## Lecture 11: Online Problems

*Reference: Online Computation and Competitive Analysis*

Borodin and El-Yaniv, Chapters 1 and 2

Until now we have been dealing with off-line algorithms, i.e., algorithms in which we had total information about the inputs. Our difficulty was that the problems we were trying to solve were (NP) Hard and we couldn't construct exact solutions in polynomial time. We therefore found polynomial time algorithms to construct solutions that were provably within a given approximation factor of optimal.

We are now going to change the type of problems we examine and switch to online algorithms. These are algorithms such as paging, some types of scheduling, etc., that need to make decisions without full knowledge of the input. They have full knowledge of the past but no (or inexact) knowledge of the future. For this type of problem we will attempt to design algorithms that are *competitive* to within a constant factor of the optimum offline algorithm, the algorithm that has perfect knowledge of the future.

Our major example of such a problem will be the *List-Accessing* problem.



### The Ski-Rental Problem

In this problem you are taking lessons to learn how to ski. After each lesson (depending upon how much you enjoy it) you will decide whether to continue to learn or to stop totally.

You have the choice of either *renting* skis at $\$x$ a time or buying skis for $\$y$ where $y > x$. If you knew in advance how many times $t$ you would ski in your life then the choice of whether to rent or buy is simple. If you will ski more than $y/x$ times then buy before you start, otherwise always rent. The cost of this *algorithm* to you is $\min\{tx, y\}$.

This type of strategy, with perfect knowledge of the future, is known as an *offline* strategy.

Now suppose you don't know how many times you will ski. What should you do? A *online strategy* will be a number $K$ such that after renting $K$ times you will then buy skis.

Setting $K = \lfloor y/x \rfloor$ guarantees that you never pay more than twice the cost of the offline strategy (why?). We say that this strategy is 2-*competitive*.



On the other hand let $K$ be any strategy. Suppose you buy the skis after renting $K$ times and then break your leg and never go skiing again. Your total ski cost is $Kx + y$ and the optimum offline cost is $\min\{Kx, y\}$. This implies that for every online strategy there is a case in which you will pay at least twice as much as the optimum offline strategy. This says that every strategy is *at least* 2-competitive. That is, the strategy on the previous page is the best one.



### The List Accessing Problem

In this problem we are given a (linked) list

$$x_1 x_2 \dots x_l$$

and a sequence $I$ of requested accesses

$$I = \sigma_1 \sigma_2 \dots \sigma_n$$

where $\forall i$, $\sigma_i \in \{x_1, x_2, \dots, x_l\}$. The *cost* of accessing $\sigma_i$ is the number of items that have to be traversed to get to that item. This is just the location of the item in the list counted from the front. Given $I$, our objective is to minimize the cost of accessing the items in the sequence in the given order.

While processing the accesses we are allowed to modify the list in two ways

1. **free transpositions:** after accessing item $\sigma_i$ in the list we can, for free, move it anywhere we want to closer to the front of the list.
2. **paid transpositions:** at any time we can swap two adjacent list items by paying one dollar.

Given an algorithm $A$, $A(I)$ will be the total cost (access costs plus paid transpositions) paid by the algorithm for performing all the accesses in $I$.

$OPT(I)$ will be the minimum cost necessary to pay to perform $I$. The algorithm that we will use to do this will have perfect knowledge of $I$ from the start. Such an algorithm is known as an *offline* algorithm.

Algorithm $A$ will be called *c-competitive* if, for some fixed $\alpha$, for all access sequences $I$

$$A(I) \leq c \cdot OPT(I) + \alpha.$$

An Algorithm $A$ will be called *online* if, at the time it performs the $i$th access it only knows $\sigma_1\sigma_2 \ldots \sigma_i$ but not $\sigma_{i+1}\sigma_2 \ldots \sigma_n$.

In the earlier section of the class finding the optimum was never the problem. Finding the optimum in *polynomial time* was the problem. Since doing that was (probably) impossible we settled for finding an approximation to $OPT$ in polynomial time.

In this section things are different. As we will see, online algorithms, because they lack information about the future, will never be as good as OPT. Our goal will be to construct on-line algorithms that are good in the sense of being *c-competitive* with small $c$.



**Some Online Algorithms**

These are algorithms that make their decision as to what to do based only upon past history of accesses performed. They have no knowledge of the future.

1. **MTF (Move-to-Front):** After accessing item $\sigma_i$ it moves $\sigma_i$ to the front of the list.
2. **TRANS (Transpose):** After accessing item $\sigma_i$ it moves $\sigma_i$ one unit forward towards the front of the list.
3. **FC (Frequency Count):** Each item has a counter that stores number of times it's been accessed. Items are stored in non-decreasing order of accesses; after item is accessed its frequency counter is updated and item moved forward (if necessary) to maintain list order.

We will prove the following two facts:

Theorem: Let $A$ be any offline algorithm.
If $A$ is $c$-competitive then $c \geq 2 - \frac{2}{l+1}$.

Theorem: MTF is $2 - \frac{1}{l}$ competitive.



Theorem: Let $A$ be any offline algorithm. If $A$ is $c$-competitive then $c \geq 2 - \frac{2}{l+1}$.

Proof: Let $A$ be any offline algorithm. We use a *cruel strategy* to construct a "bad" access sequence for $A$. Start with any arbitrary ordering of the list and set $\sigma_1$ to be the last item on the list. From then on the cruel strategy will always request the last item on the list. If this is continued for $n-1$ steps the access sequence $I$ constructed will have cost $A(I) = nl$.

We now construct a simple offline strategy STAT. It starts by simply sorting the $l$ elements in non-decreasing order of frequency; this can be done using less than $l^2/2$ paid transpositions. After that STAT never moves any more elements; the cost of accessing the $i$th most frequent element will thus be $i$.



Our analysis needs the following fact.
Lemma: Let $p_1 \geq p_2 \geq \cdots \geq p_l \geq 0$ s.t. $\sum_{i=1}^{l} p_i = 1$. Then

$$\sum_{i=1}^{l} i \cdot p_i \leq \frac{l+1}{2}$$

where equality only occurs if $\forall i$, $p_i = \frac{1}{l}$.

Now let $f_i$ be the number of times item $i$ is accessed by $I$; WLOG $f_1 \geq f_2 \geq \cdots \geq f_l \geq 0$. For all $i$ set $p_i = \frac{f_i}{\sum_{i=j}^{l} f_j} = \frac{f_i}{n}$. Then

$$\begin{aligned} OPT(I) \leq STAT(I) &\leq \frac{l^2}{2} + \sum_{i=1}^{l} i \cdot f_i \\ &= \frac{l^2}{2} + n\left(\sum_{i=1}^{l} i \cdot p_i\right) \\ &\leq \frac{l^2}{2} + n\frac{l+1}{2} \\ &= \frac{l^2}{2} + nl\left(\frac{l+1}{2l}\right) \end{aligned}$$

This implies that

$$\left(2 - \frac{2}{l+1}\right) OPT(I) - \frac{l^3}{l+1} \leq nl = A(I)$$

so $c \geq 2 - \frac{2}{l+1}$.

Theorem: MTF is $2 - \frac{1}{l}$ competitive.

To prove this we will need to introduce some new notation. Given algorithm $A$, $A_P(I)$ is the number of *paid* transpositions the algorithm uses when running on $I$, $A_F(I)$ the number of *free* transpositions, and $A_C(I)$ the cost of the accesses. Then $A(I) = A_P(I) + A_C(I)$.

Note that MTF, TRANS and FC use no paid transpositions so for them $A(I) = A_C(I)$.

We will actually prove the following:
Theorem: Let $I$ be an access sequence of $n$ requests and $OPT$ and $MTF$ start with the same list. Then

$$MTF(I) \leq 2 \cdot OPT_C(I) + OPT_P(I) - OPT_F(I) - n$$

Note that this theorem implies $MTF(I) \leq 2 \cdot OPT(I) - n$. Since $OPT(I) \leq nl$ this implies

$$MTF(I) \leq \left(2 - \frac{1}{l}\right) OPT(I)$$

proving the original theorem.



Theorem: Let $I$ be an access sequence of $n$ requests and $OPT$ and $MTF$ start with the same list. Then

$$MTF(I) \leq 2 \cdot OPT_C(I) + OPT_P(I) - OPT_F(I) - n$$

Let $L_i$ be the MTF list after the $i$th step and $L'_i$ the OPT list after the $i$th step. We will define a function $\Phi(i) = \Phi(L_i, L'_i)$ that will only be dependent upon the current state of those two lists such that $\forall i$, $\Phi_i \geq 0$ and $\Phi_0 = 0$. $\Phi$ is known as a *potential* function,

Let $t_i$ be the actual cost that the MTF algorithm pays servicing the $i$th request. We set

$$a_i = t_i + \Phi_i - \Phi_{i-1}.$$

$a_i$ is known as the *amortized* cost. The important property is that

$$MTF(I) = \sum_{i=1}^{n} t_i = \sum_{i=1}^{n} (a_i + \Phi_{i-1} - \Phi_i) = \sum_{i=1}^{n} a_i - \Phi_n \leq \sum_{i=1}^{n} a_i.$$

By bounding the $a_i$ in terms of the cost of OPT on $\sigma_i$ we can therefore bound $MTF(I)$ in terms of $OPT(I)$.

We now define the function $\Phi_i$ to be the number of *inversions* between $L_i$ and $L'_i$. An inversion is a pair $(j, k)$ such that $x_j$ is before $x_k$ in $L_i$ but $x_j$ is after $x_k$ in $L'_i$.



Theorem: (Continued) Let $I$ be an access sequence of $n$ requests and $OPT$ and $MTF$ start with the same list. Then

$$MTF(I) \leq 2 \cdot OPT_C(I) + OPT_P(I) - OPT_F(I) - n$$

Suppose now that the two algorithms have just completed processing request $i - 1$. WLOG we assume that for $1 \leq j \leq l$, the $j$th item in OPTs list is $x_j$. We now show that

$$a_i \leq (2s - 1) + P - F$$

where $s$ is the search cost by OPT on $\sigma_i$, $P$ the number of paid transpositions it makes and $F$ the number of Free ones.

Assume now that $\sigma_i = x_j$. By definition $x_j$ is the $j$th item in OPT's list ($L'$) so $s = j$. Suppose $x_j$ is in position $k$ in MTF's list ($L$) so $t_i = k$.

Let $v$ be the number of items that appear in before $x_j$ in $L_{i-1}$ but after $x_j$ in $L'_{i-1}$. Then $k - 1 - v$ items appear *before* $x_j$ in *both* lists. Since there are only $j - 1$ items preceding $x_j$ in $l'$ this implies $k - 1 - v \leq j - 1$ or

$$k - v \leq j.$$



Recall that $\Phi_i$ counts the number of inversions between the two lists at time $i$. $\Phi_i - \Phi_{i-1}$ therefore counts the number of *new* inversions minus the number of *old inversions destroyed*.

We now count the change in the number of inversions that occurs when MTF moves $x_j$ to the front of $L$ (but before OPT does anything). Moving $x_j$ to the front creates $k-1-v$ *new* inversions and destroys $v$ old ones. Taking into account $t_i$, the changes so far to the amortized cost will be

$$k + (k - 1 - v) - v = 2(k - v) - 1 \leq 2j - 1 = 2s - 1.$$

Now it's OPT's turn to change the list. Note that every paid transposition can cause the number of inversions to increase by at most 1 (it can also decrease the inversions) and every free transposition decreases the number of inversions by 1 (why?). Thus

$$a_i + \Phi_i - \Phi_{i-1} \leq k + (k - 1 - v) - v + P - F = 2s - 1 + P - F$$

and we are done.

## Randomization

We showed that no deterministic algorithm is better than 2-competitive.

We now extend our definition to randomized algorithms. A randomized algorithm $A$ is one that takes random choices (flips a coin). The *expected cost* of $A$ on access sequence $I$ is $E[A(I)]$.

**BIT:** For each element $x$ on the list associate a bit $b(x)$. before starting set each $b(x)$ to a value in $\{0, 1\}$ randomly uniformly and independently. Whenever element $x$ is accessed *flip* $b(x)$ from 0 to 1 or vice-versa. Finally, if $b(x) = 1$ move $x$ to the front and if $b(x) = 0$ do nothing.

In chapter 2 of the Borodin/El-Yaniv book it is proved that BIT breaks the 2-barrier. Note that it can do this because it uses randomization.

Theorem: If $I$ is a sequence of $n$ accesses

$$E[BIT(I)] \leq \frac{7}{4} OPT(I) - \frac{3n}{4}.$$

The proof uses an amortization technique with a slightly modified potential function.



## Paging

Consider a computer system that contains a fast memory with space for $k$ pages and a slow device (disk) with space for $n$ pages, $n >> k$.

When a page not in fast memory is referenced a *page-fault* occurs and everything stops while the page is loaded into main memory. The system's *paging algorithm* is what decides which page to evict to make space for the new space. The goal of the paging algorithm is to minimize the number of page faults.

This problem can be considered in the same way as list accessing. n access sequence $I = \sigma_1 \sigma_s \ldots \sigma_m$ is the sequence of pages requested and the cost of a paging algorithm is the number of page faults it incurs.

Fact: The offline algorithm **Longest-Forward-Distance (LFD)**, that throws away the page whose next use is furthest in the future, is an optimal paging algorithm.

We will see the following two facts.
Theorem: No Deterministic online algorithm has competitive ration less than $k$.

Theorem: The **Least Recently Used (LRU)** algorithm which evicts from fast memory the page least recently accessed is $k$ competitive.



Theorem: No deterministic online algorithm has competitive ration less than $k$.

Proof: let $A$ be a deterministic algorithm and $n = k+1$. Let $I$ be the sequence constructed by always referencing the page *not* in the fast memory. Then $A$ faults on every reference so $A(I) = m$.

LFD, which throws away the page whose next use is furthest in the future will only fault *at most* one out of every $k$ references (since $n = k + 1$) so $OPT(I) \leq m/k$ and

$$A(I) \geq k \cdot OPT(I).$$



Theorem: The **Least Recently Used (LRU)** algorithm which evicts from fast memory the page least recently accessed is $k$ competitive.

Proof: Partition the access sequence $I$ into

$$I = I_0 I_1 I_2 \ldots I_i$$

in which LRU faults *exactly* $k$ times in each of $I_1, , I_2, \ldots I_i$ and *no more* than $k$ times in $I_0$. For $1 \leq j \leq i$, let $p_{j-1}$ be the last page referenced in $I_{j-1}$.

Note that during $I_j$ algorithm $LRU$ cannot fault on any distinct page twice and can not fault on $p_j$ at all. This implies that $I_j$ contains references to $k$ different pages all of which are also different than $p_j$ (which was in the fast memory when $I_j$ started. Thus *any* paging algorithm, including the optimal one, must fault at least once on $I_k$ and

$$OPT(I) \leq K \cdot LRU(I) + k$$

so LRU is $k$-competitive.

**Randomized Paging Algorithms**

As in the list access case our algorithms can do slightly better if we allow randomness,

**MARK:** Start out with all pages marked.
If there is a request for $p$ already in the fast memory, if $p$ is unmarked then mark it.
If there is a request for $p$ not in fast memory bring it into the memory, evicting a random *unmarked* page. If there is no unmarked page then unmark *all* of the pages in fast memory and evict one of them at random.

(Under certain reasonable conditions!!!)
Theorem: If $I$ is any request sequence

$$E[MARK(I)] \leq 2H_k \cdot OPT.$$

There is an almost matching lower bound.
Theorem: If $A$ is *any* randomized algorithm and $n \geq k+1$ then the competitiveness of $A$ is not less than $H_k$.

(Note: There is a more complicated algorithm than MARK that achieves the $H_n$ bound.