

Problem Set 6

Due: Sunday, October 7, 11:59pm

Experimenting with Sketches

Overview

In this problem, we will implement and compare two different streaming algorithms for counting the number of times an item appears in a stream. (Recall, we previously looked at this problem on Problem Set 4.) The goal of this problem set is to get a chance to experiment with some real streaming algorithms, as well as to think about how best to compare two algorithms. How do we decide which algorithm is better for a given application?

Frequency of elements in a stream. Assume we have a stream consisting of N elements in total. Let $n(x)$ be the number of times that the element x appears in the stream. For this problem, the elements may not be integers. For example, one of the sample applications we will look at is counting the frequency of words in a book.

The goal is to calculate an approximate value of $n(x)$, as accurately as possible, given a limited amount of space. We will look at two algorithms for accomplishing this.

Today's goal. Your job in this problem set is to implement the two proposed algorithms (using any programming language that you like) and run some experiments to determine which is better.

What makes one algorithm *better* than another? For today, we are most interested in the accuracy of the estimates. How good a job does the algorithm do of producing an accurate estimate of $n(x)$? Trying to decide which one is better may be tricky because:

- For different distributions of input data, a different algorithm may be better. For example, one algorithm may be better on a uniform distribution, while the other may be better on an exponential distribution. (Or perhaps not!)
- For different elements in the stream, a different algorithm may be better. For example, one algorithm may produce a better estimate for elements that appear very frequently, while the other algorithm produces a better estimate for items that appear rarely.

For both algorithms, more space yields more accuracy. Thus we will be particularly interested in the trade-off between space usage and accuracy. (For example, if one data structure is twice as accurate but requires ten times as much space, that is not a good indication that it is better. We really want to compare the accuracy when they use the same space.) One goal is to understand how the accuracy improves with more space.

What to submit. For this problem set, please submit a short report that compares these two algorithms, supporting your claims with data from your experiments. You do not need to submit your code.¹

Your report should include a short description of your implementation. The implementation description does not need to repeat the explanation of how the algorithm works, or standard details (e.g., you do not need to say things like, “I used a *for* loop to iterate through the hash functions.”). You should explain any interesting design decisions you needed to make during the implementation, e.g., which hash functions you chose to use, any optimizations you added, etc.

Your report should also include a description of the experiments you ran, and the results of those experiments. Most importantly, please *interpret* the results of your experiments. Just providing many pages of data is not sufficient: the goal of a good experiment is to: (a) make a claim about the algorithms in question, and (b) support that claim with your data. The data you provide should support the claims you make (rather than the other way around).

Please support your report on IVLE.

What else? Below, I describe the two algorithms, and suggest some experiments to run. You may run additional experiments, as needed (and as motivated by your interests) to discover more about how these data structures work, or how they might be optimized. (For example, there are ways to reduce the space needed; or you might explore whether different hash functions yield different results; or you might try other input data not mentioned below; etc.)

Two Algorithms

We now describe two algorithms for estimating the number of times an element appears in a stream. Both are very similar to the algorithm considered in Problem Set 4, and both algorithms have the same structure (See Figure 1):

- We will use A hash functions: choose h_0, h_1, \dots, h_{A-1} to be hash functions mapping elements to $[0, B - 1]$ uniformly at random.
- We will also use AB counters: let $C(i, j)$ be a counter where $i \in [0, A - 1]$ and $j \in [0, B - 1]$.
- When we see element x in the stream, then for all $i \in [0, A - 1]$, we will increment the counter $C(i, h_i(x))$. (When we increment a counter, we simply add one to the value of the counter.)

The two algorithms differ only in how a query is performed:

Algorithm 1. On a query of element x , we compute an approximate of $n(x)$ by taking the **median** value of the counters that x is mapped to. That is, we return:

$$\text{query}(x) = \text{MEDIAN}\{C(i, h_i(x)) \mid i \in [0, A - 1]\}$$

(Recall that in Problem Set 4, we returned the *minimum* value from this set. Today, we are using the median.)

¹Note that you should keep your code available, in case we ask to see it. If your results look strange, we may ask you further questions about your implementation.

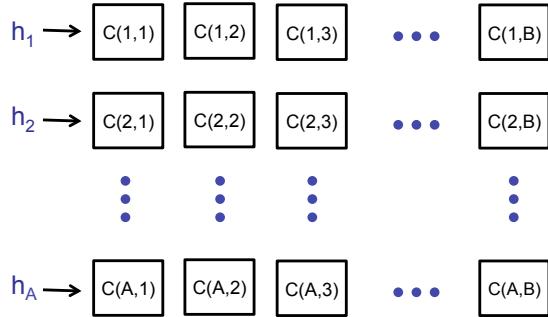


Figure 1: We use AB counters. Each of the A hash functions maps each of the incoming elements in the stream to one of the B counters in its row.

Algorithm 2. For Algorithm 2, we try to reduce the error. (The question we are trying to decide is whether this idea works!) First, consider all the counters $C[i, \cdot]$, i.e., all the counters in the i th row of counters. Assume, in this case, that B is even (and hence $B - 1$ is odd). We pair of counters with their neighbors. Define the *neighbor* of an odd-numbered counter to be the one on its left; the *neighbor* of an even numbered counter is the one on its right. More precisely:

$$\begin{aligned} \text{neighbor}(C[i, j]) &= C[i, j + 1] && \text{if } j \text{ is even} \\ \text{neighbor}(C[i, j]) &= C[i, j - 1] && \text{if } j \text{ is odd} \end{aligned}$$

Now, for the i th row of counters, we define $\text{estimate}(x, i)$ to be the difference between the counter for x and its neighbor. That is:

$$\text{estimate}(x, i) = C[i, h_i(x)] - \text{neighbor}(C[i, h_i(x)])$$

The intuition here is that we are subtracting off some of the noise.

Now on a query of element x , we compute an approximate of $n(x)$ by taking the median value of estimates. That is, we return:

$$\text{query}(x) = \text{MEDIAN}\{\text{estimate}(x, i) \mid i \in [0, A - 1]\}$$

In this case, it is important that we return the median (instead of the minimum), since the estimate might be negative for some of the rows.

Experimental Data

In order to compare these two algorithms, one important aspect is the data set being examined. The algorithms may have very different performance on different inputs. Three different data sets you might want to try are:

1. *Uniform:* All the items are uniformly distributed. That is, generate a stream of length N containing integers in the range $[0, M - 1]$ (for some fixed value of M that is much smaller than N). For each stream element, choose the element randomly from the set $[0, M - 1]$. In this case, you would expect each element to appear about N/M times.

2. *Exponential*: The items are distributed exponentially. Again, generate a stream of length N containing integers in the range $[0, M - 1]$ (for some fixed value of M that is much smaller than N). This time, however, choose element $i \in [1, M]$ with probability $1/2^{i+1}$. Thus, you will choose element 0 with probability $1/2$, element 1 with probability $1/4$, element 2 with probability $1/8$, etc. (Notice that unless N is very large, you are very unlikely to choose element M every!)
3. *Real-world data*: Choose a very long book and count the number of times each word appears in the book. You can find books in text format on Project Gutenberg. (You might, for example, use “The Complete Works of Shakespeare.”) Use both algorithms to determine how often various words appear in the book of your choice.

Other Variables

One choice that you will have to make is the values for A and B . How do these variables affect the performance of the algorithm? Does one have more of an impact on accuracy than the other? Does one have more of an impact on the frequency of producing a good answer?

Notice that the space is determined by A and B . For a fixed amount of space, what is the best choice of A and B ? How does the performance of the two algorithms compare for a given amount of space? How does the performance change as more space is used?

Hash Functions

One issue that may arise is that you need a way to generate random hash functions. There are many standard ways to do this, and feel free to look up alternatives. You can use Tabulation Hashing, or the binary matrix technique; perhaps the easiest way is to use a prime field:

Assume that you need a hash function f that maps integers to the range $[0, M]$. Choose a prime number p that is larger than M . (Some suggest that it works better if p is significantly larger than M , though in theory that should not matter as long as $p > M$.) Choose two random integer $a \in [1, p - 1], b \in [0, p - 1]$. Then, define:

$$f(x) = ((ax + b) \mod p) \mod M$$

If you need a whole sequence of hash function f_1, f_2, \dots, f_k , you can use the same prime number p , and just choose a sequence of a_1, a_2, \dots, a_k and b_1, b_2, \dots, b_k . You can then use the a_i and b_i to define the function f_i .

Regardless of which hash functions you choose, be sure to explain how you are constructing your hash functions and how you use them to implement the algorithms described.

Summary

Please summarize your report by making a recommendation. Which algorithm should I use? Are there some circumstances where I should use one instead of the other? What is your advice? (Hint: you may want to begin the report with your recommendation, rather than hiding your conclusion at the end.)