Abstract

These notes cover Morris algorithm, Reservoir Sampling and Frequency moments algorithms for calculating $F_0$ and $F_\infty$.

Main items from first lesson (including the reminder)

Morris Algorithm:[Mor77]

Our Problem. Count using log log $n$ memory, given an error $\epsilon$ and a failure probability $\delta$.

Markov inequality. $Z$ is a non-negative random variable with some expectation $E$, then $\forall t \geq 0$, $\text{Prob}[Z \geq t] \leq \frac{E}{t}$.

Chebyshev inequality. $\text{Prob}[|Z - E[Z]| \geq t] \leq \frac{\text{Var}[Z]}{t^2}$.

Algorithm 1 Morris

Init $Y \leftarrow 0$

Given event: $\begin{cases} Y \leftarrow Y + 1 & \text{w.p. } \frac{1}{2^Y} \\ \text{do nothing} & \text{w.p. } 1 - \frac{1}{2^Y} \end{cases}$

When asked how many events: return $Z = 2^Y - 1$

Analysis. We showed that: $E[Z] = E[2^Y - 1] = n$ (proof by induction), and $\text{Var}[Z] \leq C \cdot n^2$.

Note: $C$ defines some constant, and could change during calculations. For example, this is OK: $C \leq C + 1$.

Morris+:

This bound on the variance is not really good. We can reduce the variance by creating multiple counters (Morris+). Given an error $\epsilon$, we can create $k = \frac{C}{\epsilon^2}$ independent counters and then:

$$\hat{Z} = \frac{1}{k} \sum Z^{(k)}$$

$$\text{Var} (\hat{Z}) = \frac{\text{Var} (Z^{(k)})}{k} \leq \frac{C \cdot n^2}{k} \leq C \cdot \epsilon^2 n^2$$

1Derives from the Markov inequality, square both sides of the inequality.
Now, using Chebyshev we can bound: \( \text{Prob} \left[ \left| \hat{Z} - n \right| > \epsilon \cdot n \right] \leq \frac{2}{\epsilon^2 k} \).

**Morris++:**

We shall use a median trick, that can be used when \( \mathbb{E}[Z] = O \left( \text{Var} [Z]^2 \right) \).

**Algorithm 2 Morris++**

Perform \textbf{Morris+} algorithm \( S \) times (independently).

Return: \( \tilde{Z} = \text{Med} \left( \hat{Z}^{(1)}, \hat{Z}^{(2)}, \ldots, \hat{Z}^{(S)} \right) \).

**Analysis:**

The algorithm fails if \( \left| \tilde{Z} - n \right| > \epsilon n \). Let’s look at all the \( \{\hat{Z}\}_{i=1}^{S} \) sorted: \( \hat{Z}^{(\pi(1))} \leq \hat{Z}^{(\pi(2))} \leq \ldots \leq \hat{Z}^{(\pi(S/2))} \ldots \leq \hat{Z}^{(\pi(S))} \). A failure means that \( \tilde{Z} \) is too big or too small. Let’s assume that it’s too big, that is \( \tilde{Z} > (1 + \epsilon) \cdot n \). Hence, all the \( \hat{Z}^{(i)} \) that are bigger than \( \tilde{Z} \) are also too big. Let us denote this set by \( E = \{1 \leq i \leq S \mid \hat{Z}^{(i)} > (1 + \epsilon) \cdot n \} \). Instead of finding a bound on the probability of failure, we’ll find a probability bound for \( m \triangleq |E| \geq S \). We know \( m \)'s expectation, as \( m \sim \text{Binomial}(S \leq 1/\beta) \). Therefore; \( \mathbb{E}[m] \leq \frac{S}{\beta} \). From Hoeffding inequality, we get that\(^2\) \( \text{Prob} \left[ \text{Morris++ fails} \right] \leq 2e^{-\Omega(S)} \). We want this probability to be smaller than some given \( \delta \), and therefore \( S = C \cdot \log \frac{1}{\delta} \). This gives us \( \log \frac{1}{\delta} \) dependency instead of \( \frac{1}{\delta} \).

**Reservoir Sampling**

Our problem: Given a stream of non-negative numbers, that goes on to infinity - at every given moment \( t \), return \( i \) w.p \( \sum_{j=1}^{t} \frac{a_i}{Z_j} \)

**Alg**  \( \text{Init: } Z \leftarrow 0, \text{prev} \)

Given \( a_i \) : \( Z_i = Z_{i-1} + a_i \)

\( \text{prev} \leftarrow \begin{cases} i & a_i \geq Z \\ \text{prev} & \text{otherwise} \end{cases} \)

**Analysis**  \( \text{Prob}[\text{output } j \text{ at time } t \geq j] \)

\( ^2 \text{The } 2 \text{ in the inequality is given by summing the two cases in which } \tilde{Z} \text{ is too big and } \hat{Z} \text{ is too small.} \)

\( ^3 \text{the probability to assign } j \text{ at time } j \text{ to prev, and not overriding prev after time } j \)
Frequency moments:

**Our problem**  Given alphabet $\sum$, where $|\sum| = n$, and $n$ is very big, to answer for each $a \in \sum$: $f_a$ = how many times it appeared in the stream.

In our example from the first class of identifying router attack, each letter is 64 bits of origin and destination IPs. This means $n=2^{64}$ (!!)

We obviously can’t save the stream, and we even can’t save the histogram.

**Defining frequency moment**  $F_i = \sum_{a \in \sum} f_a^i$, $i$ frequency moment.

- $F_0$ = Number of distinct letters that appeared in the stream
- $F_1$ = Counter of letters in stream
- $F_2$ = $\sum f_a^2$, How distributed/centralized the distribution is (Gini Index) [AMS99]
- $F_\infty^*$ = $\max_{a \in \sum} \{f_a\}$, Heaviest Hitter in the stream

Rule of thumb: the greater the $i$, the more difficult it is to compute $F_i$. it’s really hard to compute $F_\infty^*$. Usually we’ll settle for a small set of heavy hitters.

We’ll use a hash function $h : \sum \rightarrow S$, $|\sum| \gg |S|$. (We’re choosing $h$ randomly, before seeing the stream - will do a worst case analysis, where the adversary doesn’t know our random $h$).

$F_0$: *(more idealized than implementable)*

$h : \sum \rightarrow [0,1] \forall a, h(a) \sim U([0,1])$, iid

**Algorithm 3**

Init: $h, M \leftarrow \infty$

given $a_i$, ($i^{th}$ item in stream) compute $h(a_i)$

if $h(a_i) < M$:

$M \leftarrow h(a_i)$

output: $\frac{1}{M} - 1$

**Analysis**  $S_1, ..., S_{F_0} \in \{0,1\}$, distinct elements.

$M = \min \{S_1, ..., S_{F_0}\}$

$\text{Prob}[M \in [t + dt]] = F_0(dt) \cdot (1-t)^{F_0-1}$

$\mathbb{E}[M] = \int_a^b F_0t(1-t)^{F_0-1}dt = \frac{1}{F_0+1}$

$\text{Var}(M) \leq \frac{C}{F_0^2}$
More practical

\[ h : \sum \rightarrow \{0, 1\}^{\log N}, \text{ where } N \text{ is the length of the stream.} \]

**Notation.** We denote the leading zeros of a vector \( v \in \{0, 1\}^{\log N} \) by \( LZ(v) \).

**Algorithm 4**

**Init:** \( r \leftarrow 0 \)

Given \( a_i \), Compute \( LZ(h(a_i)) \).

If \( x > r \) then

\[ r = x \]

Return \( 2^r \)

\( F_1 \)

This is a counter of the number of instances in the stream. We saw in the first lesson how to do this using \( \log \log n \) bits.

\( F_\infty \)

Let’s say we want to find top \( k \) elements. The following algorithm that handles this problem was discovered several times independently [DLOM03, KSP03, MG82].

**Algorithm 5 Batch Decrement**

**Init:** \( L \)

Allocate \( L \) counters \( C_1, \ldots, C_L \leftarrow 0, \ldots, 0 \) //each counter can be allocated to different \( a_i \)’s, at different time.

Given \( a_i : \)

\[
\begin{cases} 
C_j \leftarrow 1 & \text{if } a_i \text{ has a counter } C_j \\
\text{allocate to } a_i, C_j \leftarrow 1 & \text{else, if exists a free counter } C_j \\
\text{reduce all counters by 1 and free zero counters} & \text{otherwise}
\end{cases}
\]

**Analysis** Let’s denote \( B \) as number of bad events (meaning, decreasing all counters by 1), and \( C(t) = \text{sum of all counters at time } t \). Note: \( C(t) \) grows by 1, or decreases by \( L \).

\( B \leq \frac{N}{L} \), since \( C(t) \) is always non-negative, and can be incremented at most \( N \) times, it can be decremented at most \( N/L \) times. This guarantees that \( a_i \) which appeared more than \( \frac{N}{L} \) times will have a counter allocated to it at the end, because it was incremented more than \( \frac{N}{L} \) times, and decremented at most \( \frac{N}{L} \) times.
References


