Com S 435/535: Notes IV

1 Hashing

Let U be an universe and S be a set. We would like to store S in a hash table T. Suppose h is hash function from U to T . The idea is to use h to store S as follows: Given an element $x \in S$, we store it at location $T[h(x)]$. In future, if we want to search for a key a, then we compare a with $T[h(a)]$. For this to work, it should not be the case that there exist two distinct elements x and y from S such that $h(x) = h(y)$. Does there exist a hash function with that property? If so, how to find such hash function? Observe that if we allow $|T| = |U|$, then this is trivial. However we want $|S| \approx |T|$. Let $|S| = N$ and $|T| = M$. Our goal is find a function h from U to T such that for every $x \neq y \in S$, $h(x) \neq h(y)$. That is, the function h must be one-one on S. The solution is simple: Pick a random function. We show that a random function is one-one on S with reasonable probability (for an appropriate choice of M).

Consider the following collision pair set:

$$C = \{\langle x, y\rangle \mid h(x) = h(y), x < y, x \in S, y \in S\}$$

Note that C is a "random set" and the size of C is a random variable. Clearly h is one-one on S if and only if $C = \emptyset$ (equivalently $|C| = 0$). If we randomly pick h, what is the expectation of the size of C?

For every x and y in S ($x \neq y$), consider the following random variable

$$C_{xy} = \begin{cases} 1 \; if \; h(x) = h(y) \\ 0 \; else \end{cases}$$

Clearly,

$$|C| = \sum_{x,\; y \in S,\; x < \; y} C_{xy}$$

First let us compute $E[C_{xy}]$.

$$\begin{aligned} E[C_{xy}] &= \Pr[C_{xy} = 1] \times 1 + \Pr[C_{xy} = 0] \times 0 \\ &= \Pr[C_{xy} = 1] \\ &= \Pr[h(x) = h(y)] \end{aligned}$$

Let E denote the event that $h(x) = h(y)$, and let E_i, $1 \leq i \leq M$ denote the event that $h(x) = h(y) = i$. Note that the events E_i's ($1 \leq i \leq M$) are mutually disjoint and

$$E = \cup_{i=1}^{M} E_i$$

Thus

$$\Pr[h(x) = h(y)] = \Pr[E] = \sum_{i=1}^{M} \Pr[E_i]$$

Fix an i, and let us consider the event E_i. This happens when both $h(x)$ and $h(y)$ are equal to i. Since h is an uniformly chosen random function, probability that $h(x) = i$ is $1/M$ and the probability that $h(y) = i$ is also $1/M$. Moreover these two events are independent. Thus the probability that $h(x) = h(y) = i$ is $1/M^2$. Thus

$$\begin{aligned} \Pr[h(x) = h(y)] &= \sum_{i=1}^{M} \frac{1}{M^2} \\ &= \frac{1}{M} \end{aligned}$$

Thus $E[C(xy)] = \frac{1}{M}$. Recall that

$$C = \sum_{x<y, x\in S, y\in S} C_{xy}$$

By linearity of expectation, we have

$$\begin{aligned} E[C] &= E[\sum_{x<y, x\in S, y\in S} C_{xy}] \\ &= \sum_{x<y, x\in S, y\in S} E[C_{xy}] \\ &= \frac{N(N-1)}{2} \times \frac{1}{M} \end{aligned}$$

If we take M to be N^2, then $E[C]$ is less than $1/2$. Using Markov inequality, we obtain that

$$\Pr(C < 1) \geq \frac{1}{2}$$

. Thus, if we randomly pick a hash function then probability that there is no collision is at least half. If we randomly pick 20 hash functions then at least one of them is one-one on S is probability bigger than $1 - 1/2^{20}$. Thus if we choose the size of the table to be N^2, then we can find a hash function that is good for S by repeating the following process for 20 times. Randomly pick a hash function h. Verify that it is one-one of S, if so h is our hash function. The probability that we fail to find a good hash function is less than 1 in million.

There are at least two problems with the above approach. First, the size of the table is N^2 is too large and could be prohibitive in certain applications (think of when the set S has a million items). Secondly, we may not know the set S in advance. It is possible that S is generated by an online process, and we have to store each element of S in the table on the fly. We now address these questions.

Our goal is to make the size of the table T be equal to N. If we carry out the above analysis, then we obtain that the expected number of collisions is $\frac{N-1}{2}$. Thus we are bound to have collisions. We use *chain hashing* . The idea is as follows: For each i, $1 \leq i \leq |T|$, we store a linked list at

$T[i]$. We store an element x in the table as follows: compute $h(x)$ and store x in the linked list at $T[h(x)]$. To search for a key a, we search whether a appears in the linked list at $T[h(a)]$.

Note that the time taken to search for a key a depends on the size of the list at $T[h(a)]$. How large could it be? We will now show that with very high probability, the size of this list is at most $2\log N + 1$. Let L_i denote the list at $T[i]$.

Theorem 1. *Suppose the we randomly picked a function h from U to T and stored a set S of size N in the hash table (using chain hashing) T of size N. Then,*

$$\Pr[\exists i, |L_i| > 2\log N + 1] \leq 1/N$$

Proof. Let us fix i. We will first bound the probability the the size of L_i is more than $2\log N + 1$. Let Y be a random variable that denotes the size of L_i. Recall that an element $x \in S$, is placed in L_i when $h(x) = i$. Thus each $x \in S$ will contribute to Y when x is mapped to i via h. For each $x \in S$, let Y_x be a random variable whose value if 1 if $h(x) = i$; otherwise $h(x) = 0$. Note that $Y = \sum_{x \in S} Y_x$. Thus by linearity of expectation $E[Y] = \sum_{x \in S} E[Y_x]$. Since Y_x is a 0-1 random variable $E[Y_x] = \Pr[Y_x = 1] = \Pr[h(x) = i]$. Since h is a random function, the probability that $h(x) = i$ is $1/|T|$ which is $1/N$. Thus $E[Y_x] = 1/N$ and thus $E[Y] = 1$. Note that Y is sum of N many 0-1 independent random variables with identical expectation $1/N$. This enables us to apply Chernoff bound.

$$\begin{aligned}
\Pr[Y > 2\log N + 1] &= \Pr[Y - 1 > 2\log N] \\
&= \Pr[\frac{Y}{N} - \frac{1}{N} > 2\log N \frac{1}{N}] \\
&\leq \Pr[|\frac{Y}{N} - \frac{1}{N}| > 2\log N \frac{1}{N}] \\
&\leq 2e^{-4\log N \times \frac{1}{N} \times N} \\
&\leq \frac{1}{N^2}
\end{aligned}$$

Thus the probability that the size of L_i is more than $2\log N + 1$ is at most $1/N^2$. Let E_i denote the event that the size of L_i is more than $2\log N + 1$. Let E denote the event that for some i, the size of L_i is more than $2\log N + 1$. Note that $E = \cup E_i$. Thus

$$\begin{aligned}
\Pr[\exists i, |L_i| > 2\log N + 1] &= \Pr[E] \\
&= \Pr[\cup E_i] \\
&\leq \sum \Pr[E_i] \quad \text{By union Bound} \\
&\leq \sum \frac{1}{N^2} \\
&= \frac{1}{N}
\end{aligned}$$

□

Thus by taking a hash table of size N, we can store the set S. The worst case time to search for a key is $O(\log N)$.

2 Bloom Filters

We will now consider memory-efficient data structures to store a set S of size N. Let us assume that on average each element of S has ℓ characters. Thus to represent S we need at least $8\ell N$ bits. For example if we store S in a hash table, or in a binary tree or in a linked list the memory used the data structure is at least $8\ell N$. Can we reduce the memory? In general it is not possible. We show that if we can tolerate some errors (for a membership query), then it is indeed possible to design a data structure that uses less memory.

The idea is to build upon hashing. Let S be a set of size N and let T be a table of size M, and let h be a (randomly chosen) hash function. Recall that to store x in T, we compute $h(x)$ and store x in the linked list at $T[h(x)]$. What happens if store a bit 1, at $T[h(x)]$, instead of storing a list? That is consider the following data structure. We start with a table T of size M. Each cell of T can store one single bit, initially all cells of the table are all zeros. To store x in T, we place 1 at $T[h(x)]$.

When we want to find whether a key a appears in the set S, we look at $T[h(a)]$. If that bit is 1, then we answer "yes"; otherwise "NO". Note that if $a \in S$, then we always answer "yes", however when $a \notin S$, we may still answer "yes". This happens when there is some other element $x \in S$ such that $h(x) = h(a)$. These kind of errors are called "false positives". Let us calculate the probability of false positive. We will look at the complement event. We are given a key $a \notin S$, and we would like to compute the probability that we answer "NO". This happens when for every $x \in S$, $h(x) \neq h(a)$. A simple calculation shows that this probability is $(1 - 1/N)^N$ which equals $1/e$ which is rather very small (around 0.36). Thus this is not an ideal solution. Bloom Filters build on this idea. The man idea is to use multiple hash functions instead of a single hash functions.

Let $|S| = N$, and $|T| = M$. Uniformly at random pick k hash functions $h_1, h_2, \cdots, h_k$. Initially $T[i] = 0$ for every i, $1 \leq i \leq M$. To add x in to the Bloom Filter, set all of $T[h_1(k)], \cdots T[h_k(x)]$ to 1. Now, to check if a key a belongs to S or not check if all of $T[h_1(a)], \cdots T[h_k(a)]$ are all ones. Again, observe that if $a \in S$, we always give correct answer; however if $a \notin S$, then there is a chance that we may still answer "yes". Let us bound this probability.

Let us fix a bit of T, say t^{th} bit. Supposed we added all elements of S to the filter. What is the probability that $T[t] = 1$. Let us look at the complement event: $T[t] = 0$. We assumed that our hash functions are random and independent. This implies that, for a given $x \in S$, we randomly chose k bits of T and set them to 1. We are repeating this process N times. Thus

$$\begin{aligned} p = \Pr[T[t] = 0] &= (1 - \frac{1}{M})^{kN} \\ &= ((1 - \frac{1}{M})^{M})^{kN/M} \\ &= e^{-kN/M} \end{aligned}$$

Thus $\Pr[T[t] = 1] = (1 - p)$, where $p = e^{-kN/M}$.

Let us fix a key $a \notin S$. What is the probability that we get a false positive? This happens when all of $T[h_1(a)], \cdots T[h_k(a)]$ are ones.

$$\begin{aligned} \Pr[T[h_1(a)] = 1 \cap \cdots \cap T[h_k(a)] = 1] &= (1 - p)^k \\ &= (1 - e^{-kN/M})^k \end{aligned}$$

We can view the above probability as a function of k and compute a value of k at which the above probability is minimized. Let

$$f(k) = (1 - e^{-kN/M})^k$$

and

$$h(k) = \ln(1 - e^{-kN/M})^k = k \ln(1 - p)$$

Since $p = e^{-kN/M}$, we have that $k = \frac{-M}{N} \ln p$. Thus

$$h(k) = \frac{-M}{N} \ln p \ln(1 - p)$$

Note that the above quantity is minimized when $p = 1/2$.

Thus if we take $p = 1/2$, then if we take

$$k = \frac{M}{N} \ln 2$$

many hash functions, then the false positive probability is minimized, and the false positive probability is

$$(1 - p)^k = \frac{1}{2^{M/N \ln 2}} = (0.618)^{M/N}$$

If we take $M = 8N$, the this probability is around 0.02 and if we take $M = 16N$, then this probability is 0.0004. If we take M to be $8N$, then size of the filter (size of the table T) is $8N$. Compare this to $8\ell N$. Note that the size of the bloom filter is independent of the number of bits to represent each element of the set S.

Applications of Bloom Filter. One of the earliest application of the bloom filter was to use in early spell-checking programs, when memory was limited and it is expensive to store entire dictionary in the main memory. Instead, a Bloom Filter of the dictionary is created and stored in the memory. Any word that is not in the filter is certainly a mis-spelled word. This program occasionally errs–it may think that a misspelled word is in the dictionary.

A modern day application of Bloom Filters is in proxy web servers. This is a distributed system consisting several proxy servers; and each proxy server keeps a copy (cache) of popular web pages. When an user sends a request for a webpage, one of the proxy servers (closest to the user) intercepts the request. If the proxy server has the page, then it serves that page. If not, it will request all other (nearby) proxy servers if they have the page in their cache. If none of them have the page, then the request is sent to the source. One way to minimize the transmission is to have each proxy (periodically) send a list of all pages that they currently have to all other proxies. If a particular proxy does not have the requested webpage, then it can quickly look which proxy has the requested page, and redirect the request to that proxy server. For this, the proxies have to communicate a list of all web pages that they currently hold. This communication can be reduced by using Bloom Filter. Each proxy computes a Bloom Filter of the list of the pages that it contains, and the proxies exchange the Bloom Filters. This further reduces the communication.

(Variants of) Bloom filters have been used by `bitly` (in detecting malicious web-sites) and in FaceBook type head search. Please see
`http://word.bitly.com/post/28558800777/dablooms-an-open-source-scalable-counting`
and `https://www.facebook.com/video/video.php?v=432864835468`.

3 Hash Functions

So far we assumed that we can uniformly at random pick a function and use it as a hash function. There is a caveat with this; it is not clear how to pick a hash function uniformly at random. Secondly, once a hash function is picked, we have to store the function (along with the hash table/Bloom Filter). With very high probability a randomly chosen hash function from U to T requires $U \log T$ bits to store and this is obviously prohibitive. There are two ways to get around this problem. The first approach is to pick a random hash function that can be succinctly represented. The second approach is to design a deterministic function that exploits the "randomness in the data".

Random Functions . Let us think of S as $\{1, 2, \cdots, N\}$. Pick a prime p that is bigger than N and set the size of the hash table T to be p. Now uniformly at random pick $a \in \{0, 1, \cdots, p-1\}$ and uniformly at random pick $b \in \{0, 1, \cdots, p-1\}$. The pair $\langle a, b \rangle$ defines our hash function h as follows: $h(x) = (ax + b)\%p$. Note that to store this hash function we only need to store a and b. Another approach is to pick a random prime p between N and $2N$ and set the size of the hash table T to be p. Now the hash function h is defined as follows: $h(x) = x\%p$. To store this hash function, we only need to store the prime p. We can use the random number generator to implement these function. For example, in Java we can use the `nextInt` method from the class `Random` to generate random numbers.

Deterministic Functions. This approach assumes that the there is some randomness in the data, i.e, the data came from some distribution that is not uniform. The goal us to design a function h that disperses randomness in S. More formally let D be distribution and let S be set generated according to this distribution. Our goal is to design h such that $h(D)$ is uniform. Several hash functions have been proposed as candidates that seem to behave well in practice. One of them is FNV hash function. It can output a 32 bit hash value or 64 bit hash value. FNV-64 (64-bit hash function) works as follows.

1. Input: A string s.
2. CONSTANT: FNV64PRIME $= 2^{40} + 2^8 + 0x3b = 109951168211$
3. CONSTANT: FNV-64INIT $= 14695981039346656037$.
4. Set h to FNV-64INIT.
5. for i in range $[0, \cdots, s.length() - 1]$ DO
   - $h = h\ XOR\ s[i]$;
   - $h = (h * FNV64PRIME)\%2^{64}$.
6. Output h

   Please see the following page for more on FNV hash functions:
   `http://www.isthe.com/chongo/tech/comp/fnv/`

   Other commonly used hash functions are Jenkins, murmur, MD5, and SHA. The last two are cryptographic hash functions.