CS 466/666: Algorithm Design and Analysis. Spring 2019. Waterloo.

Lecture 6: Hashing

Hashing is not only useful in designing an efficient data structure for searching, but is also useful in designing data streaming algorithms and in derandomization.

An important concept that we will study is the notion of k-wise independent variables.

We will see applications in data streaming in the next lecture.

Hash Functions

A primary motivation of designing a hash function is to obtain an efficient data structure for searching.

We would like to achieve $O(1)$ search time using the RAM (random access machine) model.

In the RAM model, we assume that we can access an arbitrary position in an array in $O(1)$ time.

We also assume that the word size is large enough and it takes $O(1)$ time for a word operation.

Consider the scenario that we would like to store n elements (keys) from the set $M=\{0,\dots,m-1\}$.

An obvious approach is to use an array A of m elements, initially $A[i]=0$ for all i, and when a key is inserted, we set $A[i]=1$, and this supports searching in $O(1)$ time in the RAM model.

But, of course, this approach requires too much space, when $m \gg n$.

For example, consider the scenario that M is the set of all IP-addresses and n is the number of IP addresses visiting Waterloo per day.

Ideally, we would like to use only an array of size $O(n)$ and still supports searching in $O(1)$ time.

A hash function is used to map the elements of the big universe to the locations in the small table.

A hash table is a data structure that consists of the following components:

- a table T with n cells indexed by $N=\{0,1,\dots,n-1\}$, each cell can store $O(\log m)$ bits.
- a hash function $h: M \Rightarrow N$.

Ideally, we want the hash function to map different keys into different locations.

But, of course, by the pigeonhole principle, this is impossible to achieve if we do not know the keys in advance.

We say there is a collision if $x \neq y$ but $h(x) = h(y)$.

Instead, what we can hope for is to have a family of hash functions, so that the number of collisons is small with high probability, if we pick a random hash function from the family.

We assume the keys coming are independent from the hash function that we choose (i.e. there is no "adversary" who knows our internal randomness and choose a bad set of keys for our hash functions).

We do not assume that we know the keys in advance. (Actually, even if we know the keys in advance, it is non-trivial to design a good hash function; see perfect hashing later.)

A natural idea is to consider the hash family being the set of all functions from M to N, and we just pick a random function $h: M \to N$ as a hash function.

Thus, the setting is the same as the balls-and-bins setting that we just studied.

Suppose there are n keys.

Then, the expected number of keys in a location is one, and the maximum loaded location has $\Theta(\log n / \log\log n)$ keys.

We can store the keys that are hashed into the same location by a linked list.

This is called chain hashing, for which the expected search time is $O(1)$, while the maximum search time is $O(\log n / \log\log n)$.

Using the idea of power of two choices, we can use two random functions h_1 and h_2.

When we insert a value x, we look at the locations $h_1(x)$ and $h_2(x)$ and store x in a least loaded location. When we search, we search both the linked lists in location $h_1(x)$ and $h_2(x)$.

Then, we can reduce the maximum search time to $O(\log\log n)$ while not increasing the average search time by much.

Random Hash Functions ?

So far so good, but we ignored the issues of the time to evaluate $h(x)$ and the space requirement to store h (so that we could compute $h(x)$ again).

There is just no way we could do it efficiently for a random function.

Consider a random function $h: M \to N$. To store this table it requires at least $m \log n$ bits, as each element requires $\log n$ bits to remember its location. This is even more than having an array of size m. Without using so much space, there is no way to compute $h(x)$ efficiently; we don't even know which function to compute, and can't get consistent answers to the queries.

Ideally, if we use $O(n)$ cells for the hash table where each cell stores $\log m$ bits, we would like to store the function using $O(1)$ cells, so that it does not create any overhead in storage requirement.

This means that $O(\log m)$ bits can represent the hash function, and therefore the hash family should have at most poly(m) functions (instead of n^m functions from M to N).

Fortunately, by choosing a hash function from a small hash family, we can still achieve some of the properties guaranteed by the random hash functions. This is similiar in spirit to derandomization, but here we do that for the sake of efficiency of time and space.

In short, we need a succint representation of a hash function, to support fast query and require little storage space.

For this, we introduce a weaker definition of randomness.

Pairwise Independence

For a set of n independent random variables, we have $\Pr\left(\bigcap_{i=1}^{n}(X_i = x_i)\right) = \prod_{i=1}^{n} \Pr(X_i = x_i)$.

<u>Definition</u> A set of random variables $X_1, X_2, \ldots, X_n$ is k-wise independent if for any subset $I \subseteq [1,n]$ with $|I| \leq k$ and for any values x_i, $i \in I$, $\Pr\left(\bigcap_{i \in I}(X_i = x_i)\right) = \prod_{i \in I} \Pr(X_i = x_i)$.

The special case when $k=2$ is called <u>pairwise independence</u>.

To see why pairwise independent variables are easier to generate we consider two examples.

<u>Example 1</u> Given b random bits $X_1, \ldots, X_b$, one can generate $2^b - 1$ pairwise independent bits as follows:

Enumerate the $2^b - 1$ nonempty subsets of $\{1, 2, \ldots, b\}$.

For each such subset S, define $Y_S = \bigoplus_{i \in S} X_i$, where $\oplus$ denotes the mod-2 operation.

Then it is easy to see that each bit is random by the principle of deferred decision.

(The principle of deferred decision says that we can fix a bit x_i with $i \in S$, and think of the process of choosing a random string $x_1, \dots, x_b$ as first deciding the value of other bits and then deciding the value of the i-bit. By fixing all other bits, since $x_i = 1$ or 0 with probability $\frac{1}{2}$, Y_S is still equally likely to be 0 or 1.

More formally, this principle is just saying that $Pr(Y_S = 1 \mid \text{all bits except } x_i) = \frac{1}{2}$ for all possible values of the remaining bits implies that $Pr(Y_S = 1) = \frac{1}{2}$, and it follows from conditional probability.)

Consider two subsets S_1 and S_2. Let $x_i \in S_1 - S_2$. By the principle of deferred decision on x_i, we can also argue that $Pr(Y_{S_1} = c \mid Y_{S_2} = d) = \frac{1}{2}$, for any two values $c, d \in \{0,1\}$.

Therefore, $Pr(Y_{S_1} = c \cap Y_{S_2} = d) = Pr(Y_{S_1} = c \mid Y_{S_2} = d)\, Pr(Y_{S_2} = d) = \frac{1}{2} \cdot \frac{1}{2} = \frac{1}{4}$.

This proves the pairwise independence.

Example 2 Given two independent, uniform values X_1 and X_2 over $\{0, 1, \dots, p-1\}$, we can erate p pairwise independent random variables by setting $Y_i = (X_1 + i X_2) \bmod p$ for $i = 0, 1, \dots, p-1$, where p is prime.

Again, by deferred decisions, Y_i is uniformly random.

For pairwise independence, we want to show that $Pr(Y_i = a \cap Y_j = b) = \frac{1}{p^2}$.

The event $Y_i = a$ and $Y_j = b$ is equivalent to $(X_1 + i X_2) \bmod p = a$ and $(X_1 + j X_2) \bmod p = b$.

There two equations on two variables have a unique solution (because p is prime, and multiplicative inverse exists)

$$X_2 = (b-a)(j-i)^{-1} \bmod p \quad \text{and} \quad X_1 = (a - i(b-a)(j-i)^{-1}) \bmod p.$$

Since there are p^2 choices for X_1 and X_2 and there is only one choice for $Y_i = a$ and $Y_j = b$, this implies that $Pr(Y_i = a \cap Y_j = b) = \frac{1}{p^2}$, proving the pairwise independence.

Chebyshev's inequality For pairwise independent variables, we have $E[X_i X_j] = E[X_i]E[X_j]$.

This implies that $Cov(X_i, X_j) = 0$.

It follows that $Var[\sum_{i=1}^{n} X_i] = \sum_{i=1}^{n} Var[X_i]$ for pairwise independent variables.

Applying Chebyshev's inequality gives: $Pr(|X - E[X]| \geq a) \leq \frac{Var[X]}{a^2} = \frac{\sum_{i=1}^{n} Var[X_i]}{a^2}$ for $X = \sum_{i=1}^{n} X_i$.

Universal Hash Functions

Definition Let U be a universe with $|U| \geq n$ and let $V = \{0, 1, \dots, n-1\}$. A family of hash functions

$\mathcal{H}$ from U to V is said to be k-universal if, for any distinct elements $x_1, x_2, \dots, x_k$ and for a hash function h chosen uniformly at random from $\mathcal{H}$, we have $\Pr_{h \in \mathcal{H}}(h(x_1) = h(x_2) = \dots = h(x_k)) \le \frac{1}{n^{k-1}}$.

It is said to be strongly k-universal if for any values $y_1, y_2, \dots, y_k \in \{0, 1, \dots, n-1\}$ and a random hash function h from $\mathcal{H}$, we have $\Pr_{h \in \mathcal{H}}((h(x_1) = y_1) \cap (h(x_2) = y_2) \cap \dots \cap (h(x_k) = y_k)) = \frac{1}{n^k}$ for distinct x_i.

We will focus on 2-universal and strongly 2-universal families of hash functions.

2-universal and strongly 2-universal family of hash functions.

Let $U = V = \{0, 1, \dots, p-1\}$ where p is a prime number.

Let $h_{a,b}(x) = (ax+b) \bmod p$.

Let $\mathcal{H} = \{h_{a,b} \mid 0 \le a, b \le p-1\}$.

Claim $\mathcal{H}$ is strongly 2-universal.

Proof We need to prove that $\Pr((h_{a,b}(x_1) = y_1) \cap (h_{a,b}(x_2) = y_2)) = \frac{1}{p^2}$ for $x_1 \ne x_2$ and any y_1, y_2.

Satisfying $h_{a,b}(x_1) = y_1$, and $h_{a,b}(x_2) = y_2$ means $(ax_1 + b) \bmod p = y_1$ and $(ax_2 + b) \bmod p = y_2$.

Given x_1, x_2, y_1, y_2, these are two linear equations with two variables, and there is a unique solution:

$a = (y_2 - y_1)(x_2 - x_1)^{-1} \bmod p$ and $b = (y_1 - ax_1) \bmod p$.

Hence there is only one choice of (a,b) out of p^2 possibilities satisfying these conditions, proving the claim. $\square$

The above construction only defines a strongly 2-universal hash family when $|U| = |V|$. In applications, we usually want to define a mapping from a large universe to a small range.

There is an easy way to extend the above construction to this setting.

Let $U = \{0, 1, 2, \dots, p^k - 1\}$ and $V = \{0, 1, \dots, p-1\}$ for some integer k and prime p.

Interpret each element $u \in U$ as a vector $\vec{u} = (u_0, u_1, \dots, u_{k-1})$ where $0 \le u_i \le p-1$ and $\sum_{i=0}^{k-1} u_i p^i = u$.

In other words, interpret u as a "p-ary" number.

For any vector $\vec{a} = (a_0, \dots, a_{k-1})$ with $0 \le a_i \le p-1$ for $0 \le i \le k-1$, and for any b with $0 \le b \le p-1$, let $h_{\vec{a},b}(\vec{u}) = \left(\sum_{i=0}^{k-1} a_i u_i + b\right) \bmod p$. Let $\mathcal{H} = \{h_{\vec{a},b} \mid 0 \le a_i, b \le p-1 \text{ for all } 0 \le i \le k-1\}$.

Claim: $\mathcal{H}$ is strongly 2-universal.

<u>Proof</u> : We need to prove $\Pr((h_{\vec{a},b}(\vec{u}_1)=y_1)\cap(h_{\vec{a},b}(\vec{u}_2)=y_2)) = \frac{1}{p^2}$.

Assume $u_{1,0} \neq u_{2,0}$. Then these conditions are equivalent to

$$a_0 u_{1,0} + b = \left(y_1 - \sum_{j=1}^{k-1} a_j u_{1,j}\right) \bmod p \quad \text{and} \quad a_0 u_{2,0} + b = \left(y_2 - \sum_{j=1}^{k-1} a_j u_{2,j}\right) \bmod p .$$

For given $a_1, \dots, a_{k-1}, \vec{u}_1, \vec{u}_2, y_1, y_2$, this is again two equations with two variables having a unique solution.

Hence, for every $a_1, \dots, a_{k-1}$, there is exactly one choice of (a_0, b) out of p^2 possibilities satisfying the conditions.

Therefore $\Pr((h_{\vec{a},b}(\vec{u}_1)=y_1)\cap(h_{\vec{a},b}(\vec{u}_2)=y_2)) = \frac{1}{p^2}$, proving the claim. □

Finally, the above construction can be "truncated" to any table size and is still 2-universal.

Let $h_{a,b}(x) = ((ax+b) \bmod p) \bmod n$ and $\mathcal{H} = \{h_{a,b} \mid 1 \le a \le p-1,\ 0 \le b \le p-1\}$.

<u>Claim</u> $\mathcal{H}$ is 2-universal.

<u>proof</u> Exercise. See Lemma 13.6 of MU. □

Note that there is a prime between m and $2m$ for any integer m by Bertrand's postulate (see wiki), and so the above family works for any m and n by choosing a prime $m \le p \le 2m$.

Also one can define hash families for other fields, e.g. fields with 2^k elements.

<u>k-universal family of hash functions</u>

Pick random $a_0, a_1, \dots, a_{k-1} \in \{0, 1, \dots, p-1\}$. Let $h_{\vec{a}}(x) = ((a_{k-1}x^{k-1} + a_{k-2}x^{k-2} + \dots + a_1 x + a_0) \bmod p) \bmod n$.

Then it can be shown that $\mathcal{H} = \{h_{\vec{a}} \mid 0 \le a_i \le p-1,\ 0 \le i \le p-1\}$ is a k-universal hash function, and in fact very close to a strongly k-universal hash function.

<u>**Hashing using 2-universal families**</u>

Using a hash family $\mathcal{H} = \{h_{a,b}(x) \mid 1 \le a \le p-1,\ 0 \le b \le p-1\}$ where $h_{a,b}(x) = ((ax+b) \bmod p) \bmod n$, we only need to choose a, b to select a hash function. So, we can store this hash function using only $O(1)$ cells (recall that each cell can store $\log m$ bits). Also, the hash value can be evaluated very quickly, using only $O(1)$ operations on $\log(m)$-bit words.

So, these hash functions satisfy the small space requirement and also the fast evaluation requirement.

Can they give the same guarantees as random hash functions?

The expected search time of chain hashing can still be guaranteed.

Lemma Assume m elements in S are hashed into an n-bin hashing table by using a random hash function from a 2-universal family. For an arbitrary element x, let X be the number of items at bin $h(x)$.

Then $$E[X] \le \begin{cases} m/n & \text{if } x \notin S \\ 1 + (m-1)/n & \text{if } x \in S \end{cases}$$

Proof Let $X_i = 1$ if the i-th element of S is in the same bin as x and 0 otherwise.

Since the hash function is chosen randomly from a 2-universal family, it follows that $\Pr(X_i = 1) = 1/n$.

Therefore $E[X] = \sum_{i=1}^{m} E[X_i] = m/n$ if $x \notin S$ and $1 + (m-1)/n$ if $x \in S$. □

However we can not guarantee that the maximum load bin has $O(\log n / \log\log n)$ balls.

Let $X_{ij} = 1$ if items x_i and x_j are mapped to the same location.

Let $X = \sum_{1 \le i \ne j \le m} X_{ij}$ be the number of collisions between pairs of items.

Then $E[X] = \sum_{i,j} E[X_{ij}] = \sum_{i,j} \Pr(h(x_i) = h(x_j)) \underset{\substack{\uparrow \\ \text{by 2-universality}}}{\le} \sum_{i,j} \frac{1}{n} < \frac{m^2}{2n}$

By Markov's inequality $\Pr(X \ge \frac{m^2}{n}) \le \frac{1}{2}$.

Suppose that the maximum load is Y. Then with prob $\ge \frac{1}{2}$, we have $\binom{Y}{2} \le \frac{m^2}{n}$, which implies that $Y \le m\sqrt{2/n}$. When $m = n$, the maximum load is at most $\sqrt{2n}$ with prob $\ge \frac{1}{2}$.

Remark: To guarantee maximum load $O(\log n / \log\log n)$, it is enough to use a $O(\log n / \log\log n)$-universal hash family (why?), but notice that the evaluation time would also go up to $O(\log n / \log\log n)$ word operations, which may not be a good tradeoff.

Perfect Hashing

Given a fixed set S, we want to build a data structure to support only search operations with excellent worst case guarantee. Let $m = |S|$.

This problem is useful, for example in building a static dictionary.

A hash function is <u>perfect</u> if it takes a constant number of operations (on $(\log m)$-bit words) to find an item or determine that it doesn't exist.

First, convince yourself that it is not a trivial problem, e.g. defining a "sorted" function won't work because you can't evaluate the function quick enough.

Perfect hashing is easy if there is sufficient space.

<u>Lemma</u> If $h \in \mathcal{H}$ is a random hash function from a 2-universal family mapping the universe U into $[0, n-1]$, then, for any set S of size m when $m \leq \sqrt{n}$, the probability of h being perfect for S is at least $1/2$.

<u>Proof</u> The proof is similar to the analysis for maximum load.

Using the above notation and calculation, the expected number of collisions is less than $m^2/2n$.

By Markov's inequality, this implies that $\Pr(X \geq \frac{m^2}{n}) \leq \frac{1}{2}$.

When $n \geq m^2$, this means that this is perfect (no collisions) with probability at least $1/2$. $\square$

To find a perfect hash function, we can generate random hash functions from this family and check whether it is perfect. This is a Las-Vegas algorithm, on average we only need to check at most two hash functions.

However, this scheme requires $\Omega(m^2)$ bins.

Using a two-level hashing scheme, we can design a perfect hashing scheme with only $O(m)$ bins.

First, we use a hash function to map the elements into a table of m bins.

As we have seen, the maximum load could be $O(\sqrt{m})$.

The idea is to build a second hash table for the elements mapped to each bin.

If a bin has k items, by the above result the second hash table only needs $O(k^2)$ bins.

Combining these carefully will give a perfect hash function with only $O(m)$ bins.

<u>Theorem</u> The two-level approach gives a perfect hashing scheme for m items using $O(m)$ bins.

<u>Proof</u> As shown above, the number of collisions X is at most m^2/n with probability at least $1/2$.

So for $m=n$, the number of collisions is at most m with probability $1/2$.

This first level hash function can be found by trying and checking random hash functions from the family. And on average we only need to check at most two functions so that #collisions $\leq m$.

Let c_i be the number of items in the i-th bin. Then $\sum_{i=1}^{m} \binom{c_i}{2} \leq m$.

We use a second hash function that gives no collision using c_i^2 space, for each bin with $c_i > 1$.

By the above lemma we can find such a function by trying at most 2 random hash functions on average.

The total number of bins used is at most $m + \sum_{i=1}^{m} c_i^2 = m + 2\sum_{i=1}^{m}\binom{c_i}{2} + \sum_{i=1}^{m} c_i \leq m + 2m + m = 4m$. $\square$

Note that the extra space required to store the hash functions is at most $O(m)$ cells, since there are at most $m+1$ hash functions, and each requires only $O(1)$ cells to store the pairs (a,b).

Also, it is clear that finding the location takes $O(1)$ operations, $O(1)$ for the first level and $O(1)$ for the second level.

This is essentially like building an array for the m elements, even though they come from a large universe.

<u>References and Further Reading</u>

Material in this lecture is from chapter 13 of "Probability and Computing" by Mitzenmacher and Upfal.

In practice, people don't use k-universal hash family for large k because of the computational overhead.

Instead, they just use some simple hash family and observe that it works well.

A particularly simple way is called tabulation hashing, which only involves table lookups and XORs.

It is not 4-wise independent, but can be proved to have nice properties of random functions, e.g. maximum load is $O(\log n / \log\log n)$.

k-wise independent variables are used for derandomization, to turn randomized algorithms into determinisitic, by doing brute force in the sample space of size $O(n^k)$.

For some applications, an even weaker notion called "almost k-wise independence" is enough, and it has smaller sample space of size say $O(2^k \cdot n)$.

This is an important tool in derandomizing "fixed parameter algorithms" (e.g. k-path in L03).