Lecture 10: Online Algorithms II

1 Introduction

Last lecture, we made a deterministic online algorithm for the paging problem. We showed that the number of page faults made was at worst $k$-times the number made by the optimal offline algorithm, where $k$ is the size of the fast-memory cache. We additionally showed that an adversary could force this algorithm to be as bad as $k$-times optimal by appropriately devising their sequence of requests.

Now, to confuse our adversary, we introduce randomness into our algorithm, and show that we can do much better on average.

Definition 1.0.1. The Paging problem (restated)

We have memory with fixed size $k$. A sequence of requests for pages arrives one at a time. If a page is in memory, it can be served. Otherwise, it must be brought into memory at a fixed cost before it can be served.

Definition 1.0.2. A randomized online algorithm is a distribution of deterministic online algorithms $A_X$, where $X$ is a set of random event outcomes.

Definition 1.0.3. A randomized online algorithm $A$ is $\alpha$-competitive against an offline adversary if there exists a constant $C$ such that for all input sequences $\sigma$,

$$E[C_A(\sigma)] \leq \alpha C_{\text{min}}(\sigma) + C$$

(1)

where $C_A$ is the cost incurred by $A$, $C_{\text{min}}$ the minimum cost, and the expectation is over all possible coin flips for $A$ (not over the input sequences).
2 Randomized Marking Algorithm

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 \( p \) in to memory, and swap out a random, uniformly selected unmarked page.
   
   (b) Mark page \( p \) in memory and serve it.

As a page gets marked once it is used, the mark serves as an indicator that a page was used more recently than the unmarked pages. In that sense, this algorithm is somewhat like LRU.

To help our analysis, we introduce the following definitions.

Definition 2.0.4. For any sequence \( \sigma \) of input requests to \( M \), we say that a request \( \sigma_i \) is an unmarking request if all pages in memory are marked when \( \sigma_i \) arrives, and \( \sigma_i \) is a request for a page not already in memory.

Definition 2.0.5. We use the unmarking requests to divide \( \sigma = \sigma_1 \sigma_2 \sigma_3 \ldots \) into phases. For \( i > 0 \), define phase \( i \) to be the subsequence of \( \sigma \) starting at the \( i \)th unmarking request, up to but not including the \( i + 1 \)st unmarking request (or to the end of the sequence, if there are exactly \( i \) unmarking requests).

Definition 2.0.6. For each phase \( i \), we use \( S_i \) to denote the set of pages that were in memory just before the start of phase \( i \).

Remark 2.0.7. At the start of each phase, no pages in memory are marked. At the end of the phase, all \( k \) pages are marked. Therefore the phase ends exactly when \( k \) distinct pages are accessed.
Remark 2.0.8. Once a page has been accessed the first time in a phase, it will be marked and hence will remain in memory until the end of the phase. For each page, then, $M$ incurs a charge of at most 1 in a phase, the first time the page is requested.

Remark 2.0.9. Importantly, our definition of a phase depends only on the input sequence $\sigma$. The phases do not depend on the randomized choices the algorithm makes for swapping pages out. Furthermore, $S_i$ also depends only on the input sequence. $S_i$ will always consist of the previous $k$ distinct pages requested before phase $i$.

By the second remark, we need only look at the first-time requests of the pages to analyze the cost of a phase. We will divide such requests into two types: those we always have to pay for, and those we might not have to pay for.

Definition 2.0.10. A first-time request for page $p$ in phase $i$ is a clean request if $p \not\in S_i$. $M$ will have to pay a cost of 1 for such a request, no matter the results of the coin-tosses.

Definition 2.0.11. A first-time request for page $p$ in phase $i$ is a dirty request if $p \in S_i$. This means that we may not have to pay for $p$, since it’s already in memory at the start of the phase. But if $p$ was randomly ejected to make room for an earlier clean request, then we do incur a charge of 1.

Finding the expected cost of the dirty requests will help us analyze $M$.

2.1 Expected cost of a dirty request

Suppose $\sigma_j$ is a dirty request to page $p$ in phase $i$. Suppose that up to this point in the phase, there have been $s$ dirty requests and $c$ clean requests. We know that when $p$ is requested, the $c + s$ pages requested so far this phase must be in memory.

Lemma 2.1.1. The expected cost of the $s + 1$st dirty request in a phase is $\frac{c}{k-s}$, where $c$ is the number of preceding clean requests in this phase.
Proof. Use $L_i$ to denote the subset of $S_i$ of pages that have not yet been requested. As $s$ pages of $S_i$ have been requested, and the clean requests, by definition, were not part of $S_i$ to begin with, $|L_i| = |S_i| - s = k - s$.

Let $U$ denote the set of $k - s - c$ pages left in memory that have not yet been requested this phase. By definition, $U \subseteq L_i$, and specifically $U$ consists of the pages of $L_i$, minus the ones that were randomly ejected for the clean requests. Since all members of $L_i$ have stayed unmarked from the start of the phase until now, they are all equally likely to have been ejected.

Let $f$ denote the probability that a page in $L_i$ is not in memory when $\sigma_j$ arrives. Equivalently, $(1 - f)$ is the probability it is in memory, and hence is in $U$. These are equal for all such pages. Hence, we have

$$k - s - c = |U| = E[|U|] = \sum_{q \in L_i} Pr(q \in U) = \sum_{q \in L_i} (1 - f) = (1 - f)(k - s) \quad (2)$$

Solving $k - s - c = (1 - f)(k - s)$ for $f$ shows $f = \frac{c}{k-s}$.

As $f$ is the probability that any page in $L_i$ is not in memory, it is in particular the probability that $p$ is not in memory. In that case, we incur a charge of 1, and in the other case a charge of zero. Hence the expected cost for the first request to $p$ is $f = \frac{c}{k-s}$.

2.2 Bounding the expected total cost of $M$ on a phase

Suppose there are $l_i$ clean requests in phase $i$. As we have already remarked above, based on how we have defined a phase, this number depends only on the input sequence, and not on the coin tosses of $M$. As all other first-time requests are dirty, there must be $k - l_i$ dirty requests.

Lemma 2.2.1. The expected cost of $M$ on phase $i$ of $\sigma$ is no more than $l_i H_k$ where $H_k = 1 + \frac{1}{2} + \frac{1}{3} + \ldots + \frac{1}{k}$ is the $k$th Harmonic number.

Proof. The number of clean requests before a given dirty request is bounded by $l_i$, so by lemma 2.1.1, the expected cost of all dirty requests is bounded above by

$$D = \frac{l_i}{k} + \frac{l_i}{k-1} + \frac{l_i}{k-2} + \ldots + \frac{l_i}{k - (k - l_i - 1)} \quad (3)$$
The total cost of all clean requests is simply $l_i$, so the total expected cost is bounded above by

$$D + l_i = l_i(1 + \frac{1}{l_i + 1} + \frac{1}{l_i + 2} + ... + \frac{1}{k}) \leq l_i H_k$$  \hspace{1cm} (4)

\[\square\]

### 2.3 Bounding total cost of any offline algorithm on a phase

Consider any offline algorithm $A$. Let $\sigma$ be any input sequence; we will compare $A$’s behavior to $M$’s on $\sigma$. We divide $\sigma$ into phases based on $M$’s algorithm. Let $C_i(A)$ be the total cost $A$ incurs during phase $i$.

We then define a potential function $\phi_i$, which is the number of pages in $A$’s memory that are not in $M$’s memory, just before the start of phase $i$ (and $\phi_{n+1}$ the number at the end of the algorithm). This function is well defined because the pages in $M$’s memory at the start of a phase are deterministic, determined only by $\sigma$.

Using these, we establish the following bounds on the costs incurred by $A$ during a phase.

**Lemma 2.3.1.** $C_i(A) \geq l_i - \phi_i$

*Proof.* $M$ receives $l_i$ clean requests in phase $i$. These pages were not in $M$’s memory at the start of the phase. By definition of the potential function, at most $\phi_i$ of them will be in $A$’s memory. In other words, the amount that won’t be in $A$’s memory is at least $l_i - \phi_i$. \[\square\]

**Lemma 2.3.2.** $C_i(A) \geq \phi_{i+1}$

*Proof.* At the end of phase $i$, $A$ has $\phi_{i+1}$ pages in memory that $M$ does not. Hence $M$ also has $\phi_{i+1}$ pages that $A$ does not. Call that set of pages $P_i$.

Note that every page in $M$’s memory was accessed at least once during the phase, so all pages in $P_i$ must have been accessed at some point in phase $i$. But since they were all ejected by $A$ before the end of the phase, $A$ made at least $|P_i| = \phi_{i+1}$ ejections in phase $i$. \[\square\]
Lemma 2.3.3. \( C_i(A) \geq \frac{1}{2}(l_i - \phi_i + \phi_{i+1}) \)

Proof. We simply combine the previous two lemmas, dividing each side of the inequalities by two and then summing them.

\[
\frac{1}{2}C_i(A) + \frac{1}{2}C_i(A) \geq \frac{1}{2}(l_i - \phi_i) + \frac{1}{2}\phi_{i+1}
\] (5)

\[\square\]

2.4 The competitive factor of \( M \)

The total cost of an algorithm is the sum of its costs over all phases. Let \( C(M) \) be the total expected cost of \( M \). From the result of section 2.2, we see that \( C(M) \leq H_k \sum l_i \).

For any algorithm \( A \), we bound the total cost by using lemma 2.3.3 and summing over all phases. Letting \( C(A) \) be the cost incurred by \( A \),

\[
C(A) \geq \frac{1}{2}[(l_1 - \phi_1 + \phi_2) + (l_2 - \phi_2 + \phi_3) + ... + (l_n - \phi_n + \phi_{n+1})]
= \frac{1}{2}\left[\sum l_i - \phi_1 + \phi_{n+1}\right]
\geq \frac{1}{2}\sum l_i
\] (6)

Where \( \phi_1 = 0 \), as the algorithms’ memories begin in the same state. All other \( \phi_i \) terms cancel out in the telescoping series, except \( \phi_{n+1} \), the pages \( A \) may have in memory at the end of the algorithm that \( M \) does not. But as that term is nonnegative, we can drop it to attain the final bound.

So we have that \( \sum l_i \leq 2C(A) \). Combining these two bounds shows that \( C(M) \leq H_k \sum l_i \leq 2H_kC(A) \). Therefore \( M \) is \( 2H_k \)-competitive.

There exists a proof that no randomized algorithm can do better; but we won’t get into that.
3 Adwords Problem

Definition 3.0.1. The Adwords Problem

There are \( n \) bidders, and each bidder \( i \) has a known budget \( B(i) \). When request \( j \) arrives, each bidder produces a number \( bid(i, j) \), which is their bid, assumed to be much smaller than their total budget. We choose a bidder, and that amount is subtracted from their budget and added to our profit. Of course, we want to maximize our profit.

We can think of the bidders as being advertisers bidding to put up an ad. The requests could be a Google search, for example, where advertisers would want to advertise their products when someone makes a relevant search. The assumption that bids are much less than a budget, then, is reasonable, as bids would be on the order of cents, while budgets would be on the order of hundreds or thousands of dollars.

3.1 Online Greedy Algorithm for Adwords

The obvious algorithm is the greedy one.

1. Always take the highest bid from among the bidders that still have budgets.

The limiting factor in the problem is the budget – if budgets were unlimited, this greedy strategy would clearly be optimal. But with budgets, this strategy runs the risk of exhausting one bidder’s budget, after which point we may miss out on some profit.

Example 3.1.1. Suppose there are two advertisers with the same budget. For searches on laptops, bidder 1 bids \( c \) cents, and bidder 2 bids \( c + \epsilon \) cents for an arbitrarily small \( \epsilon \). For searches on headphones, bidder 1 bids 0 cents and bidder 2 bids \( c \) cents.

The greedy algorithm can be thwarted if enough searches on laptops arrive to exhaust bidder 2’s budget, and if all further searches are for headphones. In that case, the profit made is \( n(c + \epsilon) \), where \( n \) is the number of requests
needed to exhaust one of the budgets. No profit is made after the first \( n \) requests because bidder 1, the only bidder with a remaining budget, refuses to pay for headphone searches!

For the optimal solution, however, we can choose bidder 1 for the first \( n \) bids, then choose bidder 2 for the rest, netting a profit of \( c \) on each bid for an overall profit of \( 2nc \).

This shows that the profit from the Greedy algorithm could be arbitrarily close to half the optimal, showing that greedy is no better than \( \frac{1}{2} \)-competitive. But we will show it is no worse than that.

### 3.2 Competitive factor of the greedy algorithm

**Theorem 3.2.1.** Greedy is \( \frac{1}{2} \)-competitive.

**Proof.** Let \( Q \) be an instance of the adwords problem, containing a set of queries, and the bidders’ bids and budgets. We will consider the algorithm recursively. Suppose the first query is awarded to bidder \( j \) at price \( p \) by the greedy algorithm. Let \( Q' \) be the instance of the problem obtained by subtracting \( p \) from \( j \)’s budget, and removing the first query from the set.

Let \( Alg(Q) \) be the total profit made by the greedy algorithm on instance \( Q \). Let \( OPT(Q) \) be the optimal profit on instance \( Q \). It’s clear that

\[
Alg(Q') = Alg(Q) - p
\]  

(7)

Suppose, instead of taking bid \( j \) for the first query, the optimal solution instead takes bid \( k \). How much could the optimal solution for \( Q' \) differ from the optimal solution for \( Q \)? As all other bidders are identical, the only possible differences are due to \( j \) having a lower budget in \( Q' \), and \( k \) having a higher budget.

In \( Q' \), bidder \( j \) has \( p \) fewer dollars than in the optimal solution. Now, bidder \( j \) may exhaust their budget earlier, preventing them from bidding on \( p \) worth of queries.
Additionally, by choosing bidder \( k \), the optimal solution made at most \( p \) – if it had made more, the greedy algorithm would have chosen \( k \). Therefore,

\[
OPT(Q') \geq OPT(Q) - 2p
\]  

(8)

We could further recurse, considering instances \( Q'' \) and \( Q''' \), and the same arguments would hold each time. Since at each step the optimal solution gains at most twice the profit made by the greedy solution, the greedy algorithm is \( \frac{1}{2} \)-competitive.

3.3 Other Algorithms

There exist two algorithms that achieve a competitive factor of \( 1 - \frac{1}{e} \approx 0.632 \), which has been shown to be an optimal factor.

The first is the MSVV algorithm. In it, each bid is scaled by the remaining budget. Bidders with larger remaining budgets are preferred, so no one’s budget is exhausted too quickly. The scaling factor is defined to maximize the competitive factor.

**Definition 3.3.1.** For each bidder \( i \) and request \( j \), define the scaled bid \( \hat{b}_{i,j} = \text{bid}(i,j) \times \psi(f_i) \) where \( f_i \) is the fraction of unspent budget of \( i \), and \( \psi(x) = 1 - e^{-x} \).

Then the MSVV algorithm is simply

1. When query \( j \) arrives, award it to the bidder for which \( \hat{b}_{i,j} \) is maximized.

The other algorithm achieving the same bound comes from Buchbinder, Jain, and Naor. That algorithm uses linear programming and duality.

4 References and Acknowledements

The material on randomized online paging comes originally from notes of Michel Goemans for an MIT Course in Advanced Algorithms given in Fall 1991.
The proof that the greedy algorithm for adwords is a $\frac{1}{2}$-optimal approximation, follows directly from the more general result of Lehmann, Lehmann and Nisan: Here are all the references:

