15-854: Algorithms in the Real World (Spring 2014) January 30, 2014
Lec #6: Hashing Last Updated: January 30, 2014

1 Hashing

Hash functions have many applications to dictionary data structures, to load balancing and symmetry breaking, to cryptography and complexity theory. In this set of lectures, we will study:

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

The basic application is to dictionaries. Here you 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 dictionary is some subset S of this universe, and 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 really care about adding and removing key, we just care about fast query times—e.g., the actual English dictionary does not change (or changes very rarely). This is called the *static case*. Another special case is when we just add keys: the *insertion-only case*.

1.1 Desired Properties

In this lecture, let $[N]$ denote the numbers $\{0, 1, 2, \ldots, N-1\}$. As you have probably seen before, the natural approach is to take a hash function $h : U \to [M]$ (which is chosen randomly from some hash family H, and this random choice is the only source of randomness), 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 $\Pr_{h \leftarrow H}[h(x) = h(y)]$ is "small".
(ii) Small range: we want M to be small. At odds with first desired property.
(iii) Small number of bits to store a hash function h. This is at least $O(\log_2 |H|)$
(iv) h is easy to compute.

> **An Important Note:** In this class we will assume that the dictionary S is chosen adversarially. Of course we don't see S. 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.

2 Universal Hashing

The definition of universal hashing tries to capture the 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*

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

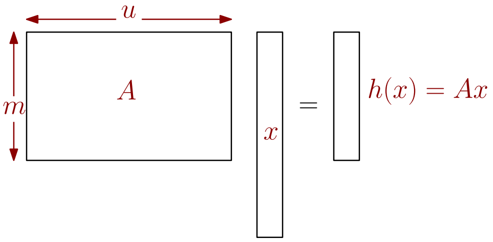
Make sure you understand the definition. This condition must hold for *every pair* of distinct keys, and the randomness is over the choice of the actual hash function h from the set H.

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.*

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

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

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 $O(N)$.[1] 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

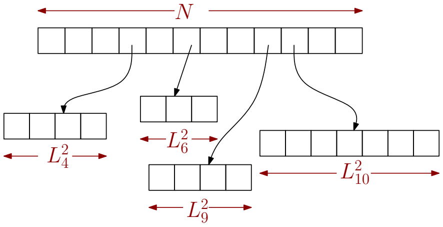
$$E[C] = E\left[\sum_{x \neq y \in S} C_{xy}\right] = E \sum_{x \neq y \in S} E\left[C_{xy}\right] \leq \binom{N}{2}\frac{1}{M}. \qquad (1)$$

[1] 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 (1)—we could ensure that all keys map to distinct locations. But that is a lot of wasted space.

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 (1) 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.

3 2-Universal and k-Universal Hashing

A couple years after their original paper, Carter and Wegman proposed a stronger requirement which they called 2-*universal* (or more generally, k-*universal*).

Definition 3 *A family H of hash functions mapping U to $[M]$ is called k-universal 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 of $k = 2$ is called pairwise independent.

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

Fact 4 *Suppose H is a k-universal family. Then*

a) H is also $(k-1)$-universal.
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-universality is indeed at least as strong a condition as universality. And one can check that the construction in Section 2.1 is not 2-universal (since $\Pr[h(\vec{0}) = \vec{0}] = 1$

and not $1/M$ as required above. In the next section we give some constructions of 2-universal and k-universal 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.*

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-universal.*

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-universal as part of your homework. Since This 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

For this construction, pick a prime p, and let $U = [p]$ and $M = p$ as well. (Hence we are hashing into a table of the same size as the universe! Seems useless, but read on.) Note that p being a prime means that $[p]$ has good algebraic properties: it forms the field $\mathbb{Z}_p$ (also denoted as $\mathrm{GF}(p)$).

> *Pick two random numbers $a, b \in \mathbb{Z}_p$. For any $x \in U$, define*
>
> $$h(x) := ax + b$$
>
> *where the calculations are done modulo p.*

To prove 2-universality, 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 $x_1 \neq x_2$. But since a, b are chosen randomly, the chance that each of them equals some specified values is at most $1/p \times 1/p = 1/p^2$, which is $1/M^2$ as desired for 2-universality.

That's cute. On the other hand, we hashed $[p] \to [p]$, which does not seem useful. That is true, but the same idea works for any field. So we could use the field $\mathrm{GF}(2^u)$ which has a correspondence with u-bit strings, and hence hash $[2^u] \to [2^u]$. 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 $m \leq u$. It requires knowledge of some algebra about Galois fields; we omit the details here.

3.1.4 Construction #4: k-universal Hashing

The construction for k-universal is not very different; let's consider hashing $[p]$ to $[p]$ once again.

> *Pick k random numbers $a_0, a_1, \ldots, a_{k-1} \in \mathbb{Z}_p$. 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 modulo p.*

The proof of k-universality is similar to that above; this is something you'll show in Homework #2. Again, we can then use $\mathrm{GF}(2^u)$ and the same ideas to get k-universal hash functions from $[2^u] \to [2^m]$.

4 Other Hashing Schemes with Good Properties

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].

4.2 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.)

5 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.[2] 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`.*

[2] 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. For a false positive on query x, all the k locations $T[h_1(x)]$, $T[h_2(x)], \ldots,$ $T[h_k(x)]$ must be set, which 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.[3]

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 Further Reading

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.

References

[BM03] Andrei Z. Broder and Michael Mitzenmacher. Survey: Network applications of bloom filters: A survey. *Internet Mathematics*, 1(4):485–509, 2003. 8

[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. 7

[PPR05] Anna Pagh, Rasmus Pagh, and S. Srinivasa Rao. An optimal bloom filter replacement. In *SODA*, pages 823–829, 2005. 8

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

[PT10] Mihai Patrascu and Mikkel Thorup. On the k-independence required by linear probing and minwise independence. In *ICALP (1)*, pages 715–726, 2010. 3

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

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

[3] 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.