Probability & Computing

Lecture 11

11/02/2020

1 Outline

- Streaming Problems
- Algorithm for finding the length of the stream
- Idea for counting distinct elements

2 Streaming Problems

In a streaming problem, we assume that we deal with a large amount of data and limited main memory for processing it. The following is the typical scenario with which we will deal. We have a sequence of items $I_1, I_2, \ldots, I_n$ that we see one at a time, and after a single pass (or sometimes a few passes), we will find a function of $(I_1, \ldots, I_n)$. We assume that we don't have enough space to store all the items; this may mean that we cannot compute the function exactly and we trade space for an approximation.

The two problems we start with are:

1. Given a sequence of n bits, find n (the length of the stream). In this problem, the value of the bits does not matter, and we can clearly do this with space of $\lceil \log_2 n \rceil$ bits, which is also required to store the exact value of n.
2. Given a stream of n numbers, which we know to be in $\{1, 2, \ldots, m\}$, find the number $d \leq m$ of distinct elements in the stream. We can

solve this problem exactly using a m bit vector, where we increment the ith bit when we see the number i. We will aim to instead find an approximate value of d using just $O(\log m)$ bits.

3 Algorithm for finding the stream length

Since $\lceil \log_2 n \rceil$ bits are both necessary and sufficient to count up to n, it appears that there is nothing more to this problem; however by using an implicit representation, we can use $O(\log \log n)$ bits of working memory to finally output an approximate value of n; the final output alone will use $\log_2 n$ bits and to find this value we only use $O(\log \log n)$ bits.

Firstly, we note that the number $\lceil k = \log_2 n \rceil$ (number of bits in n) itself can be stored using $O(\log \log n)$ bits, and if we can find this value, then we can output 2^k, which satisfies $n \leq 2^k \leq 2n$ and is hence an approximation of n.

In order to keep track of the value of $\log_2 n$, we will use a counter. Note that the value of this counter must be incremented whenever the length of the stream doubles. Now we use the idea of incrementing the counter with a probability $p(C)$, which we allow to depend on C, the value of the counter.

We know that the expected time for the counter to increment is $\dfrac{1}{p(C)}$, and so we want this to be i when the number of items increases from i to $2i$. Thus, we deduce that we should choose $p(C) = \dfrac{1}{i}$, but remembering that we don't know the value of i and that we want 2^C to approximate i, we set $p(C) = \dfrac{1}{2^C}$.

Thus, the algorithm (due to Morris) is:

- Initialize $C = 0$.
- On seeing a new bit/item, increment C with probability $\dfrac{1}{2^C}$.
- When the stream ends, output 2^C as the approximation to the length of the stream.

We must now analyze the probability that the value output is an approximation of n, and also estimate how good the approximation is.

Definition We say that an algorithm is a (ε, δ) approximation for a value v if the algorithm outputs a value w such that:

$$Pr[|w - v| > (1 + \varepsilon)v] \leq \delta.$$

Later we will show that Morris' algorithm is a (c_1, c_2) approximation for a constant c; further we can boost both the approximation and the correctness probability by maintaining several independent counters and outputing the *median* of all the counters. Using the mean works, but it turns out that the median gives a better approximation.

In class, we defined two quantities: $C(i)$, which is the value of the counter after seeing i items; and $D(i)$, which is the number of items after which the counter's value becomes i. We have $E[D(i)] = E[D(i-1)] + 2^{i-1}$ and $D(1) = 1$, which gives $E[D(i)] = 2^i$.

Also,

$$E[2^{C(i)}] = \sum_{1 \leq k \leq i} 2^k Pr[C(i) = k]$$

and

$$Pr[C(i) = k] = \frac{1}{2^{k-1}} Pr[C(i-1) = k-1] + \left(1 - \frac{1}{2^k}\right) Pr[C(i-1) = k].$$

Exercise: Using the above relations, show that $E[2^{C(i)}] = i$.

4 Idea for counting distinct elements

Suppose that we can compute a function $h : \{1, 2, \ldots, m\} \to [0, 1]$ such that each $h(i)$ is uniformly distributed and distinct $h(i)$s are independent. Then if $x_1, \ldots, x_n$ are the elements of the stream, we can compute the following: Let $m = min\{h(x_1), h(x_2), \ldots, h(x_n)\}$. Now we have $E[m] = \frac{1}{d+1}$, and hence the output $\frac{1}{m} - 1$ should be a good approximation to the number of distinct elements.

However, we have the following issues: firstly, approximations when we involve real number computation; secondly, how to choose such a randomly behaving function.

To address the first issue, the values that we compute will also have the range $\{1, 2, \ldots, M\}$ (instead of $[0, 1]$). To address the second issue, we will use a *family of 2-universal hash functions*. In a related approach, we will also compute the maximum number of leading or trailing zeroes of the hashed values. Note that if the maximum number of leading zeroes is r, then the minimum value that we've seen is at most $2^{\lceil \log_2 m \rceil - r} \sim \dfrac{m}{2^r}$.