15-451/651: Design & Analysis of Algorithms April 11, 2016
Lecture #23 last changed: April 11, 2016

Last time we looked at algorithms for finding approximately-optimal solutions for NP-hard problems. Today we'll be looking at finding approximately-optimal solutions for problems where the difficulty is not that the problem is necessarily *computationally* hard but rather that the algorithm doesn't have all the *information* it needs to solve the problem up front.

Specifically, we will be looking at *online algorithms*, which are algorithms for settings where inputs or data is arriving over time, and we need to make decisions on the fly, without knowing what will happen in the future. This is as opposed to standard problems like sorting where you have all inputs at the start. Data structures are one example of online algorithms (they need to handle sequences of requests, and to do well without knowing in advance which requests will be arriving in the future). We'll talk about some other kinds of examples today.

1 Rent or buy?

Here is a simple online problem that captures a common issue in online decision-making, called the rent-or-buy problem. Say you are just starting to go skiing. You can either rent skis for \$50 or buy them for \$500. You don't know if you'll enjoy skiing, so you decide to rent. Then you decide to go again, and again, and after a while you realize you have shelled out a lot of money renting and you wish you had bought right at the start.[1] The optimal strategy is: if you know you're going to end up skiing more than 10 times, you should buy right at the beginning. If you know you're going to go fewer than 10 times, you should just always rent. (If you know you're going to go exactly 10 then either way is equally good.) But, what if you don't know?

To talk about the quality of an online algorithm, we will look at what's called its *competitive ratio*:

Definition 1 *The* competitive ratio *of an online algorithm ALG is the worst case (i.e., maximum) over possible futures σ of the ratio:*

$$\frac{ALG(\sigma)}{OPT(\sigma)},$$

where $ALG(\sigma)$ represents the cost of ALG on σ and $OPT(\sigma)$ is the least possible cost on σ.

E.g., what is competitive ratio of the algorithm that says "buy right away"? The worst case is we only go skiing once. Here the ratio is $500/50 = 10$.

What about the algorithm that says "Rent forever"? Now the worst case is that we keep going skiing. So the competitive ratio of this algorithm is unbounded.

Here's a nice strategy: rent until you realize you should have bought, then buy. (In our case: rent 9 times, then buy). Let's call this algorithm *better-late-than-never*. Formally, if the rental cost is r and the purchase cost is p then the algorithm is to rent $\lceil p/r \rceil - 1$ times and then buy.

Theorem 2 *The algorithm* better-late-than-never *has competitive ratio ≤ 2. If the purchase cost p is an integer multiple of the rental cost r, then the competitive ratio is $2 - r/p$.*

Proof: We consider two cases. **Case 1:** if you went skiing fewer than $\lceil p/r \rceil$ times (e.g., 9 or fewer times in the case of $p = 500, r = 50$) then you are optimal. The algorithm never purchased and

[1] We are ignoring practical issues such as the type of ski you want depending on your ability level, etc.

OPT doesn't purchase either. **Case 2:** If you went skiing $\lceil p/r \rceil$ or more times, then the optimal solution would have been to buy at the start, so OPT $= p$. The algorithm paid $r(\lceil p/r \rceil - 1) + p$ (e.g., \$450 + \$500 in our specific case). This is always less than $2p$, and equals $2p - r$ if p is a multiple of r. In Case 1, the ratio of the algorithm's cost to OPT was 1, and in Case 2, the ratio of the algorithm's cost to OPT was less than 2 ($(2p-r)/p = 2 - r/p$ if p was a multiple of r). The worst of these is Case 2, and gives the claimed competitive ratio. ■

Theorem 3 *Algorithm* better-late-than-never *has the best possible competitive ratio for the ski-rental problem for deterministic algorithms when p is a multiple of r.*

Proof: Consider the event that the day you purchase is the last day you go skiing (this is a legitimate event, since (a) if the algorithm never purchases, we already know its competitive ratio is unbounded, so we may assume a purchase occurs, and (b) the algorithm is deterministic so this occurs after some specific number of rentals). Now, if you rent longer than *better-late-than-never*, then the numerator in Case 2 goes up (the algorithm's cost is larger) but the denominator stays the same, so your ratio is strictly worse. If you rent fewer times (say you rent k fewer times than *better-late-than-never* for some $k \geq 1$), then the numerator in Case 2 goes down by kr but so does the denominator, so again the ratio is worse. ■

2 The elevator problem

You go up to the elevator and press the button. But who knows how long it's going to take to come, if ever? How long should you wait until you give up and take the stairs?

Say it takes time E to get to your floor by elevator (once it comes) and it takes time S by stairs. E.g, maybe $E = 15$ sec, and $S = 45$ sec. How long should you wait until you give up? What strategy has the best competitive ratio?

Answer: wait 30 sec, then take the stairs (in general, wait for $S-E$ time). This is exactly the *better-late-than-never* strategy since we are taking the stairs once we realize we should have taken them at the start. If elevator comes in less than 30 sec, we're optimal. Otherwise, OPT $= 45$. We took 30+45 sec, so the ratio is $(30+45)/45 = 5/3$. Or, in general, the ratio is $(S - E + S)/S = 2 - E/S$.

You may have noticed this is really the *same* as rent-or-buy where stairs=buy, waiting for E time steps is like renting, and the elevator arriving is like the last time you ever ski. So, this algorithm is optimal for the same reason. Other problems like this: whether it's worth optimizing code, when your laptop should stop spinning the disk between accesses, and many others.

When $E \ll S$, this is very close to being 2-competitive. As you saw in HW#4, you can do better by randomization. Indeed, even in the case where $E \ll S$, you can get close to $\frac{e}{e-1}$-competitiveness using the zero-sum game approach from the problem of waiting for the bus.

3 An aside

Interesting article in NYT Sept 29, 2007: Talking about a book by Jason Zweig on how people's emotions affect their investing, called "Your money and your brain":

> "There is a story in the book about Harry Markowitz, Mr. Zweig said the other day. He was referring to the renowned economist who shared a Nobel for helping found modern portfolio theory and proving the importance of diversification.... Mr. Markowitz was then working at the RAND Corporation and trying to figure out how to allocate his

> retirement account. He knew what he should do: I should have computed the historical co-variances of the asset classes and drawn an efficient frontier. (That's efficient-market talk for draining as much risk as possible out of his portfolio.)
>
> But, he said, I visualized my grief if the stock market went way up and I wasn't in it or if it went way down and I was completely in it. So I split my contributions 50/50 between stocks and bonds. As Mr. Zweig notes dryly, Mr. Markowitz had proved incapable of applying his breakthrough theory to his own money."

So, he wasn't applying his own theory but he *was* using competitive analysis: 50/50 guarantees you end up with at least half as much as if you had known in advance what would happen, which is best possible Competitive Ratio you can achieve.

4 List Update

This is a nice problem that illustrates some of the ideas one can use to analyze online algorithms. Here are the ground rules for the problem:

- There's a list of n items. (For simplicity we fix n). The positions in the list are numbered $1, 2, \ldots, n$. Position 1 is the front of the list. The initial ordering of the items in the list is the same for any algorithm.
- An item x can be accessed. The operation is called $\texttt{Access}(x)$. The cost of the operation is the position i of x in the list.
- After doing an access, the algorithm is allowed to rearrange the list by doing swaps of adjacent elements. The cost of a swap is 1.

So an on-line algorithm is specified by describing which swaps are done after an element is accessed. (Without loss of generality we can give off-line optimum algorithm the power to do its sawps any time it wants, and not associate them with any particular access.)

The goal is to devise and analyze an on-line algorithm for doing all the accesses $\texttt{Access}(\sigma_1)$, $\texttt{Access}(\sigma_2)$, $\texttt{Access}(\sigma_3), \ldots$ with a small competitive factor.

Here are several algorithms to consider.

- **Do Nothing:** Don't reorder the list.
- **Single Exchange:** After accessing x, if x is not at the front of the list, swap it with its neighbor toward the front.
- **Frequency Count:** Maintain a frequency of access for each item. Keep the list ordered by non-increasing frequency from front to back.
- **Move To Front (MTF):** After an access to an element x, do a series of swaps to move x to the front of the list.

It's easy to construct sample sequences to show that Do Nothing, Single Exchange and Frequency Count have competitive factors that are $\Omega(n)$.

Theorem 4 *MTF is a 4-competitive algorithm for the list-update problem.*

Proof: We'll use the potential function method. There will be a potential function that depends on the state of the MTF algorithm and the state of the "opponent" algorithm B, which can be any algorithm, even one which can see the future. Using this potential, we'll show that the amortized cost to MTF of an access is at most 4 times the cost of that access to B.

What is the potential function Φ? Define

$$\Phi_t = 2 \cdot (\text{The number of inversions between } B\text{'s list and MTF's list at time } t)$$

Recall that an inversion is a pair of distinct elements (x, y) that appear in one order in B's list and in a different order in MTF's list. It's a measure of the similarity of the lists.

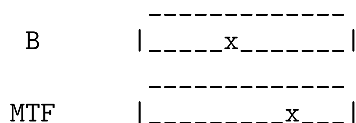
We can first analyze the amortized cost to MTF of `Access`(x) (where it pays for the list traversal and its swaps, but B only does its access). Then we separately analyze the amortized cost to MTF that is incurred when B does any swaps. (Note that in the latter case MTF incurs zero cost, but it will have a non-zero amortized cost, since the potential function may change. To be complete the analysis must take this into account.). In each case we'll show that the amortized cost to MTF (which is the actual cost, plus the increase in the potential) is at most 4 times the cost to B.

For any particular step, let C_{MTF} and C_B be the actual costs of MTF and B on this step, and $AC_{MTF} = C_{MTF} + \Delta\Phi$ be the amortized cost. Here $\Delta\Phi = \Phi_{new} - \Phi_{old}$ is the increase in Φ. Hence observe that $\Delta\Phi$ may be negative, and the amortized cost may be less than the actual cost. We want to show that

$$AC_{MTF} \leq 4 \cdot C_B$$

We can then sum the amortized costs, which would equal the actual cost of the entire sequence of accesses to MTF plus the final potential (non-negative) minus the initial potential (zero). This would be the four times total cost of B, which would give the result.

Analysis of `Access`(x). First look at what happens when MTS accesses x and brings it to the front of its list. Say the picture looks like this:



Look at the elements that lie before x in MTF's list, and partition them as follows:

$$S = \{y \mid y \text{ is before } x \text{ in MTF and } y \text{ is before } x \text{ in } B\}$$
$$T = \{y \mid y \text{ is before } x \text{ in MTF and } y \text{ is after } x \text{ in } B\}$$

What is the cost of the access to MTF in terms of these sets?

$$C_{MTF} = 1 + \underbrace{|S| + |T|}_{\text{find cost}} + \underbrace{|S| + |T|}_{\text{swap cost}} = 1 + 2(|S| + |T|).$$

On the other hand, since all of S lies before x in B, the cost of the algorithm B is at least

$$C_B \geq |S| + 1.$$

What happens to the potential as a result of this operation? Well, here's MTF after the operation:

```
        -------------
MTF    |x____________|
```

The only changes in the inversions involve element x, because all other pairs stay in the same relative order. Hence, for every element of S the the number of inversions increases by 1, and for every element of T the number of inversions decreases by 1. Hence the increase in Φ is precisely:

$$\Delta(\Phi) = 2 \times (|S| - |T|)$$

Now the amortized cost is

$$\begin{aligned} AC_{MTF} &= C_{MTF} + \Delta(\Phi) = 2(|S| - |T|) + 1 + 2(|S| + |T|) \\ &= 1 + 4|S| \leq 4(1 + |S|) \leq 4\,C_B \end{aligned}$$

This completes the amortized analysis of `Access`(x).

Analysis of B swapping. Now we perform all the swaps that B does. We do the analysis one swap at a time. For each such swap, observe that $C_{MTF} = 0$ and $C_B = 1$. Moreover, $\Delta(\Phi) \leq 2$, since the swap may introduce at most one new inversion. Hence,

$$AC_{MTF} \leq 2\,C_B \leq 4\,C_B$$

Putting the parts together. Summing the amortized costs we get:

$$\text{Total Cost to MTF } + \Phi_{final} - \Phi_{init} \leq 4(\text{Total Cost to } B)$$

But $\Phi_{init} = 0$, since we start off with the same list as B. And $\Phi_{final} \geq 0$. Hence $\Phi_{final} - \Phi_{init} \geq 0$. Hence,

$$\text{Total Cost to MTF} \leq 4 \times (\text{Total Cost to } B).$$

Hence the MTF algorithm is 4-competitive. ■

5 Paging

In paging, we have a disk with N pages, and fast memory with space for $k < N$ pages. When a memory request is made, if the page isn't in the fast memory, we have a page fault. We then need to bring the page into the fast memory and throw something else out if our space is full. Our goal is to minimize the number of misses. The algorithmic question is: what should we throw out? E.g., say $k = 3$ and the request sequence is 1,2,3,2,4,3,4,1,2,3,4. What would be the right thing to do in this case if we knew the future? Answer: throw out the thing whose next request is farthest in the future.

A standard online algorithm is LRU: "throw out the least recently used page". E.g., what would it do on above case? What's a bad case for LRU? 1,2,3,4,1,2,3,4,1,2,3,4... Notice that in this case, the algorithm makes a page fault every time and yet if we knew the future we could have thrown out a page whose next request was 3 time steps ahead. More generally, this type of example shows that the competitive ratio of LRU is at least k. In fact, you can show this is actually the worst-case for LRU, so the competitive ratio of LRU is exactly k (it's not hard to show but we won't prove it here).

In fact, it's not hard to show that you can't do better than a competitive ratio of k with a deterministic algorithm: you just set $N = k + 1$ and consider a request sequence that always

requests whichever page the algorithm *doesn't* have in its fast memory. By design, this will cause the algorithm to have a page fault every time. However, if we knew the future, every time we had a page fault we could always throw out the item whose next request is farthest in the future. Since there are k pages in our fast memory, for one of them, this next request has to be at least k time steps in the future, and since $N = k + 1$, this means we won't have a page fault for at least $k - 1$ more steps (until that one is requested). So, the algorithm that knows the future has a page fault at most once every k steps, and the ratio is k.

Here is a neat *randomized* algorithm with a competitive ratio of $O(\log k)$. Specifically, for any request sequence σ, we have $\mathrm{E}[ALG(\sigma)]/OPT(\sigma) = O(\log k)$.

Algorithm "Marking":

- Assume the initial state is pages $1, \ldots, k$ in fast memory. Start with all pages unmarked.
- When a page is requested,
  - if it's in fast memory already, mark it.
  - if it's not, then throw out a random unmarked page. (If all pages in fast memory are marked, unmark everything first. For analysis purposes, call this the end of a "phase"). Then bring in the page and mark it.

We can think of this as a 1-bit randomized LRU, where marks represent "recently used" vs "not recently used".

We will show the proof for the special case of $N = k + 1$. For general N, the proof follows similar lines but just is a bit more complicated.

Proof:(for $N = k + 1$). In a phase, you have $k + 1$ different pages requested so OPT has at least one page fault. For the algorithm, the page not in fast memory is a *random* unmarked page. Every time a page is requested: if it was already marked, then there is no page fault for sure. If it wasn't marked, then the probability of a page fault is $1/i$ where i is the number of unmarked pages. So, within a phase, the expected total cost is $1/k + 1/(k - 1) + ... + 1/2 + 1 = O(\log k)$. ■