

Lecture 10: Online Algorithms - Part II

1 The Paging Problem

We have a section of fast memory that can hold k pages of data. The rest of the pages must be stored on disk or in slower memory. When a page p is requested, the following occurs:

- If p is already loaded in fast memory, p is served without cost.
- Else, a page fault occurs. A page in memory must be swapped out so p can be swapped in. This results in a cost of 1.

Definition 1.0.1 *A strategy for the paging problem is a procedure for choosing which page in fast memory should be removed when a page fault occurs.*

Definition 1.0.2 *An online algorithm for the paging problem is **C-competitive** if the number of faults that occur is always $\leq C \times$ the number of faults that occur in the offline algorithm, regardless of the input sequence.*

In the last lecture, we showed that the least recently used (LRU) strategy is k -competitive.

2 A Randomized Strategy for the Paging Problem

Using a randomized strategy allows us to do better than k -competitive for the paging problem.

Definition 2.0.3 *A randomized online algorithm, A , is a probability distribution over deterministic online algorithms A_x where x is the sequence of A 's coin tosses over the course of the algorithm.*

2.1 Adversary Models

Although the C -competitive definition does not explicitly mention an adversary, there is one that is present implicitly. Since the inequality must hold over the entire set of input sequences, this set would include any arbitrarily bad sequence an adversary could generate.

For the purposes of evaluating the randomized model, there are the following possible adversary models:

1. The adversary generates and presents the entire input sequence to the algorithm. The algorithm then executes and tosses coins. The adversary cannot make any changes to the sequence as the algorithm is executing. This is known as an **oblivious adversary**.
2. The adversary presents one page request at a time to the algorithm. After each request, the algorithm tosses one coin and executes one iteration. The adversary can see the result of this iteration and may use this in generating its next page request. This is known as a **adaptive adversary**.
3. The algorithm does all its coin tosses in advance. The adversary has access to the results of all tosses and can generate a sequence of requests based on it. This model is essentially deterministic from the adversary's point of view.

For our analysis, we will use an **oblivious adversary**.

2.2 Competitiveness for a Randomized Algorithm

Definition 2.2.1 *A randomized online algorithm A_x is said to be α -competitive against an oblivious adversary if $\exists c$ such that*

\forall input sequences σ , $E[C_{A_x}(\sigma)] \leq \alpha \text{OptOffline}(\sigma) + c$

2.3 The Randomized Marking Algorithm

The following describes our randomized strategy for the paging problem:

1. Mark all pages in memory
2. When a page p is requested:
 - If page p is in memory, serve it, and then mark it.
 - Else,
 - If all pages in memory are marked, un-mark all pages.
 - Swap page p with a uniformly randomly selected unmarked page in memory.
 - Mark page p .

2.4 Analysis of the Randomized Marking Algorithm

Suppose we have a sequence σ of requests to the randomized marking algorithm (RMA).

Definition 2.4.1 σ_i is an **unmarking request** if when σ_i arrives, all pages are marked and if σ_i is a request for a page not in memory.

First, we want to divide the sequence of page requests into phases where each page ends with the next unmarking request.

Let S_i be the set of all pages in memory before phase i begins.

- A phase ends when k pages are marked. This implies that k distinct pages are marked in a phase.
- Since a page is marked once it enters a phase, a page stays in memory until the end of the phase.

The cost of sequence $\sigma =$ the total cost of all phases.

Claim 2.4.2 *Regardless of the coin tosses of the RMA, a phase is always k distinct page faults.*

Definition 2.4.3 *A clean page request is a request for a page that does not belong to S_i*

Definition 2.4.4 *A dirty page request is a request for a page that does belong to S_i*

Since a clean request is never in memory at the start of a phase, it will always cause a fault to occur. Therefore, the cost of a clean request will always be 1.

A dirty request was, by definition, in memory at the beginning of the phase. Depending on your algorithm, it may still be in memory or it might have gotten swapped out. Therefore, the cost of a dirty request could be either 1 or 0 depending on whether that page has been swapped out or not.

To analyse this algorithm, we need to determine how likely we are to have to pay for a dirty request. In other words, we need to calculate the expected cost of a dirty request.

Let σ_i be a dirty request (DR) for page p at time j within phase i .

Assume that there have been s dirty and c clean request so far before time j .

Paying 0 for request p means that p is still in memory (and unmarked) after c clean and s dirty requests. p must therefore be one of:

$|\text{Unmarked}| = k - s - c$ pages in L_i where L_i is the set of $k - s$ pages in S_i that have not been requested so far in phase i .

All pages in L_i are equally likely to be in memory with probability $1 - f$ and thrown out with probability f .

$$\begin{aligned}
|Unmarked| = k - s - c = E[|Unmarked|] &= \sum_{q \in L_i} (1 - f) \\
&= (1 - f)(k - s)
\end{aligned}$$

so

$$f = \frac{c}{k - s}$$

is the probability that our page was thrown out. Thus the expected cost of the $s + 1^{st}$ dirty request is:

$$\frac{c}{k - s}$$

If there are l_i clean requests total in phase i , there are $k - l_i$ dirty requests. Therefore we can bound the expected cost of all dirty requests in a phase by :

$$\leq D = \frac{l_i}{k} + \frac{l_i}{k - 1} + \dots + \frac{l_i}{k - (k - l_i - 1)}$$

So the cost of all dirty and clean requests is

$$\begin{aligned}
&\leq D + l_i = l_i \left(\frac{1}{k} + \frac{1}{k - 1} + \dots + \frac{1}{k - (k - l_i - 1)} \right) \\
&\leq l_i \times H_k
\end{aligned}$$

2.5 Comparison of RMA and Offline Algorithm on a Phase

Let A be the optimal offline algorithm.

A potential function Φ_i is the number of pages in A 's memory that are not in RMA's memory just before phase i begins.

RMA receives l_i clean requests in phase i . By definition, these were not in RMA's memory at the start of phase i . Therefore, at least $l_i - \Phi_i$ are not in A 's memory at the start of phase i . So:

$$C_i(A) \geq l_i - \Phi_i$$

A has, by definition, Φ_{i+1} pages that were not in RMA's memory at the end of phase i . RMA has a set P_i of Φ_{i+1} pages not in A's memory. Every page in RMA's memory at the end of phase i was accessed in phase i . Thus A tossed P_i pages out of memory in phase i . So:

$$C_i(A) \geq \Phi_{i+1}$$

where

$$|P_i| = \Phi_{i+1}$$

Adding the two equations together gives us:

$$2C_i(A) \geq l_i + \Phi_{i+1} - \Phi_i$$

Summing this over all i phases yields:

$$2C_A \geq \sum_i l_i$$

or

$$C_A \geq .5 \left(\sum_i l_i \right)$$

Giving us the cost of algorithm A. Since we computed the cost of the RMA earlier to be $\leq l_i \times H_k$, we have just shown that the RMA is $2 \times H_k$ competitive, which is equivalent to $2 \times \log k$ competitive.

Our randomized algorithm is successful because it places its luck in its series of coin tosses, and not in the particular sequence of requests it receives. This effectively eliminates an adversary's ability to create a worst case. All of the cases where we perform very poorly are absorbed into the overall expected cost, which when averaged with the cases with very low cost, yields an algorithm with an overall good competitiveness.

3 The Scheduling Problem

We are given n jobs $\{j_1, j_2, \dots, j_n\}$ that take processing times $\{p_1, p_2, \dots, p_n\}$ all < 0 . We have m identical machines $\{m_1, m_2, \dots, m_m\}$. The goal is to

schedule all the jobs on the machines such that we minimize the makespan.

Definition 3.0.1 *The makespan is the time at which the last job completes.*

3.1 The List Scheduling Algorithm

The algorithm follows a greedy strategy. As jobs $j_1..j_n$ come in, whenever there is an available machine, it assigns the next job in the sequence to it. This is not an optimal solution to the problem. The following case proves this fact.

Case - We are given $m(m - 1)$ jobs of size 1, and 1 job of size m . The optimal solution would first assign the large job to one machine. It would then split all remaining jobs across the other machines. This results in a makespan of m . Instead, LS would begin assigning the small jobs to all available machines. Finally, it would assign the large job at the end. This solution results in a makespan of $2m - 1$.

Theorem 3.1.1 *List Scheduling produces a $2 - \frac{1}{m} \times OPT$ scheduling*

Proof 3.1.2 *Given job j , let S_j be the start time of the job and c_j be the time of completion.*

Let J_k be the job that completes last. This implies that no machine is idle before time S_k .

Let P_k be the processing time of the k^{th} job.

Let C_{max}^ denote the completion time under the optimal scheduling and C_{max}^{LS} denote the completion time under the List Scheduling algorithm.*

Trivially we can say that:

$$C_{max}^* \geq P_k$$

Our solution must be at least as costly as the perfect solution, if it exists, so we can also say:

$$C_{max}^* \geq \frac{1}{m} \sum_{i=1}^n P_i$$

We can prove our closer bound for LS to be:

$$\begin{aligned}C_{max}^{LS} &= C_k \\&= S_k + P_k \\&\leq \frac{1}{m} \sum_{j \neq k} P_j + P_k \\&= \frac{1}{m} \sum_{j \neq k} P_j + \left(1 - \frac{1}{m}\right) P_k \\&\leq C_{max} + \left(1 - \frac{1}{m}\right) C_{max} \\&= \left(2 - \frac{1}{m}\right) C_{max}\end{aligned}$$

Acknowledgement

Note that the material presented in class on paging is based largely on lectures by Michel Goemans at MIT in 1992.