Randomized Algorithms 2020-1
Lecture 6

Streaming Algorithms, K-wise Independence, Card Guessing [*]

Moni Naor

We discussed some aspects of the memory checking problem, where a processor has small secret memory and access to a large memory of size n and the goal is to discover whether the memory malfunctions. The memory malfunctioning means that it does not return the correct value that was last stored at a given location (i.e. the last value stored at that location). The processor is given a sequence of read operations (get the value stored at location a) and write operations (store value v at location a) and should either perform them as given or declare that the memory malfunctioned (but should do the declare 'faulty' only if the memory indeed malfunctioned). The memory is controlled by an adversary and the processor has some secret memory and randomness, which the adversary cannot access. The secret memory of the processor should be as small as possible.

In your homework you discussed the offline solution which is a reduction to the multi-set comparison: the goal is to discover at the end of the sequence operations whether a malfunction occurred. The transformation considered in class involved adding to each cell in memory in addition to the value stored a "timestamp", which is the time when it was (last) written. In more details, the transformation involves:

- Scanning all memory cells at the beginning on the sequence and a writing a '0' is written in each cell.
- After each 'read' operation a 'write' operation is performed with *current_time* as the timestamp.
- Before each 'write' operation to location a, a 'read' operation to location a is performed.
- At the end of the sequence of operations the memory is scanned again and all cells are read.

Whenever a 'read' operation is performed and a timestamp t is returned, verify that $t < \mathit{current\_time}$ (otherwise it is a clear malfunction of the memory). Call the property that this is always the case the **monotone property**:

Consider the following two sets:

$$R = \{(v, a, t) | \text{location } a \text{ was read with value } v \text{ and timestamp } t\}$$

[*] These notes summarize the material covered in class, usually skipping proofs, details, examples and so forth, and possibly adding some remarks, or pointers. In the interest of brevity, most references and credits were omitted.

$$W = \{(v, a, t) | \text{location } a \text{ was written with value } v \text{ and timestamp } t\}$$

The algorithm suggested in class tested whether these two sets are equal and that the monotone property holds. We claimed that: If the monotone property holds, then $W = R$ iff the memory functioned properly.

The argument of why the reduction works is that since time grows, if at any point the read is wrong, its timestamp is smaller than the current time and the tuple will never be inserted to to the write set, since the time only grows.

Regarding the online case, without cryptographic assumptions, where the goal is to discover a wrong value *as soon as it is read*. We discussed first the case where you verify the full memory, say of size T. Here we want a family of hash functions H, s.t. for any two *different* $x, x' \in \{0,1\}^T$ the probability that $h(x) = h(x')$ is at most ϵ where $h \in_R H$. We want the size of H to be small as well as the range of H to be small, since the secret memory of the processor will store both the description of h and the value $h(x)$.

A pretty good example of such a family of hash functions is based (again) on polynomials. Take a finite field of size Q. Think of each vector as being mapped in a 1-1 manner to $A = a_0, a_2, \ldots a_\ell$ where $a_i \in GF[Q]$ (we must have that $Q^\ell \geq 2^T$). Now the hash function h_r is defined by a point r in the field $GF[Q]$ and the value of $h_r(A) = \sum_{i=0}^{\ell} a_i r^i$ (all operations are in the field).

The probability that $h_r(A) = h_r(A')$ (when $A \neq A'$) is bounded by ℓ/Q (the number of points that two different polynomials of degree ℓ can agree on). This means that to get the probability of collision to be bounded by ϵ you need Q to be about T/ϵ. So the storage needed is $O(\log T + \log(1/\epsilon))$.

Now we want to verify a memory of size n, we will divide it into n/T words, each of size T bits and authenticate each word separately. This means that reading any place in memory implies reading T bits and storing S bits where

$$S = \frac{n}{T}(\log n + \log(1/\epsilon).$$

This is almost the best possible (see M. Naor and G. Rothblum, *The complexity of online memory checking*, Journal of the ACM 2009, who showed that $S \cdot T$ is necessarily $\Omega(n)$).

With cryptographic assumptions we can do better, but still there is a logarithmic cost, based on various tree structures. No good lower bounds are known here, so it is an open problem.

1 Frequency Moments

We saw an elegant algorithm for computing F_2. The algorithm requires a four-wise independent hash function, that is a family of functions H mapping the universe to $\{-1, 1\}$, s.t. for any four different elements x_1, x_2, x_3, x_4 the values $h(x_1), h(x_2), h(x_3), h(x_4)$ are independent, where the randomness is over the choice of $h \in H$. We discussed such constructions and noted that if you map the universe to vectors over $GF[2]$ such that every four of them are linearly independent, then the function $h_R : U \mapsto \{1, 1\}$ defined by taking a random vector R and considering the inner product of R is four-wise independent.

In general we know that to achieve *exact* k-wise independence for even a 2 valued function (as needed in the F_2 algorithm) on a domain of size m requires $\log m + \lfloor k/2 \rfloor$ bits to describe h and this is tight.

2 Card Guessing

The scenario we are considering is where a deck of n distinct cards (for simplicity labeled $1, 2, \ldots n$) is shuffled and the cards from the deck are drawn one by one. A player called 'guesser' tries to guess the next card, for n rounds and gets a point for each correct guess. We are interested in the expected number of points the guesser can have.

Suppose that the guesser has *perfect memory* and can recall all the cards that it has seen, then what is expected number of correct guesses? At any point it picks one of the cards that have not appeared as a guess. If there are i cards, the probability of guessing correctly is $1/i$ and the expected number of guesses is

$$1 + \frac{1}{2} + \frac{1}{3} \ldots + \frac{1}{n} = H_n \approx \ln n.$$

Note that any guess of an unseen card has the same probability of success, so there is really not much of a strategy here.

Question: What can you say about high concentration in this case?

Now consider the opposite scenario, where the guesser has no memory at all. I.e. before it turns over a card it has no idea which cards have already appeared. But we will give it for free the round number. So the best strategy it may have is represented by a fixed guess g_i for the round i. The probability that this is correct is $1/n$, so the expectation over of all n rounds is 1.

Note: By guessing say '1' all the time the guesser can assure getting exactly 1.

Now suppose that the guesser has m bits of memory. That is it has a state M represented by m bits; whenever it sees a card that was turned over it moves to the next state and outputs a guess. What is the expected number of points now? How many bits do we need in order to achieve something like what the guesser with the perfect memory makes.

One observation is that we can pretend as if there are only m cards, i.e. ignore all cards of face value larger than m and have a perfect recall on the first s cards. This gives us $H_m \approx \ln m$. Is this the best possible? As we will see, you can do much better.

Remember the last k cards: It is possible to know for certain the last card using $\log n$ bits: keep track of the sum of the cards ($bmodn$). Just before the last card arrive can figure out the unique missing one. We can generalize this to the k last cards using $O(k \log n)$ bits. For instance, the polynomial from the set comparison protocol. We track the value of the polynomial at k

points, starting with the value of the points of the set $A = \{1, 2, \dots n\}$.[1] Can use this to get $H_{m'}$ on expectation for $m' = m/\log n$.

The two methods are compatible: of the m bits: use (say) $m/2$ for the first method and $m/2$ for the second one. The last card with face value in $1, ..., m/2$ is expected when there are $2n/m$ cards left. Or the expected number of cards with face value at most $m/2$ appearing in the the last $m/2 \log n$ cards is $\frac{m^2}{2} \log n < 1$. So for $m \leq \sqrt{n}$ the two useful periods do not overlap, with reasonable probability. We get $\min\{2H_{m'}, H_n\}$. For $s = \sqrt{n}$ this is almost as much as for $m = n$.

3 Homework

1. Consider the simultaneous message model for evaluating a function $f(x, y)$: Alice and Bob share a random string. They receive inputs x and y respectively and each should send a message to a referee, Charlie, who should evaluate the function $f(x, y)$. They may also have their own private source of randomness. The goal is for Alice and Bob to send short messages to Charlie.

   We will consider the equality function.

   (a) Consider first the case that Alice and Bob share a random string. The protocol where Alice and Bob send to Charlie an inner product of their input with a common random string r and Charlie decides based on the equality of the messages he receives. What happens to this protocol if Eve, who selects the inputs and whose goal is to make Charlie compute the wrong value, knows the common string r when she selects x and y?

   (b) Suggest a non-trivial protocol for this case, i.e. one whose communication complexity is sublinear in the input length.

2. Consider a deck of cards where half of the cards are red and half are black. The deck is shuffled and the cards turned one by one. At any point in time you can stop and guess that the next card is 'red'. What is the probability of success of best strategy?

   Hint: this is an easy problem...

[1] In more detail, consider the polynomial $P_A(x) = \Pi_{a \in A}(x - a)$ on the set $A = \{1, 2, \dots, n\}$ over a finite field of size at least $n + k + 1$. The algorithm is:

- Pick $k + 1$ many points x: $n + 1, n + 2, \dots, n + k + 1$, Evaluate $P_A(x)$ at these points and store each value separately.
- As card y goes by: divide the value at point $x = n + i$ by $(n + i - y)$.
- When k many cards left: reconstruct the k degree polynomial using the $k+1$ points, and recover the remaining values.