Recitation 3: Competitive Analysis

1 Competitive Analysis

Let $R = \{r_1, r_2, ...\}$ be a sequence of requests processed by an algorithm. An *online* algorithm is one that processes each request with no knowledge of subsequent requests. To evaluate the performance of an online algorithm, we can compare it with an optimal *offline* algorithm OPT which knows the sequence of requests beforehand and processes each request optimally with this knowledge. This type of comparison is known as *competitive analysis*. We say an algorithm A is α-competitive if, for any sequence of requests R,

$$C_A(R) \leq \alpha \cdot C_{OPT}(R) + k$$

where $C_A(R)$ denotes the cost of algorithm A on request sequence R, and k is some constant. The smaller the competitive ratio α, the better the algorithm.

In lecture we saw a competitive analysis of the MTF (move-to-front) self-organizing list. Here, we will explore an online approach to the scheduling problem and the LRU paging algorithm.

1.1 The Scheduling Problem

We consider a simple version of the scheduling problem. We have a cluster of m identical machines which can process jobs. A sequence of jobs $\sigma = J_1, J_2, ..., J_n$ arrives online, one job at a time. Each job J_i has a known processing time p_i. When a job arrives, we must immediately schedule it on one of the machines. Our goal is to minimize the time at which the last job finishes, also known as the makespan.

We consider the greedy algorithm, which always assigns an incoming job to the least-loaded machine, and we prove the following lemma:

Lemma. The greedy algorithm is 2-competitive.

Proof. Suppose we are given an arbitrary job sequence σ. Let $T_G(\sigma)$ be the makespan of the schedule produced by the greedy algorithm, and let $T_{OPT}(\sigma)$ be the makespan of the schedule produced by an optimal offline algorithm.

Let p_{max} denote the maximum processing time of all jobs. We let t_1 denote the length of the time interval, starting at time $t = 0$, during which all machines are busy in the online schedule. Let $t_2 = T_G(\sigma) - t_1$. Using the definition of the greedy algorithm, we know that the job that finishes last in the greedy schedule starts at some time $t <= t_1$. We can

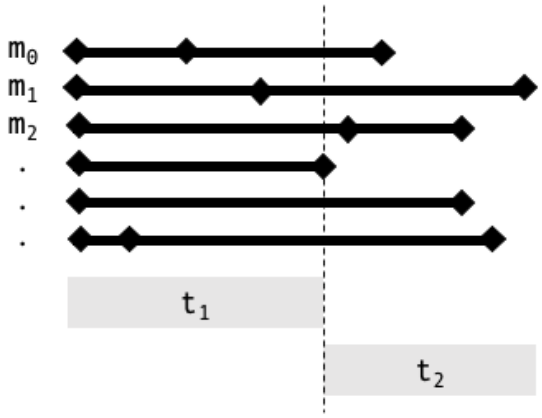

Figure 1: An example of scheduling assignments given by the greedy algorithm.

see this by considering the following contradiction: if the job that finishes last had started at a time after t_1, then all machines must be occupied until that time, so t_1 would not be the length of the time interval during which all machines are busy. This implies that the processing time of the last job is at least t_2. Thus, $t_2 \leq p_{max}$.

We also note that the minimum possible makespan is equal to $\frac{1}{m}\Sigma_{i=1}^{n} p_k$. Clearly, $t_1 \leq \frac{1}{m}\Sigma_{i=1}^{n} p_k$.

We can see that $T_{OPT}(\sigma) \geq \max\{\frac{1}{m}\Sigma_{i=1}^{n} p_k, p_{max}\}$.

Thus, $T_G(\sigma) = t_1 + t_2 \leq 2\max\{\frac{1}{m}\Sigma_{i=1}^{n} p_k, p_{max}\} \leq 2 * T_{OPT}(\sigma)$, so the greedy algorithm is 2-competitive.

1.2 The Paging Problem and the LRU Algorithm

1.2.1 The Paging Problem

Computers often use *caching* to speed up memory accesses. In the paging problem, we have two types of memory: a small, fast memory unit (the cache) and a large, slow memory unit. The cache holds k pages of memory. We are given a sequence of page requests. For each request, if the page is in the cache, we have a *hit* and can quickly return the page. Otherwise, we have a *fault* and must retrieve the page from the slow memory. After retrieving, we choose some page to evict from the cache (if it is filled), and replace it with the requested page. Our goal is to determine which page to evict from the cache to minimize the number of page faults across a sequence of requests.

1.2.2 Proving k-competitiveness of the LRU algorithm

One approach to this problem is the Least Recently Used (LRU) algorithm: upon faulting, we evict the page which was accessed least recently. It turns out this simple algorithm is k-competitive.

Let the optimal offline algorithm be OPT. Given a sequence of requests R, we wish to show that if the LRU algorithm faults k times, then OPT faults at least once.

How can we say when the optimal algorithm will fault? The key insight is that *any sequence of $k+1$ distinct requests necessarily results in a fault for OPT*. Why? You can only store k pages in the cache; if $k+1$ distinct pages are requested, at least one of them won't be found. This is known as the *pigeonhole principle*.

We can leverage the pigeonhole principle to "force" the optimal algorithm to fault. The intuition is that using the LRU algorithm, whenever we have k faults, we can show that $k+1$ distinct pages were requested, causing a fault for OPT.

To be more formal, split our sequence of requests R into chunks $R_0, R_1, R_2, \ldots$ such that the LRU algorithm has at most k faults in R_0 and exactly k faults in the remaining chunks.

Lemma. Given a sequence of requests R_i in which the LRU algorithm has exactly k faults, OPT will have at least one fault.

Proof.

Let p be the last distinct request before the LRU algorithm's first fault. If we can show that R_i contains requests for k distinct pages that are not p, then by the pigeonhole principle, any algorithm must have at least 1 page fault in R_i.

Case 1: The LRU algorithm faulted on k distinct pages in R_i.

If none of the faults are p, then we are done. If one of them is p, then our algorithm must have evicted p at some point. Since p was the most recently used at the beginning of R_i, the LRU algorithm would only have evicted p if there were k different requests between the beginning of R_i and the eviction of p. Therefore, R_i contains k distinct pages that are not p.

Case 2: The LRU algorithm faulted on some page x more than once in R_i.

After the first fault, x would be the most recently used, and by the LRU algorithm, there would need to be k requests that are not x between the beginning of R_i and the eviction of x. This gives $k+1$ distinct requests in R_i, at least k of which are not p.

Note that for our proof we did not know any details about the behavior of the optimal algorithm. If you're curious, it turns out that the optimal offline algorithm for the paging problem is the "farthest-in-future" eviction policy.

Exercise. Prove that no deterministic online algorithm for the paging problem can be less than k-competitive.

Hint: Consider the *cruel adversary*, who initially requests a page not in the cache, and thereafter always requests the page the algorithm just evicted.