Lecture 7:
Adversarially Robust Streaming Algorithms

DS-563 / CD-543 @ Boston University
Instructor: Krzysztof Onak

Spring 2024

1 What we want to compute

Input: A stream of at most m elements from $[n] = \{1, \ldots, n\}$. For simplicity, $m = O(\mathrm{poly}(n))$.

Want to approximate: The number of distinct elements (a.k.a. F_0). Denote it $F_0(S)$ for a stream S. More specifically, for some parameter $\epsilon \in (0, 1)$, we want to output an estimate $\widehat{F}$ such that

$$(1 - \epsilon)F_0(S) \leq \widehat{F} \leq (1 + \epsilon)F_0(S).$$

Deletions: Sometimes deletions of items are allowed as well. In this setting, every stream item is of the form "insert x" or "delete x."

2 Adversarially robust streaming algorithms

Most streaming algorithms we have seen use randomness to "compress" the input into a small sketch that uses much less space than what would be needed to store the entire input. In fact, for many problems, including distinct elements, it is possible to show that an algorithm has to essentially store its entire input to be able to provide even approximate estimates if it is deterministic. Therefore, randomness is crucial for designing efficient streaming algorithms for many problems.

So far, any analysis we performed assumed that there is a fixed stream and we analyze the correctness of our estimates at the end of this stream. In some applications, however, we may want to provide continuous estimates as we go through the stream. (Think about, for instance, monitoring internet traffic.) This could be problematic if the stream is not fixed but future elements of the stream may depend on previous estimates. Estimates may leak information about the internal randomness of the algorithm. This information could be used by an adversarial actor to break our estimates. Alternately, we may want to use the streaming algorithm as a subroutine for another algorithm and this type of breakage could occur accidentally.

Is there a way to make our algorithms robust to this type of information leakage? In this and next lecture, we discuss various techniques that allow for achieving this goal without turning to space–inefficient deterministic algorithms.

We start by introducing a model which describes the challenge we are facing.

Model: a game between two players, the Algorithm and the Adversary.

In each round:

- First, the Adversary sends the next element of the stream to the Algorithm.
- Second, the Algorithm sends an updated estimate to the Adversary.

Outcome: The Adversary wins if *at least one* of the estimates sent by the Algorithm is not a $(1 \pm \epsilon)$-approximation to the current value.

Resources:

- The Algorithm: as small space as possible (it's a streaming algorithm after all!)
- The Adversary: can base next updates on her own random coin tosses and previous estimates of the Algorithm, does not know the Algorithm's personal coin tosses other than through the estimates that are made public

Goal: turn non-robust algorithms into robust

- An *non-robust* algorithm here is an algorithm about which we only know that it works with good probability for any fixed stream.
- An *(adversarially) robust* algorithm is an algorithm that, with good probability, provides good estimates in the adaptive setting, i.e., wins the game described above with good probability against any adversary.

3 Simple solution: use a different copy in each step

$\mathcal{A} =$ non-robust algorithm that provides $(1 \pm \epsilon)$-approximation w.p. $1 - \delta/m$

Solution: Keep and update m independent copies of $\mathcal{A}$. In round i, return the estimate from the i-th copy.

Problem: multiplicative overhead over $\mathcal{A}$ is $\Theta(m)$, so we could just store the entire input...

4 Technique 1: "Sketch/algorithm switching" (Insertion Only)

$\mathcal{A} =$ non-robust algorithm that provides $(1 \pm \epsilon/20)$-approximation w.p. $1 - \delta/m^2$

Algorithm 1: Insertion–only $F_0(S)$ approximation

```
estimate ← 0
index ← 1
t = O(ε^{-1} log m) independent copies A_1, ..., A_t of A
foreach stream item x do
    pass x to each A_i and process it independently
    if estimate from A_index ≥ (1 + ε/2)estimate then
        estimate ← estimate from A_index
        index ← index + 1
    output estimate
```

Observation (space usage): If all estimates of $\mathcal{A}_i$'s that we look at are good, then $\sim \log_{1+\epsilon/2} m = O(\epsilon^{-1} \log m)$ copies of $\mathcal{A}$ suffice because $F_0(S) \in \{0, \ldots, m\}$.
Why: The estimate we provide cannot increase too many times from 0 and then roughly 1 to roughly m, which bounds the maximum number of distinct elements in the stream.

To be continued in the next lecture