Lecture 9: Online Algorithms

All of the algorithms we have studied so far operate on the assumption that they always have access to their entire input. This is not always the case in real life; it is also important to consider algorithms which receive their input in pieces and must make decisions based on this limited information. These are known as online algorithms.

1 Online Problem Examples

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

1. The Canadian Traveler’s Problem
2. The Ski Rental 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, turning this into a simple pathfinding problem. The traveler can run any shortest path algorithm, ignoring the roads which are blocked, and 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 they have already passed through an endpoint of said road section.
1.2 Ski Rental Problem

It costs $1/day to rent skis and $T$ to buy them. After $S$ days, you will break your leg and never ski again.

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 $T < S$, then buy.
- If $T > S$, rent every day until you break your leg.
- If $S = T$, then buying and renting will both cost the same.

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 Online Algorithm Performance

Since online algorithms need to make decisions before they have all the necessary information, it is often impossible to find an optimal solution even when we disregard time complexity. What, then, is a good measure of an online algorithm? We can measure the relative effectiveness of an online algorithm using competitive analysis.

**Definition 2.0.1.** An online algorithm \( A \) is said to be \( \alpha \)-competitive if for any input sequence \( \langle \sigma \rangle \), \( \text{COST}_A(\sigma) \leq \alpha \text{COST}_{\text{MIN}}(\sigma) \), where \( \text{COST}_A(\sigma) \) denotes cost of online algorithm \( A \) on \( \langle \sigma \rangle \) and \( \text{COST}_{\text{MIN}}(\sigma) \) denotes the cost of the optimal offline algorithm on \( \langle \sigma \rangle \).

2.1 Ski Rental Problem

For this problem, there exists an optimal offline algorithm \( \text{MIN} \), which, as we have already discussed, is to simply buy if and only if \( S \geq T \). Thus we either rent for \( S \) days and spend \( $S \), or we buy immediately and spend \( $T \). Hence the value of \( \text{MIN} \) on any given input is simply \( \min\{S, T\} \). Let us consider two possible online algorithms for the ski rental problem.

Define simple algorithm \( A_1 \) as follows:

- • Buy on the very first day.

Regardless of the value of \( S \), \( A_1 \) will spend \( T \). Note that if \( S \geq T \) then \( \text{MIN} \) would have elected to buy as well, and the two algorithms have made an identical decision. \( A_1 \) will only deviate from \( \text{MIN} \) when \( S < T \). The worst case scenario is that we break our leg the very first day. In this case \( \text{MIN} \) provides the optimal solution of $1, while \( A_1 \) led to a cost of $T. Thus \( A_1 \), in the worst-case, is a factor of \( T \) greater than \( \text{MIN} \). By our definition above, we may say that \( A_1 \) is \( T \)-competitive.

Define simple algorithm \( A_2 \) as follows:

- • Rent for the first \( T - 1 \) days.
• If we still have not broken our leg by day $T$, then buy on day $T$.

If $S < T$, then $A_2$ rented every day, resulting in a cost of $S$, which is optimal. If we break our leg after day $T$, then $A_2$ spent $T - 1$ while renting, in addition to $T$ when buying, resulting in a total cost of $2T - 1$. The optimal strategy here would have been to buy immediately, which would result in a cost of $T$. Since $2T - 1 < 2T$, we can say that $A_2$ is 2-competitive.

### 2.2 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.

There are many possible online algorithm strategies:

• 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.

• Most Recently Used (MRU). This algorithm works like LRU, except it swaps out the most recently used page.

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

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

• Random Selection. This algorithm selects a page to remove at random.

The listing above is not comprehensive; there are many other paging algorithms as well.

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

We will compare LRU to LFD, but first we must demonstrate that LFD is in fact an optimal offline algorithm.

**Theorem 2.2.1.** LFD is an optimal algorithm for the offline paging problem.

**Proof.** Consider a finite sequence of page requests. Suppose there exists an 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, there exists an \( i \) such that \( \sigma_i \) is not in fast memory and \( A \) removes some page \( p \) and LFD removes page \( q \), with \( p \neq q \).

Define \( t \) as the first time at which 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 \( p \) at time \( j \), not \( q \)).

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 \( t \) \( A_1 \) 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 \( COST_{A_1}(\sigma) = COST_A(\sigma) \), 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 \( t \), \( COST_{A_1}(\sigma) \leq COST_A(\sigma) + 1 \), 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 \(A_i\). More precisely we have \(COST_{LFD}(\sigma) = COST_{A_n}(\sigma) \leq COST_{A_{n-1}}(\sigma) \leq \ldots \leq COST_{A_1}(\sigma) \leq COST_{LFD}(\sigma)\), which a contradiction.

We may thus conclude that there does not exist any algorithm \(A\) which outperforms \(LFD\). \(\Box\)

Claim 2.2.2. \(LRU\) is \(k\)-competitive.

**Proof.** Assume an input sequence with non-trivial quantity of distinct pages \((k + 1\) or more). First observe that both algorithms start with at least \(k\) successes (to fill up the initial \(k\) slots). We want to show that \(COST_{LRU}(\sigma) \leq k \ast COST_{LFD}(\sigma)\).

Let \(\langle \sigma \rangle = \sigma_1, \sigma_2, \ldots, \sigma_i, [\sigma_{i+1}, \ldots, \sigma_j], [\sigma_{j+1}, \ldots, \sigma_m], \ldots\),

where \(\sigma_i\) is the first page fault and \(\sigma_j\) is page fault \(k + 1\), and \(\sigma_m\) is page fault \(2k + 1\). We will refer to the time encapsulated within a bracket as a “phase”. Note that it is the final element within a bracket that is a page fault; the first element within a bracket is not necessarily a page fault.

Our goal is to show that \(LFD\) makes at least one page fault per phase, as \(LRU\) makes precisely \(k\) page faults in each phase by our construction.

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 phase. \(k + 1\) distinct pages are requested, including \(p\), during the phase, since \(k\) pages must be requested in between the two faults on page \(p\) for \(LRU\) to fault on it twice. At least \(k + 1\) distinct pages requested implies \(LFD\) must fault at least once, since \(LFD\) cannot have all of these pages already in memory.

- **Case 2:** \(LRU\) faults on \(k\) distinct pages in the bracket. Let \(p_1\) denote the last page fault before the request phase (in previous phase).
Case 2a: There exists a page fault on $p_1$ by LRU at some time in the current phase. Consider $p_1$ and the bracket following it. We may then conclude that $k$ distinct pages were requested prior to the fault on $p_1$, otherwise LRU would not have removed $p_1$ and would not have faulted on it. Thus including $p_1$, $k + 1$ distinct pages were requested in the phase, thus at least 1 page fault is incurred by LFD.

Case 2b: There does not exist any page fault by LRU on $p_1$ at some time in the current phase, so we need $k-1$ distinct pages to max out the remaining $k-1$ slots ($p_1$ is in memory at the start of the phase and never faulted on, so it is occupying one page slot throughout the entire phase). Since we have requested $k$ distinct pages, and have only $k - 1$ spaces for them, we can be certain at least 1 page fault is incurred by LFD within the phase.

Thus, LRU is $k$-competitive, as LFD faults at least once for every $k$ faults made by LRU.

Indeed, we cannot construct a deterministic algorithm that is better than $k$-competitive, which we will now show.

**Theorem 2.2.3.** For any deterministic online algorithm $A$, there exists a sequence of requests $\langle \sigma \rangle$ such that $\text{COST}_A(\sigma)$ is arbitrarily large and $\text{COST}_A(\sigma) \geq k \ast \text{COST}_{\text{MIN}}(\sigma)$, as long as there are at least $k + 1$ pages in our universe.

**Proof.** Given a deterministic online algorithm $A$, we will construct a sequence to force $A$ to produce a solution that is at least $k$ times worse than the solution of $\text{MIN}$ for the same sequence.

- First, request pages $p_1$ through $p_k$.
- Request $p_{k+1}$, and denote the page $A$ will remove to make room for $p_{k+1}$ as $q$, which we can determine since $A$ is deterministic.
- Request $q$, and denote the page $A$ will remove to make room for $q$ as the 'new q' and repeat this step $k \ast d$ times (for some integer $d$).
The sequence constructed above has length \( j = k(d+1) \), and \( A \) executing on this sequence yields \( kd \) page faults, as the first \( k \) requests are the only non-faults. Hence \( \text{COST}_A(\sigma) = kd \).

If the optimal offline algorithm \( MIN \) faults on some page \( \sigma_i \), it will not fault on the next at least \( k - 1 \) requests. Hence,

\[
\text{COST}_{MIN}(\sigma) \leq j/k = (d + 1) = (d + 1) \times k/d = \frac{\text{COST}_A(\sigma)}{k} \times \frac{d+1}{d}
\]

Multiply both sides by \( k \) and taking the limit as \( d \to \infty \) yields

\[
k \times \text{COST}_{MIN}(\sigma_j) \leq \text{COST}_A(\sigma_j).
\]

Therefore no deterministic algorithm can be better than \( k \)-competitive.

In addition to providing a bound on the competitiveness of a deterministic online algorithm, this example shows that an adversary could, knowing exactly how the algorithm works, construct a sequence of page requests which will force the algorithm to perform badly.

### 3 Randomized Online Algorithms

Philosophically, measuring performance against this kind of adversary is not a realistic model. This adversary is too strong. Suppose instead that we have an adversary, but we can toss a coin whose outcome the adversary cannot predict.

**Definition 3.0.4.** A randomized online algorithm \( A \) is a distribution of a deterministic online algorithm \( A_x \) where \( x \) is \( As \)et of random event outcomes.

An “oblivious offline adversary” knows how algorithm \( A \) works, but does not have any prior knowledge of the random variable \( x \). We now formally define competitive analysis for randomized online algorithms.
Definition 3.0.5. 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: $E[COST_A(\sigma)] \leq \alpha \cdot COST_{min}(\sigma) + C$, where the expected value is determined across the distribution of all possible values for $x$.

3.1 Randomized Marking Algorithm for Paging

Algorithm M:

1. Begin by initializing all pages as marked.

2. When page $p$ is requested:
   (a) If $p$ is not in memory:
      i. If all pages in memory are marked, unmark them all.
      ii. Swap $p$ into memory and swap out a random, uniformly selected unmarked page.
   (b) Mark page $p$ in memory and serve it.

4 Acknowledgement

These lecture notes are heavily adapted from the notes of several students, who took Lenore Cowen’s Comp 260 in previous years:

- John Trafton (2009)
- Alex Tong (2016)

The original lecture notes were adapted from the notes of a class taught by Michel Goemans at MIT in 1992.