Lecture Notes on Hashing, and Bloom Filters

Mehrnoosh Sameki *December 3, 2013*

1 Hash Tables

1.1 Introduction

A hash table is a data structure for storing a set of items, so that we can quickly determine whether an item is or is not in the set. The basic idea is to pick a hash function h that maps every possible item x to a small integer $h(x)$. Then we store x in slot $h(x)$ in an array. The array is the hash table.

Lets be a little more specific. We want to store a set of n items. Each item is an element of some finite set U called the universe; we use u to denote the size of the universe, which is just the number of items in U. A hash table is an array $T[1..m]$, where m is another positive integer, which we call the table size. Typically, m is much smaller than u. A hash function is any function mapping each possible item in U to a slot in the hash table. We say that an item x hashes to the slot $T[h(x)]$.

Of course, if $u = m$, then we can always just use the trivial hash function $h(x) = x$. In other words, use the item itself as the index into the table. This is called a direct access table, or more commonly, an array. In most applications, though, the universe of possible keys is orders of magnitude too large for this approach to be practical. Even when it is possible to allocate enough memory, we usually need to store only a small fraction of the universe. Rather than wasting lots of space, we should make m roughly equal to n, the number of items in the set we want to maintain. What we would like is for every item in our set to hash to a different position in the array. Unfortunately, unless $m = u$, this is too much to hope for, so we have to deal with collisions. We say that two items x and y collide if the have the same hash value: $h(x) = h(y)$. Since we obviously can not store two items in the same slot of an array, we need to describe some methods for resolving collisions. The two most common methods are called chaining and open addressing.

1.2 Chaining

In a chained hash table, each entry $T[i]$ is not just a single item, but rather (a pointer to) a linked list of all the items that hash to $T[i]$. Let $l(x)$ denote the length of the list $T[h(x)]$. To see if an item x is in the hash table, we scan the entire list $T[h(x)]$. The worst-case time required to search for x is $O(1)$ to compute $h(x)$ plus $O(1)$ for every element in $T[h(x)]$, or $O(1 + l(x))$ overall. Inserting and deleting x also take $O(1 + l(x))$ time.

In the worst case, every item would be hashed to the same value, so we would get just one long list of n items. In principle, for any deterministic hashing scheme, a malicious adversary can always present a set of items with exactly this property. In order to defeat such malicious behavior, we would like to use a hash function that is as random as possible. Choosing a truly random hash function is completely impractical, but there are several heuristics for producing hash functions that behave randomly, or at least close to randomly on real data. Thus, we will analyze the performance as though our hash function were truly random

1.2.1 Analysis of Collision Resolution by Chaining

- Theorem H1: In a hash table in which collisions are resolved by chaining, an unsuccessful search takes expected time $\theta(1 + \alpha)$, under the assumption of simple uniform hashing.
- Theorem H2: In a hash table in which collisions are resolved by chaining, a successful search takes time $\theta(1 + \alpha)$, on the average, under the assumption of simple uniform hashing.

1.3 Open Addressing

Another method used to resolve collisions in hash tables is called open addressing. Here, rather than building secondary data structures, we resolve collisions by looking elsewhere in the table. Specifically, we have a sequence of hash functions $< h_0, h_1, h_2, ..., h_{m-1} >$, such that for any item x, the $probesequence$ $< h_0(x), h_1(x), h_2(x), ..., h_{m-1}(x) >$ is a permutation of $< 0, 1, 2, ..., m-1 >$. In other words, different hash functions in the sequence always map x to different locations in the hash table. We search for x using the following algorithm, which returns the array index i if $T[i] = x$, "absent" if x is not in the table but there is an empty slot, and "full" if x is not in the table and there no no empty slots.

Algorithm 1 Open Address Search(x)

```
for $i \leftarrow 0tom - 1$ do
  if $T[h_i(x)] = x$ then
    $returnh_i(x)$
  end if
  if $T[h_i(x)] = null$ then
    $returnabsent$
  end if
end for
$returnfull$
```

The algorithm for inserting a new item into the table is similar; only the second-to-last line is changed to $T[h_i(x)] \leftarrow x$. Notice that for an open-addressed hash table, the load factor is never bigger than 1. Just as with chaining, we would like to pretend that the sequence of hash values is random. For purposes of analysis, there is a stronger uniform hashing assumption that gives us constant expected search and insertion time.

Strong uniform hashing assumption: For any item x, the probe sequence $< h_0, h_1, h_2, ..., h_{m-1} >$ is equally likely to be any permutation of the set $< 0, 1, 2, ..., m-1 >$.

1.3.1 Collision Resolution by Open Addressing

- Theorem H3: Given an open-address hash table with load factor $\alpha < 1$, the expected number of probes in an unsuccessful search is at most $1/(1-\alpha)$, assuming uniform hashing.
- Corollary H4: Inserting an element into an open-address hash table with load factor α requires at most $1/(1-\alpha)$ probes on average, assuming uniform hashing.
- Theorem H4: Given an open-address hash table with load factor $\alpha < 1$, the expected number of probes in a successful search is at most $(1/\alpha)ln(1/(1-\alpha))$ assuming uniform hashing and assuming that each key in the table is equally likely to be searched for

2 Bloom Filters

2.1 Definition

A Bloom filter is probabilistic data structure which acts as a set, and is a method for answering set membership. It basically contains a bit vector, v, of n bits with k independent hash functions, $h_1(x)$, $h_2(x)$, $\ldots$, $h_k(x)$. So,

$$\forall i,\ h_i : U \to \{0, \ldots, n-1\}$$

The required operations are implemented in the following manner:

- Initialization:
  1. Set all n bits of v to FALSE.
- Insertion of $x \in U$:
  1. Compute $h_1(x)$, $h_2(x)$, $\ldots$, $h_k(x)$.

2. Set $v[h_1(x)] = v[h_2(x)] = \ldots = v[h_k(x)] = TRUE$.

- Lookup of $x \in U$:

  1. Compute $h_1(x)$, $h_2(x)$, ..., $h_k(x)$.
     (a) If any of $v[h_i(x)]$ is false, then x is not in the filter.
     (b) Otherwise, report x in filter. Given the nature of compression, and query may return a positive result when no such x exists.

- Deletion of $x \in U$: There is no deletion in bloom filter!

2.2 Usage

Bloom filters are being used in a lot of problems:

- Web cache sharing: Each cache periodically broadcasts its summary to all other members of the distributed cache. Using all summaries received, a cache node has a (partially outdated, partially wrong) global image about the set of files stored in the aggregated cache. The Squid Web Proxy Cache uses "Cache Digests" based on a similar idea.

- Free text searching: Basically, the set of words that appear in a text is succinctly represented using a Bloom filter

- Query filtering and routing: The Secure wide-area Discovery Service, subsystem of Ninja project, organizes service providers in a hierarchy. Bloom filters are used as summaries for the set of services offered by a node. Summaries are sent upwards in the hierarchy and aggregated. A query is a description for a specific service, also represented as a Bloom filter. Thus, when a member node of the hierarchy generates/receives a query, it has enough information at hand to decide where to forward the query: downward, to one of its descendants (if a solution to the query is present in the filter for the corresponding node), or upward, toward its parent (otherwise).

2.3 Error Types

- False Negative: Answering element is "not there" on an element that is in set. Never happens!

- False Positive: Answering "is there" on an element that is in not in the set. Can happen!

2.4 Analysis

Bloom filters can report false positives (ie. report $a \in X$ when $a \notin X$). The probability of false positives,p, is called the *false positive rate* [4], and the key to understanding them is to analyse their relationship to n.

$$\boxed{\text{FACT: } \left(1 - \tfrac{1}{x}\right)^y \approx e^{-x/y}}$$

The probability that one hash fails to set a given a bit is $\left(1 - \frac{1}{n}\right)$. Using the above fact, p is

$$\begin{aligned} Pr[\text{bit is still zero}] &= \left(1 - \frac{1}{n}\right)^{\overbrace{km}^{no.\ of\ hashes}} \\ &\approx e^{-\frac{km}{n}} \end{aligned}$$

$$\boxed{\text{Note: This assumes that hash functions are independent and random.}}$$

If we let $p = e^{-km/n}$ then we can say that

$$\begin{aligned} Pr[\text{false positive}] &= Pr[\text{All k bits are 1}] & (1)\\ &= \underbrace{\left(1-\left(1-\frac{1}{n}\right)^{km}\right)^{k}}_{Prob\ one\ bit\ is\ 1} & (2)\\ &\approx \left(1-e^{-\frac{km}{n}}\right)^{k} & (3)\\ &= (1-p)^{k} & (4) \end{aligned}$$

Now minimize f, the false positive rate, by finding the optimal number of hash functions. That is, minimize f as a function of k by taking the derivative. To simplify the math, minimize the logarithm[1] of f with respect to k.

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

We find the optimal k, or right number of hash functions to use, when the derivative is 0. This occurs when $k = \frac{\ln 2n}{m}$. Substitute this value into (1.4), above,

$$f\left(\ln 2\frac{n}{m}\right) = \left(\frac{1}{2}\right)^{k} = (0.6185)^{\frac{n}{m}}$$

References

[1] Thomas H. Cormen, Clifford Stein, Ronald L. Rivest, and Charles E. Leiserson. *Introduction to Algorithms*. McGraw-Hill Higher Education, 2nd edition, 2001.

[2] B. Bloom. "*Space/time trade-offs in hash coding with allowable errors*," Communications of the ACM, 13(7):422-426, 1970.

[3] L. Fan, P. Cao, J. Almeida and A. Z. Broder, “*Summary Cache: A Scalable Wide-area Cache Sharing Protocol*," in Proceedings of ACM SIGCOMM ’98.

[4] M. Mitzenmacher. "*Compressed Bloom Filters*," in Proceedings of PODC 2001.

[1] I was reminded that minimizing the logarithm of a function is equivalent to minimizing the function itself.