CS174 Lecture 19

Paging, Online Algorithms and Adversaries

You should be familiar with the paging problem. A computer has a cache memory which can hold k pages. Then there is a much larger slow memory (or disk) which can hold an arbitrary number of pages. When the computer accesses a memory item, if the page containing the item is in cache, the access is extremely fast and is said to be a *hit*. If the item isn't in memory, then the page containing it must be moved into cache, and some other page moved out. This is called a cache *miss*.

When a miss occurs, a decision must be made about which cache page to move back to main memory. The algorithm that does this is called an *online* algorithm. An online algorithm doesn't have a fixed input. Instead, an online algorithm receives its input incrementally, and must make output decisions in between the inputs. Each memory access request is an input to the paging algorithm. The algorithm must decide which page to move out, and then wait for the next memory request.

Because the behavior of an online algorithm depends on a stream of inputs interleaved with the online algorithm's outputs, in order to analyze the online algorithm, its necessary to have a model of what produces those inputs. Normally, we think of the inputs as deriving from an agent (a person or a program, or a person running a program) called an adversary. The word adversary suggests that the agent is working against you. That's usually not the case, but algorithm designers like to build algorithms that work well in any situation. That means in the worst possible situation. Assuming an adversarial agent can guarantee the performance of an online algorithm in all situations.

Deterministic Paging Algorithms

Some common deterministic online paging algorithms are:

1. LRU (Least Recently Used): Evict the item in cache whose last request was the longest time ago.

2. FIFO (First-In, First-Out): Evict the item that was the first item paged in, i.e. which has been in cache for the longest time.

3. LFU (Least Frequently Used): Evict the item in the cache that has been requested least often.

Both 1 and 3 are intuitive assuming pages that were not referenced much in the past are unlikely to be referenced much in future. They assume that page access is relatively independent of time. FIFO assumes memory access is localized in time, i.e. that the processor is likely to work fairly heavily on pages for some time interval and then stop using them.

Note that both 1 and 3 require the maintenance of the smallest item in some ordering (frequency or time), probably with a heap.

To analyze paging algorithms, we already noted that we need to model the adversary, in this case the agent that chooses where the memory requests are. We will also need a new method for analyzing performance. Asymptotic analysis is of limited usefulness because the "input size" measure doesn't work very well. Online algorithms work with a long stream of "short" inputs. Normally, you don't want their complexity to depend on the length of the stream, which may grow without bound. The individual inputs are usually small and of fixed size, so measuring complexity in terms of them isn't helpful. What is very helpful is to understand how the online algorithm compares with an "offline" algorithm for the same task. The offline algorithm is assumed to know the future as well as the past.

Competitive Ratios

A competitive ratio is the ratio of some performance measure between an online and an offline algorithm for the same task. For the paging problem, the most natural performance measure is the number of cache misses.

Let $\rho = (\rho_1, \rho_2,\ldots,\rho_N)$ be a sequence of page requests.

Let $f_A(\rho_1, \rho_2,\ldots,\rho_N)$ denote the number of times that the online algorithm A misses on the page request sequence.

Let $f_O(\rho_1, \rho_2,\ldots,\rho_N)$ denote the number of times that an optimal offline algorithm O misses on the page request sequence.

The algorithm A is said to be C-*competitive* if there is a constant b such that for every sequence of requests $(\rho_1, \rho_2,\ldots,\rho_N)$, then

$$f_A(\rho_1, \rho_2,\ldots,\rho_N) - C\times f_O(\rho_1, \rho_2,\ldots,\rho_N) \leq b$$

This is something like a big-O bound. The constant b is there to allow a bit of slop for small values of N, and to ensure that C need only track the asymptotic ratio of the two functions. An algorithm will be C-competitive for all C values larger than some minimum (actually an infimum, the "minimum" over an uncountable set). The infimum of C's such that the algorithm A is C-competitive is denoted C_A.

In case it isn't clear, a small C is "good". It means that the online algorithm is only a little worse than the optimal offline algorithm. A big C means that the online algorithm has many times more cache misses than the optimal offline algorithm.

An optimal offline algorithm

If you can look into the future, it's not hard to figure out how to do paging. You don't have much to lose by ejecting the page whose next request is as far as possible in the future. This will cause a miss then, but ejecting anything else will cause a miss sooner. It would seem that this scheme will maximize the time between misses, and therefore minimize the total number of misses. It turns out that it does, and so is an optimal offline paging algorithm. This offline algorithm is called MIN.

If k is the number of cache pages, LRU and FIFO are both k-competitive. That may not sound very impressive, but LFU is even worse. It does not have a bounded competitive ratio. In fact LRU and FIFO are really doing quite well in light of the following result:

Theorem

Let A be any deterministic online algorithm for paging, then $C_A \geq k$.

So LRU and FIFO are both as good as deterministic algorithms can be.

These results suggest that it's a good idea to look for randomized online algorithms. Before we go further, we need to say something more about the adversaries for randomized algorithms.

Oblivious and Adaptive Adversaries

Recall that the adversary is the agent that chooses the sequence of page requests. For a deterministic paging algorithm, the adversary need only output a deterministic sequence of requests. The paging algorithm is deterministic, and so always responds the same way to each request sequence. There is no reason for the adversary to make decisions based on what the paging algorithm does, because the adversary could figure that out (by simulating the paging algorithm if needed).

But when the paging algorithm is deterministic, there are two possible adversaries. An *oblivious adversary* produces a sequence of paging requests that is independent of the behavior of the paging algorithm. It is just like the adversary for a deterministic algorithm.

An *adaptive adversary* produces page requests that depend on all the actions of the paging algorithm up to the current time, and those depend on random choices which are not known ahead of time.

An oblivious adversary models most user programs. Programs are normally not aware of residency of pages in cache, and their memory access patterns do not depend on that.

An adaptive adversary models the operating system and certain other cache-aware applications.

The adaptive adversary is perhaps too strong a model, even for programs to which is applies. As an adversary, it tries to make the online algorithm perform as poorly as possible. In this case, it tries to cause cache misses by watching the cache contents. That is very strange behavior for an operating system, even a poorly designed one. In any case, we consider only the oblivious adversary here.

Paging Against an Oblivious Adversary

First, the bad news:

Theorem

Let R be a randomized algorithm for paging, and let C_R be its competitive ratio against an oblivious adversary. Then $C_R \geq H_k$, where H_k is the k^{th} harmonic number.

Recalling that H_k is about ln k, that implies that any online randomized paging algorithm must do about ln k times worse than the optimal offline algorithm for some page request sequence.

Now the good news:

Theorem

There is a probabilistic, online paging algorithm called the Marker algorithm which is $2H_k$-competitive.

The marker algorithm uses a single bit called the marker bit m_i to the i^{th} cache location.

Marker Algorithm

1. Do forever:
    2. Set marker bits $m_i = 0$ for i = 1,…,k.
    3 Do until all $m_i = 1$ for i = 1,…,k
        4 Accept a page request
        5 If requested page is in cache location i,
            6 Then set $m_i = 1$
        7 Otherwise,
            8 Choose an unmarked cache location j at random
            9 Evict cache location j, and bring new page into j
            10 Set $m_j = 1$

You can think of the marker bits as "protecting" cache pages from being evicted. Initially, all the cache pages are unprotected. Each cache hit causes that page to be protected. A cache miss causes an unprotected page to be thrown out, and then the new page is inserted in cache and protected right away. When all the pages are protected, the algorithm can't throw anything out, so it just unprotects everything, and the process starts over.

Intuitively, this algorithm will tend to keep frequently-used pages in cache. If a page is referenced at least every k/2 requests, even if all the intermediate requests cause cache misses, there is only a 50% chance that that page will be evicted (by the k/2 random, distinct evictions).

Proof of Competitiveness of the Marker Algorithm

Lets now prove the last theorem, namely that the Marker algorithm is $2H_k$-competitive. We consider two algorithms, the Marker algorithm itself and an optimal offline algorithm. The cache has k locations, and there is a series of requests $\rho_1, \rho_2,\ldots,\rho_N$.

Rounds

The marker algorithm proceeds in a series of rounds, corresponding to the outer loop of the code above. At the beginning of each round, all the marker bits are reset. Suppose a round begins with

ρ_i and ends with ρ_j, that is, ρ_j is the start of the next round. During the round, every location in the cache was marked. Each request for a new page causes it to be in the cache and for its marker bit to be set, whether it started in cache or not. So all k marker bits are set after there are k distinct page requests in ρ_i,..., ρ_{j-1}, and ρ_j must be a request for a $(k+1)^{st}$ page.

Consider the requests in any round. Call an item *stale* if it is unmarked, but was marked in the last round. Stale items were requested in the last round, but not yet in this round. They may or may not still be in the cache of the Marker algorithm. Call an item *clean* if it is neither stale nor marked. A clean item was not requested in the last round, nor yet in this round, and will not currently be in the cache of the Marker algorithm.

Let l be the number of requests to clean items in a round. We will show that the number of misses for the optimal offline algorithm is at least $l/2$. We will also show that the number of misses of the Marker algorithm is at most lH_k. That proves that the Marker algorithm is $2H_k$-competitive.

Offline algorithm performance

Let S_O denote the set of items in the cache of the offline algorithm, and S_M denote the set of items in the Marker algorithm's cache. Define

$d_I = |S_O - S_M|$ at the beginning of the round,

$d_F = |S_O - S_M|$ at the end of the round.

Let M_O be the number of misses of the offline algorithm during the round. As we noted above, clean items are not in the Marker algorithm's cache at the beginning of a round, and there are l of these. The contents of the offline algorithm's cache differs from the Marker algorithm's in d_I items at the start of the round. So at least $l - d_I$ of the clean items will not be in the offline algorithm's cache either. Each of these items will be requested during the round and will generate a cache miss for the offline algorithm. So the offline algorithm has at least $l - d_I$ cache misses.

Another type of cache miss for the offline algorithm occurs because it looks ahead. The contents of the Marker cache S_M at the end of the round are all and only the k pages requested during the round. But the offline algorithm's cache contents S_O can be different, which means it must have thrown out some of those pages. The variable d_F counts pages that were requested in this round and which have already been thrown out by the offline algorithm. Each of those must have been a cache miss (in order to cause a page to be thrown out). So the number of misses of the offline algorithm is at least d_F. Putting both arguments together, we see that the number of misses for the offline algorithm is at least:

$\max(l - d_I, d_F)$

The max function is a little tricky to use in recurrences, so instead we notice that the max of two numbers is at least their average. Therefore the number of cache misses is at least:

$\max(l - d_I, d_F) \geq (l - d_I + d_F)/2$

This is a bound for the number of misses in a round. We would like to get the average or

amortized number of misses for many rounds. If we add up this sum for many rounds, we notice that the d_F for one round is equal to the d_I for the next. So all the d_I's and d_F's cancel except the first and the last, and their contribution is negligible if we sum over enough rounds. It follows that the average or amortized number of cache misses for the optimal offline algorithm is

$l/2$

Marker algorithm performance

First of all, notice that every request to a clean page causes a cache miss. There are l of these. There is another type of cache miss which is a request for a stale page that has been evicted in the current round. There are $k - l$ requests for stale pages in the round, because an item requested for the first time in a round is unmarked at that time (and therefore is either stale or clean). The probability of stale page requests causing a cache miss is maximized when all the requests for clean pages (which cause certain cache misses) come before the requests for stale pages (which may or may not). So we assume that happens.

There are $k - l$ requests for stale pages. Since the l clean page requests go first, when the first stale page request happens, l out of k of the pages have been evicted (at random), so the probability that the first stale page request causes a miss is

l/k

At the second step, the expected number of misses is therefore $l + l/k$ and the probability of another miss is

expected number of misses / $k = (l + l/k)/k = l(k+1)/k^2$

since $(k+1)/k \leq k/(k-1)$ which we can bound the above as $l(k+1)/k^2 \leq l(k/(k-1))/k = l/(k-1)$. If we repeat this process, we find the next term is bounded by $l(k-2)$ etc, and in general, the probability of a miss at the i^{th} stale page request is at most

$l/(k - i + 1)$

The sum from $i = 1$ to $k - l$ of this value is $l(H_k - H_l)$. So the total expected number of misses for the Marker algorithm, counting both clean and stale page requests is

$l + l(H_k - H_l)$

and since H_l is at least 1, the above value is at most lH_k which completes the proof of the bound.