15-854: Algorithms in the Real World (Fall 2015) October 21, 2015
Lec #13: Hashing Last Updated: October 21, 2015

1 Hashing

Hashing is a basic computer science technique used in many different contexts, from dictionary data structures to load balancing and symmetry breaking, to cryptography and complexity theory. In the next few lectures we will study the following:

- Desirable properties of hash families
- Constructions of hash families with these properties
- Applications of hash functions in various contexts

1.1 Maintaining a Dictionary

To understand the desired properties, let us keep one application in mind. We want to maintain a dictionary. We have a large universe of “keys” (say the set of all strings of length at most 80 using the Roman alphabet), denoted by U. The actual dictionary (say the set of all English words) is some subset S of this universe. S is typically much smaller than U. The operations we want to implement are:

- `add(x)`: add the key x to S.
- `query(q)`: is the key $q \in S$?
- `delete(x)`: remove the key x from S.

In some cases, we don’t care about adding and removing keys, we just care about fast query times—e.g., the actual English dictionary does not change (or changes very gradually). This is called the *static case*. Another special case is when we just add keys: the *insertion-only case*. The general case is called the *dynamic case*.

1.2 Desired Properties

In this lecture, let $[N]$ denote the numbers $\{0, 1, 2, \ldots, N-1\}$. One natural approach is to choose a hash function $h : U \to [M]$, and store the key $x \in S$ at (or near) the location $h(x)$.

What do we want from hash functions:

(i) *Small probability of distinct keys colliding:* if $x \neq y \in S$ then $\Pr[h(x) = h(y)]$ is “small”.
(ii) *Small range:* we want the hash table size M to be small. At odds with first desired property.
(iii) *Small number of bits to store the hash function h.*
(iv) *h is easy to compute.*

What is the probability taken over? There are two choices: (a) we could want two random keys $x \neq y$ not to collide (i.e., $\Pr_{x,y \in U | x \neq y}[h(x) = h(y)] \leq \mathit{blah}$). But the keys in our dictionary are not random keys, so this guarantee would be useless. Or (b) we could choose the hash function randomly (from some set H of hash functions) and want that $\Pr_{h \leftarrow U}[h(x) = h(y)] \leq \mathit{blah}$. This latter approach is the one we take. [1]

[1] In practice people use hashing schemes based on cryptographic hash functions like MD5 and SHA (of which there are many variants), or Google’s CityHash/FarmHash. These hash functions are deterministic, and so can only give

An Important Note: We will assume that the dictionary S is chosen "adversarially", we have no control over it. We choose h randomly from the family H. This is the only randomness in the process. Of course the adversary does not see h. Then we look at the performance of our random h on this worst-case S. It's like we're playing a game, and both of us are choosing our actions simultaneously, and we want our minimax behavior to be as good as possible.

1.3 An Ideal: The Perfectly Random Hash Function

Consider the completely random hash function: for each $e \in S$, we choose a uniformly random location in $[M]$, and set $h(e)$ to be that location. Clearly, this has great properties

- Low collision probability: $\Pr_h[h(a) = h(b)] = 1/M$ for any $a \neq b$, since having fixed where a maps to, there is a $1/M$ chance that b is mapped to the same location.
- In fact, even conditioned on knowing where any set of elements $A \subseteq S$ maps to, the position of any $e \in S \setminus A$ is still random:

$$\Pr_h[h(e) = \alpha \mid \wedge_{a \in A} h(a) = \alpha_a] = 1/M.$$

The problem? Storing this hash function requires storing $\lg_2 M$ bits for each $e \in S$ and hence $|S| \log_2 M$ overall. Moreover, it is not clear how to compute $h(\cdot)$ fast, other than doing a table-lookup.

However, perfectly random hash functions are good to keep in mind: we often develop algorithms assuming we have a perfectly random hash function. Then we see what properties we needed for the analysis (e.g., low collision probability, or small sets of elements behave as though they are independent), and find good hash functions that have these properties.

2 Universal Hashing

The definition of universal hashing tries to capture the most basic desired property that distinct keys do not collide too often. It was proposed by Carter and Wegman (1979).

Definition 1 *A family H of hash functions mapping U to $[M]$ is called universal if for any two keys $x \neq y \in U$, we have*

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

Make sure you understand the definition. This condition must hold for *every pair* of distinct keys, and the randomness is over the choice of the actual hash function h from the set H.

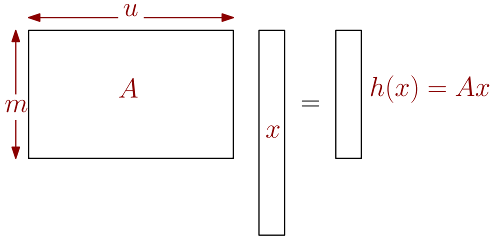
2.1 A Construction

A simple construction of universal hashing is the following. Consider the case where $|U| = 2^u$ and $M = 2^m$. The hash functions are defined as follows.

> *Take an $u \times m$ matrix A and fill it with random bits. For $x \in U$, view x as a u-bit vector in $\{0,1\}^u$, and define*
>
> $$h(x) := Ax$$
>
> *where the calculations are done modulo 2.*

the former kind of guarantee, if at all. Moreover, one can hope that there are few collisions on the dataset being used in any application.

Since the hash function is completely defined by the matrix A, there are 2^{um} hash functions in this family H.



Theorem 2 *The family of hash functions H defined above is universal.*

Proof: Consider $x \neq y \in \{0,1\}^u$. We claim that the probability that $h(x) = h(y)$ is at most $1/M$. Since h is a linear map, $h(x) = h(y) \iff h(x-y) = \vec{0}$. Equivalently, we want to show that for any non-zero vector $z \in \{0,1\}^n$,

$$\Pr_{h\leftarrow H}[h(z) = \vec{0}] \quad = \quad \Pr[Az = \vec{0}] \quad \leq \quad 1/M.$$

If the columns of A are $A_1, A_2, \ldots, A_u$, then $Az = \sum_{i\in[u]} z_i \cdot A_i$. Say that $z_{i^\star}$ equals 1 (since z is non-zero, there is at least one such coordinate. Now fix all the entries of A except column $i^\star$. For Az to be zero, it must be the case that $A_{i^\star} = \sum_{i \neq i^\star} z_i A_i$. But $A_{i^\star}$ contains m random bits, and each one matches the corresponding bit on the right with probability $1/2$. Hence the probability of $Az = \vec{0}$ is $1/2^m = 1/M$. □

BTW, note that $h(\vec{0}) = \vec{0}$, so picking a random function from the hash family H does not map each key to a random place. (The definition of universality does not require this.) It just ensures that the probability of collisions is small.

2.2 Application #1: Hashing with Open Addressing

The condition of universality may not seem like much, but it gives a lot of power. As mentioned above, one of the main applications of universal hashing is to dictionary data structures. We When many keys hash to the same location, the hash table can store only one of them. So we need some way of "resolving" these collisions, and storing these extra keys. There are many solutions, which you've probably seen before.

Hashing with separate chaining: An easy way to resolve collisions, also easy to analyze, but it may increase the space usage of the data structure. Here we maintain a linked list of all the "additional" keys. So the lookup time at location i becomes proportional to $|\{x \in S \mid h(x) = i\}|$, the number of keys in the dictionary S that map to i. Hence, when we perform a lookup on key q, we will spend expected time proportional to

$$E_{h\leftarrow H}\big[|\{x \in S \mid h(x) = h(q)\}|\big] = \sum_{x\in S} \Pr_{h\leftarrow H}\big[h(x) = h(q)\big] \leq \frac{|S|}{M}.$$

Hence, with a table of size $M = N = |S|$, lookups take expected constant time. (Also observe that item deletion is easy with separate chaining.)

Aside: What are other ways of resolving collisions? One way that requires no extra space is *open addressing*, where colliding keys are stored in the array itself. Where? That depends. The most basic idea is *linear probing*: When you are inserting x and $h(x)$ is occupied, you look for the smallest index i such that $(h(x)+1) \bmod M$ is free, and store $h(x)$ there. When querying for q, you look at $h(q)$ and scan linearly until you find q or an empty space. Observe that deletions are not quite as simple any more. It is known that linear probing can also be done in expected constant time, but universal hashing does not suffice to prove this bound: 5-universal hashing is necessary [PT10] and sufficient [PPR11].

One can use other probe sequences: e.g., not probe each location but choose some step size s and look at $h(x), h(x)+s \pmod M, h(x)+2s \pmod M, \ldots$. Or *quadratic probing*, where you look at $h(x), h(x)+1 \pmod M, h(x)+4 \pmod M, \ldots, h(x)+i^2 \pmod M$. Or you can use a random pattern for each key, chosen according to its own hash function.

One can also try to store multiple (usually a constant number of) keys in the same table location. And there's a different approach called *cuckoo hashing*, which we will discuss in the next lecture.

2.3 Application #2: Perfect Hashing

The results for separate chaining mentioned above hold in expectation (over the choice of the hash function). Can we hope for worst-case bounds? For a static dictionary S with $|S| = N$, there is an elegant solution that gives worst-case constant lookup time, and uses only tables of total size $O(N)$.[2] And it only uses universal hashing, combined with a two-level hashing idea. Here's how.

First, we claim that if we hash a set S of size N into a table of size $O(N)$ using a universal hash family, with probability at least 1/2 no location will have more than $O(\sqrt{N})$ keys mapped to it. Why? For $x, y \in S$, let C_{xy} be the indicator random variable for whether $h(x) = h(y)$, i.e., they "collide". The total number of collisions is $C = \sum_{x \neq y \in S} C_{xy}$, and its expectation is

$$E[C] = E\left[\sum_{x \neq y \in S} C_{xy}\right] = E \sum_{x \neq y \in S} E\left[C_{xy}\right] \leq \binom{N}{2} \frac{1}{M}. \tag{1}$$

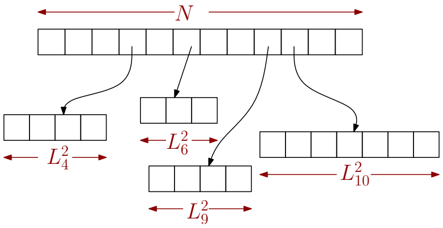
For $M = N$, say, this is at most $N/2$. So by Markov's inequality, we have $\Pr[C \geq N] \leq 1/2$. Moreover, if some location did have $\sqrt{2N}$ keys hashing to it, that location itself would result in $\binom{\sqrt{2N}}{2} \geq N$ collisions. Hence, with probability at least half, the maximum load at any location is at most $\sqrt{2N}$.

In fact, things are even better than that. If the load at location i is L_i, then the total number of collisions is $\sum_{i \in [M]} \binom{L_i}{2}$. And by the argument above, this is smaller than N (with probability 1/2). Hence $\sum_i L_i^2 \leq 3N$. Fix this first-level hash function $h^* : U \to [N]$.

Now we can take all the L_i keys that map into location i of the main table, build a special second-level table for them of size $M_i = O(L_i^2)$, and use the calculation (4) with $M = O(L_i^2)$ to argue that using a universal hash family for this second-level hashing from these L_i keys to $[M_i]$ will map all of them into separate locations. So we can choose a good hash function h_i^* for the keys that h^* maps to location i.

To look up q, we look at location $i = h^*(q)$, and then check location $h_i^*(q)$—this takes two hash function evaluations. Total space: N for the first table, then $\sum_i O(L_i^2)$ for the second level tables, which is again $O(N)$. (All space is measured in the number of keys.) We also need to store the hash functions, of course, which adds linear overhead.

[2] If we allow ourselves a table of size $M = \Omega(N^2)$, this is easy, because by the tightness of the birthday paradox—or the calculation in (4)—we could ensure that all keys map to distinct locations. But that is a lot of wasted space.



3 Pairwise and k-wise Independent Hashing

A couple years after their original paper, Carter and Wegman proposed a stronger requirement which we will call *pairwise-independent*.[3] Let us define a general notion of being *k-wise-independent*.

Definition 3 *A family H of hash functions mapping U to $[M]$ is called k-wise-independent if for any k distinct keys $x_1, x_2, \ldots, x_k \in U$, and any k values $\alpha_1, \alpha_2, \ldots, \alpha_k \in [M]$ (not necessarily distinct), we have*

$$\Pr_{h \leftarrow H}\left[h(x_1) = \alpha_1 \ \wedge\ h(x_2) = \alpha_2 \ \wedge\ \cdots\ \wedge\ h(x_k) = \alpha_k\right] \leq \frac{1}{M^k}.$$

Such a hash family is also called k-wise independent. The case $k = 2$ is called pairwise independent.

The following facts about k-wise independent hash families are simple to prove.

Fact 4 *Suppose H is a k-wise independent family for $k \geq 2$. Then*

a) H is also $(k-1)$-wise independent.
b) For any $x \in U$ and $\alpha \in [M]$, $\Pr[h(x) = \alpha] = 1/M$.
c) H is universal.

From part (c) above, we see that 2-wise independence is indeed at least as strong a condition as universality. And one can check that the construction in Section 2.1 is not 2-wise independent (since then it would also be 1-wise independent by Fact 4(a), but $\Pr[h(\vec{0}) = \vec{0}] = 1 \neq 1/M$). In the next section we give some constructions of 2-wise independent and k-wise independent hash families.

3.1 Some Constructions

3.1.1 Construction #1: A Variant on a Familiar Theme

The first construction is a simple modification of the universal hash family we saw in Section 2.1 for the case where $|U| = 2^u$ and $M = 2^m$.

> *Take an $u \times m$ matrix A and fill it with random bits. Pick a random m-bit vector $b \in \{0,1\}^m$. For $x \in U = \{0,1\}^u$, define*
>
> $$h(x) := Ax + b$$
>
> *where the calculations are done modulo 2.*

[3] They called it *strongly universal* but this terminology does not naturally generalize to k-tuples. (They use k-strongly universal to mean something else, so that doesn't help.)

The hash function is defined by the matrix A containing um random bits, and vector b containing m random bits, there are $2^{(u+1)m}$ hash functions in this family H.

Claim 5 *The family H is 2-wise independent.*

PROOF: Exercise. □

3.1.2 Construction #2: Using Fewer Bits

In the above construction, describing the hash function requires $O(um)$ bits. A natural question is whether we can do better. Indeed we can. Here is a related construction:

> *Take an $u \times m$ matrix A. Fill the first row $A_{1,\star}$ and the first column $A_{\star,1}$ with random bits. For any other entry i, j for $i > 1$ and $j > 1$, define $A_{i,j} = A_{i-1,j-1}$. So all entries in each "northwest-southeast" diagonal in A are the same.*
>
> *Also pick a random m-bit vector $b \in \{0,1\}^m$. For $x \in U = \{0,1\}^u$, define*
>
> $$h(x) := Ax + b$$
>
> *where the calculations are done modulo 2.*

Hence the hash family H consists of $2^{(u+m-1)+m}$ hash functions, one for each choice of A and b. You will prove that this family H is 2-wise independent as part of your homework. Here we need $O(u+m)$ random bits, and hence the space to store the hash function is comparable to the space to store a constant number of elements from U and $[M]$. Much better than $O(um)$!

3.1.3 Construction #3: Using Finite Fields

Suppose we want to map the universe $U = \{0,1\}^u$ to $[M] = \{0,1\}^m$. For this construction, we will work with the Galois field $GF(2^u)$ (and we associate strings in U with elements of the field in the natural way). First, we construct a 2-wise independent map from U to U as follows.

> *Pick two random numbers $a, b \in GF(2^u)$. For any $x \in U$, define*
>
> $$h(x) := ax + b$$
>
> *where the calculations are done over the field $GF(2^u)$.*

To prove 2-wise independence, note that for $x_1 \neq x_2 \in U$,

$$\begin{pmatrix} h(x_1) \\ h(x_2) \end{pmatrix} = \begin{pmatrix} 1 & x_1 \\ 1 & x_2 \end{pmatrix} \begin{pmatrix} a \\ b \end{pmatrix}$$

To calculate $\Pr[h(x_1) = \alpha_1 \ \wedge \ h(x_2) = \alpha_2]$, we get

$$\Pr\left[\begin{pmatrix} 1 & x_1 \\ 1 & x_2 \end{pmatrix} \begin{pmatrix} a \\ b \end{pmatrix} = \begin{pmatrix} \alpha_1 \\ \alpha_2 \end{pmatrix}\right] = \Pr\left[\begin{pmatrix} a \\ b \end{pmatrix} = \begin{pmatrix} 1 & x_1 \\ 1 & x_2 \end{pmatrix}^{-1} \begin{pmatrix} \alpha_1 \\ \alpha_2 \end{pmatrix}\right]$$

where the matrix is invertible because $x_1 \neq x_2$, and we're working over a field. But since a, b are chosen randomly, the chance that each of them equals some specified values is at most $1/2^u \times 1/2^u = 1/2^{2u}$, which is $1/|U|^2$ as desired for 2-wise independence.

That's cute. On the other hand, we hashed $U \to U$, which does not seem useful. But now we could truncate the last $u - m$ bits of the hash value to get a hash family mapping $[2^u]$ to $[2^m]$ for $m \leq u$; you can check this is 2-wise independent too.

3.1.4 Construction #4: k-universal Hashing

The construction for k-universal is not very different; let's consider hashing $GF(2^u) \to GF(2^u)$ once again.

> *Pick k random numbers $a_0, a_1, \ldots, a_{k-1} \in GF(2^u)$. For any $x \in U$, define*
>
> $$h(x) := a_0 + a_1 x + a_2 x^2 + \ldots + a_{k-1} x^{k-1}$$
>
> *where the calculations are done over the field $GF(2^u)$.*

The proof of k-universality is similar to that above; this is something you'll show in the homework. (In fact you could use any finite field $GF(p^s)$ you want.)

4 Other Hashing Schemes with Good Properties

While the above properties (universality and k-universality) are the most popular to prove that algorithms work well, here are some other hashing schemes which are commonly used, and have other good features.

4.1 Simple Tabulation Hashing

One proposal that has been around for some time (even considered by Carter and Wegman in their 1979 paper on universal hashing) is that of tabulation hashing. In this case, imagine $U = [k]^u$ and $M = 2^m$.

> **Tabulation Hashing.** Initialize a 2-dimensional $u \times k$ array T with each of the uk entries having a random m-bit string. Then for the key $x = x_1 x_2 \ldots x_u$, define its hash as
>
> $$h(x) := T[1, x_1] \oplus T[2, x_2] \oplus \ldots \oplus T[u, x_u].$$

Note that the hash function is completely defined by the table, which contains $u \cdot k \cdot m$ random bits. Hence the size of this hash family is 2^{kmu}. Is this any good? We can look at the independence properties of this family, for one.

Theorem 6 *The hash family H for tabulation hashing is* 3*-wise independent but not* 4*-wise independent.*

However, this is one case where independence properties of the hash family do not capture how good it is. A recent paper of Patrascu and Thorup [PT12] showed that the performance of many natural applications (linear probing, cuckoo hashing, balls-into-bins) using tabulation hashing almost matches the performance of these applications using truly random hash functions. An extension called *twisted tabulation* gives a better behavior for some applications [PT13].

4.1.1 A 5-universal variant

Thorup and Zhang show that if we just write $x = x_1 x_2$, and use the hash function

$$h(x) = T[1, x_1] \oplus T[2, x_2] \oplus T[3, x_1 + x_2]$$

which is slight variant on simple tabulation, then we get 5-universality. Recall that 5-universal is good for some applications, like for hashing with linear probing.

4.2 A Practical Almost-Universal Scheme

One hashing scheme that is not universal (but almost is), and is very easily implementable is as follows. As usual, we are hashing $U \to [M]$. Consider the common case where both $|U|$ and M are powers of 2; i.e., $|U| = 2^u$ and $M = 2^m$.

Pick a random **odd** number a in $[M]$. Define

$$h_a(x) := (ax \bmod U) \text{ div } (U/M)$$

Note that this construction clearly gives us an answer in $[M]$. It is also easy to implement: e.g., the div operation can be implemented by shifting to the right $u - m$ times. But is this any good? It turns out the collision probability is only twice as bad as ideal.

Theorem 7 ([DHKP97]) *For the hash family H defined as above, for $x \neq y \in U$,*

$$\Pr_{h \leftarrow H}[h(x) = h(y)] \leq \frac{2}{M}.$$

(The proof is not very difficult, you should try it as a bonus problem.)

4.3 Further Reading on Fast/Practical Hashing

There has been a lot of work on making hashing fast and practical, while also maintaining good provable properties – and also to understand why certain hashing schemes work well in practice. Check out papers by, e.g., Martin Dietzfelbinger, Rasmus Pagh, Mikkel Thorup, and Mihai Patrascu, and the references therein.

5 Application: Bloom Filters

A central application of hashing is for dictionary data structures, as we saw earlier. In some cases it is acceptable to have a data structure that occasionally has mistakes.

A *Bloom filter* is one such data structure.[4] It has the feature that it only has false positives (it may report that a key is present when it is not, but never the other way). Compensating for this presence of errors is the fact that it is simple and fast. A common application of a Bloom filter is as a "filter" (hence the name): if you believe that most queries are not going to belong to the dictionary, then you could first use this data structure on the queries: if the answer is `No` you know for sure the query key is absent. And if the answer is `Yes` you can use a slower data structure to confirm. For example, the Google Chrome browser uses a Bloom filter to maintain its list of potentially malicious websites.

> *Here's the data structure. You keep an array T of M bits, initially all entries are zero. Moreover, you have k hash functions $h_1, h_2, \ldots, h_k : U \to [M]$; for this analysis assume they are completely random hash functions.*
>
> *To add a key $x \in S \subseteq U$ to the dictionary, set bits $T[h_1(x)], T[h_2(x)], \ldots, T[h_k(x)]$ to 1.*
>
> *Now, when a query comes for key $x \in U$, check if all the entries $T[h_i(x)]$ are set to 1; if so, answer `Yes` else answer `No`.*

[4] It was invented by Burton H. Bloom in 1970.

Just that simple. Note that if the key x was in the dictionary S, all those bits would be on, and hence we would always answer `Yes`. However, it could be that other keys have caused all the k bits in positions $h_1(x), h_2(x), \ldots, h_k(x)$ to be set. What is the probability of that?

As usual, assume that $|S| = N$. For any key in S, h_1 does not hash this key to the location $\ell \in [M]$ with probability $(1 - 1/M)$. If the bit $T[\ell] = 0$, the same must be is true for all N keys, and all k hash functions—this happens with probability

$$\left(1 - \frac{1}{M}\right)^{kN} \approx e^{-kN/M}.$$

Denote this probability by p. So each location is 0 with probability p, and hence the expected fraction of zeros in the table is p. One can show, by a concentration bound, that the fraction of zeros is close to p with very high probability. (See, e.g., [BM03] for details.)

Now for a false positive on some query x, the bits $T[h_1(x)]$, $T[h_2(x)], \ldots,$ $T[h_k(x)]$ in all the k random locations must be set. Since there are a p fraction of zeros in the table, this happens with probability

$$(1 - p)^k \approx (1 - e^{-kN/M})^k. \tag{2}$$

Just to get a sense of the numbers, suppose $M = 2N$. Then the false positive probability is about $(1 - e^{-k/2})^k$—minimizing this as a function of k gives us a false positive rate of 38%; for $M = 8N$ this falls to 2%. In general, taking derivatives tells us that the optimal setting of k is $k = (\ln 2) \cdot (M/N)$, which gives false-positive probability of $(0.6185)^{M/N}$. In other words, if the false-positive probability is ε, then the number of bits we use is $M \approx 1.44N \log(1/\varepsilon)$—about $1.44 \log(1/\varepsilon)$ bits per entry.[5]

Bloom filters often arise in applications because of their simplicity and wide applicability; see this survey by Broder and Mitzenmacher [BM03] on many applications in networking.

6 Application: Cuckoo Hashing

Cuckoo hashing is a form of hashing without any false positives/negatives. It was was invented by Pagh and Rodler [PR04]. Due to its simplicity, and its good performance in practice, it has become very popular algorithm. Again, we want to maintain a dictionary $S \subseteq U$ with N keys.

> *Take two tables T_1 and T_2, both of size $M = O(N)$, and two hash functions $h_1, h_2 : U \to [M]$ from hash family H.*[6]
>
> *When an element $x \in S$ is inserted, if either $T_1[h_1(x)]$ or $T_2[h_2(x)]$ is empty, put the element x in that location. If both locations are occupied, say y is the element in $T_1[h_1(x)]$, then place x in $T_1[h_1(x)]$, and "bump" y from it. Whenever an element z is bumped from one of its locations $T_i[h_i(z)]$ (for some $i \in \{1, 2\}$), place it in the other location $T_{3-i}[h_{3-i}(z)]$. If an insert causes more than $6 \log N$ bumps, we stop the process, pick a new pair of hash functions, and rehash all the elements in the table.*

[5] The best possible space usage in this model is $\log(1/\varepsilon)$ bit per key, so Bloom filters are off by 44%. See the paper by Pagh, Pagh and Rao [PPR05] for an optimal data structure.

[6] We assume that H is fully-random, but you can check that choosing H to be $O(\log N)$-universal suffices.

> *If x is queried, if either $T_1[h_1(x)]$ or $T_2[h_2(x)]$ contains x say* `Yes` *else say* `No`*.*
>
> *If x is deleted, remove it from whichever of $T_1[h_1(x)]$ or $T_2[h_2(x)]$ contains it.*

Note that deletes and lookups are both constant-time operations. It only remains to bound the time to perform inserts. It turns out that inserts are not very expensive, since we usually perform few bumps, and the complete rebuild of the table is extremely rare. This is formalized in the following theorem.

Theorem 8 *The expected time to perform an insert operation is $O(1)$ if $M \geq 4N$.*

See these notes from Erik Demaine's class for a proof. [We'll sketch it on the board.] You can also see these notes on Cuckoo Hashing for Undergraduates by Rasmus Pagh for a different proof.

6.1 Discussion of Cuckoo Hashing

In the above description, we used two tables to make the exposition clearer; we could just use a single table T of size $4M$ and two hash functions h_1, h_2, and a result similar to Theorem 8 will still hold. Now this starts to look more like the two-choices paradigm from the previous section: the difference being that we are allowed to move balls around, and also have to move these balls on-the-fly.

One question that we care about is the occupancy rate: the theorem says we can store N objects in $4N$ locations and get constant insert time and expected constant insert time. That is only using 25% of the memory! Can we do better? How close to 100% can we get? It turns out that you can get close to 50% occupancy, but better than 50% causes the linear-time bounds to fail. What happens if we use d hash functions instead of 2? With $d = 3$, experimentally it seems that one can get more than 90% occupancy and still linear-time bounds hold. And what happens when we are allowed to put, say, two or four items in each location? Again there are experimental conjectures, but the theory is not fleshed out yet. See this survey for more open questions and pointers.

Moreover, do we really need $O(\log N)$-universal hash functions? Such functions are expensive to compute and store. Patrascu and Thorup [PT12] showed we can use *simple tabulation hashing* instead, and it gives performance very similar to that of truly-random hash functions. Cohen and Kane show that we cannot get away with 6-universal hash functions.

7 Another Application: Load Balancing

7.1 Load Balancing: The Model

Another central application of hashing is in load balancing. Suppose there are N jobs to be sent to M machines, and consider the case where $M = N$. So there exists a way to send each job to a machine and maintain load 1. But if we hash the jobs to machines, we will get some additional load due to randomness. How much?

Let's use the same formalism as last time.

- Jobs are indexed by keys from the universe U, and we have a set S of $|S| = N$ jobs to schedule. Let us imagine that each job has the same (unit) size.

- There are M machines, indexed by $[M] = \{0, 1, 2, \ldots, M-1\}$.
- We have a family H of hash functions $\{h_1, h_2, \ldots, h_k\}$, with each $h_i : U \to [M]$. We randomly pick a hash function $h \leftarrow H$, and then each job $x \in U$ is mapped to machine $h(x)$.

We want to analyze the "load" of this strategy. It is clear that the best way to schedule N jobs on M machines is to assign N/M jobs to each machine. Can we show that there exist hash families H such that for every subset set S of jobs, the load on all machines is $\approx N/M$ with high probability? Let's see.

> *Notation: We will often call jobs as "balls" and machines as "bins". Think of throwing balls into bins. You want no bin to get many balls. The "load" of a bin is the number of balls that map to it.*

7.2 Load-Balancing Using Hashing

To begin, consider the simplest case of $N = M$. We would like each machine to have $N/M = 1$ jobs, the average load. Suppose the hash functions were truly random: each $x \in U$ was mapped independently to a random machine in $[M]$. What is the maximum load in that case? Surprisingly, you can show:

Theorem 9 *The max-loaded bin has $O(\frac{\log N}{\log \log N})$ balls with probability at least $1 - 1/N$.*

Proof: The proof is a simple counting argument. The probability that some particular bin i has at least k balls is at most

$$\binom{N}{k}\left(\frac{1}{N}\right)^k \leq \frac{N^k}{k!} \cdot \frac{1}{N^k} \leq \frac{1}{k!} \leq 1/k^{k/2}$$

which is (say) $\leq 1/N^2$ for $k = \frac{8 \log N}{\log \log N}$. To see this, note that

$$k^{k/2} \geq (\sqrt{\log N})^{4 \log N / \log \log N} \geq 2^{2 \log N} = N^2.$$

So union bounding over all the bins, the chance of some bin having more than $8\frac{\log N}{\log \log N}$ balls is $1/N$. (I've been sloppy with constants, you can get better constants using Stirling's approximation.) $\Box$

Moreover, you can show that *this is tight.* The max-load with $M = N$ is at least $\Omega(\frac{\log N}{\log \log N})$ with high probability. So even with truly random hash functions, the load is much above the average.

Observe that the calculation showing that the maximum load is $O(\frac{\log N}{\log \log N})$ only used that every set of $O(\frac{\log N}{\log \log N})$ balls behaves independently. This means that we do not need the hash family to be fully independent: it suffices to use $O(\frac{\log N}{\log \log N})$-universal hash family to assign balls to bins.

Still, storing and computing with $O(\frac{\log N}{\log \log N})$-universal hash families is expensive. What happens if we use universal hash families to map balls to bins? Or k-universal for $k \ll \frac{\log N}{\log \log N}$? How does the maximum load change? For that it will be useful to look at some *concentration* bounds.

7.3 Concentration Bounds

What is a concentration bound? Loosely, you want to say that some random variable stays "close to" its expectation "most of the time".

7.3.1 Markov's Inequality: Using the Expectation

The most basic bound is *Markov's inequality*, which says that any non-negative random variable is "not much higher" than its expectation with "reasonable" probability. If X is a non-negative random variable, then

$$\Pr[\ X \geq kE[X]\ ] \leq \frac{1}{k}.$$

7.3.2 Chebyshev's Inequality: Using the Variance

Using the second moment (this is $E[X^2]$) of the random variable, we can say something stronger. Define $\mathrm{Var}(X) = \sigma^2 = (E[X^2] - E[X]^2)$, the variance of the random variable.

$$\Pr[\ |X - E[X]| \geq k\sigma\ ] \leq \frac{1}{k^2}.$$

Now we don't need X to be non-negative. This is *Chebyshev's inequality.* The proof, interestingly, just applies Markov's to the r.v. $Y = (X - E[X])^2$.

Example: Let's get some practice with using Chebyshev's inequality:

Lemma 10 *Suppose you map N balls into $M = N$ bins using a 2-universal hash family H, maximum load over all bins is $O(\sqrt{N})$ with probability at least* $1/2$.

Proof: Let L_i be the load of bin i. Let X_{ij} be the indicator random variable for whether ball j fell into bin i. Note that $E[X_{ij}] = 1/M$, and hence $E[L_i] = \sum_{j=1}^{N} E[X_{ij}] = N/M$. Moreover, since the variables are pairwise-independent, we have $\mathrm{Var}(L_i) = \sum_{j=1}^{N} \mathrm{Var}(X_{ij})$. (Verify this for yourself!) And $\mathrm{Var}(X_{ij}) = E[X_{ij}^2] - E[X_{ij}]^2 = E[X_{ij}] - E[X_{ij}]^2 = (1/M - 1/M^2)$. So $\mathrm{Var}(L_i) = N(\frac{1}{M} - \frac{1}{M}^2)$.

Now Chebyshev says the probability of deviation $|L_i - N/M|$ being more than $\sqrt{2M} \cdot \sqrt{\mathrm{Var}(L_i)} \leq \sqrt{2N}$ is at most $\frac{1}{2M}$. And taking a union bound over all M bins means that with probability at least half, all bins have at most $N/M + \sqrt{2N}$ balls. □

Hmm. If we use 2-universal hash families to throw N balls into N bins, we are guaranteed a maximum load $O(\sqrt{N})$ when $N = M$. (For this proof we used that any two balls behaved uniformly and independently of each other.) But using fully random hash functions — or even $O(\frac{\log N}{\log\log N})$-universal hash functions — gives us max-load $\Theta(\frac{\log N}{\log\log N})$.

How does the max-load change when we increase the independence from 2 to fully random? For this, let us give better concentration bounds, which use information about the higher moments of the random variables.

7.3.3 Higher-Moment Chebyshev

A *higher-moment Chebyshev* shows that for any random variable X and even powers $p \geq 2$,

$$\Pr[\ |X - E[X]| \geq D\ ] \leq \frac{E[(X - E[X])^p]}{D^p}.$$

You can use this to show better bounds for hashing using p-universal hash families.

7.3.4 Chernoff/Hoeffding Concentration Bounds

Perhaps the most useful concentration bound is when X is the sum of *bounded independent* random variables. Hoeffding's bound that is often useful, though it's not the most powerful.

Before we give this, recall the Central Limit Theorem (CLT): it says that if we take a large number of copies $X_1, X_2, \ldots, X_n$ of some independent random variable with mean μ and variance σ^2, then their average behaves like a standard Normal r.v. in the limit; i.e.,

$$\lim_{n\to\infty} \frac{(\sum_i X_i)/n - \mu}{\sigma^2} \sim N(0,1).$$

And the standard Normal is very concentrated: the probability of being outside $\mu \pm k\sigma$, i.e., being k standard deviations away from the mean, drops *exponentially* in k. You should think of the bound below as one quantitative version of the CLT.

Theorem 11 (Hoeffding's Bound) *Suppose $X = X_1 + X_2 + \ldots + X_n$, where the X_is are independent random variables taking on values in the interval $[0,1]$. Let $\mu = E[X] = \sum_i E[X_i]$. Then*

$$\Pr[X > \mu + \lambda] \leq \exp\left(-\frac{\lambda^2}{2\mu + \lambda}\right) \tag{3}$$

$$\Pr[X < \mu - \lambda] \leq \exp\left(-\frac{\lambda^2}{3\mu}\right) \tag{4}$$

A comment on Hoeffding's bound.[7] Suppose $\lambda = c\mu$. Then we see that the probability of deviating by $c\mu$ drops *exponentially* in c. Compare this to Markov's or Chebyshev's, which only give an inverse polynomial dependence ($1/c$ and $1/c^2$ respectively).

> **Example:** Suppose each X_i is either 0 with probability 1/2, or 1 with probability 1/2. (Such an X_i is called a Bernoulli r.v.) Let $X = \sum_{i=1}^n X_i$. If you think of 1 has "heads" and 0 as "tails" then X is the number of heads in a sequence of n coin flips. If n is large, by the Central Limit Theorem we expect this number to be very close to the average. Let's see how we get this.
>
> Each $E[X_i] = 1/2$, hence $\mu := E[X] = n/2$. The bound (3) above says that
>
> $$\Pr[X > n/2 + \lambda] \leq \exp\left(-\frac{\lambda^2}{2(n/2) + \lambda}\right)$$
>
> Clearly $\lambda > n/2$ is not interesting (for such large λ, the probability is zero), so focus on $\lambda \leq n/2$. In this case $2(n/2) + \lambda \leq 3(n/2)$, so the RHS above is at most $e^{-\lambda^2/(3n/2)}$.
>
> E.g., for $\lambda = 30\sqrt{n}$, the probability of $X \geq n/2 + 30\sqrt{n}$ is at most $e^{-(900n)/(3n/2)} = e^{-600}$. (Smaller than current estimates of the number of particles in the universe.) A similar calculation using (4) shows that $\Pr[X < n/2 - \lambda] \leq e^{-\lambda^2/(3n/2)}$. Since the standard-deviation $\sigma = \sqrt{n}$ in this case, these results are qualitatively similar to what the CLT would give in the limit.

The price we pay for such a strong concentration are the two requirements that (a) the r.v.s be independent and (b) bounded in $[0,1]$. We can relax these conditions slightly, but some constraints

[7] Plus a jargon alert: such "exponential" concentration bounds for sums of independent random variables go by the name "Chernoff Bounds" in the computer science literature. It stems from the use of one such bound originally proved by Herman Chernoff. But we will be well-behaved and call these "concentration bounds".

are needed for sure. E.g., if X_i's are not independent, then you could take the X_i's to be *all equal*, with value 0 w.p. 1/2 and 1 w.p. 1/2. Then $\sum_{i=1}^{n} X_i$ would be either 0 or n each with probability 1/2, and you cannot get the kind of concentration around $n/2$ that we get in the example above. Similarly, we do need some "boundedness" assumption. Else imagine that each X_i is independent, but now 0 w.p. $1-1/2n$ and $2n$ w.p. $1/2n$. The expectation $E[X_i] = 1$ and hence $\mu = \sum_i E[X_i] = n$. But again you don't get the kind of concentration given by the above bound.

> **Example:** Back to balls-and-bins: let's see how to use Hoeffding's bound to get an estimate of the load of the max-loaded bin.
>
> So, in the $N = M$ case, let L_i be the load on bin i, as above. It is the sum of N i.i.d. random variables X_{ij}, each one taking values in $\{0, 1\}$. We can apply Hoeffding's bound. Here, $\mu = E[X] = N/M = 1$. Set $\lambda = 6 \log N$. Then we get
>
> $$\Pr[L_i > \mu + \lambda] \leq \exp\left(-\frac{\lambda^2}{2\mu + \lambda}\right) \leq \exp\left(-\frac{(6 \log N)^2}{2 + 6 \log N}\right) \leq e^{-2 \log N} = \frac{1}{N^2}.$$
>
> Now we can take a union bound over all the N bins to claim that with probability at least $1 - 1/N$, the maximum load is at most $O(\log N)$. This is weaker than what we showed in Theorem 9, but it still is almost right.[8]

8 Load Balancing using Two-Choice Hashing

Just like in cuckoo hashing, here's a more nuanced way to use hashing for load balancing — use two hash functions instead of one! The setting now is: N balls, $M = N$ bins. However, when you consider a ball, you pick *two* bins (or in general, d bins) independently and uniformly at random, and put the ball in the less loaded of the two bins. The main theorem is:

Theorem 12 (Azar, Broder, Karlin, Upfal [ABKU99]) *For any $d \geq 2$, the d-choice process gives a maximum load of*

$$\frac{\ln \ln N}{\ln d} \pm O(1)$$

with probability at least $1 - O(\frac{1}{N})$.

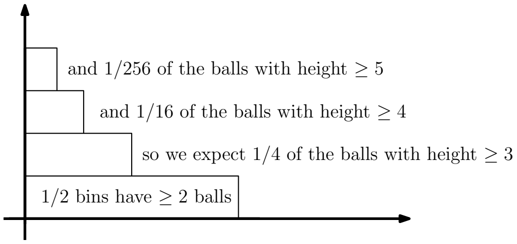
It's pretty amazing: just by looking at two bins instead of one, and choosing the better bin, gives us an exponential improvement on the maximum load: from $\approx \log N$ to $\log \log N$. Moreover, this analysis is tight. (Finally, looking at $d > 2$ bins does not help much further.)

Why is this result important? It is clearly useful for load balancing; if we want to distribute jobs among machines, two hash functions are qualitatively superior to one. We can even use it to give a simple data structure with good worst-case performance: if we hash N keys into a table with $M = N$ bins, but we store up to $O(\log \log N)$ keys in each bin and use two hash functions instead of just one, we can do inserts, deletes and queries in time $O(\log \log N)$.

8.1 Some Intuition

The intuition behind the proof is the following picture: Consider a ball to have height h if it is placed in a bin that has $h - 1$ balls before this ball was added into it. (We would like to show that no ball has height more than $\frac{\ln \ln N}{\ln 2} + O(1)$.) A bin has height $\geq h$ if it contains a ball of height h.

[8] We stated Hoeffding's bound in its most easy-to-use format. The actual bound is tighter and shows load $O(\frac{\log N}{\log \log N})$. See the last bound on page 2 of these notes if you are interested.


and 1/256 of the balls with height ≥ 5
and 1/16 of the balls with height ≥ 4
so we expect 1/4 of the balls with height ≥ 3
1/2 bins have ≥ 2 balls

How many bins can have height (at the end) at least 2? Clearly, at most $N/2$, at most half the bins. Now what is the chance that a ball has height at least 3? When it arrives, it must choose two bins, both having height at least 2. That happens with probability at most $\frac{1}{2} \times \frac{1}{2} = \frac{1}{4}$. Hence we expect about $\frac{N}{4}$ balls to have height at least 3 — and the expected number of bins of height at least 3 is at most $N/4 = N/(2^2)$. Using the same argument, we expect at most $N \cdot (1/4)^2 = N/16 = N/(2^{2^2})$ bins to have height at least 4, and $N/2^{2^{h-2}}$ bins to have height at least h. For $h = \log\log N + 2$, we expect only one ball to have height h.

Of course this is only in expectation, but one can make this intuition formal using concentration bounds, showing that the process does behave (more or less) like we expect. See these notes for a proof.

8.2 A Proof Based on Random Graphs*

Another way to show that the maximum load is $O(\log\log N)$—note that the constant is worse—is to use an first-principles analysis based on properties of random graphs. (If we have time we will cover this in class.)

We build a random graph G as follows: the N vertices of G correspond to the N bins, and the edges correspond to balls—each time we probe two bins we connect them with an edge in G. For technical reasons, we'll just consider what happens if we throw fewer balls (only $N/512$ balls) into N bins—also, let's imagine that each ball chooses two *distinct* bins each time.

Theorem 13 *If we throw $\frac{N}{512}$ balls into N bins using the best-of-two-bins method, the maximum load of any bin is $O(\log\log N)$ whp.*

Hence for N balls and N bins, the maximum load should be at most 512 times as much, whp. (It's as though after every $N/512$ balls, we forget about the current loads and zero out our counters—not zeroing out these counters can only give us a more evenly balanced allocation; I'll try to put in a formal proof later.)

To prove Theorem 13, we need two results about the random graph G obtained by throwing in $N/512$ random edges into N vertices. Both the proofs are simple but surprisingly effective counting arguments, you can see them here.

Lemma 14 *The size of G's largest connected component is $O(\log N)$ whp.*

Lemma 15 *There exists a suitably large constant $K > 0$ such that for all subsets S of the vertex set with $|S| \geq K$, the induced graph $G[S]$ contains at most $5|S|/2$ edges, and hence has average degree at most 5, whp.*

Good, let us now show begin the proof of Theorem 13 in earnest. Given the random graph G, suppose we repeatedly perform the following operation in rounds:

> *In each round, remove all vertices of degree ≤ 10 in the current graph.*

We stop when there are no more vertices of small degree.

Lemma 16 *This process ends after $O(\log\log N)$ rounds whp, and the number of remaining vertices in each remaining component is at most K.*

Proof: Condition on the events in the two previous lemmas. Any component C of size at least K in the current graph has average degree at most 5; by Markov at least half the vertices have degree at most 10 and will be removed. So as long as we have at least K nodes in a component, we halve its size. But the size of each component was $O(\log N)$ to begin, so this takes $O(\log\log N)$ rounds before each component has size at most K. □

Lemma 17 *If a node/bin survives i rounds before it is deleted, its load due to edges that have already been deleted is at most $10i$. If a node/bin is never deleted, its load is at most $10i^* + K$, where i^* is the total number of rounds.*

Proof: Consider the nodes removed in round 1: their degree was at most 10, so even if all those balls went to such nodes, their final load would be at most 10. Now, consider any node x that survived this round. While many edges incident to it might have been removed in this round, we claim that at most 10 of those would have contributed to x's load. Indeed, each of the other endpoints of those edges went to bins with final load at most 10. So at most 10 of them would choose x as their less loaded bin before it is better for them to go elsewhere.

Now, suppose y is deleted in round 2: then again its load can be at most 20: ten because it survived the previous round, and 10 from its own degree in this round. On the other hand, if y survives, then consider all the edges incident to y that were deleted in previous rounds. Each of them went to nodes that were deleted in rounds 1 or 2, and hence had maximum load at most 20. Thus at most 20 of these edges could contribute to y's load before it was better for them to go to the other endpoint. The same inductive argument holds for any round $i \leq i^*$.

Finally, the process ends when each component has size at most K, so the degree of any node is at most K. Even if all these edges contribute to the load of a bin, it is only $10i^* + K$. □

By Lemma 16, the number of rounds is $i^* = O(\log\log N)$ whp, so by Lemma 17 the maximum load is also $O(\log\log N)$ whp.

8.3 Generalizations

It's very interesting to see how the process behaves when we change things a little. Suppose we divide the N bins into d groups of size N/d each. (Imagine there is some arbitrary but fixed ordering on these groups.) Each ball picks one random bin from each group, and goes into the least loaded bin as before. But if there are ties then it chooses the bin in the "earliest" group (according to this ordering on the groups). This subtle change (due to Vöcking [Vöc03]) now gives us load:

$$2\,\frac{\log\log n}{d} + O(1).$$

Instead of $\log d$ in the denominator, you now get a d. The intuition is again fairly clean. [Draw on the board.]

What about the case when $M \neq N$? For the one-choice setting, we saw that the maximum load was $\frac{N}{M} + O(\sqrt{\frac{N \log m}{M}})$ with high probability. It turns out that the d-choice setting gives us load at most

$$\frac{N}{M} + \frac{\log\log M}{\log d} + O(1)$$

with high probability [BCSV06]. So the deviation of the max-load from the average is now independent of the number of balls, which is very desirable!

Finally, supposed you really wanted to get a constant load. One thing to try is: when the i^{th} ball comes in, pick a random bin for it. If this bin has load at most $\lceil i/M \rceil$, put ball i into it. Else pick a random bin again. Clearly, this process will result in a maximum load of $\lceil N/M \rceil + 1$. But how many random bin choices will it do? [BKSS13] shows that at most $O(N)$ random bins will have to be selected. (The problem is that some steps might require a lot of choices. And since we are not using simple hashing schemes, once a ball is placed somewhere it is not clear how to locate it.) This leads is to the next section, where you want to just use two hash functions, but maintain only low load — in fact only one ball per bin!

References

[ABKU99] Yossi Azar, Andrei Z. Broder, Anna R. Karlin, and Eli Upfal. Balanced allocations. *SIAM J. Comput.*, 29(1):180–200, 1999. 14

[BCSV06] Petra Berenbrink, Artur Czumaj, Angelika Steger, and Berthold Vöcking. Balanced allocations: The heavily loaded case. *SIAM J. Comput.*, 35(6):1350–1385, 2006. 17

[BKSS13] Petra Berenbrink, Kamyar Khodamoradi, Thomas Sauerwald, and Alexandre Stauffer. Balls-into-bins with nearly optimal load distribution. In *SPAA*, pages 326–335, 2013. 17

[BM03] Andrei Z. Broder and Michael Mitzenmacher. Survey: Network applications of bloom filters: A survey. *Internet Mathematics*, 1(4):485–509, 2003. 9

[DHKP97] Martin Dietzfelbinger, Torben Hagerup, Jyrki Katajainen, and Martti Penttonen. A reliable randomized algorithm for the closest-pair problem. *J. Algorithms*, 25(1):19–51, 1997. 8

[PPR05] Anna Pagh, Rasmus Pagh, and S. Srinivasa Rao. An optimal bloom filter replacement. In *SODA*, pages 823–829, 2005. 9

[PPR11] Anna Pagh, Rasmus Pagh, and Milan Ruzic. Linear probing with 5-wise independence. *SIAM Review*, 53(3):547–558, 2011. 4

[PR04] Rasmus Pagh and Flemming Friche Rodler. Cuckoo hashing. *J. Algorithms*, 51(2):122–144, 2004. 9

[PT10] Mihai Patrascu and Mikkel Thorup. On the k-independence required by linear probing and minwise independence. In *ICALP (1)*, pages 715–726, 2010. 4

[PT12] Mihai Patrascu and Mikkel Thorup. The power of simple tabulation hashing. *J. ACM*, 59(3):14, 2012. 7, 10

[PT13] Mihai Patrascu and Mikkel Thorup. Twisted tabulation hashing. In *SODA*, pages 209–228, 2013. 7

[Vöc03] Berthold Vöcking. How asymmetry helps load balancing. *J. ACM*, 50(4):568–589, 2003. 16