CS166
Spring 2018

Handout 13
May 17, 2018

Problem Set Five: Randomized Data Structures

This final problem set of the quarter explores randomized data structures and the mathematical techniques useful in analyzing them. By the time you've finished this problem set, you'll have a much deeper appreciation for just how clever and powerful these data structures can be!

Due Thursday, May 24 at 2:30PM.

Problem One: Count Sketches with 2-Independent Hashing (3 Points)

In our analysis of count sketches from lecture, we made the following simplification when determining the variance of our estimate:

$$\mathrm{Var}[\sum_{j \ne i} \boldsymbol{a}_j s(x_i) s(x_j) X_j] = \sum_{j \ne i} \mathrm{Var}[\boldsymbol{a}_j s(x_i) s(x_j) X_j]$$

In general, the variance of a sum is not the sum of the variances, so this step required some justification. The justification we provided in lecture was that if we let s and h be 3-independent hash functions, then the terms in the sum were pairwise independent. However, to write the variance of a sum as a sum of variances, we don't actually need the terms of the sum to be independent of one another. They just need to be ***uncorrelated***.

Prove that any two terms in the above summation are uncorrelated under the assumption that both s and h are drawn uniformly and independently from 2-independent families of hash functions. This shows that 2-independent hashing is sufficient for count sketching, and does so in a way that doesn't require us to redo any of the remaining analysis.

As a refresher, two random variables X and Y are uncorrelated if $\mathrm{E}[XY] = \mathrm{E}[X]\mathrm{E}[Y]$.

Problem Two: Cardinality Estimation (12 Points)

A ***cardinality estimator*** is a data structure that supports the following two operations:

- *ds.*`see`(x), which records that the value x has been seen; and
- *ds.*`estimate`(), which returns an estimate of the number of distinct x's we've seen.

Imagine that we are given a data stream consisting of elements drawn from some universe $\mathscr{U}$. Not all elements of $\mathscr{U}$ will necessarily be present in the stream, and some elements of $\mathscr{U}$ may appear multiple times. We'll denote the number of unique elements in the stream as $\boldsymbol{F_0}$.

Here's an initial data structure for cardinality estimation. We'll begin by choosing a hash function h uniformly at random from family of 2-independent hash functions $\mathscr{H}$ from $\mathscr{U}$ to the open interval of real numbers (0, 1). For simplicity's sake, we'll assume that there are no hash collisions, which isn't too unreasonable given that the codomain is infinite.

Our data structure works by hashing the elements it `see`s using h and doing some internal bookkeeping to keep track of the kth-smallest unique hash code seen so far. The fact that we're tracking *unique* hash codes is important; we'd like it to be the case that if we call `see`(x) multiple times, it has the same effect as just calling `see`(x) a single time. (The fancy term for this is that the `see` operation is ***idempotent***.) We'll implement `estimate`() by returning k / h_k, where h_k denotes the kth smallest hash code seen so far.

i. Explain, intuitively, why you'd expect k / h_k to be a good estimate of F_0. As a hint, think about two different ways of counting up how many elements should have hash codes less than h_k.

Let $\varepsilon \in (0, 1)$ be some accuracy parameter that's provided to us.

ii. Prove that $\Pr[\ k / h_k \ >\ (1 + \varepsilon)F_0\ ] \le 4 / k\varepsilon^2$. This shows that by tuning k, we can make it unlikely that we overestimate the true value of F_0.

Some hints and things to think about as you work through the above problem:

- At some point you should end up with an expression involving $(1 + \varepsilon)^{-1}$. We strongly recommend applying the inequality $(1 + \varepsilon)^{-1} \le 1 - {}^{\varepsilon}/_{2}$, which holds for any $\varepsilon \in (0, 1)$.
- Find a random variable X that's a sum of indicator variables where
$$\Pr[\ k / h_k \ >\ (1 + \varepsilon)F_0\ ] \ \le\ \Pr[\ X > k\ ].$$
- Use Chebyshev's inequality.

Using a proof analogous to the one you did in part (ii) of this problem, we can also prove that

$$\Pr[\ k / h_k < (1 - \varepsilon)F_0\ ] \le 2 / k\varepsilon^2.$$

The proof is very similar to the one you did in part (ii), so we won't ask you to write this one up. However, these two bounds collectively imply that by tuning k, you can make it fairly likely that you get an estimate within $\pm\varepsilon F_0$ of the true value! All that's left to do now is to tune our confidence in our answer.

iii. Using the above data structure as a starting point, design a data structure with tunable parameters $\varepsilon \in (0, 1)$ and $\delta \in (0, 1)$ such that

  - `see`(x) takes time $O(\log \varepsilon^{-1} \cdot \log \delta^{-1})$;
  - `estimate`(x) takes time $O(\log \delta^{-1})$, and if we let C denote the estimate returned this way, then
$$\Pr[\ (1 - \varepsilon)F_0 \ \le\ C \ \le\ (1 + \varepsilon)F_0\ ] \ge 1 - \delta; \text{ and}$$
  - the total space usage is $\Theta(\varepsilon^{-2} \log \delta^{-1})$.

Problem Three: Hashing in the Real World (10 Points)

You've now seen a number of different approaches for building hash tables and their mathematical analysis. How well do these hash tables hold up in practice? In this problem, you'll find out.

The starter files for this programming assignment are available at

`/usr/class/cs166/assignments/ps5`

Your task is to implement the following flavors of hash table:

- ***Chained hashing:*** The standard hash table usually taught in CS106B/X, CS107, and CS161. Chances are you've implemented one of these before, so hopefully this will just be a warm up.
- ***Second-choice hashing:*** This is a variation on chained hashing. You'll maintain two hash functions h_1 and h_2. When inserting a key x, compute $h_1(x)$ and $h_2(x)$ and insert x into whichever bucket is less loaded. To do a lookup, search for x both in the bucket given by $h_1(x)$ and $h_2(x)$.
- ***Linear probing:*** The open-addressing scheme described in class.
- ***Robin Hood hashing:*** A modification on linear probing described in lecture. When storing elements in the table, annotate each with its intended bucket. When doing an insertion, start off as normal, but if you ever find that the slot you're currently scanning is occupied and the element there is closer to its intended location, place the newly-inserted element at that location, displacing the older element, then continue onward with the scan from the current position to place the displaced element. You may end up displacing several elements in a single insertion.
- ***Cuckoo hashing:*** The dynamic perfect hashing scheme described in class.

We've provided a test harness that will test your hash tables with different load factors (you don't need to worry about resizing the tables – we'll always size them appropriately for you) and different choices of hash functions. Once you've finished implementing your solutions, crank up the optimization level to the maximum (`-O3`), run our driver code, and submit a (brief) writeup answering the following questions:

- The theory predicts that linear probing and cuckoo hashing degenerate rapidly beyond a certain load factor. How accurate is that in practice?
- How does second-choice hashing compare to chained hashing across the range of load factors? Why do you think that is?
- How does Robin Hood hashing compare to linear probing across the range of load factors? Why do you think that is?
- In theory, cuckoo hashing requires much stronger classes of hash functions than the other types of hash tables we've covered. Do you see this in practice?
- In theory, cuckoo hashing's performance rapidly degrades as the load factor approaches $\alpha = ½$. Do you see that in practice?

As always, to receive full credit for this assignment, your code should compile cleanly without warnings on the `myth` machines and should not have any memory leaks.

(There's advice and hints for this problem on the next page.)

Some things to keep in mind:

- Our driver code will provide the number of buckets to use as a parameter to the constructors of your hash table. We've chosen these numbers specifically to test the performance of your hash table under different load factors. As a result, you should *not* resize your hash tables dynamically. Our starter files will always leave at least a few slots empty in your tables, so, for example, you don't need to handle the case where you have a linear probing hash table that's at 100% capacity.
- The file comments for each of the hash tables contain information about specific implementation requirements. For example, we'd like you to implement deletions in Robin Hood hashing using backwards-shift deletion and deletions in linear probing tables using tombstone deletion.
- Hash functions are represented using the `HashFunction` type. `HashFunction` essentially acts like a function pointer, so if you have a variable of type `HashFunction` named `h`, you can invoke it by calling `h(key)`. The hash values are distributed over the range $[0, 2^{31})$, so you will need to mod hash codes by your table sizes.
- You can assume that the keys you're hashing will be nonnegative integers. Feel free to reserve negative integers as sentinel values.
- In Robin-Hood hashing, remember that you can – and should – terminate searches early in many cases by looking at where the currently-scanned element is relative to where it should be.
- You are welcome to use the C++ standard library types and functions if you'd like, though the standard hash containers (`std::unordered_map`, `std::unordered_set`, etc.) are, understandably, off-limits.