Online Algorithms

Introduction

So far in class, we have only considered the following algorithm design: we are given an input I, are allowed to perform some computation, and then produce an output O. For many problems, this is an appropriate framework. However, there are also many cases in the real world in which the algorithm does not know the entire input yet, but still has to make partial decisions about the output.

Algorithms which have to make their decisions gradually as data arrives are called **online algorithms**. For instance, the cache should contain frequently accessed items. Unfortunately, we do not know which items will be accessed. Other examples include: scheduling problems, traffic routing in networks, and more.

We define the **competitive ratio** for online algorithms to capture how much worse the algorithm does compared to one that knows about the future.

Definition. Let OPT be the optimum offline algorithm (knowing about the future), and ALG our online algorithm. Let C_{OPT} and C_{ALG} denote the costs incurred by those two algorithms. The competitive-ratio (CR) is defined as

$$\text{CR} = \frac{C_{\text{ALG}}}{C_{\text{OPT}}}$$

ALG is c-competitive if there is a constant α such that for any OPT and all inputs σ

$$C_{\text{ALG}}(\sigma) \leq c\, C_{\text{OPT}}(\sigma) + \alpha$$

The Ski Rental Problem

Say you are just starting to go skiing. Can either rent skis for \$50 a day or buy them for \$500. What to do, buy or rent?

Consider the worst case - you bought the skis right away, but then lost your interest

$$\text{CR} = \frac{C_{\text{ALG}}}{C_{\text{OPT}}} = \frac{500}{50} = 10$$

Another case, you decide to rent. Then you decide to go skiing again, and again, and after a while you realize you had rather bought skis right at the start.

$$\text{CR} = \frac{C_{\text{ALG}}}{C_{\text{OPT}}} = \frac{d\,50}{500} = \frac{d}{10}$$

Optimal strategy is: if you know you're going to end up skiing more than 10 days, you should buy right at the beginning. If you know you're going to go < 10 days, you should just rent. But, what if you don't know?

In the Ski Rental Problem, we assume that we are going skiing for some number d of days total. Each day, we can either rent skis for R dollars, or buy them for $B > R$ dollars. Once we have bought the skis, we can use them for free forever afterwards.

Our deterministic algorithm can be described as "rent for d days, then buy". Executing this algorithm gives us a cost of $B + d\,R$. Next, we compute the cost of the optimum algorithm, which is $\min(B,\ (d+1)\,R)$

$$\text{CR} = \frac{d\,R + B}{\text{MIN}(B,\ (d+1)\,R)} = \text{MAX}\left(1 + \frac{d\,R}{B},\ 1 + \frac{B-R}{(d+1)\,R}\right)$$

$$\text{CR} = 1 + \text{MAX}\left(\frac{d\,R}{B},\ \frac{B-R}{(d+1)\,R}\right)$$

The first term in the maximum is monotone increasing in d, and the second monotone decreasing in d. Hence, the best ratio is achieved for the d for which the two terms are equal.

$$\frac{d\,R}{B} = \frac{B-R}{(d+1)\,R}$$

$$d\,(d+1) = -\frac{B}{R}\left(1 - \frac{B}{R}\right)$$

Solving for d, yields

$$d + 1 = \frac{B}{R}$$

This suggests that the optimum online algorithm is to buy on day number B/R. Substituting the value of d back, we obtain

$$\text{CR} = 1 + \frac{d\,R}{B} = 1 + \frac{R}{B}\left(\frac{B}{R} - 1\right) = 2 - \frac{R}{B}$$

<u>Claim</u>: this strategy has the best possible competitive ratio for deterministic algorithms.

For the above example (rent for 9 days, then buy)

$$\text{CR} = 2 - \frac{R}{B} = 2 - \frac{50}{500} = 1.9$$

What is we rented for 8 days, and then buy?

$$\text{CR} = \frac{C_{\text{ALG}}}{C_{\text{OPT}}} = \frac{8 \times 50 + 500}{9 \times 50} = 2$$

What is we rented for 10 days, and then buy?

$$\text{CR} = \frac{C_{\text{ALG}}}{C_{\text{OPT}}} = \frac{10 \times 50 + 500}{500} = 2$$

Proof.

Case 1: $C_{\text{OPT}} = B$ i.e $B \le (d+1)\,R$ or $B - R \le d\,R$

$$\text{CR} = \frac{B + d\,R}{\text{MIN}(B,\ (d+1)\,R)} = \frac{B + d\,R}{B} \ge \frac{B + B - R}{B} = 2 - \frac{R}{B}$$

Case 2: $C_{\text{OPT}} = (d+1)\,R$ i.e $B \ge (d+1)\,R$ or $\frac{1}{(d+1)\,R} \ge \frac{1}{B}$

$$\text{CR} = \frac{B + d\,R}{(d+1)\,R} = \frac{(d+1)\,R - R + B}{(d+1)\,R} = 1 + \frac{B-R}{(d+1)\,R} \ge 1 + \frac{B-R}{B} = 2 - \frac{R}{B}$$

■ The List Update Problem

Among the first papers to study online algorithms was one by Sleator and Tarjan, studying algorithms for online list accessing and paging.

Here, we focus on accessing the elements of a linked list of the size *n*. Specifically, if the *k*-th element of the list is accessed, then the cost incurred for this access is *k*. Immediately after the access, we can move that item to any position closer to the front of the list at no extra cost. This is called a free exchange. The algorithm can also exchange any two consecutive items at a cost of 1. These are called paid exchanges.

The goal is to devise and analyze an on-line algorithm for doing accesses with a small competitive factor.

Clearly, frequently accessed elements should be toward the front of the list. The problem, of course, is that we do not know which items will be requested in the future. In the absence of this knowledge, one could think of several natural online heuristics.

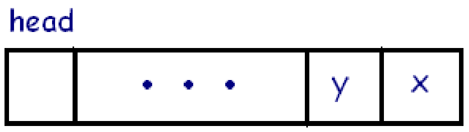
Transposition (TRANS): Always move the most recently accessed element one position forward, by swapping it with its neighbor.

Frequency Count(FC): Maintain a frequence of access for each item. Keep the list sorted by deacreasing fequency.

Move To Front (MTF): Always move the most recently accessed element to the front of the list.

To analyze the above heuristics, consider the following model: the request sequence is $\sigma(1)$, $\sigma(2)$, ... The values $\sigma(i)$ and the sequence length t are unknown. If $\sigma(i)$ is in position k, then the access cost is k. Afterwards, the element can be moved forward (closer to the front) for free.

Transposition (TRANS).



*W*e could have a sequence that always accesses the last element in the list: $\sigma(x)$, $\sigma(y)$, ... $\sigma(x)$, $\sigma(y)$, altogether t-pairs. This results in repeated swaps between the last two elements.

$$C_{\text{ALG}} = t\,n$$

The optimum solution could move the last two elements to front on the first call

$$C_{\text{OPT}} = 2\,n + 2\,t$$

The competitive ratio

$$\text{CR} = \frac{C_{\text{ALG}}}{C_{\text{OPT}}} = \frac{t\,n}{2\,n + 2\,t} = \Omega(n) \text{ as } t \to \infty$$

*Frequency Count(FC). W*e could construct a sequence in the following way: access the first element $k > n$ times, the second - $(k-1)$ times, and finally the last element - $(k-n+1)$ times. Observe, the FC heuristic will never reorganize the list. The cost is given by

$$C_{\text{ALG}} = k + 2\,(k-1) + 3\,(k-2) + \ldots + n(k-n+1) \geq (k-n)\,(1 + 2 + \ldots + n) = \Omega\left(k\,n^2\right)$$

The optimum solution would move each element to the front when it is accessed for the first time

$$C_{\text{OPT}} = k + [2 + (k-2)] + [3 + (k-3)] + \ldots = k\,n$$

$$\text{CR} = \frac{C_{\text{ALG}}}{C_{\text{OPT}}} = \frac{k\,n^2}{k\,n} = n$$

Theorem. *MTF is a 2-competitive algorithm*

$$C_{\text{MTF}} \le 2\, C_{\text{OPT}}$$

Proof.

We observe that if the lists of OPT and MTF are identical, then accesses to any element cost exactly the same. Thus, good moves by MTF are ones that make the lists more similar. An element in OPT has a cheap access, if the element is close to the front. Thus, moving the element to the front will make the lists much more similar in this case.

The amortized cost AC is the actual cost c_k plus a change in the potential $\Delta\Phi\ (s_k)$

$$\text{AC} = c_k + \Delta\Phi\ (s_k) = c_k + \Phi\ (s_k) - \Phi\ (s_{k-1})$$

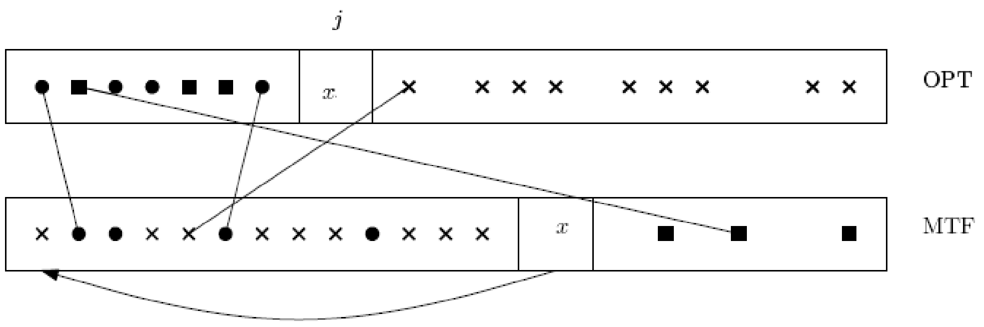
We will measure similarity of the lists with a potential function

$$\Phi(k) \;=\; \text{the number of inversions between OPT and MFT after accessing } k \text{ item}$$

An inversion is a pair of distinct elements that appear in one order in one list and in a different order in the other list. Example,

$$a,\ c,\ d,\ e$$
$$c,\ e,\ a,\ d$$

the number of inversions is 3. Observe that in the worst case the number of inversions between two arbitrary given lists is $\binom{n}{2}$. The following picture shows list configurations at access i, which is an item x:



Consider all elements before x in MTF and find them in OPT. Let

S = {y | y is before x in MTF and y is before x in OPT} - shown as solid circles.

T = {z | z is before x in MTF and z is after x in B} - shown as crosses.

Then we have

$$C_{\text{MTF}}(i) = S + T + 1$$

Next, find a change in potential $\Phi(i) - \Phi(i-1)$. Moving x to front will eliminate T inversions (crosses) and create new S inversion (solid circles). Thus,

$$\Phi(i) - \Phi(i-1) = S - T$$

But this is not all - OPT is also allowed to rearrange its list. Since OPT performs only paid exchanges, each paid exchange creates one inversion. Let P denote the number of paid exchanges performed by OPT, hence

$$\Phi(i) - \Phi(i-1) = S - T + P$$

Moreover,

$$C_{\text{OPT}}(i) = j + P \geq S + 1 + P$$

where j is a positon of x at access i. Putting it all together we get

$$\text{AC} = C_{\text{MTF}}(i) + \Phi(i) - \Phi(i-1) = S + T + 1 + (S - T + P) = 2\,S + 1 + P \leq 2\,C_{\text{OPT}}(i) - 1$$

Finally, we add up over all requests

$$\sum_{i=1}^{n} C_{\text{MTF}}(i) + \Phi(i) - \Phi(i-1) \leq \sum_{i=1}^{n} 2\,C_{\text{OPT}}(i) - 1$$

$$C_{\text{MTF}} + \sum_{i=1}^{n} \Phi(i) - \Phi(i-1) \leq 2\,C_{\text{OPT}} - \sum_{i=1}^{n} 1$$

$$C_{\text{MTF}} - \Phi(0) + \Phi(n-1) \leq 2\,C_{\text{OPT}} - n$$

$$C_{\text{MTF}} + \Phi(n-1) \leq 2\,C_{\text{OPT}} - n$$

$$C_{\text{MTF}} \leq 2\,C_{\text{OPT}} - n - \Phi(n-1) \leq 2\,C_{\text{OPT}}$$

■ The Cat && Mouse Game

There is one cat and one mouse which has n hiding places. A cat has a sequence of probes for finding a mouse. At each time step, the cat comes to one of the places. If it's the one a mouse is hiding in, the mouse has to move. The cost is the number of times the mouse has moved. Find a good randomized strategy for the mouse.

Example, $n = 4$, the mouse is in 1 and cat's sequence is $\{2, 1, 3, 4, 2, 1, 3\}$.

Note: for any deterministic algorithm for the mouse, there exists a sequence for the cat that causes the mouse to move every time. What's the simplest randomized algorithm for this problem?

Definition. The online algorithm ALG is c-competitive if there is a constant α such that for all inputs σ

$$E[C_{\text{ALG}}(\sigma)] \leq c\, C_{\text{OPT}}(\sigma) + \alpha$$

RAND: Mouse starts in a random place. Each time the mouse is hit by the cat, the mouse moves to a random other place.

What is a strategy for the cat? The cat visits places 1, 2, ..., $n-1$ repeatedly. We should have moved to point n at the start for a cost of 1. But, we expect to move an expected $n-1$ times. Here's why. Consider two cases. Case 1 - the mouse is at n, Case 2 - it's not at n. The probability that we're on n at the start is $1/n$ and our cost is 0 in this case. The probability that we're not at n is $1-1/n$. In this case what's the expected number of times we get hit before we land on n? $(n-1)$ - it's like flipping a coin of bias $1/(n-1)$ until it gets a head.

An optimal mouse will initially choose to hide in spot n, thus $C_{\text{OPT}} = 1$.

So the RAND algorithm is $\Omega(n)$ competitive, which is not what we're looking for.

<u>Claim</u>. No algorithm can get $o(\log n)$ ratio.

Proof. What if a cat probes randomly. Then, no matter what a mouse does, it has $1/n$ probability of being forced to move. So, in t time steps, expected cost to move is t/n. But, what is t? how long does it take a cat to hit every place? This is the coupon collector's problem.

Let p_i be the probability of seeing a new place after seeing i places

$$p_i = \frac{n-i}{n}$$

Let X_i be a random variable representing the number of probes to see a new place after seeing i places

$$E[X] = E[X_0] + \ldots + E[X_{n-1}] = (\text{by the mean time of failure})\ \frac{1}{p_0} + \frac{1}{p_1} + \ldots + \frac{1}{p_{n-1}}$$

$$E[X] = \frac{n}{n} + \frac{n}{n-1} + \ldots + \frac{n}{1} = n\left(\frac{1}{n} + \frac{1}{n-1} + \ldots + 1\right) = \Theta(n \log n)$$

Thus every online mouse will move at least

$$\frac{t}{n} = \frac{n \log n}{n} = \log n$$

times for a single move of OPT.

Marking. A better online algorithm.

1. The mouse starts in a random hiding place
2. If the cat looks in a place, mark that place
3. If the cat looks in the mouse's place, the mouse moves to a random unmarked place
4. When all places are marked, unmark them and restart

Claim. Marking is $O(\log n)$- competitive. For all cat probe sequences σ

$$E[C_{\text{ALG}}(\sigma)] \leq O(\log n)\, C_{\text{OPT}}(\sigma)$$

Proof. We divide the analysis into *phases*. The last probe of a phase is the one that causes all of the marks to be cleared. Note that the set of probes in a phase must hit every spot.

There are two types of probes

- probing a marked place (do not count this, since a cat knows our strategy)
- probing a unmarked place

At the first probe, the cat find the mouse with $1/n$ probability. At the second probe, the cat find the mouse with $1/(n-1)$ probability. And so on. So, the total expected number of moves per phase is

$$E[X] = \frac{1}{n} + \frac{1}{n-1} + \ldots + 1 = H_n = O(\log n)$$

An optimal mouse will hide in the last spot probed in the cat’s sequence, and thus the expected cost to OPT is 1.