CSC 5450 Randomness and Computation

Week 9: Hashing

Plan ① Hash functions, bloom filters

② pairwise independence, universal hash functions, perfect hash functions

③ applications: heavy hitter, derandomization

Hash Functions [MU 5.5, MR 8.4]

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.

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 CUHK 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 $N=\{0,1,\dots,n-1\}$, 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>collison</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 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 in week 2.

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

So far so good, but we have ignored the issued of the storage and the evaluation time of a hash function.

Before we address these issues, let's first see some simple applications to get a better idea what we do with a hash function.

Approximate Set Membership (MU 5.5.2-5.5.3)

In the approximate set membership problem, we have a set $S = \{s_1, s_2, \ldots, s_m\}$ of m elements from a large universe U, and we would like to create a data structure using as little space as possible, and answer queries of the form "Is $x \in S$?".

We show how to use hash functions as a "fingerprinting" function to solve the problem, assuming that the hash functions behave like random functions and use little space (unrealistic assumptions)

Also we accept a small probability of having a "false positive", i.e. return yes even the element is not in S.

Suppose we use b bits to represent each member, and store them in a sorted list.

The probability that an element not in S having the same fingerprint as some element in S, i.e. the probability of a false positive is $1 - (1 - \frac{1}{2^b})^m \approx 1 - e^{-m/2^b}$.

For this to be at most ε, we need to set $b \geq \log_2 \frac{m}{\ln(1/(1-\varepsilon))}$. So we need $\Omega(\log m)$ bits.

On the other hand, if $b = 2\log_2 m$, then the false positive probability is $1-\left(1-\frac{1}{m^2}\right)^m < \frac{1}{m}$.

To conclude, we can "compress" S using $O(m \log m)$ bits with a small false positive error.

Bloom Filters

Bloom filter is a data structure for approximate set membership with more interesting tradeoffs.

A bloom filter consits of an array of n bits, $A[0]$ to $A[n-1]$, initially all set to 0.

For each element $s \in S$, it uses k independent hash functions $h_1, h_2, \ldots, h_k$, and set the bits $A[h_1(s)], A[h_2(s)], \ldots, A[h_k(s)]$ to one.

To check whether an element $x \in S$, we check whether $A[h_1(x)], A[h_2(x)], \ldots, A[h_k(x)]$ are all equal to one. If yes, say $x \in S$; if $A[h_i(x)] = 0$ for some i, say $x \notin S$.

Obviously when it says no it is always correct. We just need to bound the false positive probability.

First we hash all the elements in S into the bloom filter.

Let p be the probability that a specific bit is 0. Let m be the number of keys to store.

Then $p = \left(1-\frac{1}{n}\right)^{km} \approx e^{-km/n}$. By using Poisson approximation and Chernoff's bound for Poisson variables (similar to what we did in week 2), we can prove that with high probability the number of 0s (i.e. number of empty bins) is very close to np.

The probability of a false positive is $f = (1-p)^k$.

Suppose we are given m and n and want to optimize k so as to minimize f. On one hand, having k larger is more likely to hit a 0. On the other hand, having small k will have more 0s.

Let $g = \ln f = k \ln(1-e^{-km/n})$. Take $\frac{dg}{dk} = \ln(1-e^{-km/n}) + \frac{km}{n}\left(\frac{e^{-km/n}}{1-e^{-km/n}}\right)$.

It can be checked that the derivative is zero when $k = (\ln 2)\cdot(n/m)$ and this is the global minimum.

The false positive probability f is $(1/2)^k \approx (0.6185)^{n/m}$.

Choosing $n \approx O(m)$ will give us a data structure using $O(m)$ bits with small constant error probability. This may be preferable to the previous scheme which requires $\Omega(\log m)$ bits, especially when false positive is acceptable (e.g. checking whether a password is easy).

(constant bits per element)

Random Hash Functions ?

In the previous applications, we assume the hash functions behave like random functions, but at the same time use very little time to evaluate $h(x)$ (to support search operations in $O(1)$ time) and require very little space to store it (to support compression for approximate set membership).

But it is just impossible to do either one of these.

Consider a random function $h: M \to N$. To store this table it requires at least $m \log n$ bits, each element requires $\log n$ bits to remember its location. This is even more than storing 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, which implies that 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 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 (MU 13.1)

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

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

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

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

Example 1 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 (page 9 of MU).

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 and d.

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 $\{0,1,\ldots,p-1\}$, 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 deferred decisions, Y_i is uniformly random.

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

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

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

$X_2 = (b-a)(j-i)^{-1} \bmod p$ 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 $\Pr(Y_i = a \wedge Y_j = b) = \frac{1}{p^2}$, 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 (MU 13.3)

Definition Let U be a universe with $|U| \geq n$ 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 for a hash function h chosen uniformly at random from $\mathcal{H}$, we have $\Pr(h(x_1) = h(x_2) = \ldots = h(x_k)) \leq \frac{1}{n^{k-1}}$.

It 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 $\Pr((h(x_1) = y_1) \wedge (h(x_2) = y_2) \wedge \ldots \wedge (h(x_k) = y_k)) = \frac{1}{n^k}$ 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 $h_{a,b}(x) = (ax + b) \bmod p$, where p is a prime number.

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

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

Proof We need to prove that $\Pr((h_{a,b}(x_1) = y_1) \wedge (h_{a,b}(x_2) = y_2)) = \frac{1}{p^2}$ for $x_1 \neq x_2$ 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:

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

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

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 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

$$h_{\vec{a},b}(\vec{u}) = \left(\sum_{i=0}^{k-1} a_i u_i + b \right) \bmod p. \quad \text{Let } \mathcal{H} = \{ h_{\vec{a},b} \mid 0 \le a_i, b \le p-1 \text{ for all } 0 \le i \le k-1 \}.$$

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

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

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 only one choice of (a_0,b) out of p^2 possibilities satisfying the conditions.

Therefore $\Pr\left((h_{\vec{a},b}(\vec{u}_1)=y_1) \cap (h_{\vec{a},b}(\vec{u}_2)=y_2) \right) = \frac{1}{p^2}$, 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 $\mathcal{H} = \{ h_{a,b} \mid 1 \le a \le p-1, 0 \le b \le p-1 \}$.

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

Proof 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 also define hash families for other fields, e.g. fields with 2^k elements, see MU.

k-universal family of hash functions

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

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.

Hashing using 2-universal families

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

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

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 $\ge \frac{1}{2}$.

Perfect Hashing (MU 13.3.3)

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

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.

Lemma 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 $m \le \sqrt{n}$, the probability of h being perfect for S is at least $1/2$.

Proof 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 $\Pr(X \ge \frac{m^2}{n}) \le \frac{1}{2}$.

When $n \ge m^2$, this means that this is perfect (no collisions) 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. This is a Las-Vegas algorithm, on average we only need to check at most two hash functions.

However, this schema 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 use 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.

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

Proof 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.

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

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

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 $m + \sum_{i=1}^{m} c_i^2 = m + 2\sum_{i=1}^{m} \binom{c_i}{2} + \sum_{i=1}^{m} c_i \le m + 2m + m = 4m$. □

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.

Heavy Hitters (MU 13.4)

We show an application of hash functions in designing sublinear space algorithms.

The idea used is similar to that used in bloom filters.

Given a data stream $X_1, X_2, X_3, \ldots, X_T$, each $X_t = (i_t, c_t)$ where i_t is the ID of the t-th data and c_t is the weight associated with it, e.g. i_t is the IP address and c_t is the number of bytes of the data transmitted.

Our goal is to find all the <u>heavy hitters</u> of the data stream. Let $Q = \sum_t c_t$ be the total count.

Given a number f, we say an ID i is a heavy hitter if $\sum_{t: i_t = i} c_t \ge f$. (e.g. $f \ge 0.01Q$)

Let $\text{Count}(i,T) = \sum_{t=1, i_t=i}^{T} c_i$ be the total count for ID i.

We will show a sublinear space algorithm with the following properties:

1. All heavy hitters are reported, i.e. those i with $\text{Count}(i,T) \ge f$,
2. If $\text{Count}(i,T) \le f - \varepsilon Q$, then i is reported with probability at most δ.

That is, report all heavy hitters with probability one, while having small false positive error.

The idea is to use 2-universal hash functions, similar to the construction of bloom filters.

We will use k hash functions, $h_1, h_2, \ldots, h_k$, each maps the universe into $\{1, 2, \ldots, \ell\}$.

We maintain $k \cdot \ell$ counters, each counter $C_{a,j}$ adds the weight of the items that are mapped to the j-th entry by the a-th hash function. Initially $C_{a,j} = 0$ for $1 \le a \le k$, $1 \le j \le \ell$.

The algorithm is simple. Given $X_t = (i_t, c_t)$, for each $1 \le a \le k$, increment $C_{a, h_a(i_t)}$ by c_t.

The algorithm is simple. Given $X_t=(i_t,c_t)$, for each $1\le a\le k$, increment $C_{a,h_a(i_t)}$ by c_t.

After we read all the data, we report all i with $\min_{j=h_a(i),\,1\le a\le k} C_{a,j} \ge f$. In words, we report an ID if the minimum counter associated with it is at least f.

It should be clear that a heavy hitter is always reported, since by our algorithm all the counters associated with it must be at least f.

The key is to show that if an ID is not a heavy hitter, then it is reported with small probability.

Let us consider a specific ID i. Suppose $\text{Count}(i,T) \le f-\varepsilon Q$. What is the prob that i is reported?

Consider $C_{a,h_a(i)}$ for a specific a. Since i is reported, we must have $C_{a,H_a(i)} \ge f$.

Let Z_a be the value of this counter that is incremented by other IDs.

Since h_a is chosen from a 2-universal family, another item is mapped to $H_a(i)$ with probability $\le \frac{1}{\ell}$.

Therefore $E[Z_a] \le Q/\ell$.

By Markov's inequality, $\Pr(Z_a \ge \varepsilon Q) \le \frac{1}{\varepsilon\ell}$

So $\Pr(\min Z_a \ge \varepsilon Q) \le (\frac{1}{\varepsilon\ell})^k$

Set $\ell=\frac{e}{\varepsilon}$ and $k=\ln(\frac{1}{\delta})$, this probability is at most δ.

So, we just need to maintain $O(\frac{1}{\varepsilon}\ln\frac{1}{\delta})$ counters for this task, which is just a constant for constant ε and δ.

The hash functions can also be stored and computed efficiently.

We will discuss more data streaming algorithms next week.

Derandomization

k-wise independent distributions are a standard tool for derandomization.

Definition For $k\in\mathbb{N}$, a family of functions $\mathcal{H}=\{h:[N]\to[M]\}$ is k-wise independent if for all distinct $x_1,x_2,\dots,x_k\in[N]$, the random variables $h(x_1),h(x_2),\dots,h(x_k)$ are independent and uniformly distributed in $[M]$ if h is a random function in $\mathcal{H}$.

Construction Let $\mathbb{F}$ be a finite field. Define the family of functions $\mathcal{H}=\{h_{a_0,a_1,\dots,a_{k-1}}:\mathbb{F}\to\mathbb{F}\}$, where each $h_{a_0,a_1,\dots,a_{k-1}}(x)=a_0+a_1x+a_2x^2+\dots+a_{k-1}x^{k-1}$ for $a_0,\dots,a_{k-1}\in\mathbb{F}$.

Claim The family $\mathcal{H}$ given in the above construction is k-wise independent.

Proof We will prove that given any subset of k distinct elements $x_1, x_2, \ldots, x_k \in \mathbb{F}$ and for any subset of k (not necessarily) distinct elements $y_1, y_2, \ldots, y_k \in \mathbb{F}$, there is exactly one polynomial h of degree at most $k-1$ such that $h(x_i) = y_i$ for $1 \le i \le k$. This would imply that knowing any subset of less than k variables, the remaining variables can still be of any values with equal probabilities.

Such a polynomial exists by using the Lagrange interpolation formula, $h(x) = \sum_{i=1}^{k} y_i \cdot \prod_{j \neq i} \frac{x - x_j}{x_i - x_j}$.

This is unique, because the difference between two polynomials h and g such that $h(x_i) = g(x_i)$ for $1 \le i \le k$ must be the zero polynomial (as $h-g$ is a degree $k-1$ polynomial with k roots). □

k-wise independent bits: Suppose we want to generate n bits that are k-wise independent. Let $\mathbb{F}$ be a finite field with $2^l \ge n$ elements. Pick k random elements $a_0, a_1, \ldots, a_{k-1}$ from this field. Let h be the function $h_{a_0, a_1, \ldots, a_{k-1}}$. Compute $h(x)$ for all x in the finite field. Then we obtain 2^l random variables that are k-wise independent. Since $\mathbb{F}$ is a finite field of size 2^l, the last bits in these variables are k-wise independent. Note that we only need $O(k \log n)$ bits (for $a_0, \ldots, a_{k-1}$) to generate n bits that are k-wise independent.

It is an easy exercise of the probabilistic method to prove that there is a 2-coloring of the edges of the complete graph with n vertices, so that the total number of monochromatic copies of K_4 (complete subgraph with 4 vertices) is at most $\binom{n}{4} 2^{-5}$.

Can we find such a coloring in deterministic polynomial time?

To derandomize such a statement, the observation is that in order for the expected value to be $\binom{n}{6} 2^{-5}$, we just need the n bits to be 6-wise independent since each K_4 only has 6 edges.

That is, if we take the expectation over a set of n-bit strings, where the n bits in each string are 6-wise independent, then the expected value is still $\binom{n}{6} 2^{-5}$.

By the above construction, there exists such a set with $2^{O(k \log n)} = n^{O(k)}$ polynomially many such strings.

We know that there is a good string in this set, and we can find one by exhausive search, which still runs in polynomial time.

To summarize, this method is to reduce a large sample space to a small sample space.

Almost k-wise independence

The sample space of k-wise independent is $\Theta(n^k)$ (there is a matching lower bound).

There is a much more effective construction if we only require almost k-wise independence: A Sample

space S of n bits is almost k-wise independent if when $X = x_1 \dots x_n$ is chosen uniformly from S_n, then $| \Pr(x_1 = c_1, x_2 = c_2, \dots, x_k = c_k) - 2^{-k} | \leq \varepsilon$.

There exists a sample space of size $O\left(\frac{k^2 \log^2 n}{\varepsilon^2}\right)$ for n almost k-wise independent random variables with ε-error. See [1]. This is very useful in obtaining efficient deterministic algorithms from randomized algorithms, e.g. an important tool in fixed parameter algorithms.

Remark

While k-wise independence can guarantee some nice properties of the hash functions, there exist some constructions of hash functions which are not even 4-wise independent, but still enjoy some nice properties (e.g. maximum load $O(\log n / \log \log n)$, using two hash functions with maximum load $O(\ln \ln n)$, etc). A particularly simple construction can be found in [2].

References

[1] Alon, Goldreich, Håstad, Peralta. Simple constructions of almost k-wise independent random variables.

[2] Patrascu, Thorup. The power of simple tabulated hashing.