Advanced Algorithm Design: Online Algorithm

Lecture by Prof. Moses Charikar
Transcribed by Kevin Lin

February 18, 2013

In this lecture, we describe **online algorithms**, algorithms that process its input piece-by-piece in a serial fashion where the entire input is not available from the start. We study list update and caching and develop common methods to analyse these algorithms.

1 List Update

List update was developed first by Sleator and Tarjan when developing online algorithms for data structures. We study this in the setting below:

1.1 Setup

There is a list of n elements in the form of a linked list. The cost of accessing the kth element is k. We also have flexibility of rearranging the list as we access elements. More specifically, we can choose the perform the following actions when accessing this k^{th} element:

- Move the kth element to any position in front of it for free (no cost) (ie: one position forward, two positions forward, all the way forward, etc.).
- Swap consecutive elements in the list for cost of 1. (This is called a **Paid Exchange**)

The task we want to solve is given a sequence signal σ, design an online algorithm to minimize $C_A(\sigma)$.

Definition. $C_A(\sigma)$: Cost of A on sequence σ.

Definition. $C_{OPT}(\sigma)$: Minimum cost solution for σ knowing σ in advance.

Definition. Strictly α-competitive: A has a stricly competitive ratio of α if $C_A(\sigma) \leq \alpha \cdot C_{OPT}(\sigma)$

Definition. α**-competitive**: A has a competitive ratio of α if $C_A(\sigma) \leq \alpha \cdot C_{OPT}(\sigma) + \beta$ where β is independent of σ.

Notes:

- We generally will not use the swap function at all. We specify it so we allow other possible algorithms (such as the optimal algorithm) to use them.
- The definition of α-competitive is more general and focuses on the average cost. β can be a function of any other parameters of the problem besides σ (for example, it cannot depend on $|\sigma|$). We hope that for large σ, β becomes insignificant.
- We are allowed to store *any* amount of external information when running an online algorithm in the setting we described. We can also take as much time (ie: exponential) time to calculate anything we want when running these algorithms. As it turns out though, we typically do not need to do either of these things.
- For this lecture, we are not concerned with how the list is initialized. All we need to know is that both A and OPT share the same initial list.

1.2 Optimal Algorithm

What if we knew σ beforehand? As it turns out, finding this cost is NP-hard. The details are not shown here. (This is shown by reduction where we show it is NP-Hard to differentiate between the yes's and no's for some defined yes-instance and no-instance.) The point is that $C_{OPT}(\sigma)$ is hard to compute, and we might feel hopeless since calculating $C_A(\sigma)$ is an "even harder" than calculating $C_{OPT}(\sigma)$.
Instead, we put implicit lower-bounds on the solution. In some sense, by comparing the ratio between $C_A(\sigma)$ and an approximation of $C_{OPT}(\sigma)$, we are trying to answer how much we lose due to our inability to predict the future.

1.3 Algorithms

Here are some examples of possible algorithms we might be interested in using to solve List Update

- MTF (Move to Front): After accessing the kth element, move it all the way to the very front of the list.
- MoTF (Move One to Front): After accessing the kth element, move it one position forward in the list.
- Frequency Count: Keep the list sorted with respect to frequency. (The highest frequency is in the front.)

We can actually eliminate two of these ideas right away since they have poor competitive ratios. MoTF is clearly bad since we might continually call the nth and $(n\text{-}1)^{\text{th}}$ element. A better algorithm should've recognized that it would be best to move this two elements to the front of the list so save costs. Frequency count is also clearly bad since (intuitively), it will react too slowly to new changes. In fact, frequency count can be shown to be $\Omega(n)$-competitive.

Theorem 1.1. MTF is 2-Competitive

Proof Idea: We first describe one method that would **not** work. One method that would be unsuccessful would be to try showing $C_{MTF}(\sigma_i) \leq 2 \cdot C_{OPT}(\sigma_i)$. We cannot hope to prove this since we cannot be certain on what the optimal algorithm does, and in some sense, it is too good to be true. Instead, we use potential functions as some kind of counter roughly measuring how different our algorithm is from the optimal algorithm. The formula we seek to show is:

$$C_{MTF}(\sigma_i) + \phi_i - \phi_{i\text{-}1} \leq 2 \cdot C_{OPT}(\sigma_i) \tag{1}$$

In particular, if we add up (1) for all i, we get:

$$C_{MTF}(\sigma) + \phi_{|\sigma|} - \phi_0 \leq 2 \cdot C_{OPT}(\sigma) \tag{2}$$

where $|\sigma|$ denotes the length of the entire request sequence, $\phi_{|\sigma|} \geq 0$, $\phi_0 = 0$.

Proof: All that remains to prove is (1). Let our potential function ϕ denote the number of inversions between the OPT's and MTF's lists. When handling the ith request:

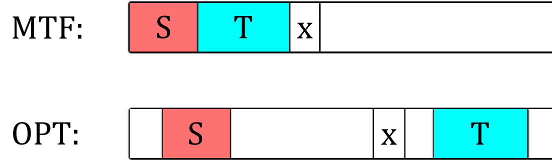

Figure 1: An illustration of Go-CART on the climate data. (a) A list of the 18 collected climate factors; (b) The estimated subregions for 125 locations projected onto the US map (two locations belong to the same subregion if they are denoted by the same shape and color), with the estimated graphs for subregions 2, 3, and 65; (c) the rescaled partition pattern mapped onto a unit square; (d) estimated graph with data pooled from all 125 geographic locations. All tuning parameters are chosen by minimizing the held-out likelihood under a conditional Gaussian model.

where $S = \{y \mid y$ precedes x in both MTF and OPT$\}$ and $T = \{y \mid y$ precedes x in MTF but not in OPT$\}$. (The graphics are to show volume not actual element positions.) We can easily see for $\sigma_i = x$,

$$C_{MTF}(\sigma_i) = |S| + |T| + 1 \tag{3}$$

$$C_{OPT}(\sigma_i) \geq |S| + 1 + P \tag{4}$$

where P refers to the unknown number of paid exchanges made by the optimal algorithm. These equations simply refer to the position of x which can be visualized by the graphic above.

We now want to know how ϕ changes. Let MTF make its move first. Clearly only all pairs involving x might possibly have their inversions changed. Looking at the elements in S and T, if we first look at just MTF's actions before we consider OPT, we

count ϕ_i - $\phi_{i\text{-}1} \leq |S|$ - $|T|$.

After OPT makes its move, our number of inversions might decrease from $|S|$ - $|T|$ (if element *x* moves to the front). We need to count the number of possible paid exchange OPT might do though. Hence, after looking at OPT's actions, we get the final inequality:

$$\phi_i - \phi_{i\text{-}1} \leq |S| - |T| + P. \tag{5}$$

Thus, using (5), we get:

$$\begin{aligned} C_{MTF}(\sigma_i) + \phi_i - \phi_{i\text{-}1} &\leq (|S| + |T| + 1) + (|S| - |T| + P) && (6)\\ &\leq 2 \cdot |S| + P + 1 && (7)\\ &\leq 2 \cdot (|S| + P + 1) && (8)\\ &\leq 2 \cdot C_{OPT}(\sigma_i) && (9)\end{aligned}$$

Hence, we have shown (1) and we are done. We should note that finding these particular potential functions is almost an artful in itself since there is no clear heuristic to follow when constructing them. ■

1.4 Randomness

Notice that in our algorithm, we did not use any randomness at all. In fact, we can show that for any deterministic algorithm, a factor of 2 is the best.

Theorem 1.2. The lowest competitive ratio for a deterministic algorithm to solve List Update is 2.
Proof Idea: We will only present the proof idea. We use an adversary-argument to show why this bound holds. Suppose we run any deterministic algorithm. Since the algorithm is deterministic, an adversary looking at the currently-stored list will know how the algorithm will act in the future. Thus, the adversary makes a specific request list σ to make the algorithm give the worst possible performance. Specifically, the adversary keeps requesting the last element in the list in that iteration. Clearly MTF would then pay *n* for every request.

Using that same σ, what would OPT do? Keep in mind that OPT can see the entire σ before it starts handling the requests. A good approximation to what OPT would do is to first sort the entire list by each element's frequency count in σ with the more frequently-requested items in front. It can be shown that aside from the constant cost to sort this list, OPT would pay *n*/2 on average. ■

In general, we could randomize an algorithm and analyse its performance, but in general, randomized algorithms are hard to analyse in an online setting. These randomized algorithms are essentially algorithms that are allowed to toss (weighted) coins and view the result of the coin to determine its next action. When evaluating these models, our competitive ratio definition would need to be altered slight:

Definition. α**-competitive**: A (a randomized algorithm) has a competitive ratio of α if $\mathbb{E}[C_A(\sigma)] \leq \alpha \cdot C_{OPT}(\sigma) + \beta$ where β is independent of σ, and $\mathbb{E}$ is taken over all possible random choices of A. We would like to have adversary-arguments similar to the deterministic case to see what are the best possible bounds. Unfortunately, there are many different adversarial-models since we need to specify whether or not the adversary knows things like the outcomes of the coin tosses, etc. Here is a list of different models with a brief description for each.

- **Oblivious Adversary**: Adversary knows how the randomized algorithm works but does not know the exact choices the algorithm made. This will be the most common model we use when we use the adversary-model.
- **Adaptive Online Adversary**: Adversary decides the request as the randomized algorithm works.
- **Adaptive Offline Adversary**: Similar to the one before but in an offline setting for the adversary.

In current algorithm literature, the best known randomized algorithm to solve List Hash has a competitive ratio of approximately 1.6. Lastly, we present a very simple example of a randomized List Update algorithm:

```
Data: A linked list with n elements
Set one randomized bit (0 or 1) for every element in the list;
while Still requesting elements do
    Return the requested element;
    if the bit for the requested element is 0 then
        Move the requested element to front
    end
    Flip the bit of the requested element
end
```

Algorithm 1: Algorithm "**BIT**"

The only randomization BIT has is the randomized bits assigned to each element in the very beginning. We will not prove it here, but surprisingly, BIT is 1.75-competitive.

2 Caching

There is additional information at `theory.stanford.edu/~trevisan/cs261/lecture17.pdf`.

2.1 Setup

There are n pages (of memory) stored in your computer across all types of memory. The cache can only has a capacity of storing k pages at any time. When someone requests a certain page, if the page is already in the cache, it is a *hit*. If not, then we have to first evict a certain page in cache, retrieve the requested page from hard disk,

and then place this request page into the cache. We pay a cost of 1 doing this. The task we want to solve is given a sequence signal σ of page requests, design an online algorithm to minimize $C_A(\sigma)$, the number of cache misses.

2.2 Common Algorithms

We present some reasonable online algorithms used to solve this problem. Both of these algorithms will require you to store additional information.

- Least Recently Used (LRU): When choosing a page to evict in cache, chose the page which the long has passed since its last request.
- Least Frequently Used (LFU): When choosing a page to evict in cache, chose the page that has been least frequently requested.

Similar to our analysis in List Update, we can immediately get rid of LFU for its poor competitive ratio. Consider the following scenario. (*k-1*) of the pages in cache have a really high frequency count after running the algorithm for some amount of time. If x and y denote two distinct pages with extremely low frequency count, a request sequence of x and y alternating will result in a cache miss every single time since the cache keeps cycling through these 2 pages. Thus, LFU can be shown to be $\Omega(|\sigma|)$-competitive. We present a popular offline algorithm for Caching:

- Furtherest in Future: When choosing a page to evict in cache, choose the page that is requested furthest in the future.

Again, we can see the entire σ when we are in the offline setting. As opposed to List Update, we actually can prove that this offline algorithm *is* the optimal algorithm, so there is no need to use potential functions. We will not prove this claim here.

Theorem 2.1. LRU is K-competitive.
Proof Idea: We can partition the request sequence into phases where each phase starts at the first element after the previous phase and contains the maximum set of pages to ensure that there are k distinct pages within each phase. In particular, by choosing the phases to be the maximum set of pages to ensure k distinct pages within in phase, we can guarantee that when we request the first page in the following phase, OPT *must* have perform at least 1 eviction. On the other hand, when LRU handles this phase, it performs at most k evictions since there are k distinct elements in this phase. Formally, by construction of phases:

- OPT has cot ≥ 1 per phase
- LRU has cost $\leq k$ per phase.

We note that the $\leq$ for LRU is needed since some of the pages requested in the phase might already be the cache. ■

Now we are interested if we could have done any better than k-competitive. This ratio might seem pretty bad, but we should keep in mind that this pessimism is dictated by the way we chose to analyse online algorithms.

Theorem 2.2. Any deterministic algorithm has a competitive ration $\geq k$.
Proof: Consider (*k+1*) pages. Consider any deterministic algorithm A. Let an adversary always request a page that is not in cache when running A. This can be any page. (Again, similar to when analysing deterministic algorithms in List Update, the adversary can plan out how to choose the requests to generate this behaviour since the algorithm is deterministic.) Thus, the algorithm incurs a cost of 1 for every request.

Now, consider the optimal algorithm. Since we know the optimal algorithm uses "Furthest in the Future", we evict the page furthest in the future. this means that once we have a fault, we won't have a fault for at least k requests in the future. It is clear the following holds:

$$C_{OPT}(\sigma) \leq \left\lceil \frac{|\sigma|}{k} \right\rceil \tag{10}$$

2.3 Randomness

We now describe a simple random "Marking" algorithm:

Data: n pages and a cache of capacity $k < n$
Until the cache fills up, attach an unset bit to all pages in the cache;
while *Requesting page p* **do**
  if *Page p does not exist in cache* **then**
    if *There is one unmarked page in cache* **then**
      Evict one random unmarked page;
    else
      Unmark all pages and evict a random unmarked page;
    end
    Bring in page p and mark it
  end
end

Algorithm 2: Algorithm "**Marking**"

Next lecture, we will show that this algorithm is actually O(log k)-competitive. We will also show that the lower bound for all randomized algorithms for Caching is Ω(log k)-competitive.