Advanced Algorithms: Lecture 8 Notes

Dana Randall Spring 2024

Sarthak Mohanty

Online Algorithms

Before this class, our algorithm design has been as follows: take in some input, put it though some black box, and produce some output.

In the last section, we expanded our definition a little, by allowing some randomized “bits" to be fed in alongside our input. We saw that this approach allowed for algorithms that came with security and time complexity benefits.

Now, instead of allowing our algorithm more infomration, we *restrict* some information to our algorithm.

Definition. An **online algorithm** is one that produces its output step-by-step, as more parts of the input become available. More formally, the input of an online algorithm is a sequence of requests $\sigma = \sigma_1, \ldots, \sigma_n$. The algorithm needs to handle these requests in order. When handing σ_t, the algorithm does not know the future requests $\sigma_{t'}$ for $t' > t$.

While you may not have encountered online algorithms formally yet, there are many areas you have seen similar motifs. For example, online algorithms and machine learning are both concerned with problems of making decisions from limited information. In fact, there are a collection of results in Computational Learning Theory that fit nicely into the “online algorithms" framework, such as the Winnow algorithm. [1].

[1] Professor Vempala, who taught this class last semester, covered this here

Paging

Computer systems often have multi-level storage systems. Earlier, there were only RAM and long-term storage devices. Nowadays there are also several levels of cache memory (L1, L2, L3 cache).
The setting for this problem is as follows: we have a two-level computer memory system, composed of fast memory (of size k), and (slow memory of size $n > k$.) We need to answer a request sequence $\sigma = \sigma_1\sigma_2 \dots \sigma_m$, where σ_i is an item to be paged.

Any page someone wants needs to be moved to fast memory. If fast memory (cache) is full and I need a page that is not in it, we encounter a page fault where I now need to swap in a page.

Offline Approach

Suppose I already have a list of all the pages I intend on accessing (i.e. P5, P7, P5, P1, P1000, etc.). Because we know the order of pages to be accessed, we can deterministically find an optimal strategy to minimize page faults. One such strategy is **longest forward distance** (LFD), where we replace the page whose next request is latest (in the future).

Theorem. LFD is an optimal page replacement policy.
Proof. Fix a sequence σ. For all i, let $C_0^{(i)}$ denote the state of the cache after request σ_i is served, and let $C_0 = (C_0^{(i)})_{1 \le i \le n}$.

Let $C_t = (C_t^{(i)})_i$ be defined inductively as follows: up to and including time t, use LFD to choose which page to evict when there is a page fault. This determines the state of the cache at each timestep until time t. Then, follow the choice of evictions of C_{t-1} for all times $> t$ except one defined as follows: assume that at time t algorithm C_{t-1} evicts page u whereas LFD evicts page v. Let t' be the first time when, either C_{t-1} evicts page v, or C_{t-1} brings page u back into the cache (by evicting some page u'), whichever comes first. In the first case, at time t' C_t will evict page u, and in the second case, at time t' C_t will evict page u'. Either way, after serving request at time t' the state of the cache is the same for C_{t-1} and for C_t. By definition of LFD, between time t

and time t' there can be no request for page v, so C_t is well-defined and incurs no more pages faults than C_{t-1}. By induction C_t is at least as good as C_0. For $t = n$, C_n is exactly LFD, so LFD is at least as good as C_0. This was for an arbitrary C_0, so LFD is optimal.

Online Approach

Definition: Let $OPT(\sigma)$ and $ALG(\sigma)$ denote the cost incurred by the best possible offline algorithm OPT, and the online algorithm ALG, respectively. The **competitive ratio** of a deterministic online algorithm ALG is defined as

$$\max_{\sigma} \frac{ALG(\sigma)}{OPT(\sigma)}.$$

Sometimes we choose to ignore constant additive factors in the cost of the algorithm. This leads to the following alternative definition of competitive ratio. In this case, when the algorithm's cost involves no constant additive factor, we call the corresponding ratio a strong competitive ratio.

Definition: A (deterministic) online algorithm is **c-competitive** iff

$$\exists a \geq 0 \text{ s.t.} \underbrace{\forall \sigma, ALG(\sigma) \leq c \cdot OPT(\sigma) + a}_{\text{c-competition}}$$

We first state a lower bound for deterministic online algorithms:

Theorem. For any deterministic online algorithm A, the competitive ratio of $ALG \geq k$.

Proof. We first bound the performance of LFD as $\leq \frac{|\sigma|}{k}$ for k distinct pages.

Next, note that for *any* deterministic online algorithm A an adversary can construct a sequence where the next request is always the page that is not contained in the cache of A. Thus there exists some σ such that $A(\sigma) = |\sigma|$.

Thus, the competitive ratio r is given by

$$r = \frac{ALG(\sigma)}{OPT(\sigma)} \geq \frac{|\sigma|}{|\sigma|/k} = k.$$

From the above result, we can categorize common paging strategies by whether they are k-competitive or not. Some k-competitive strategies include

- Least Recently Used (LRU): Replace the page that hasn't been used for the longest time
- First in First Out (FIFO): Replace the page that has been int he fast memory longest

Strategies that don't achieve this competitive bound include

- Last In First Out (LIFO): Replace the page most recently moved to fast memory.
- Least Frequently Used (LFU): Replace the page that has been used the least

Marking Algorithms

Marking algorithms are a class of online paging algorithms that achieve the k-competitive bound. In fact, all the above mentioned competitive algorithms are some form of a marking algorithm.

A marking algorithms works as follows:

- Partition the sequence of accesses σ into phases, where each phase has exactly k *distinct* pages and mark each new page brought in.
- At the end of a phase, unmark all pages in fast memory.

Note that a marking algorithm never evicts a page which is already marked. This key observation leads to an easy analysis of these algorithms:

Theorem. All marking algorithms are k-competitive.

Proof. Any marking algorithm costs, or has the number of page faults, at most k per phase. Any adversary costs OPT at least 1 per phase. Let p be the first request in a phase. OPT must have p in cache after this request. There are k distinct requests so OPT must fault.

Randomization[2]

The competitive ratio that we've achieved thus far is pretty substandard. However, it works rather well in practice. This is because competitive ratio measures the "worst-case" performance, but the distribution of the input sequence usually makes it so our average performance is pretty ok.

The issue remains that if an adversary wanted to defeat our system, it would be pretty easy. We know already that one way to beat this adversary is randomize our decision process. Of course, this requires the adversary to produce the sequence in advance – after knowing our algorithm, but before the algorithm response to the sequence. This is called an *oblivious adversary*.

For example, consider this modified Marking algorithm: When a request to page arrives, if it is in cache then mark it. If not then remove an unmarked page at random, fetch from disk, and mark it. If all pages in cache are marked when a page faults, unmark all pages first before applying the above strategy.

This algorithm is $2H_k$ competitive, where $H_k = 1 + \frac{1}{2} + \cdots + \frac{1}{k}$ is the kth harmonic number. (Proof omitted).

More Adversaries

Besides the oblivious adversary, there are two other types:

[2]Although we did not discuss this section in class, I think it is important to know

Definition: The **oblivious adversary** must construct the request sequence in advance (based only on the description of the online algorithm but before any moves are made!).

Since a randomized algorithm is not totally predictable, randomization makes it hard for the oblivious adversary to generate a difficult request sequence for the algorithm.

Definition: The **adaptive online adversary** makes the next request based on the algorithm's answers to previous ones, but serves it immediately.

This "medium adversary" can be considered as an active attacker. When you make a decision, they see that decision, and can respond appropriately.

Definition: The **adaptive offline adversary** makes the next request based on the algorithm's answers to previous ones, but serves them optimally at the end.

This adversary is extremely strong. After observing the entire sequence of moves by the algorithm, the adversary then serves (or responds to) them optimally.

There are two important theorems[3] on the power of the last two adversaries:

Theorem. If there is a randomized algorithm that is α-competitive against any adaptive offline adversary, then there also exists an α-competitive deterministic algorithm.

Theorem. If G is a c-competitive randomized algorithm against any adaptive online adversary, and there is a randomized d-competitive algorithm against any oblivious adversary, then G is a randomized $(c \cdot d)$-competitive algorithm against any adaptive offline adversary.

These two theorems basically indicate that randomization provides no power

[3]Ben-David et. al, On the Power of Randomization in Online Algorithms, STOC 1990

against the adaptive offline adversary, and provides little (if any) power against the adaptive online adversary.