CS166 Handout 11
Spring 2016 May 12, 2016

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 19 at the start of lecture.

Problem One: The Tug-of-War Sketch (10 Points)

In lecture, we saw that if $\boldsymbol{a}$ is a vector, then $\|\boldsymbol{a}\|_2$ is the L_2-norm of $\boldsymbol{a}$, which is defined as

$$\|\boldsymbol{a}\|_2 = \sqrt{\sum_{i=1}^{n} \boldsymbol{a}_i^2}$$

A related quantity is the ***second frequency moment*** of $\boldsymbol{a}$, which is defined as

$$F_2 = \sum_{i=1}^{n} \boldsymbol{a}_i^2$$

The second frequency moment of a frequency vector is useful for getting a sense of how skewed the distribution is, since it's quite sensitive to large values. A common application of second frequency moments is measuring the difference between two different data streams – by computing the second frequency moment of the difference between the underlying frequency vectors, you can get a good estimate for how dissimilar the data are.

In this problem, you'll work through the development and analysis of a data structure called the ***tug-of-war sketch*** that estimates F_2 for a data stream. As you'll see, the data structure strongly resembles a count sketch, and its analysis touches on many of the techniques from our analysis of linear probing.

The goal of this problem is to design a data structure supporting these two operations:

- *ds.**increment***(x), which increments the frequency of the element x, and
- *ds.**estimate***(), which returns an estimate of F_2.

As with the count[-min] sketch, we'll start with a very simple data structure for solving this problem, then replicate it in a way gives it a $1 - \delta$ probability of estimating F_2 to within a factor of $(1 \pm \varepsilon)$. The initial data structure is so simple that it's almost comical. Let $\mathscr{H}$ be a k-independent family of hash functions from some set $\mathscr{U}$ to $\{-1, +1\}$. (We'll choose k later based on how the analysis goes). Our data structure will simply be an integer counter C, initially 0, paired with hash function h chosen uniformly at random from $\mathscr{H}$. To perform *ds.**increment***(x), we add $h(x)$ to C. To perform *ds.**estimate***(), we return C^2. That's it. (Notice the similarity to the count sketch.)

i. Prove that if $\mathscr{H}$ is a family of hash functions that's at least 2-independent, then $E[C^2] = F_2$.

As per our normal approach, let's see if we can use some concentration inequalities to get a bound on the accuracy of our estimation. Suppose we are ultimately interested in getting an estimate that's within a $(1 \pm \varepsilon)$ factor of the true answer (that is, we'd like to have $(1 - \varepsilon) \le C^2 \le (1 + \varepsilon)$, or, equivalently, that $|C^2 - F_2| \le \varepsilon F_2$). Since we're trying to bound a two-sided error, Markov's inequality isn't likely to be useful, but we can probably make some progress using Chebyshev's inequality.

ii. Prove that if $\mathscr{H}$ is a family of hash functions that's at least 4-independent, then $\mathrm{Var}[C^2] \le 2F_2^2$.

iii. Use Chebyshev's inequality to bound the probability that $|C^2 - F_2| \ge \varepsilon F_2$, then explain why the bound you got isn't a particularly useful one.

We're now at a point where we have an unbiased estimator for F_2, but one whose variance is too large to be useful. To address this, we'll use a time-honored tradition for reducing the variance: run a bunch of copies of this estimator in parallel and take the average.

Let's run w independent copies of this estimator and denote the counters as $C_1, C_2, \ldots, C_w$ and the (independently, uniformly-randomly-chosen) hash functions as $h_1, h_2, \ldots, h_w$. As our estimate, we'll return

$$D = \frac{1}{w}\sum_{i=1}^{w} C_i^2$$

In other words, we'll simply return the average of all the estimates.

iv. Prove that $E[D] = F_2$ and that $\mathrm{Var}[D] \le 2F_2^2 / w$. This shows we now have a new, unbiased estimator for F_2 with lower variance than our original estimator.

v. Show that there is a way to set $w = O(\varepsilon^{-2})$ such that the estimate D satisfies $|D - F_2| \le \varepsilon F_2$ with probability at least p for some fixed universal constant $\frac{1}{2} < p \le 1$.

We now have a estimator for F_2 whose precision can be controlled by a parameter ε. All that's left to do now is to find a way to ensure that we're accurate with high probability.

vi. Show that there's a way to run $O(\log \delta^{-1})$ copies of the above data structure such that a final estimate of F_2 can be produced that, with probability $1 - \delta$, estimates F_2 to within a factor of $(1 \pm \varepsilon)$. This final data structure is the tug-of-war sketch.

Your final data structure only needs space for $O(\varepsilon^{-2} \log \delta^{-1})$ hash functions and counters, which makes it extremely space efficient!

The technique covered in parts (iv), (v), and (vi) of this question are often referred to as the ***median-of-means*** method. It turns out it's actually a pretty general technique – given any randomized data structure that's an unbiased estimator of some quantity, you can run several copies of it in parallel in the way you described to get a tightly controlled bound.

Problem Two: 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, 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?

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

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 if you'd like, though the standard hash containers (`std::unordered_map`, `std::unordered_set`, etc.) are, understandably, off-limits.