Advanced Algorithms 4 December 2018

Lecture 12: Online Algorithms and Competitive Analysis I

Lecturer: Mohsen Ghaffari *Scribe: Davin Choo*

In this section, we study the class of online problems where one has to commit to provably good decisions as data arrive in an online fashion. To measure the effectiveness of online algorithms, we compare the quality of the produced solution against the solution from an optimal offline algorithm that knows the whole sequence of information a priori. The tool we will use for doing such a comparison is *competitive analysis.*

Remark We do not assume that the optimal offline algorithm has to be computationally efficient. Under the competitive analysis framework, only the *quality of the best possible solution* matters.

Definition 1 (α-competitive online algorithm)**.** *Let σ be an input sequence, c be a cost function, $\mathcal{A}$ be the online algorithm and OPT be the optimal offline algorithm. Then, denote $c_{\mathcal{A}}(\sigma)$ as the cost incurred by $\mathcal{A}$ on σ and $c_{OPT}(\sigma)$ as the cost incurred by OPT on the same sequence. We say that an online algorithm is α-competitive if for any input sequence σ, $c_{\mathcal{A}}(\sigma) \leq \alpha \cdot c_{OPT}(\sigma)$.*

1 Warm up: Ski rental

Definition 2 (Ski rental problem)**.** *Suppose we wish to ski every day but we do not have any skiing equipment initially. On each day, we could:*

- *Rent the equipment for a day at CHF* 1
- *Buy the equipment (once and for all) at CHF B*

In the toy setting where we may break our leg on each day (and cannot ski thereafter), let d be the (unknown) total number of days we ski. What is the best online strategy for renting/buying?

Claim 3. *$\mathcal{A}$ = "Rent for B days, then buy on day $B + 1$" is a 2-competitive algorithm.*

Proof. If $d \leq B$, the optimal offline strategy is to rent everyday, incurring d. $\mathcal{A}$ will rent for d days and also incur d. If $d > B$, the optimal offline strategy is to buy the equipment immediately, incurring B. $\mathcal{A}$ will rent for B days then buy, incurring $2B$. Thus, for any d, $c_{\mathcal{A}}(d) \leq 2 \cdot c_{OPT}(d)$. □

2 Linear search

Definition 4 (Linear search problem)**.** *Suppose we have a stack of n papers on the desk. Given a query, we do a linear search from the top of the stack. Suppose the query is the i-th paper in the stack and it costs i units of work to reach it. There are two types of swaps we can make to the stack:*

Free swap *Move the queried paper from position i to the top of the stack for 0 cost.*

Paid swap *For any consecutive pair of items (a, b) before i, swap their relative order to (b, a) for 1 cost.*

What is the best online strategy for manipulating the stack to minimize total cost on a sequence of queries?

Remark One can reason that the free swap costs 0 because we already incurred a cost of i to reach it.

2.1 Amortized analysis

Amortized analysis[1] is a way to analyze the complexity of an algorithm on a sequence of operations. Instead of looking the worst case performance on a single operation, it measures the total cost for a *batch* of operations.

[1]See https://en.wikipedia.org/wiki/Amortized_analysis

The dynamic resizing process of hash tables is a classical example of amortized analysis. An insertion or deletion operation will typically cost $\mathcal{O}(1)$ unless the hash table is almost full or almost empty, in which case we double or halve the hash table, incurring a runtime of $\mathcal{O}(m)$ time for doubling or halving a hash table of size m.

Worst case analysis tells us that dynamic resizing will incur $\mathcal{O}(m)$ run time per operation. However, resizing only occurs after $\mathcal{O}(m)$ insertion/deletion operations, each costing $\mathcal{O}(1)$. Amortized analysis allows us to conclude that this dynamic resizing runs in amortized $\mathcal{O}(1)$ time. There are two equivalent ways to see it:

- Split the $\mathcal{O}(m)$ resizing overhead and "charge" $\mathcal{O}(1)$ to each of the earlier $\mathcal{O}(m)$ operations.
- The *total* run time for every sequential chunk of m operations is $\mathcal{O}(m)$. Hence, each step takes $\mathcal{O}(m)/m = \mathcal{O}(1)$ amortized run time.

2.2 Move-to-Front — A 2-competitive algorithm for linear search

Move-to-Front (MTF) [ST85] is an online algorithm for the linear search problem where we move the queried item to the top of the stack (and do no other swaps). Before we analyze MTF, let us first define a potential function Φ and look at examples to gain some intuition.

Let Φ_t be the number of pairs of papers (i, j) that are ordered differently in MTF's stack and OPT's stack at time step t. By definition, $\Phi_t \geq 0$ for any t. We also know that $\Phi_0 = 0$ since MTF and OPT operate on the same initial stack sequence.

Example One way to interpret Φ is to count the number of inversions between MTF's stack and OPT's stack. Suppose we have the following stacks (visualized horizontally) with $n = 6$:

	1	2	3	4	5	6
MTF's stack	a	b	c	d	e	f
OPT's stack	a	b	e	d	c	f

We have the inversions (c, d), (c, e) and (d, e), so $\Phi = 3$.

Scenario 1 We swap (b, e) in OPT's stack — A new inversion (b, e) was created due to the swap.

	1	2	3	4	5	6
MTF's stack	a	b	c	d	e	f
OPT's stack	a	e	b	d	c	f

Now, we have the inversions (b, e), (c, d), (c, e) and (d, e), so $\Phi = 4$.

Scenario 2 We swap (e, d) in OPT's stack — The inversion (d, e) was destroyed due to the swap.

	1	2	3	4	5	6
MTF's stack	a	b	c	d	e	f
OPT's stack	a	b	d	e	c	f

Now, we have the inversions (c, d) and (c, e), so $\Phi = 2$.

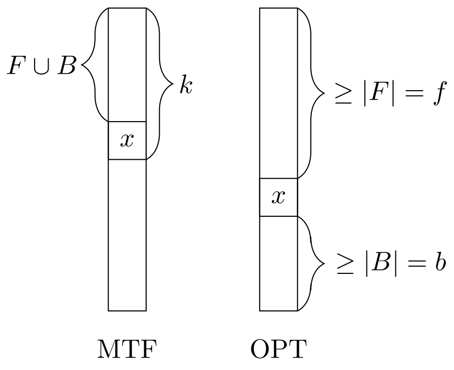
In either case, we see that any paid swap results in ± 1 inversions, which changes Φ by ± 1.

Claim 5. *MTF is 2-competitive.*

Proof. We will consider the potential function Φ as before and perform amortized analysis on any given input sequence σ. Let $a_t = c_{MTF}(t) + (\Phi_t - \Phi_{t-1})$ be the amortized cost of MTF at time step t, where $c_{MTF}(t)$ is the cost by MTF at time t. Suppose the queried item at time step t is at position k in MTF. Denote:

$$\begin{aligned} F &= \{\text{Items in front of } x \text{ in MTF's stack and } \textit{in front} \text{ of } x \text{ in OPT's stack}\} \\ B &= \{\text{Items in front of } x \text{ in MTF's stack and } \textit{behind } x \text{ in OPT's stack}\} \end{aligned}$$

Let $|F| = f$ and $|B| = b$. There are $k - 1$ items in front x, so $f + b = k - 1$.



Since x is the k-th item, MTF will incur $c_{MTF}(t) = k$ to reach item x, then move it to the top. On the other hand, OPT needs to spend at least $f + 1$ to reach x. Suppose OPT does p paid swaps, then $c_{OPT}(t) \geq f + 1 + p$.

To measure the change in potential, we first look at the swaps done by MTF and how OPT's swaps can affect them. In MTF, moving x to the top destroys b inversions and creates f inversions, so the change in Φ due to MTF is $f - b$. If OPT chooses to do a free swap, Φ decreases as both stacks now have x before any element in F. For every paid swap that OPT performs, Φ changes by one since inversions only locally affect the swapped pair. That is, the change in Φ due to MTF is $\leq p$.

Thus, the effect on Φ from both processes is: $\Delta(\Phi_t) \leq (f - b) + p$. Putting together, we have $c_{OPT}(t) \geq f + 1 + p$ and $a_t = c_{MTF}(t) + (\Phi_t - \Phi_{t-1}) = k + \Delta(\Phi_t) \leq 2f + 1 + p$. Hence,

$$\begin{array}{rrcll}
 & a_t & \leq & 2 \cdot c_{OPT}(t) & \text{Since } a_t = 2f + 1 + p \text{ and } c_{OPT}(t) \geq f + 1 + p \\
\Rightarrow & \sum_{t=1}^{|\sigma|} a_t & \leq & \sum_{t=1}^{|\sigma|} 2 \cdot c_{OPT}(t) & \text{Summing over all inputs in the sequence } \sigma \\
\Rightarrow & \sum_{t=1}^{|\sigma|} c_{MTF}(t) + (\Phi_t - \Phi_{t-1}) & \leq & 2 \cdot c_{OPT}(\sigma) & \text{Since } a_t = c_{MTF}(t) + (\Phi_t - \Phi_{t-1}) \\
\Rightarrow & \sum_{t=1}^{|\sigma|} c_{MTF}(t) + (\Phi_{|\sigma|} - \Phi_0) & \leq & 2 \cdot c_{OPT}(\sigma) & \text{Telescoping} \\
\Rightarrow & \sum_{t=1}^{|\sigma|} c_{MTF}(t) & \leq & 2 \cdot c_{OPT}(\sigma) & \text{Since } \Phi_t \geq 0 = \Phi_0 \\
\Rightarrow & c_{MTF}(\sigma) & \leq & 2 \cdot c_{OPT}(\sigma) & \text{Since } c_{MTF}(\sigma) = \sum_{t=1}^{|\sigma|} c_{MTF}(t)
\end{array}$$

□

3 Paging

Definition 6 (Paging problem [ST85]). *Suppose we have a fast memory (cache) that can fit k pages and an unbounded sized slow memory. Accessing items in the cache costs 0 units of time while accessing items in the slow memory costs 1 unit of time. After accessing an item in the slow memory, we can bring it into the cache by evicting an incumbent item if the cache was full. What is the best online strategy for maintaining items in the cache to minimize the total access cost on a sequence of queries?*

Denote *cache miss* as accessing an item that is not in the cache. Any sensible strategy should aim to reduce the number of cache misses. For example, if $k = 3$ and $\sigma = \{1, 2, 3, 4, \dots, 2, 3, 4\}$, keeping item 1 in the cache will incur several cache misses. Instead, the strategy should aim to keep items $\{2, 3, 4\}$ in the cache. We formalize this notion in the following definition of *conservative strategy.*

Definition 7 (Conservative strategy). *A strategy is conservative if on any consecutive subsequence that has only k distinct pages, there are at most k cache misses.*

Remark Some natural paging strategies such as "Least Recently Used (LRU)" and "First In First Out (FIFO)" are conservative.

Claim 8. *If $\mathcal{A}$ is a deterministic online algorithm that is α-competitive, then $\alpha \geq k$.*

Proof. Consider the following input sequence σ on $k + 1$ pages: Since the cache has size k, at least one item is not in the cache at any point in time. Iteratively pick $\sigma(t + 1)$ as the item not in the cache after time step t.

Since $\mathcal{A}$ is deterministic, the adversary can simulate $\mathcal{A}$ for $|\sigma|$ steps and build σ accordingly. By construction, $c_{\mathcal{A}}(\sigma) = |\sigma|$.

On the other hand, since OPT can see the entire sequence σ, OPT can choose to evict the page that is requested furthest in the future. Then, in every k steps, OPT has ≤ 1 cache miss. Thus, $c_{OPT} \leq \frac{|\sigma|}{k}$.

Hence, $k \cdot c_{OPT} \leq c_{\mathcal{A}}(\sigma)$. $\square$

Claim 9. *Any conservative online algorithm $\mathcal{A}$ is k-competitive.*

Proof. For any given input sequence σ, partition σ into *maximal phases* — $P_1, P_2, \ldots$ — where each phase has k distinct pages, and a new phase is created only if the next element is different from the ones in the current phase. Let x_i be the first item that does not belong in Phase i.

$$\sigma = \underbrace{\boxed{k \text{ distinct pages}}}_{\text{Phase 1}}\underbrace{\boxed{x_1}\boxed{k \text{ distinct pages}}}_{\text{Phase 2}}\boxed{x_2}\boxed{\quad\cdots\quad}$$

By construction, OPT has to pay ≥ 1 to handle the elements in $P_i \cup \{x_i\}$, for any i. On the other hand, since $\mathcal{A}$ is conservative, $\mathcal{A}$ has $\leq k$ cache misses per phase.

Hence, $c_{\mathcal{A}}(\sigma) \leq k \cdot \text{Number of phases} \leq k \cdot c_{OPT}(\sigma)$. $\square$

Remark A randomized algorithm can achieve $\mathcal{O}(\log k)$-competitiveness. This will be covered in the next lecture.

4 Types of adversaries

Since online algorithms are analyzed on all possible input sequences, it helps to consider adversarial inputs that may induce the worst case performance for a given online algorithm $\mathcal{A}$. To this end, one may wish to classify the classes of adversaries designing the input sequences (in increasing power):

Oblivious The adversary designs the input sequence σ at the beginning. It does not know any randomness used by algorithm $\mathcal{A}$.

Adaptive At each time step t, the adversary knows all randomness used by algorithm $\mathcal{A}$ thus far. In particular, it knows the exact state of the algorithm. With these in mind, it then picks the $(t+1)$-th element in the input sequence.

Fully adaptive The adversary knows all possible randomness that will be used by the algorithm $\mathcal{A}$ when running on the full input sequence σ. For instance, assume the adversary has access to the same pseudorandom number generator used by $\mathcal{A}$ and can invoke it arbitrarily many times while designing the adversarial input sequence σ.

Remark If $\mathcal{A}$ is deterministic, then all three classes of adversaries have the same power.

References

[ST85] Daniel D Sleator and Robert E Tarjan. Amortized efficiency of list update and paging rules. *Communications of the ACM*, 28(2):202–208, 1985.