Lecture 7: Online Algorithms

1 Classic Online Problems

There are a number of classic online algorithms problems. We will look at three specific examples today.

1. The Canadian Travelers problem
2. The Ski Renters problem
3. The Paging problem

1.1 Canadian Traveler’s Problem

It is winter in Canada. A Canadian traveler has a road map and must get from point A to point B, but some sections of road are blocked by snow.

In the offline version of this problem, the traveler has full information of which road sections are blocked. Thus, a traditional shortest path algorithm can be employed to determine an optimal path from point A to point B.

In the online version of the problem, at each intersection, the traveler must decide which way to go with awareness only of a given road section being blocked by snow if he/she has already passed through an endpoint of said road section.

1.2 Ski Renters Problem

Skis cost $1 a day to rent and $T to buy, after $S$ days you fall off the mountain breaking both legs and can no longer ski for the rest of the season. Each day
you are given a choice to continue renting or buy the skis for $T$. Minimize the dollar amount spent on skis this season.

In the offline version of this problem, you know the value of $S$ prior to your first day of skiing. As a result the decision is easy (if $T \leq S$ then buy, if $T \leq S$, rent every day until you break your leg, and if $S = T$ apply either strategy).

The online version of this problem is not so trivial. In this version of the problem we do not know $S$ in advance. Observe that any deterministic strategy we apply can be described as follows: we will rent for $k$ days, and then buy on day $k + 1$ if we have not broken our leg by then.

1.3 Paging Problem

We are allowed $k$ pages of data to be cached in "fast" memory storage and all other data pages must be stored in slow memory and swapped into the fast memory if and when they are requested. The user requests a sequence of pages $\langle \sigma \rangle$.

If page $\sigma_i$ is in fast memory, the cost for retrieving it is zero, and if $\sigma_i$ is not in fast memory, then $\sigma_i$ must be swapped in, and some other page $\sigma_j$ must be removed to create space for it, and the retrieval cost is 1. The latter case is referred to as a page fault. The cost of an algorithm on a given sequence $\langle \sigma \rangle$ is simply the number of page faults.

In the offline formulation of this problem, we know the values of our sequence in advance. In the online formulation, we only learn what pages will be requested as the requests arrive; that is we learn $\sigma_i$’s value at time $i$, and need to base our decision on knowledge of only the values of $\sigma_1, \sigma_2, ... \sigma_i$ without knowing the values of $\sigma_{i+1}, \sigma_{i+2}, ... \sigma_n$.

2 Measuring Performance of Online Algorithms

The first question is one of measurement. What is a good measure of an online algorithm? We measure the relative effectiveness of an online algorithm by
comparing its $\alpha$ value as defined below.

### 2.1 $\alpha$-Competitive

**Definition 2.1.1 $\alpha$-competitive.** An online is $\alpha$-*competitive* if for all input sequences $\sigma$, the cost $C_A(\sigma) \leq \alpha \cdot C_{\min}(\sigma)$ where $C_A(\sigma)$ is the cost of the online algorithm $A$ on input sequence $\sigma$, and $C_{\min}(\sigma)$ is the cost of the optimal offline algorithm on the same input sequence $\sigma$.

### 2.2 Ski-Renters Problem

The Optimal Offline Algorithm: Since we know both $T$ and $S$, we compare the two, if $T \leq S$, buy the skis on the first day, otherwise rent until day $S$. This results in a cost of $T$ if $T \leq S$, and a cost of $S$ otherwise.

**Online Strategy $A_1$** Always buy on day 1.

In this case $C_A(\sigma) = T$. Since $C_{\min} \geq 1$, this strategy is $T$-competitive.

**Online Strategy $A_2$** Rent for $T - 1$ days, then buy on day $T$.

If $S \leq T$, then $C_A(\sigma) = C_{\min}(\sigma)$, otherwise $C_{\min}(\sigma) = T$ and $C_A(\sigma) = 2T = 2 \cdot C_{\min}(\sigma)$, therefore this strategy is 2-competitive.

### 2.3 Paging Problem

The optimal offline algorithm knows the future, and applies a strategy which we will refer to as Longest Forward Distance ($LFD$). At any given time $i$, $LFD$ will remove either a page that is never again requested within the sequence if such a page exists, otherwise it will remove the page that is requested again the farthest into the future, the $LFD$ page.

There are many possible online algorithm strategies:
1. Random Selection. This algorithm selects a page to remove at random.

2. Least Frequently Used (LFU). This algorithm selects the page that has been requested the fewest number of times thus far.

3. First-In First-Out (FIFO). This algorithm will always remove the page that has been in memory for the greatest amount of time.

4. Least Recently Used (LRU). This algorithm will keep track of how long it has been since each page has been accessed, and will swap out the least-recently used page.

5. Last-In First-Out (LIFO). This algorithm will always remove the page that has been in memory for the least amount of time.

$LIFO$ can be shown to perform very poorly. Assume $k$ page slots and consider the following sequence: $\langle \sigma_1, \ldots, \sigma_{k-1}, p, q, p, p, q, p, q, \ldots \rangle$, where $\sigma_i \neq p, \sigma_i \neq q, \forall i \in [1..k-1]$.

In the above sequence, $LIFO$ will choose to swap out $p$ to make room for $q$, then it will swap out $q$ to make room for $p$, etc, for as long a sequence as we choose to construct, ensuring that there is no bound on the worst-case scenario of our competitive analysis.

**Theorem 2.3.1** \(LFD\) is an optimal algorithm for the offline paging problem.

**Proof 2.3.2** Consider a finite sequence of page requests. Suppose \(\exists\) algorithm \(A\) that produces fewer page faults than \(LFD\) on some input sequence. In order to produce a different result, at some unique point in time \(A\) and \(LFD\) must choose to do something different. Specifically, \(\exists\ i\ such\ that\ \sigma_i\ is\ not\ in\ fast\ memory\ and\ A\ removes\ some\ page\ p\ and\ \text{LFD}\ removes\ page\ q,\ \text{with}\ p \neq q\).

Define \(t\ as\ the\ first\ time\ that\ algorithm\ A\ removes\ page\ q\ after\ time\ i\).

Let time \(j\ be\ the\ time\ at\ which\ the\ next\ request\ for\ p\ after\ time\ i\ occurs,\ and\ let\ l\ be\ the\ time\ at\ which\ the\ next\ request\ for\ q\ after\ time\ i\ occurs. Note
that By definition of LFD, \( i < j < l \) (otherwise LFD would have removed \( q \) at time \( j \), not \( p \)).

Now we define algorithm \( A_1 \) which is identical to \( A \), except at time \( i \) \( A_1 \) removes \( q \) instead of \( p \) (mimicking LFD); and at time \( t \) \( A^* \) removes \( p \) (assuming \( \sigma_i \neq p \); if \( \sigma_i = p \) then \( A_1 \) would do nothing at time \( t \) as there would be no page fault). Our analysis can now be divided into two cases:

- **Case 1:** if \( t < j \), then \( \text{COST}_{A_1}(\sigma) = \text{COST}_{A}(\sigma) \), both have a page fault at time \( j \) when requesting \( p \).
- **Case 2:** if \( j \leq t < l \), \( A \) faults at time \( t \) and removes \( q \), while \( A_1 \) does not incur a page fault. Thus after \( t \), \( \text{COST}_{A_1}(\sigma) \leq \text{COST}_{A}(\sigma) + 1 \), that is, \( A_1 \) suffers one fewer page faults than \( A \).

Thus, \( A_1 \) is either same or better performance-wise than \( A \), but the page fault sets of \( A_1 \) and LFD are identical at least to time \( t \) which is strictly greater than \( i \). This argument can be repeated iteratively to generate an algorithm \( A_{i+1} \) from \( A_i \), where \( A_{i+1} \) matches LFD for a strictly greater length of time than \( A_i \) \( \forall i \). Since the sequence has finite length, it must be the case that eventually, we will derive algorithm \( A_n \) which matches LFD for the entire sequence which has the same or better performance than \( A \). More precisely we have \( \text{COST}_{LFD}(\sigma) = \text{COST}_{A_n}(\sigma) \leq \text{COST}_{A_{n-1}}(\sigma) \leq ... \leq \text{COST}_{A_1}(\sigma) \leq \text{COST}_{LFD}(\sigma) \), which a contradiction.

We may thus conclude \( \neg \exists \) any algorithm \( A \) which outperforms LFD.

**Theorem 2.3.3** LRU is a k-competitive online strategy.

**Proof 2.3.4** Bracket \( \sigma \) (The input sequence) into groups containing \( k \) page faults. i.e. \( \sigma = \sigma_1, ... \sigma_i, [k \text{ page faults}], [k \text{ page faults}], ... \)

Let \( \sigma_i \) be the first page fault. If we can show that LFD has at least 1 page fault in each bracket then LRU is k-competitive because we defined the brackets as each containing \( k \) faults in LRU. Proof by case analysis. Since there are \( k \) page faults per phase, one of two cases must hold.

- **Case 1:** LRU faults twice on some page \( p \) in the course of a phase. LRU must have requested at least \( k + 1 \) pages in that interval (otherwise it
would not have thrown out p). So LFD sees requests for at least $k + 1$ pages in this phase so it must have at least one fault. As LFD cannot contain all $k + 1$ pages in $k$ memory.

- **Case 2:** LRU faults on $k$ distinct pages in the course of a phase. Let $p_i$ denote the last page fault before the current phase.
  - Case 1: There is a page fault on $p_i$ by LRU during the current phase. Look at $p_i$ and the bracket after it. Because of the way LRU works, we may conclude that $k$ distinct pages were requested prior to the fault on $p_i$ as otherwise LRU would not have removed $p_i$ and would not have faulted on it. Thus looking at $p_i$, $k + 1$ distinct pages were requested in the phase, thus at least 1 page fault is incurred by LFD.
  - Case 2: There does not exist a page fault on $p_i$ by LRU on $p_i$ at some time in the current phase, so we need $k - 1$ distinct pages to max out the remaining $k-1$ slots.

Since $C_{LFD}(\text{phase}) > 1$ (Cost of LFD on a phase), and $C_{LRU}(\text{phase}) = k$, LRU is a $k$-competitive strategy.

**Theorem 2.3.5** Cannot do better than $k$-competitive with a deterministic algorithm.

**Claim 2.3.6** For any deterministic online algorithm $A$ there exists a sequence of requests $\sigma$ such that the cost $C_A(\sigma)$ is arbitrarily large and you can force $C_A(\sigma) \geq k * C_{\text{min}}(\sigma)$ even if we have only $k + 1$ pages in memory.

**Proof 2.3.7** Given an algorithm $A$, we can construct an input sequence $\sigma$ that forces $a$ to do $k$ times worse than the optimal solution (offline) algorithm for the same sequence. This strategy is fairly simple, it can be stated as the last page removed by $A$ is the next page requested. We can do this $k * (d + 1)$ times where $d$ is an integer (the first $k$ pages may be hits), resulting in $C_A(\sigma) = k * d$. 
In the optimal offline strategy, if OPT faults on some page \( p \), it will not fault on the next at least \( k - 1 \) page requests. Therefore \( \text{Cost}_{\text{OPT}}(\sigma) \leq \frac{k^* (d+1)}{k} \frac{\text{Cost}_A(\sigma)}{k} \).

\[ k * \text{Cost}_{\text{OPT}}(\sigma) \leq \text{Cost}_A(\sigma) \]

Therefore any deterministic algorithm \( A \) can do no better than \( k \)-competitive.

3 Randomized Online Algorithms

Philosophically, measuring performance against this kind of adversary is not a realistic model. This adversary is too strong. Suppose we have an adversary, but we can toss a coin that that adversary cannot predict. We can average over all coin tosses based on a single bad sequence. This is often a more realistic model.

Definition 3.0.8 A **randomized online algorithm** \( A \) is a distribution of a deterministic online algorithm \( A_x \) where \( x \) is \( A \)'s set or random event outcomes.

This is against what we call an “oblivious offline adversary” who can set up any sequence of requests \( \sigma \), but does not know the coin tosses used in \( A_x \) ahead of time.

Definition 3.0.9 A randomized online algorithm is \( \alpha \) **competitive** against an oblivious offline adversary if there exists a constant \( C \) such that for all input sequences the following holds: \( E[C_A(\sigma)] \leq \alpha * C_{\min}(\sigma) + C \) where the expected value is determined across the distribution of all possible values for \( x \).

Note: there is also an adaptive adversary where you (the algorithm) get to choose what happens, but he (the adaptive adversary) gets to set later requests. This is a stronger adversary than the oblivious adversary.
3.1 Randomized Marking Algorithm for The Paging Problem

1. Mark all pages in memory (asserts reset on first page fault).

2. When page p is requested
   - if page p is in memory, serve it and mark it.
   - Otherwise: if all pages in memory are marked, unmark all pages, swap page p with a uniformly randomly selected unmarked page in memory.

This algorithm will be analyzed in Online Algorithms part 2.

4 Acknowledgement

These lecture notes are heavily adapted from the notes of a classmate, John Trafton, who took Lenore Cowen’s Comp 260 in 2009, whose notes were adapted from a class taught by Michel Goemans at MIT in 1992.