CPSC 320 Notes: Bloom Filters, or Trading Off Memory for…Wrong Answers

November 28, 2016

You work for the well-known web infrastructure company Wawaiki. Your company's business is caching the files associated with requested URLs so browsers can load pages faster. Recent analysis of your caching logs show that more than 3/4 of requested URLs are never requested again.[1]

You're trying to figure out a way to avoid caching the files associated with URLs until the **second** time you see the URL.

1 Data Structures Question? The Answer Is Probably Hash Tables

One approach would be to keep a hash set of URLs (a hash table where the keys are URLs and there are no values, since you just care about presence/absence).

Determine how much memory we will use to store 1 billion URLs in our hash set.

Assume:

- the hash table's load factor is 1
- the hash table uses chaining to resolve collisions
- a pointer on your system takes 8 bytes (64 bits)
- the average URL is a 78 bytes long (including a nil-terminator)[2]

[1]"One-hit wonders" and the solution we use here are real. See Algorithmic Nuggets in Content Delivery by Maggs and Sitaraman: https://people.cs.umass.edu/r̃amesh/Site/HOME_files/CCRpaper_1.pdf.

[2]See http://www.supermind.org/blog/740/average-length-of-a-url-part-2. Note that URLs are ASCII; so, 1 byte per character.

2 A Bit Smaller

Let's design a hash set except that each entry in the table is a single bit, where a 1 indicates that the key is present in the set and a 0 indicates that it is not.

1. Determine how much memory we will **now** use to store 1 billion URLs. Assume we make the table have 10 times as many entries as there are elements in the table.

2. Why might the load factor (the number of non-zero entries divided by the total number of entries) be less than 0.1?

3. If I ask whether a URL is in this strange hash set, the answer can be wrong. Under what circumstances might the answer be wrong? Under what circumstances will it **never** be wrong?

4. With the load factor indicated above, if you ask the hash set a query that can be wrong, what is the probability that the answer **will** be wrong?

   Do your best to find a formula here, and then move on!

5. What should we **do** if the answer is wrong? (Can we "fall back" to a better solution?)

3 If Hash Is Good, More Hash Is Better

Imagine we have two totally different, excellent hash functions.

Originally, to insert a key into the table, we: hash the key, mod that by the table size, and set the entry at the resulting index to 1.

Now, we simply do that twice, once for the first hash function and once for the second.

1. Determine how much memory we will **now** use to store 1 billion URLs. Assume the load factor is still 0.1, and that "load factor" still means "number of keys inserted into the table divided by table size".

2. If I ask whether a URL is in this strange hash set, the answer can be wrong. Under what circumstances might the answer be wrong? Under what circumstances will it **never** be wrong?

3. With the load factor indicated above, if you ask the hash set a query that can be wrong, what is the probability that the answer **will** be wrong? (Assume that each key (independently) has equal probability of being mapped to each entry in the table by each hash function (independently).)

   This is hard to answer precisely. To derive a relatively simple bound on the answer, try imagining that there were **no** collisions among any of the indices chosen by either hash function for any of the already-inserted elements.

4 Bloom Filters

A Bloom Filter is a generalized version of what we've just done. It is a bit vector V with m entries and k (ideally-independent) hash functions. To insert a new element i, we set $V[h_j(i)] = 1$ for each of the k hash functions h_j. To check whether an element i is in the set, we AND together each value $V[h_j(i)]$ for all k hash functions.

The optimal number of hash functions is $k = \frac{m}{n} \ln 2$. For that number of hash functions, and given a choice of false positive rate, the ratio $\frac{m}{n} = -\frac{\ln p}{(\ln 2)^2}$.

1. Compute the approximate number of bits per entry and the number of hash functions needed for a false positive rate of 1%.

2. What happens to the false positive rate as users search for more and more URLs? How should we resolve this problem?

5 Big Picture/Challenges

We've just looked at three big ideas in data structures and algorithms:

1. Randomized algorithms can give you enormous power and flexibility. Check out the awesome and heavily-used Miller-Rabin primality test to learn more about one randomized algorithm. (Testing primality deterministically takes polynomial time, but the randomized test will almost certainly serve your needs better if you want to find a big prime!)

2. Universal hash functions make it easy to use multiple different hash functions on the same piece of data. Once we can do that, lots of nifty ideas become possible. Check out Cuckoo Hashing for one cool example! (There is also such a think as a Cuckoo Filter.)

3. If we abandon a restriction to correctness (i.e., soundness, completeness, or both), we can often make progress on problems that are otherwise intractable. For example, the Daikon invariant detector exploits unsoundness to document invariants in software, a problem that is uncomputable in general.