CS 466/666 Algorithm Design and Analysis . Spring 2020

Lecture 5: Hashing

Hashing is not only useful in designing efficient data structures 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 random 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 $\{0, 1, \ldots, M-1\}$.

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 $m >> n$.

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 <u>hash function</u> is used to map the elements of the big universe to the locations in the small table.

A <u>hash table</u> is a data structure that consists of the following components:

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

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 <u>collision</u> if $x \neq y$ but $h(x) = h(y)$.

Instead, what we can hope for is to have a <u>family</u> 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 our 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 <u>chain hashing</u>, 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 locations $h_1(x)$ 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 more than a factor of two.

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 $\text{poly}(m)$ 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 similar 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 time and requires little storage space.

For this, we introduce a weaker notion of randomness.

---

k-wise independence

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

k-wise independence is a weaker notion where it only requires the above conditions to hold for any subset of (up to) k variables, rather than all possible subsets of variables.

Definition 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, $i \in I$, it holds that $\Pr\left(\bigcap_{i\in I} (X_i = x_i)\right) = \prod_{i\in I} \Pr(X_i = x_i)$.

The special case when $k=2$ is called pairwise independence.

We see two examples why pairwise independent random variables are easier to be generated.

Example 1 Given b random bits $X_1, \ldots, X_b$, we can generate $2^b - 1$ pairwise independent bits as follows:

Enumerate the $2^b - 1$ non-empty 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 Y_S is random by the principle of deferred decisions.

(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 $x_1, \ldots, x_b$ as first deciding the value of other bits and then deciding the value of the i-th bit. By fixing all other bits, since $x_i = 1$ or 0 with probability $1/2$, 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. We would like to show that the two bits Y_{S_1} and Y_{S_2} are independent.

We assume without loss of generality that $S_1 - S_2 \neq \phi$. Let $x_i \in S_1 - S_2$.

By the principle of deferred decision on X_i, we can argue that $\Pr(Y_{S_1} = c \mid Y_{S_2} = d) = \frac{1}{2}$ for any $c, d \in \{0,1\}$.

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

which proves the pairwise independence of Y_S.

Are the variables Y_S 3-wise independent ?

Example 2 Given two independent uniformly random variables X_1, X_2 over $\{0,1,\ldots,p-1\}$, we can generate p pairwise independent random variables by setting $Y_i = (X_1 + iX_2) \bmod p$ for $i = 0,1,\ldots,p-1$, where p is a prime.

Again, by the principle of deferred decisions. Y_i is uniformly random, i.e. $\Pr(Y_i = a) = \frac{1}{p}$ for any $0 \le a \le p-1$.

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

The event $Y_i = a$ and $Y_j = b$ is equivalent to $(X_1 + iX_2) \bmod p = a$ and $(X_1 + jX_2) \bmod p = b$.

These two equations on two variables have a unique solution (because p is prime, and so multiplicative inverse exist)

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

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 $\Pr(Y_i = a \text{ and } Y_j = b) = \frac{1}{p^2}$. proving the pairwise independence.

Intuitively, this construction can be thought of as generating a random line over a field, i.e. if we only know one point Y_i of the line then another point Y_j is still random, but once we know two points of the line then we can figure out all other points of the line.

Note that in both examples, we used significantly fewer random bits to generate many random variables.

Chebyshev's inequality for Sum of pairwise independent random variables

We cannot apply Chernoff bounds for pairwise independent random variables.

An important remark is that Chebyshev's inequality still applies for pairwise independent random variables.

This is because for pairwise independent variables $X_1, X_2, \ldots, X_n$, we have $E[X_i X_j] = E[X_i]E[X_j]$ $\forall\, i \ne j$.

This implies that $\mathrm{Cov}(X_i, X_j) = 0$ $\forall\, i \ne j$, and so $\mathrm{Var}[\sum_{i=1}^{n} X_i] = \sum_{i=1}^{n} \mathrm{Var}[X_i]$ for pairwise independent variables.

Applying Chebyshev's inequality gives $\Pr(|X - E[X]| \ge a) \le \mathrm{Var}[X]/a^2 = \sum_{i=1}^{n} \mathrm{Var}[X_i]/a^2$ for $X = \sum_{i=1}^{n} X_i$.

This will be important in analysis of data streaming algorithms.

Universal Hash Functions

Definition Let U be a universe with $|U| \ge n$ and let $V = \{0,1,\ldots,n-1\}$. A family of hash functions $\mathcal{H}$ from U to V is said to be k-universal if, for any distinct elements $x_1, x_2, \ldots, x_k$ and a hash function h chosen uniformly random from $\mathcal{H}$, we have $\Pr_{h \in \mathcal{H}}(h(x_1) = h(x_2) = \ldots = h(x_k)) \le \frac{1}{n^{k-1}}$.

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

We can think of $\mathcal{H}$ is stronly k-universal if the random variables $h(0), h(1), \ldots, h(|U|-1)$ are k-wise independent, when h is chosen uniformly random from $\mathcal{H}$.

With this connection, it is not surprising that the constructions for generating k-wise independent random variables can be used to construct universal hash functions.

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

2-universal and strongly 2-universal families of hash functions

We begin with a construction where the size of the universe and the size of the table are the same.

Let $U = V = \{0, 1, \ldots, p-1\}$ 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 $\Pr_{a,b}\left((h_{a,b}(x_1)=y_1) \cap (h_{a,b}(x_2)=y_2)\right) = \frac{1}{p^2}$ for any y_1, y_2 and any $x_1 \neq x_2$.

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 the conditions, proving the claim. □

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 table.

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

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

Interpret each element $u \in U$ as a vector $\vec{u} = (u_0, \ldots, u_{k-1})$ 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 where u_0 is the least significant "digit" and u_{k-1} is the most significant "digit".

For any vector $\vec{a} = (a_0, \ldots, a_{k-1})$ where $0 \le a_i \le p-1$ for every $0 \le i \le k-1$, and any $0 \le b \le p-1$,

let $h_{\vec{a},b}(\vec{u}) = \left(\sum_{i=0}^{k-1} a_i u_i + b\right) \bmod p$.

Let $\mathcal{H} = \{ h_{\vec{a},b} \mid 0 \le a_i \le p-1 \text{ for every } 0 \le i \le k-1, \text{ and } 0 \le b \le p-1 \}$.

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

Proof: We need to prove that $\Pr_{h \in \mathcal{H}}\left(h_{\vec{a},b}(\vec{u}) = y \text{ and } h_{\vec{a},b}(\vec{w}) = z \right) = \frac{1}{p^2}$ for any y, z and $\vec{u} \neq \vec{w}$.

Assume without loss of generality that $u_0 \neq w_0$. Then these conditions are equivalent to

$$a_0 u_0 + b = \left(y - \sum_{j=1}^{k-1} a_j u_j \right) \bmod p \quad \text{and} \quad a_0 w_0 + b = \left(z - \sum_{j=1}^{k-1} b_j w_j \right) \bmod p.$$

Fixing $a_1, \ldots, a_{k-1}, \vec{u}, \vec{w}, y, z$, this is again two linear equations with two variables having a unique solution.

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

Therefore, $\Pr_{h \in \mathcal{H}}(h_{\vec{a},b}(\vec{u}) = y \text{ and } h_{\vec{a},b}(\vec{w}) = z) = 1/p^2$, 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 $\mathcal{H} = \{ h_{a,b} \mid 0 \leq a \leq p-1 \text{ and } 0 \leq b \leq p-1 \}$.

Claim $\mathcal{H}$ is 2-universal (but not strongly 2-universal).

Proof Exercise. See Lemma 13.6 of MU. □

Note that there is a prime number between m and $2m$ for any integer m by Bertrand's postulate (wiki).

and so the above family works for any m and n by choosing a prime p such that $m \leq p \leq 2m$.

Also, we can define hash families for other fields, e.g. fields with a power of two elements.

k-universal families of hash functions

The idea is similar. Instead of generating a random line (a degree one polynomial), we can generate a random degree k polynomial.

By polynomial interpolation, we know that given any k distinct points $x_1, x_2, \ldots, x_k$ and k values $y_1, y_2, \ldots, y_k$, there is a unique degree k polynomial p with $p(x_i) = y_i$ for $1 \leq i \leq k$.

This means that if we pick a random degree k polynomial, the outputs are k-wise independent.

More precisely, we can construct a k-universal hash family as follows.

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

Then it can be shown that $\mathcal{H} = \{ h_{\vec{a}} \mid 0 \leq a_i \leq p-1 \text{ for } 0 \leq i \leq k-1 \}$ is a k-universal hash family.

Hashing using 2-universal families

Let $|U| = m$ and $|V| = n$. Let p be a prime number such that $m \leq p \leq 2m$.

Using a hash family $\mathcal{H} = \{ h_{a,b}(x) \mid 0 \leq a, b \leq p-1 \}$ where $h_{a,b}(x) = \big((ax+b) \bmod p \big) \bmod n$,

we only need to choose a, b to select a hash function.

So, we can store this hash function using only 2 cells (recall that each cell can store $\log_2 m$ bits).

Also, the hash value can be evaluated very quickly, using only $O(1)$ operations on $\log_2(m)$-bit words.

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

Can they provide the same guarantees as random hash functions? Yes and no.

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

Lemma Assume m elements in S are hashed into an n-bin hash 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, $E[X] \le \begin{cases} \frac{m}{n} & \text{if } x \notin S \\ 1 + \frac{m-1}{n} & \text{if } x \in S \end{cases}$

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

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

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

The above statement is simple but it should be read carefully where we used the 2-universal property.

Consider the stupid hashing scheme where we choose a random bin and put all the balls there.

It is still true that the expected number of balls in a bin is $\frac{m}{n}$, but the search time would be m always.

Maximum load: However we cannot guarantee that the maximum loaded bin has $O(\log n / \log\log n)$ balls.

We can still use the 2-universal property to give a non-trivial bound.

Let $X_{ij} = 1$ if item i and item j are mapped to the same bin, and 0 otherwise.

Let $X = \sum_{i,j} X_{ij}$ be the number of collision pairs

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

$\frac{Y(Y-1)}{2} \le \frac{m^2}{n}$

By Markov's inequality, $\Pr(X \ge m^2/n) \le 1/2$, or equivalently $\Pr(X < m^2/n) \ge 1/2$.

Suppose the maximum load is Y. Then there are at least $\binom{Y}{2}$ collision pairs.

Therefore, with probability $\ge 1/2$, we have $\binom{Y}{2} < \frac{m^2}{n}$, which implies that $Y \le m\sqrt{\frac{2}{n}}$.

When $m = n$, this implies that the maximum load is $\sqrt{2n}$ with probability $1/2$.

Remark: To guarantee that the maximum load is $O(\log n / \log\log n)$ with high probability, we can use a $\Omega(\log n / \log\log n)$-universal hash family (why?).

But then the evaluation time is $\Omega(\log n / \log\log n)$, which is not a good tradeoff.

Perfect Hashing

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

This problem is useful in building a static dictionary.

A hash function is <u>perfect</u> if it takes a constant number of word operations (on $\log_2 m$-bit words) to find an item or determine that it doesn't exist.

First, convince yourself that it is not a trivial problem.

Next, we observe that 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 of 2-universal family.

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

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

When $n \geq m^2$, this means that this is perfect (i.e. no collision pairs) with probability at least $1/2$. □

To find a perfect hash function, we can generate random hash functions from this family and check whether it is perfect. On average we only need to check at most 2 hash functions.

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

The new idea is to use a two-level hashing scheme to do perfect hashing with only $O(m)$ bins.

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

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

The idea is to build a new/second-level hash table for each bin with multiple elements.

If a bin has k items, by the above lemma, the second-level hash table for that bin only needs $O(k^2)$ bins.

Combining these 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 collision pairs X in the first level is at most m^2/n with prob $\geq 1/2$.

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

The first level hash function can be found by trying and checking random hash function from a 2-universal family.

On average, we only need to check at most 2 functions to find a first level hash function with $\leq m$ collision pairs.

Let c_i be the number of items in the i-th bin.

Then, # collision pairs $= \sum_{i=1}^{m} \binom{c_i}{2} \leq m$.

We use a second-level hash function that gives no collisions 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 $m + \sum_{i=1}^{m} c_i^2 = m + 2\sum_{i=1}^{m}\binom{c_i}{2} + \sum_{i=1}^{m} c_i \le m + 2m + m = 4m$. □

Space : 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 (e.g. to store the pairs (a,b)).

Search time : Finding the location takes $O(1)$ operations, $O(1)$ for each level.

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

References and further reading (project ideas)

Material in this lecture is from chapter 13 of MU.

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 $O(\log n / \log\log n)$.

There are also other beautiful hashing techniques such as cuckoo hashing.

k-wise independent variables are used for derandomization, to turn randomized algorithms into deterministic, by doing brute force search 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 $O(2^k \cdot n)$.

This has become a standard tool in derandomizing "fixed parameter algorithms".