Notes on Hashing

Owen Jow · Last updated May 05, 2018

This document is intended to give a high-level overview of hashing, approximately at the level I'd imagine you need to know for the CS 170 final. As a disclaimer, I have not referred to the final in terms of its hashing content and accordingly am not sure what you *actually* need to know. This is just a guess.

As another disclaimer, this note is no substitute for watching lecture, doing homework, or reviewing section worksheets. It's meant as more of a refresher, something to help organize your thoughts.

Overview

Hashing

Generally, we have studied hashing in the context of search queries. Under this setup, we are given n key-value pairs $(k_1, v_1), ..., (k_n, v_n)$ [for distinct keys k_i] and would like to construct a data structure that we can later use to perform the operation `query(`k_i`)` $\rightarrow v_i$ in constant time.

Naively, we could store all of the (k_i, v_i) tuples in an array, sort them by their k_i components, and later perform binary search to find the key we're looking for. However, each query would then take $O(\log n)$ time (because binary search). This sounds good, but it's actually bad. For one because queries are going to happen all the time, and for another *because we can do better.*

How? With hash functions. A hash function follows the form

$$h : U \rightarrow [m]$$

where U is the universe (the set of possible keys) and $[m]$ (shorthand for $\{1, ..., m\}$, or $\{0, ..., m-1\}$ depending on indexing) is one of m "slot" indices. Computing $h(k_i)$ is known as "hashing" k_i.

Here is a basic hashing scheme for information storage and retrieval:

Initialization:

1. Allocate an array of size m.
2. For each $i \in [n]$: store the tuple (k_i, v_i) in slot $h(k_i) \in [m]$.

Access:

1. Hash key k_i; get a slot $h(k_i) \in [m]$.
2. Go to the slot, get the value, and go home.

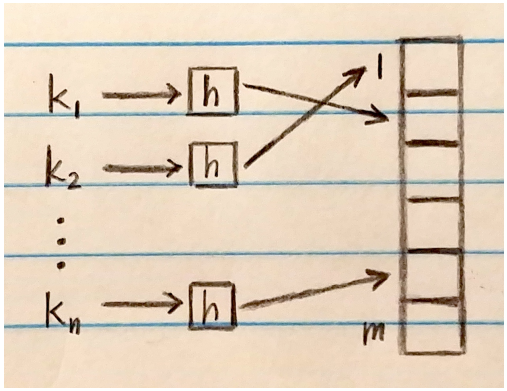

Figure 1: *basic hashing. In each slot we store a linked list of* (k_i, v_i) *tuples.*

In this scheme, each access time consists of *the time it takes to evaluate* $h(k_i)$ plus *the time it takes to find* (k_i, v_i) *at slot* $h(k_i)$. The former should be $O(1)$. The latter depends on how many items are at slot $h(k_i)$, which in turn depends on the collisions our hash function creates.

To assert an overall $O(1)$ access time, we need to spread the data out as much as possible (avoid a bunch of collisions in any given slot)! Unfortunately, the following lemma exists:

For every hash function, there exists a set of keys that are all mapped to the same slot.

Note: this is assuming that the universe is sufficiently large. But it usually will be.

For most inputs, our hash function might do pretty well. However, by this lemma there's always an input that will make our hashing scheme useless (no better than storing everything in a list). And we want our search query algorithm to have guarantees for *all* inputs, not just random inputs.

Thus, instead of hardcoding some h and using it, we will pick h at random as part of the algorithm. Specifically, we will pick h from a family of hash functions $\mathcal{H}$. Assuming each $h \in \mathcal{H}$ performs well for most inputs, we will then have a good probability of getting a good h for the data we receive.

Algorithm 1 No randomization

procedure H(k)
    Use black magic to turn k into a number from 1 to m
procedure INITIALIZE($(k_1, v_1), ..., (k_n, v_n)$)
    Allocate an array of size m
    For each $i \in [n]$: store the tuple (k_i, v_i) in slot $h(k_i) \in [m]$

Algorithm 2 Randomization

procedure INITIALIZE($(k_1, v_1), ..., (k_n, v_n)$)
    $h \leftarrow$ random $h \in \mathcal{H}$
    Allocate an array of size m
    For each $i \in [n]$: store the tuple (k_i, v_i) in slot $h(k_i) \in [m]$

Without randomization, there is surely an input (i.e. a bunch of (k_i, v_i) pairs) that will defeat us. *With* randomization, there is no such input.

As it happens, **if** the probability of a collision between any two distinct keys is $\leq \frac{1}{m}$ *for a hash function chosen uniformly randomly from* $\mathcal{H}$, the expected number of collisions per slot is $\leq \frac{n}{m}$.

n: the number of (k_i, v_i) pairs we're hashing
m: the number of slots we're hashing to

This is a great result – if $n = m$ the expected number of collisions per slot is ≤ 1 – so let's make sure $\mathcal{H}$ meets this condition. Such a hash family is called **universal**.

Universal Hashing

Formally, a family of hash functions $\mathcal{H}$ is **universal** if for all $k_1 \neq k_2 \in U$,

$$\Pr_{h \in \mathcal{H}}[h(k_1) = h(k_2)] \leq \frac{1}{m}$$

Note: a simple universal hash family is the set of **all** functions from U to $[m]$. However, to store a generic random function from U to $[m]$ would require $|U| \log m$ bits (because we would have to encode all of the mappings). Furthermore, to sample one of these functions we would need to randomize $|U| \log m$ bits. Since $|U|$ is often massive, this is just too expensive. We should find a smaller universal hash family to sample from.

For example, the family of key-indexed inner product functions seen in lecture.

Our hashing scheme has now become

Algorithm 3 Universal Hashing

1: $h \leftarrow$ `null`
2: **procedure** INITIALIZE($(k_1, v_1), ..., (k_n, v_n)$)
3: $h \leftarrow$ random h from **universal** hash family $\mathcal{H}$
4: Allocate an array of size m
5: For each $i \in [n]$: store the tuple (k_i, v_i) in slot $h(k_i) \in [m]$
6: **procedure** QUERY(k_i)
7: Go to slot $h(k_i)$ and find (k_i, v_i)
8: Return v_i

With universal hashing, in expectation we will have a constant number of collisions in each slot – and therefore an expected constant access time overall! Best of all, this will work for all data (hence the "universal" moniker).

Still, maybe we would like a stronger guarantee than just expectation. How about a *worst-case* guarantee? This is where perfect hashing makes its entrance.

Perfect Hashing

In perfect hashing, we guarantee zero collisions and thus constant access time in the worst case. There are multiple ways to achieve perfect hashing; I'll cover the one we discussed in class.

During the initialization step of our current universal hashing algorithm, (k_i, v_i) pairs are distributed over a hash table as depicted in Figure 2 (to give an example).

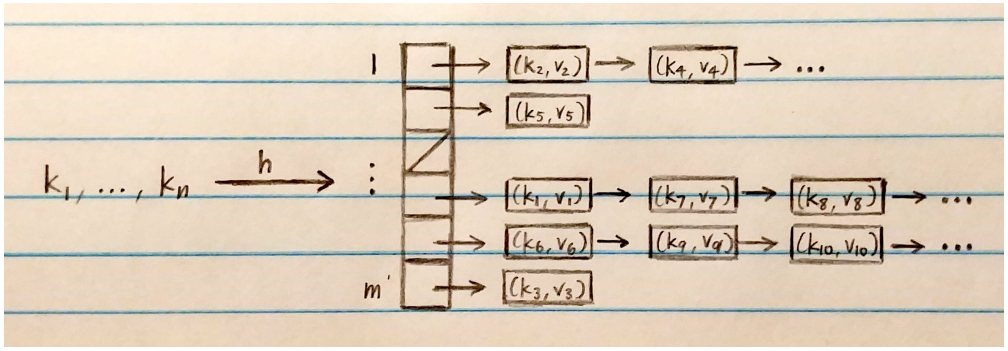

Figure 2: *basic hashing re-illustrated.*

Instead of laying collisions out into a list, perfect hashing invokes another hash function which maps each slot's elements into a second-level array. If the first-layer hash function maps l_i elements into slot i, then the second-layer hash function $h_i : U \to [l_i^2]$ for slot i will map these l_i elements into an array of size l_i^2. Why l_i^2? Because then the probability of collision will be low.

To be clear, we first hash each element into a table of size m. Then we re-hash the elements in each slot into another, bigger hash table such that there are no collisions. The sizes of the second-level hash tables do not all have to be the same.

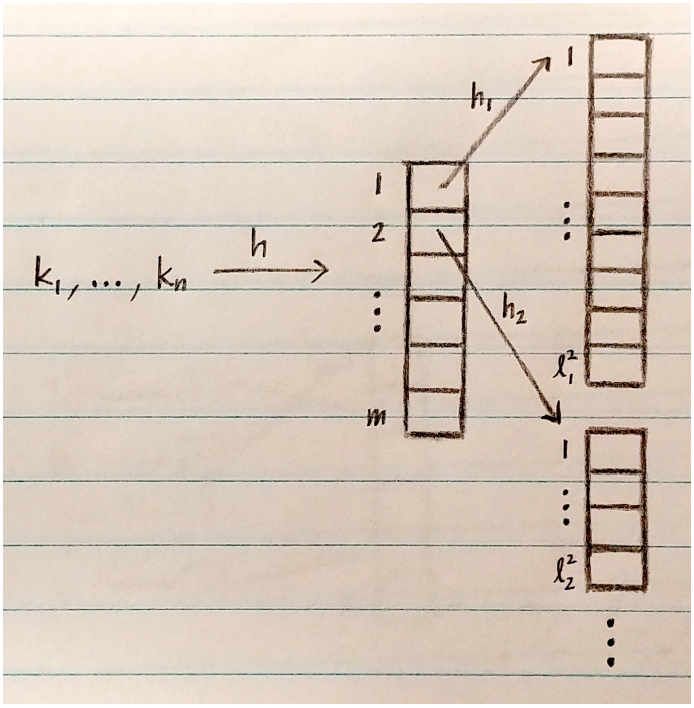

Figure 3: *perfect hashing. The length of each second-level hash table is the square of the length of the original linked list at the slot. All hash functions are sampled from universal families.*

The total size of the two-layer hash table will then be $\sum_{i=1}^{m} l_i^2$, since we expand each of m slots to a size that is the square of the number of elements that are originally mapped into it. We would like $\sum_{i=1}^{m} l_i^2$ to be on the order $O(n)$.

Overall, the algorithm comes together as follows:

Algorithm 4 Perfect Hashing

procedure INITIALIZE($(k_1, v_1), ..., (k_n, v_n)$)
    $h \leftarrow$ random h from universal hash family $\mathcal{H}$
    Hash each of the k_i's; get a distribution $l_1, ..., l_m$
        (Check that $\sum_{i=1}^{m} l_i^2 < 100 \cdot n$; if not, resample h and hash again)
    for each slot $j = 1, ..., m$ **do**
        $h_j \leftarrow$ random h from universal hash family $\mathcal{H}_j$
        Use h_j to hash every k_i originally mapped to slot j
            (Check that there are no collisions; if there are, resample h_j and hash again)
        Store each v_i in slot $h_j(k_i)$ in the second-layer table for slot j
procedure QUERY(k_i)
    Return the value at $h_{h(k_i)}(k_i)$

Note that if we do end up with collisions, we resample the hash function and try again. We continue in this way until we have zero collisions. Therefore zero collisions is guaranteed.

The only question is "how long does it take to find hash functions that work?" But since we're using universal hash functions and second-layer arrays of length l_i^2, it shouldn't take *too* long; after all, the probability of choosing an h_i with zero collisions is greater than $\frac{1}{2}$.

Perfect hashing uses randomness to find good (indeed perfect) hash functions for the given data.

Common Confusions

Hashing

- *Why can't we just map elements to the indices* $1, 2, 3, ...$ *in order?*
  - This spreads the items out, but doesn't address the issue of search queries. Once items are stored, how do we access them again in constant time? How do you know which element goes with which index?
  - If your answer is "use a dictionary," that doesn't help since dictionaries *are* hash maps. You'd only end up adding a layer of indirection on top of the already-proposed solution.
- *Do we sample a different hash function every time we hash something?*
  - We only sample the hash function once. Once we start mapping things, we need the mappings to stay the same – otherwise we might lose track of where the items are.
- *Does the hash function map all of the keys at once?*
  - No, a hash function only maps one element at a time. In case you are confused by the notation $h : \{1, ..., p\} \to \{1, ..., m\}$, it means we're mapping a single element in the set $\{1, ..., p\}$ to a single element in the set $\{1, ..., m\}$.

Universal Hashing

- *If a hash family consists of functions that take $[m]$ to $[m-1]$, aren't we mapping a larger space to a smaller space, meaning collisions are guaranteed? How can such a family be universal?*
  - In general the universe *will* be much larger than our hash table, and our hash function *will* map multiple elements in the universe to the same slot. This is fine, because universality is defined in terms of *pairwise* collision probabilities between distinct keys. The condition for universality says "what's the **probability** that **two** different keys are mapped to the same slot?"
  - It's totally irrelevant that the overall number of [theoretical] collisions is greater than zero; all that matters for universality is that our pairwise probabilities hold.
  - Zero collisions is a property of perfect hashing, not universal hashing.
- *Is the randomness in choosing a hash function included in the $\frac{1}{m}$ probability calculation?*
  - Yes, intrinsically; it is where the stochasticity comes from. It's the reason we have a probability at all. That's why the probability is "over a choice of hash function."

Perfect Hashing

- *Why do we use extra space for the second layer? Why not have the length of each secondary table be the number of elements in the slot?*
  - Mainly because we're using randomly sampled hash functions to map elements to the second-layer arrays. We want the probability of collisions to be low, so we should hash to as large as a space as we reasonably can.
  - We're using hash functions for the same reasons as the original problem. Again, note that it doesn't make sense to just map elements to slots $1, 2, 3, ...$ (see the hashing FAQ).
- *Are each of the second-layer arrays the same size as the first-layer array?*
  - No, they are not. The array for slot i has length l_i^2 (the square of the number of elements that were originally mapped to slot i), so most will probably be different lengths, and they definitely won't all be the same length as the first layer.
- *Why is the probability of choosing an h_i with zero collisions greater than $\frac{1}{2}$?*
  - First, note that this refers to the choice of h_i from a universal hash family which maps to a space of size l_i^2 (where l_i is the number of elements we're mapping).
  - In this setting, there are $\binom{l_i}{2}$ pairs of distinct elements that could collide, and for each pair the chance they collide (given a random hash function from a universal family) is $\leq \frac{1}{l_i^2}$. Therefore, the probability of at least one collision is

$$\leq \binom{l_i}{2} \cdot \frac{1}{l_i^2} = \frac{l_i^2 - l_i}{2l_i^2} < \frac{1}{2} \quad \text{(by the union bound)}$$

    And the probability of *zero* collisions is $> \frac{1}{2}$.

Appendix

Proofs

- If $\mathcal{H}$ is a universal hash family, then $\forall\ k_1, ..., k_n \in U$ we have

$$\mathbb{E}_{h\in\mathcal{H}}[\#\text{ collisions in any slot}] \leq \frac{n}{m}$$

Proof. Let $I_{ij} = \begin{cases} 1 & \text{if } h(k_i) = h(k_j) \\ 0 & \text{otherwise.} \end{cases}$

Then

$$\begin{aligned} \mathbb{E}_{h\in\mathcal{H}}[\#\text{ collisions in any slot}] &= \mathbb{E}_{h\in\mathcal{H}}[I_{12} + I_{13} + ... + I_{1n}] \\ &= \sum_{i=2}^{n} \mathbb{E}_{h\in\mathcal{H}}[I_{1i}] \\ &\leq \sum_{i=2}^{n} \frac{1}{m} \\ &\leq \frac{n}{m} \end{aligned}$$

- In expectation, $\sum_{i=1}^{m} l_i^2$ is on the order $O(n)$.

Proof.

$$\begin{aligned} \mathbb{E}_{h\in\mathcal{H}}\left[\sum_{i=1}^{m} l_i^2\right] &= \mathbb{E}_{h\in\mathcal{H}}[\#\text{ ordered } (k_i, k_j) \text{ tuples that collide}] \\ &= \sum_{i=1}^{n}\sum_{j=1}^{n} \Pr[h(k_i) = h(k_j)] \\ &\leq \underset{(k_i=k_j)}{n} + \underset{(k_i\neq k_j)}{n(n-1)\cdot\frac{1}{m}} \\ &= 2n - 1 \end{aligned}$$

if m (the first layer size) $= n$.