

Lecture 27 Perfect Hashing, Bloom Filters: 05.03.05

*Lecturer: Christos**Scribe: Jeremy Rahe*

Disclaimer: *These notes have not been subjected to the usual scrutiny reserved for formal publications. They may be distributed outside this class only with the permission of the Instructor.*

27.1 Hashing

A hash function is a partition function that maps a large domain of key values into a much smaller domain of hash values. For example, a hash function $h : U \rightarrow \{0, 1, \dots, n - 1\}$ might map the universal set U of all English words of 20 letters or less into the set of 2-byte integers. Here the domain has size 26^{20} while the codomain has size 2^{16} .

A hash function is typically intended to be used as a cheap approximation for equality-testing. Suppose you wanted to implement a spell-checker. Instead of storing the full table of words present in the dictionary, you could instead apply the hash function to each word in the dictionary and store just the table of hash values of keys. Then, to check if a string is in the dictionary, you compute the hash value for that string and see if the value is present in the table of hash values that represents the dictionary. While a match of hash values doesn't guarantee that the string is a word in the dictionary, if a good hash function is chosen, then the probability of error is quite small.

If you don't want to sacrifice accuracy, hashing can still be used to speed up searching. So, instead of storing just the hash values generated by the dictionary, you could store the entire dictionary of words, indexed by hash value. Then, checking for a word in the dictionary becomes a matter of going through a (small) list of words with the same hash value to see if there is a match.

Definition 1 *Hash Table:* A hash table is an array of size equal to the size of the codomain of the hash function (i.e., it has a location for each hash value). Each location holds a pointer to a list of elements in the domain that map to the corresponding hash value.

It should be noted generally the number of values from the domain that are inserted into the hash table is small relative to the size of the domain being typically comparable to or smaller than the size of the codomain. In the example of the English-word dictionary, our hash table has 65,000 different values and our English dictionary might only have 50,000-100,000 words from the possible 26^{20} total 20-character combinations. Thus, assuming equal distribution of the words over the hash table values, there will generally be only one or two words with that hash value. This leads to the result that hash table lookups are generally considered to take constant time since the vast majority of lookups can be handled with just one comparison.

One can analyze hashing like Balls and Bins. Each operation takes place in time $O(\text{max balls in bin})$. Similarly, using two hash functions to select two possible hash values and placing the value in the less crowded bin results in savings owing to the "Power of Two Choices."

Obtaining equal distribution of words over hash table values is not trivial. For example, one could take the first four characters of a word and use that as a hashing function. This will result in equal distribution of the

domain over the codomain. But it is a bad choice for hashing English words, since the first four-characters of words are not evenly distributed across all words.

One could try mapping each new word to a random number, but then you have to keep track of the numbers somehow, so this isn't much help.

Or one could choose a *random* function from the domain to the codomain. This has the desirable result that the probability that two words map to the same value is $1/n$, where n is the size of the codomain; and we don't need to know what actual values from the domain will occur. However, we cannot encode the random function that we choose in any finite amount of space, so this isn't a viable option either.

A third option, known as Universal Hash Functions, is a variant of the previous option that doesn't take infinite space.

For any fixed hash function, if the keys are chosen in such a way that all r keys hash to the same slot, the average retrieval time can grow to $O(r)$. By using a random choice of function from a carefully selected class, where the choice is independent of key, one can achieve good performance on average, regardless of the choice of keys.

Let \mathcal{H} be a finite collection of hash functions that map a given universe into $\{0, 1, \dots, n - 1\}$.

Definition 2 (Universal Hash Function) *A collection \mathcal{H} is said to be universal if for each pair of distinct keys, $x, y \in U$, the number of hash functions $h \in \mathcal{H}$ for which $h(x) = h(y)$ is precisely $|\mathcal{H}|/n$.*

The following theorem applies:

Theorem 1 *If h is chosen from a universal collection of hash functions and is used to hash r keys into a table of size n , where $r \leq n$, the expected number of collisions involving a particular key x is less than 1.*

Proof:

For each pair, y, z of distinct keys, let c_{yz} be a random variable (the indicator function) that is 1 if $h(y) = h(z)$, i.e., if y and z collide using h , and 0 otherwise. Since, by definition, a single pair of keys collides with probability $1/n$, we have

$$E[c_{yz}] = 1/n.$$

Let C_x be the total number of collisions involving key x in a hash table T of size m containing r keys. It follows that

$$E[C_x] = \sum_{y \in T, y \neq x} E[c_{xy}] = \frac{r-1}{n}.$$

Since $r \leq n$, we have $E[C_x] < 1$.

To construct a universal class of hash functions, choose the table size n to be prime, since the integers mod n form a field. Decompose a key x into $s + 1$ bytes. so that $x = \langle x_0, x_1, \dots, x_s \rangle$. This is possible provided only that the maximum value stored in a byte is less than n . Let $a = \langle a_0, a_1, \dots, a_s \rangle$ denote a sequence of $s + 1$ elements chosen randomly from the set $\{0, 1, \dots, n - 1\}$. For each such a , define a corresponding hash function $h_a \in \mathcal{H}$:

$$h_a(x) = \sum_{i=0}^s a_i x_i \text{ mod } m.$$

With this definition,

$\mathcal{H} = \cup_a \{h_a\}$ has n^{s+1} members.

Theorem 2 *The class \mathcal{H} defined above is a universal class of hash functions*

Proof:

For any pair of distinct keys x, y , with $x_0 \neq y_0$, and for any fixed values of a_1, a_2, \dots, a_s , there is exactly one value of a_0 that satisfies the equation $h(x) = h(y)$. This a_0 is the solution to the equation

$$a_0(x_0 - y_0) \equiv -\sum_{i=1}^s a_i(x_i - y_i) \pmod{n}.$$

Note that since m is prime, the nonzero quantity $x_0 - y_0$ has a multiplicative inverse modulo n , and there is a unique solution for a_0 modulo m . Therefore, each pair of keys x and y collides for exactly n^s values of a , since they collide exactly once for each possible value of $\langle a_1, a_2, \dots, a_s \rangle$. Since there are n^{s+1} possible values for the sequence a , keys x and y collide with probability exactly $n^s/n^{s+1} = 1/n$. Therefore \mathcal{H} is universal.

27.2 Bloom Filters

A normal hash table implemented as an array of pointers to, say, linked lists allows inserting and deleting an element by inserting or deleting from the linked list of the hash value associated with the element.

Suppose you removed the requirement that you can delete elements from the hash table. Can you take advantage of relaxing this requirement to get something else in return?

A Bloom Filter, named after its discoverer, Burton H. Bloom, is such an object. It is a space-efficient probabilistic data structure that is used to test whether or not an element is a member of a set. False positives are possible, but false negatives are not. Elements can be added to the set, but not removed. The more elements that are added to the set, the larger the probability of false positives.

A Bloom Filter consists of a k element bitvector that represents a kind of hash table.

We have two operations using the m hash functions:

insert(a), with $T(h_i(a)) = 1, i = 1, \dots, m$, and

lookup(a), test if $T(h_i(a)) = 1, i = 1, \dots, m$.

The insert operation on $x \in S$ computes the values of m hash functions, h_i , on x . We get a positive result to our lookup iff all bits in the bitvector are 1.

In other words, an empty Bloom filter is a bit array of k bits, all set to 0. There are also m different hash functions defined, each of which maps an element to one of the k array positions. Each element of the domain is mapped to a set of m bits in the bitvector. When an element is inserted into the Bloom filter, each of those bits is set to 1. Checking for the presence of an element in the Bloom filter then requires finding which m bits the element corresponds to, and checking to see if all of them are set to 1.

Bloom filters have been used for spell-checking in a space-efficient way. A Bloom filter trained with a list of all correct words will accept all correct words and reject almost all incorrect words. They also are used in semi-joins for databases.

Bloom filters have the property that the time needed to either add items or to check whether an item is in the set is a fixed constant, $O(m)$, completely independent of the number of items already in the set. In a hardware implementation, the Bloom filter's m lookups are independent and can be parallelized.

For optimal performance, ideally a Bloom filter should have half of its bits be zero.

Instead of using single bits to represent presence or absence, one can use saturating counters. The Broder and Mitzenmacher paper shows that 4-bit counters are sufficient for most applications.

See the paper by Broder and Mitzenmacher:

Network Applications of Bloom Filters: A Survey. A. Broder and M. Mitzenmacher. Allerton Conference 2002.

(<http://www.eecs.harvard.edu/~michaelm/NETWORK/postscripts/BloomFilterSurvey.pdf>)

Bloom filters are very efficient. They have the property that, if we allow 1% false positive error, an optimal value of m requires only about 9.6 bits per element, regardless of the size of the elements. If one desires a lower false positive rate than 1%, increasing by about 4.8 bits per element will decrease the false positive rate by a factor of 10. This follows from the analysis below.

27.2.1 Probability of false positives

We can analyze the probability of a false positive. Assume that a hash function selects each array position with equal probability. The probability that a certain bit is not set to one by a certain hash function during the insertion of an element is then $1 - \frac{1}{k}$.

The probability that it is not set by any of the hash functions is $(1 - \frac{1}{k})^m$.

If we have inserted n elements, the probability that a certain bit is still 0 is $(1 - \frac{1}{k})^{mn}$.

Therefore, the probability that it is 1 is $1 - (1 - \frac{1}{k})^{mn}$.

To test to see if an element that is not in the set, each of the m array positions computed by the hash functions is 1 with a probability as above. Assuming approximate independence, the probability of all of them being 1, which would cause the algorithm to indicate in error that the element is in the set, is then $(1 - (1 - \frac{1}{k})^{mn})^m \approx (1 - e^{-mn/k})^m$.

The probability of false positives decreases as k (the number of bits in the array) increases, and increases as n (the number of inserted elements) increases.

We can find the value of m that minimizes the expression. Let $p \doteq e^{-mn/k}$. Then we can solve to get $m = -k/n \ln p$. Hence, $(1 - p)^m = e^{m \ln(1-p)} = e^{-k/n \ln(1-p) \ln p}$, which is maximized by taking $p = 1/2$.

For given k and n , the number, m , of hash functions that will minimize the probability is $\frac{k}{n} \ln 2 \approx \frac{9k}{13n} \approx 0.7 \frac{k}{n}$, equivalent to a probability of $(\frac{1}{2^{\ln 2}})^{k/n} \approx 0.62^{k/n}$.

27.2.2 Cheap Tricks with Bloom Filters

Denote by $B(S_i), i = 1, 2$ the Bloom filter of some subsets S_i in the domain. If we want a Bloom filter $B(S_1 \cup S_2)$, we just take the bitwise **OR** of the two filters.

Given a Bloom filter, if we want a version that uses half as many bits, we take the first half and **OR** it with the second half of the bitvector.