CS787: Advanced Algorithms

Scribe: Yiying Zhang, Dan Rosendorf **Lecturer:** Shuchi Chawla
Topic: Streaming algorithms (contd.) and Online algorithms **Date:** October 29, 2007

19.1 Introduction

In the last lecture we talked about streaming algorithms and using unbiased estimators and sampling to solve problems in that setting. Recall the setting of streaming algorithms. We have a stream of elements indexed 0 to T where T is some large number and the elements are drawn from $[n]$ where again n is large. We have limited space and time to work with any element typically to $O(poly(log(n), log(T)))$. Also recall that we define $m_i = |\{j : a_j = i\}|$ to be the frequency of element i.

19.2 More on sampling

For our sampling approach we want to take some k-elements at random from $[n]$ and keep only track of frequencies of the elements in this sample at the end extrapolating data from it. The problem with this approach is that sometimes we have $[n]$ too large to be able to choose a good sample ahead of time since we expect to only see a small subset of $[n]$ to show up in our stream. An example of this are IP addresses on a router. In such a case we really wish to choose our sample only from the distinct elements we actually see.

Let us define $S = \{i \in [n] | m_i \geq 1\}$ to be the set of all elements that occur in our stream. Also $\forall t \leq T$ define $S_t = \{i \in [n] | \exists j \leq t(a_j = i)\}$ the set of all elements seen up to time t. To make the problem slightly easier we will limit ourselves to try and choose an element e_t uniformly at random from S_t.

To do this assume first we have a perfect hash function $h : [n] \to [n]$. Then we can can just let $e_t = argmin_{i \in S_t} h(i)$. Since we don't actually have a perfect hash function though we will need to use a family of hash functions and we will want them to be minwise independent.

Definition 19.2.1 *We say a family of m hash functions $h_i : [n] \to [n]$, $i \leq m$ is minwise independent if $\forall X \subset [n]$ and $x \in X$,*

$$\Pr_{1 \leq i \leq m}[h_i(x) = \min(h(X))] = \frac{1}{|X|}.$$

Unfortunately no such family of size $O(log(n))$ is known so we shall need to settle for a slightly weaker condition called ϵ-minwise independent.

Definition 19.2.2 *We say a family of m hash functions $h_i : [n] \to [n]$, $i \leq m$ is ϵ-minwise independent if $\forall X \subset [n]$ and $x \in X$,*

$$\Pr_{1 \leq i \leq m}[h_i(x) = \min(h(X))] = \frac{1}{|X|}(1 \pm \epsilon).$$

Since such families are known to exist of size $O(log(n)log(1/\epsilon))$ we have the necessary tool we need to choose uniformly at random from S_t. We need only keep track of our element with least hash and at each t check whether $h(a_t) < h(e_{t-1})$ and if so let $e_t = a_t$ otherwise do nothing. If we need to keep a sample of k distinct elements we need only use k different hash functions.

19.3 Online algorithms

In this section we will talk about online algorithms. In the case of online algorithms we again have some data coming to us in a stream but in this case the basic problem is we need to make decisions before we see all of the stream. That is we are trying to optimize but we need to commit to partial steps during the way. To make this concept clearer let us look at an example.

19.3.1 Rent or Buy (Ski rental)

In the *rent or buy* problem sometimes also called the *ski rental* problem we need to figure out whether to rent or buy skis. Suppose we go skiing every winter and thus need skis. We can either buy a pair for some cost B and then use them forever, or we can rent them every winter for cost R but then we will have to rent them again when we want to go skiing next year. If we know how many times we will go skiing in our lifetime it is easy to choose whether to buy or rent just by comparing the total cost of each option, but we do not actually know how many times we will go until it's far too late. Thus we need to come up with another solution.

An easy algorithm is to just rent every winter until we have payed B in rent and then buy the skis once that point comes. While this algorithm doesn't seem very smart the amount we pay is always within a factor of 2 from the optimal solution. Since if we go skiing fewer times then $\frac{B}{R}$ then we have actually chosen the optimal course of action and if not then the optimal solution is to buy the skis in the first place so the optimal cost is B and we have paid exactly $2B$ (assuming $\frac{B}{R}$ is integer).

Note that the 2 here is not an approximation ratio it is something we call a competitive ratio. That is the ratio of our cost to the best cost we can get given that we would actually know everything in advance.

Definition 19.3.1 *Assume we have a problem and some algorithm Alg such that given an instance I of our problem $ALG(I)$ gives the cost of the solution Alg comes up with. Furthermore assume $OPT(I)$ is the cost of the best possible solution of the problem for instance I. Then*

$$\max_I \frac{ALG(I)}{OPT(I)}$$

is called the competitive ratio of the algorithm Alg.

There is also an alternate definition which is sometimes called the strong competitive ratio.

Definition 19.3.2 *Assume the same as above, then we also call r the competitive ratio, if*

$$ALG(I) \leq r\, OPT(I) + c$$

holds for all I and some constant c independent of I. When c is 0, r is the strong competitive ratio.

19.3.2 List Update

One important application area of online algorithms is maintaining data structures efficiently. With incoming search requests and updates over time, we want to develop an algorithm that could dynamically adjust the data structure so as to optimize search time. Think of a binary search tree with n elements, the search time is generally no more than $\log n$. But if we know the frequency of search terms, we could change the structure of the tree so that the more frequent elements are at a higher level, and thus we have a better search term.

Here we study an online algorithm for an easier data structure, *list*. We have a list of size n. For simplicity, we also assume that there are no insertion or deletion to the list. Over time, we have a sequence of requests. At every step, the algorithm gets a request for element i. We pay a cost i to move to position i. There is no cost on moving i forward to any position before its original position. And there is a cost of 1 for a single transposition of i with a position after it. Our goal is to minimize the total cost over time.

We observe that if the distribution of input search terms is known ahead, we can order the list from most frequently requested element to the least frequently requested one. The problem is that we do not know this distribution ahead of time. We need to develop some online algorithm (with no prior knowledge of input distribution) to minimize the total cost.

Algorithm 1 (Move-To-Front): The first intuition is that every time when we see an element we move it to the front of the list. This seemingly naive algorithm turns out to have quite good performance (2-competitive ratio as wee see later).

Algorithm 2 (Frequency-Ordering): In this algorithm, we order elements in decreasing order of frequency of searched elements so far. At each step, when we see an element, we add one to its frequency and if necessary move it forward to a appropriate position. This algorithm seems to be better than the previous one, however it turns out to have poor performance ($\Omega(n)$-competitive ratio as we see later).

Algorithm 3 (Move-Ahead-One): In this algorithm, we move an element forward one position every time we see it. This algorithm also performs poorly ($\Omega(n)$-competitive ratio as we see later).

Now we give proof of the competitive ratios of these three algorithms. The analysis of online algorithms often include the process of a competing adversary trying to find a input on which the algorithm works poorly but there exists a good optimal algorithm (with prior knowledge).

Claim 1: Move-Ahead-One has $\Omega(n)$-competitive ratio.

Proof: Suppose the list is $1, 2, ..., n$. A bad input sequence (one that the adversary comes up) is $n, n-1, n, n-1, n, n-1, ...$ Move-Ahead-One will repeatedly move element n to position $n-1$, then move element $n-1$ to position $n-1$, then move element n to position $n-1$... This gives a cost of n at each step. If we know the input sequence, an optimal solution will be put element n and $n-1$ at the front two positions, giving a cost of $O(1)$ at each step. This shows that Move-Ahead-One has $\Omega(n)$-competitive ratio. ■

Claim 2: Frequency-Ordering has $\Omega(n)$-competitive ratio.

Proof: The bad input sequence the adversary constructs is as follow. In the first $\binom{n}{2}$ steps, element

i is requested $n-i+1$ time in the order from 1 to n, i.e. 1 is requested n times, then 2 is requested $n-1$ times ... n is requested 1 time. Notice that the cost of these steps for Frequency-Ordering is the same as the optimal algorithm with knowledge of input sequence.

¿From then on, every element is requested n times in the order from n to 1. After the first $\binom{n}{2}$ steps, Frequency-Ordering algorithm gives the list $1, 2, ..., n$. Then when element n is requested n times, it moves from the tail to the front of the list, costing totally $n(n+1)/2 = O(n^2)$. When element $n-1$ is requested for the first time, it is at the tail of the list. For n requests of $n-1$, the element moves from tail to front, costing $n(n+1)/2 = O(n^2)$. This analysis applies to all the n elements, giving a total cost of $O(n^3)$. However, if we know the input sequence, we can construct the optimal algorithm to bring element n to front the first time it is requested after the first $\binom{n}{2}$ steps, which cost $O(n)$ for element n. The same analysis apply to the rest of the elements, giving a total cost of $O(n^2)$. This shows that Frequency-Ordering gives $\Omega(n)$-competitive ratio. ■

Claim 3: Move-To-Front has 2-competitive ratio.

Proof: To prove this claim, we first compare Move-To-Front to a static list (say $1, 2, ..., n$), and show the competitive ratio between them is 2.

Here we use an idea similar to amortized analysis. The idea is that we pay some cost for the elements being affected by a move-to-front act. Later on when we search for an element, we use the money we paid for it before. More specifically, when we move an element i at position p to the front, the $p-1$ elements originally before position p are moved backward for 1 position. Thus, we pay \$1 cost for each element that is at a position before p in the Move-To-Front list before moving i and is at a position before i in the static list, i.e. whose value is smaller than i. Notice that we have at most $i-1$ such elements. We get \$1 cost for each element that is at a position before p in the Move-To-Front list before moving i and is at a position after i in the static list. We have at least $p-i$ such elements. The amount money we pay can be also thought of as a potential function, which is always non-negative.

Now let TC_t be the cost of Move-To-Front plus the total amount of money we pay after the first t steps. We have $TC_0 = 0$. For step t, the amount of money we pay is no more than $i-1$ and the amount of money we get is no less than $p-i$. The cost of search element i in the static list is i. Therefore,

$$TC_t - TC_{t-1} \leq p + (i-1) - (p-i) = 2i - 1$$

$$TC_t - TC_{t-1} \leq 2 * (cost\ of\ static\ list\ at\ step\ t)$$

When we sum up $TC_t - TC_{t-1}$ over t, we have

$$TC_t \leq 2 * (total\ cost\ of\ static\ list)$$

This shows that Move-To-Front has 2-competitive ratio as compare to a static list.

Now we generalize this analysis to comparing Move-To-Front to an arbitrary algorithm. Let σ be the sequence of requests. Let $OPT(\sigma)$ be the optimal algorithm for sequence σ. At step t, consider

Move-To-Front's list L_{MTF} and optimal algorithm's list L_{OPT}. We define $\Pi(L_{MTF})$ as the number of different positions in the Move-To-Front list and in the optimal list.

$$\Pi(L_{MTF}) = |\{(x, y)|x \textit{ is ahead of } y \textit{ in } L_{MTF} \textit{ and } y \textit{ is ahead of } x \textit{ in } L_{OPT}\}|$$

At step t, consider the difference between step $t-1$ and t, $\Delta cost_{MTF} + \Delta\Pi(L_{MTF})$. Here, $\Delta\Pi(L_{MTF})$ is the potential function, i.e. the amount of money we pay at step t. Suppose at step t, element i is searched and it is at position p in L_{MTF} and at position p' in L_{OPT}.

Now consider the first case when optimal algorithm doesn't move i or move to a position that is before its original position. The cost we pay for moving i to the front in Move-To-Front is for the elements that are before i in L_{OPT} and are originally before i in L_{MTF}. There are at most $p' - 1$ such elements. We get money for elements that are after i in L_{OPT} and are originally before i in L_{MTF}. There are at least $p - p'$ such elements. Thus, we have

$$\Delta\Pi(L_{MTF}) \leq (p' - 1) - (p - p') = 2p' - p - 1$$

If optimal algorithm exchange i with some position behind, we gain in addition the cost of exchanging positions in the optimal algorithm. Therefore, we have

$$\Delta\Pi(L_{MTF}) \leq 2p' - p - 1 + (\textit{number of paid exchange})$$

And the cost of searching i in L_{MTF} is p. All together, we have

$$\Delta cost_{MTF} + \Delta\Pi(L_{MTF}) \leq p + 2p' - p - 1 + (\textit{number of paid exchange})$$
$$\Delta cost_{MTF} + \Delta\Pi(L_{MTF}) \leq 2p' - 1 + (\textit{number of paid exchange})$$

When we sum this up over t, we have $(\textit{Total cost of } MTF) \leq 2 * (\textit{total cost of } OPT)$. This gives a 2-competitive ratio. ■

In fact, 2-competitive ratio is the best for any deterministic algorithm. However, using randomized algorithms, we can reach a 1.6-competitive ratio. We can briefly reason the advantage of randomized algorithm being that the adversary cannot see the output of a randomized algorithm and thus cannot come up a competing input sequence, while for deterministic algorithms, the adversary can observe the output and come up with a competing input sequence.

Next time, we will see another example of online algorithm, namely the cashing problem. We will develop both a deterministic algorithm and a randomized algorithm for it.