U$ in $A[H(k)]$.

Note that by the pigeonhole principle, unless $M \geq \mid U \mid$, there will be inevitable **collisions** of keys $k_1 \neq k_2,\ H(k_1) = H(k_2)$. The way we resolve this in our dictionary is called **collision resolution**. There are two main ways we deal with these:

1. **Separate Chaining** - Instead of storing values in side our array A, we store references to data structures (usually a linked list) that store all keys that hash to the same value. We can simply traverse the data structure when performing lookups and deletes and append to it when adding.
2. **Open Addressing** - If the index we are hashing to is taken, we probe in some way (linearly, quadratically, double hashing) to find an empty index. Deletion involves marking

the previous index as a tombstone that frees the index. Lookup involves probing until an empty index is reached or the element is found (tombstones indices continue the lookup).

For analysis of our hashing, we will simply use separate chaining as it tends to be a lot simpler to visualize.

Formally, we wish for our hashing to have the following properties:

1. $H(x)$ should have a relatively uniform distribution across M. This will ultimately make collisions less common.
2. The space of M should be $O(N)$ while achieving property 1.
3. $H(x)$ should be relatively fast, with the ideal being constant time.

The Problem

First, let's deliver the unfortunate news that if U is even remotely large - $O(n^2)$, then $\exists$ a set S such that $\forall x \in S, H(x) = \text{the same value}$.

We can show this via the Pigeonhole Principle: let us fill the M spots in our array with $N-1$ elements each. This, of course, is the best case scenario. Then, wherever our next element hashes to, that index will contain a set S that all hashes to the same value. Our hand is forced to enter this scenario at $\mid U \mid = M(N-1)+1$ elements. Given we want M to be $O(n)$, this is roughly $O(n^2)$.

The Solution

However, not all hope is lost! While we cannot guarantee decent behavior in the absolute worst-case scenario, we can get around this via pseudo-randomness, so we get good expected performance.

How do we do this? We construct a deterministic (otherwise, we couldn't perform lookups), but pseudo-random hash function depending on the input. In essence, for every key $k \in U$, our hash function H will choose a pseudo-random hash function h in some space of hash functions and use it to hash the key.

Because we are using pseudo-random hash functions, unless the adversary is aware of our random state, they cannot construct a predefined set S that all hashes to the same value simply by analyzing H. Of course, they may still be extremely lucky, but given any sequence of inserts and lookups (insertion setting), H will perform well *in expectation*.

Formal Definition of Universal Hashing

We consider a family of hash functions H to be a **universal hash family** if, for all distinct keys $x \neq y \in U$, then for a random variable hash function $h \in H$,

$$P(h(x) = h(y)) \leq \frac{1}{M}$$

Given a uniform probability distribution of hash functions $h \in H$, then we can simplify the probability to the ratio of the cardinality between two sets:

$$\frac{|\{h \in H \mid h(x) = h(y)\}|}{|H|} \leq \frac{1}{M}$$

That is, for every pair $x \neq y \in U$, show that the number of hash functions in which $x,\ y$ collide divided by the total number of hash functions is less than $\frac{1}{M}$,

For example, these would be universal hash families, because $\frac{1}{2},\ 0,\ \frac{1}{3} \leq \frac{1}{2}$.

	a	b
h_1	0	0
h_2	0	1

	a	b
h_1	0	1
h_2	1	0

	a	b
h_1	0	0
h_2	1	0
h_3	0	1

These, however, would not be considered universal because $1,\ \frac{2}{3} \nleq \frac{1}{2}$:

	a	b
h_1	0	0
h_3	1	1

	a	b	c
h_1	0	0	1
h_2	1	1	0
h_3	1	0	1

Universal hash function have the special property that if we have a dictionary S of size N, hashing into an array of size M, then for $x \in U$, the expected number of collisions between $H(x)$ and S is $\leq \frac{N}{M}$.

We can show this using the property of universal hash functions where $P(h(x) = h(y)) \leq \frac{1}{M}$. Let us have a random variable

$$C_{xy} = \begin{cases} 0 \text{ if } x, y \text{ do not collide} \\ \\ 1 \text{ if } x, y \text{ do collide} \end{cases}$$

Then, let C_x be the total number of collisions resulting in hashing x into our dictionary:

$$C_x = \sum_{y \in S,\ y \neq x} C_{xy}$$

What we are trying to find, then is $E[C_x]$.
Given $E[C_{xy}] \leq \frac{1}{M}$ by properties of universal hashing, then by linearity of expectation:

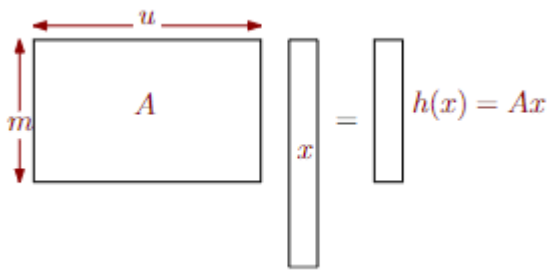
$$E[C_x] = \sum_{y \in S,\, y \neq x} [C_{xy}] \leq \frac{N}{M}$$

as we have N distinct $y \in S$. This also shows that inserts, lookups, and deletes can be done in constant time of $\frac{N}{M}$ (assuming relatively static dictionary) assuming H can be computed in constant time. This also extends to any sequence of n operations taking $O(n)$ time in total.

An Example of Universal Hashing

So what does one of these hashing schemes actually look like?
Consider if $M = 2^k, k \in \mathbb{N}$. Then, we can index using k bits.

Therefore, for a key of length u in binary, we can generate a pseudo-random $k \times u$ matrix A of 0s and 1s.



Then, our hash function $h(x) = Ax$, taking addition $\mod 2$ and outputs a k length binary column vector we can use as the index.

Proof of Correctness

Consider a pseudo-random A for Ax, Ay where $x \neq y$. This means that there must exist at least one bit in x such that the i^{th} bit in x does not equal the i^{th} bit in y. As such, one bit must be 0 and the other must be 1. Without loss of generality, we can label the $x_i = 0$ and $y_i = 1$.

Consider the i^{th} column of A and all other columns "set." For $H(x)$, the answer is not affected by the i^{th} column; however, for $H(y)$, any 1 in a row will flip the output of $H(y)$ in that row. For each row in the i^{th} column, then there is a $\frac{1}{2}$ chance to match the output of $H(x)$ by either flipping or not flipping the bit. Therefore, the probability there exists a collision is $\left(\frac{1}{2}\right)^k = \frac{1}{2^k} = \frac{1}{M}$, which is what we wanted to show.

Small Extension

We can extend the idea of universality to l-wise universality if it satisfies:

$$P(h(x_1) = h(y_1)) \ \wedge \ h(x_2) = h(y_2)) \ldots \ \wedge \ h(x_l) = h(y_l)) \leq \frac{1}{M^l}$$

If a hash function H is l-universal for n, then it is universal for all naturals $\leq n$.
The above scheme is universal but not 2-universal.

#hashing #algo #collisions #universal-hashing #dictionary