CSE 347 Lecture 22

On-line Algorithms II: Competitive Analysis

1. The elevator problem
2. Caching
3. Load Balancing

Sometimes in on-line algorithms, as we make decisions in response to our inputs that arrive one at a time, our decisions impact our subsequent performance. While this obviously happens in data structures, we may face similar difficulties in other problems. Here's an everyday example:

Suppose you are in a hurry, and you have to decide whether to take the elevator or to take the stairs. Of course, on the one hand you spend time waiting you don't get back if you hold out for the elevator, but on the other hand, once you start up the stairs, you can't generally dash back to the elevator if it comes — you're committed to taking the stairs all the way. What should you do?

Let's suppose taking the elevator takes E seconds (once it arrives — say $E = 10$) and let's suppose that the stairs take S seconds, perhaps $S = 45$. If you knew when the elevator would come, what is the optimal strategy? If the elevator comes in

less than $S-E$ seconds, then taking it costs $E+S-E=S$ seconds, so the elevator is faster. Otherwise, $E+$(time waiting) $\geq E+S-E=S$, so the stairs are faster. But we don't know the future. If we take the stairs right away, we'd spend S seconds, but if the elevator comes one second later, we are doing much worse than optimal. By contrast, if we stick it out, the elevator might be out of order, and we might wait forever. In this case, to quantify the performance we use the <u>competitive ratio</u>: the maximum, over all input sequences, of the ratio $\frac{\text{Algorithms' cost}}{\text{OPT}}$ where OPT is the best strategy in hindsight (equiv., if we knew the future). So, taking the stairs has a competitive ratio of $\frac{S}{E}$ (= 4.5 here) whereas waiting no matter what has a ratio of ∞. How well can we do?

Here is a good strategy: wait until we <u>could</u> have taken the stairs ($S-E-1$ secs) and if the elevator doesn't come, we take the stairs then. Observe: our algorithm matches OPT unless we take the stairs, in which case OPT only pays S, and we pay $S+S-E-1$. Thus, our competitive ratio is $\frac{2S-E-1}{S} = 2-\frac{E+1}{S} \leq 2$.

<u>Claim</u> No deterministic strategy has a competitive ratio better than $2-\frac{E+1}{S}$.

<u>Proof</u>: Suppose we wait an additional Δ seconds. Then

our competitive ratio is $\frac{2S-E-1+\Delta}{S} = 2-\frac{E+1}{S}+\frac{\Delta}{S} > 2-\frac{E+1}{S}$.
Likewise, if we wait Δ fewer seconds, the elevator may come immediately after, in which case OPT takes the elevator, and the ratio is $\frac{2S-E-1-\Delta}{E+S-E-\Delta} \geq \frac{2S-E-1}{S} = 2-\frac{E+1}{S}$. ▨

Caching

A cache is a small amount of fast memory that reduces the delay for repeated accesses to the same address, if it is stored in the cache. In this case, the access is a "cache hit," and otherwise (if the value must be fetched from slow memory) it is a "miss." The problem in caching is, how do we manage our k slots to minimize the number of misses? If we knew the full sequence of accesses, the optimal policy is to "evict" (forget) the value that will be next accessed furthest in the future; this is easy to show by an exchange argument: evicting anything else just incurs a miss sooner. We'll denote the number of misses incurred by this policy on the sequence σ by $f(\sigma)$. Commonly used policies such as evicting the Least Recently Used value achieve a competitive ratio of k, and deterministic policies can't do better. But, here is a randomized policy with a $O(\log k)$ competitive ratio: each value can be "marked" or "unmarked" (initially, it's unmarked)

On a request to an address a:
- If a is not in the cache
  - If there are unmarked addresses, evict one at rand.
  - Else, unmark everything, and evict one at random

| | Put a in the vacant slot
Mark a and return its value.

When we unmark everything, we call this the "end of a phase" (and the beginning of the next phase).

Claim: A phase ends with the $k+1^{th}$ distinct address request.

Proof? Until we have k marked addresses, each request to a distinct address results in one more marked address in the cache. The $k+1^{th}$ then sees all k slots marked, and triggers the end of the phase. ▨

We'll call an unmarked item "fresh" if it wasn't marked in the previous phase, and otherwise we call it "stale."

Let c_j denote the number of accesses to fresh addresses in the jth phase, and suppose σ has r phases.

Lemma: $f(\sigma) \geq \frac{1}{2}\sum_{j=1}^{r} c_j$

Proof: We'll let f_j denote the number of misses incurred by the optimal policy during the jth phase of σ. Thus, $f(\sigma) = \sum_{j=1}^{r} f_j$. Now, by our claim, there are k distinct requests during any j^{th} phase; in the $j+1^{th}$, the c_{j+1} fresh requests are (by definition) distinct from these k. Therefore, between phases j & $j+1$, there are $k + c_{j+1}$ distinct requests, so any policy (including the optimal policy) incurs $\geq c_{j+1}$ misses. (In phase 1, we incur c_1 misses on our empty cache.) So, letting $f_0 = 0$, we find for $j = 0, \ldots, r-1$ that $f_j + f_{j+1} \geq c_{j+1}$, and thus $\sum_{j=0}^{r-1}(f_j + f_{j+1}) \geq \sum_{j=0}^{r-1} c_{j+1} = \sum_{j=1}^{r} c_j$, and $\sum_{j=0}^{r-1}(f_j + f_{j+1}) \leq 2\sum_{j=1}^{r} f_j$ ▨

Now that we've bounded the performance of the optimal

policy, we only need to show that we're unlikely to accidentally evict many of the $k-c_j$ stale addresses that the optimum policy avoids missing in phase j. Let $M(\sigma)$ denote the (random) number of misses incurred by our randomized marking policy, and X_j denote the (random) number of these incurred during phase j.

<u>Theorem</u> $\mathbb{E}[M(\sigma)] \le H_k \sum_{j=1}^{r} c_j \le O(\log k \cdot f(\sigma))$

<u>Proof</u>: We write $M(\sigma) = \sum_{j=1}^{r} X_j$. Now, during the jth phase, we incur c_j misses to the fresh addresses. For the stale addresses, on the ith such request, suppose that $c \le c_j$ of the fresh requests have been made. Then there are $k-c-(i-1)$ (unmarked) stale addresses in the cache out of the $k-(i-1)$ stale addresses. Thus, since each of these $k-(i-1)$ stale addresses was equally likely to be evicted on each of the c fresh requests, the particular address of the ith stale request incurs a miss with probability $\frac{c}{k-(i-1)} \le \frac{c_j}{k-(i-1)}$. Thus, our number of misses on the jth phase is $\mathbb{E}[X_j] \le c_j + \sum_{i=1}^{k-c_j} \frac{c_j}{k-(i-1)} = c_j\left[1 + \sum_{\ell=c_j+1}^{k} \frac{1}{\ell}\right] \le c_j H_k$

Therefore, in total across all phases, our algorithm incurs $\mathbb{E}[M(\sigma)] = \sum_{j=1}^{r} \mathbb{E}[X_j] \le \sum_{j=1}^{r} c_j H_k \le 2 H_k f(\sigma)$ by our lemma. We know $H_k \le 1 + \ln k$, so indeed, $\mathbb{E}[M(\sigma)] \le 2(1+\ln k) f(\sigma)$ ▨

That is: the competitive ratio is $2(1+\ln k)$. Not bad, considering that $f(\sigma)$ knows the future!

Load Balancing

Here is yet another variant of scheduling: we have m id-

entical machines, $M_1, \ldots, M_m$, and jobs with arbitrary processing times $t_1, t_2, \ldots, t_j, \ldots, t_n$ arrive one at a time. When a job arrives, we must assign it to one of the machines. This time, our objective is to minimize the <u>makespan</u>: if machine i is assigned the set of jobs A_i, it runs for time $T_i = \sum_{j \in A_i} t_j$. The makespan is the largest of these, $\max_{i=1,\ldots,m} T_i$. (T_i is the <u>load</u> of the ith machine.)

Unlike caching, this is a problem where even if we knew the entire set of jobs in advance, minimizing the makespan is NP-complete. So there is no nice way to compute the optimum. But, we can bound it.

What are some lower bounds on the makespan? One is $\max_j t_j$ — some machine must be assigned this job. Another is $\frac{1}{m}\sum_{j=1}^{n} t_j$: there is $\sum_{j=1}^{n} t_j$ total work to be done, and some machine must get at least a $\frac{1}{m}$-fraction of it.

A simple greedy algorithm turns out to do well: assign the new job j to the machine i with the smallest load T_i.

<u>Theorem</u> This algorithm has a competitive ratio of 2.

<u>Proof</u>: First observe that when we assign job j to machine i, its previous load, $T_i - t_j$, was minimal: every other machine had at least this load. Therefore, $\sum_{k=1}^{m} T_k \geq m(T_i - t_j)$ in our final assignment. But then, $\sum_{k=1}^{m} T_k = \sum_{j=1}^{n} t_j$ is our total work, so $T_i - t_j \leq \frac{1}{m}\sum_{j=1}^{n} t_j \leq T_{OPT}$. Thus, we always have $T_i \leq T_{OPT} + t_j$. We also know for every job j, $t_j \leq T_{OPT}$, so

moreover, $T_i \leq T_{OPT} + T_{OPT} = 2T_{OPT}$ at all times. Thus, in our final allocation, $\max_i T_i \leq 2T_{OPT}$, as needed ▨