6.854 Advanced Algorithms

Lecture 19: 11/17/2003 Lecturer: Erik Demaine, David Karger

Scribes: Kevin Zatloukal

Online Algorithms

19.1 Introduction

All of the algorithms we have studied so far in this course have taken the entire input up front and used it to compute an answer. In this section, we will study algorithms that are given the input a piece at a time and are forced to make a decision based solely on the input they have seen so far. The goal is for the quality of those decisions to be close to the quality we could achieve if we were given the entire input up front. We will call algorithms that get the input a piece at a time *online* algorithms and those that get the input all at once *offline* algorithms.

As mentioned above, our focus will be on finding online algorithms for which the quality of the decisions made can be guaranteed to be close to what we can achieve offline. The following definition makes this notion precise.

Definition 1 *We say that an online algorithm A has* competitive ratio α *(or A is α-competitive) if, for every input sequence σ, $C_A(\sigma) \leq \alpha\, C_{OPT}(\sigma)$, the cost of the solution produced by A is less than α times the cost of the solution produced by the optimal offline algorithm (OPT).*

This defines the competitive ratio in a worst-case sense, as is usual for deterministic algorithms. For randomized algorithms, we will consider the expected cost of the algorithm.

Definition 2 *A randomized online algorithm A has competitive ratio α if, for every input sequence σ, $\mathrm{E}\left[C_A(\sigma)\right] \leq \alpha\, C_{OPT}(\sigma)$.*

The analysis of online algorithms is similar to that of approximation algorithms in that we are comparing the quality of our solution to that of the (possibly unknown) optimal algorithm. As with approximation algorithms, our focus is on the quality of the solution rather than on running time or space (beyond basic polynomiality). Futhermore, in both cases, we are comparing our algorithm against an optimal algorithm that can cheat. In approximation algorithms, OPT may not run in polynomial time. In online algorithms, OPT can see the future because it is given all of the input up front.

Example 1 *Although we did not present the algorithm in this way orginally, we have already seen one example of an online algorithm: our approximation algorithm for load balancing. We can describe this as an online algorthm as follows.*

The problem is to assign a sequence of jobs to one of m identical machines. The jobs arrive at different times, and we must immediately choose the machine to which we will assign it. Each job comes with a load, the amount of computational resources it will consume. And all jobs are permanent – they never terminate. As before, the goal is to minimize the maximum load on any machine.

Previously, we saw a simple greedy algorithm for load balancing: we assign each job to the least loaded machine. Since this algorithm does not require looking at any of the jobs yet to arrive, it is an online algorithm. We proved before that, over any sequence of jobs, the maximum load produced by our algorithm is at most twice the maximum load of OPT. This means that the algorithm is 2-competitive.

19.2 Ski Rental Problem

After finals week, suppose that you head to a ski resort. You have the entire vacation as well as the Independent Activities Period to ski. Unfortunately, you know from past experience that, at some point, the fun will come to a premature end when fate steps in and breaks your leg. On each day until then, you have to make an important decision: should you rent ski equipment for \$1 or buy your own for B dollars? If you keep renting long enough, you will eventually find that you have spent more than B dollars, so it would have been cheaper to buy your own equipment at the beginning. However, if you buy your own, then you might break your leg that very day, wasting $B - 1$ dollars.

One idea would be to always buy on the first day. However, if you break your leg that day, then you spent B dollars while the optimum algorithm would have rented and spent only \$1, so this algorithm is only B-competitive.

A better idea is to rent for B days and then buy on day $B + 1$. To analyze this algorithm, suppose that you break your leg on day d. If $d \leq B$, then we always rented, which was the optimal decision. If $d > B$, then we will pay $2B$. The optimal decision would have been to buy on the first day, which would cost B dollars. But we only spent twice that, so this algorithm is 2-competitive.

19.3 Linear Search

Suppose that you have a large stack of papers on your desk. Every so often, you need one of them. To find it, you search down through the stack. Is there some way that you can put the stack back together in order to reduce the amount of time you spend searching in the future?

This can also be seen as a simple data structures problem. Suppose that we have a linked list of n items. We receive a sequence of m requests for items in the list. If the requested item is currently at index i in the list, then it will cost i to find it, searching from the front of the list. However, we have the option of rearranging the list (at some cost) in order to reduce the cost of future searches.

When managing a stack of papers, a natural strategy is to put the paper we found on the top of the stack (leaving the rest of the stack as is). If that paper is requested again in the near future, then the cost of accessing it will be small. This algorithm is called move-to-front (MTF). We will prove below that MTF is very effective. But first, let's consider a variant with simpler analysis. Suppose that instead of moving the accessed item all the way to the front, we simply move it forward by one. In other words, we transpose it with the item just before.

Theorem 1 *Transpose is not competitive.*

Proof: Let the list of items be $[1, \ldots, n-1, n]$. Consider the request sequence $n, n-1, n, n-1, \ldots$. The first request (for item n) costs n and changes the list to $[1, \ldots, n, n-1]$. The second request (for item $n-1$) costs n again. This takes us back to where we started: a list of $[1, \ldots, n-1, n]$ and the next request for item n. We can see that each request will cost n, for a total cost of nm.

The optimal algorithm first moves n and $n-1$ to the front for a cost of $2n$. (We will define the cost of rearranging the list below.) Once this is done, requests for item $n-1$ cost 1 and requests for item n cost 2. Thus, the total cost will be $2n + \frac{3}{2}m$. As m becomes arbitrarily large, the competitive ratio tends to $nm/\frac{3}{2}m = \frac{2}{3}n$. This is $\Theta(n)$, which is the worst possible since the cost of any algorithm must be at least m and at most nm. ■

Before we can analyze the performance of MTF, we need to define the cost of rearranging the list. For this analysis, we will use the Sleator-Tarjan cost model. As above, searching to index i in the list costs i. Once we have found the requested item, it can then be moved forward arbitrarily far at no cost. These are called "free swaps". We can also swap any other items (once we have advanced to or past them in the list) at a cost of 1. These are called "paid swaps". The reader may feel that this model is somewhat questionable. We will have more to say on this below.

Theorem 2 *MTF is 2-competitive in the Sleator-Tarjan model.*

Proof: In analyzing the competiveness of MTF, we are not interested in the cost of any particular operation, but only the cost of a sequence of operations. The reader may recall that we encountered this situation before during our study of data structures. Hence, it is not surprising that the technique that was so useful to us in that context, amortized analysis, will be useful to us again here.

We will compute the amortized cost of MTF using a potential function. We will consider running MTF and OPT in parallel. After each item is processed by both algorithms, we will compare the two lists. We define the potential after i requests, Φ_i, to be number of inversions between the lists, i.e., the number of pairs (i, j) such that item i is before item j in the MTF's list but after in OPT's list. Initially, the two lists are identical, so $\Phi_0 = 0$. Also note that $\Phi_i \geq 0$.

Now, let's consider the i-th request, for some item x, in the MTF algorithm. Let k_{MTF} be the index of x in the list before the request is processed. Consider the $k_{\text{MTF}} - 1$ items before x in the list. Let f be the number of these items that come before x in OPT's list and b be the number that come after. Since every such item is either before or after x, we have $f + b = k_{\text{MTF}} - 1$. After x is processed by MTF, it will be moved to the front, so all of these items will now be behind x in MFT's list. The b items behind x in OPT's list are no longer inversions, so these swaps decrease Φ by b, but the f items in front of x in OPT's list have become inversions, so these swaps increase Φ by f. So we have $\Delta\Phi_i = \Phi_i - \Phi_{i-1} = f - b$. Thus, the amortized cost of the i-th request in MTF is $C^i_{\text{MTF}} = k_{\text{MTF}} + \Delta\Phi_i = k_{\text{MTF}} + f - b = (f + b + 1) + f - b = 2f + 1$.

Next, let's consider this request for OPT. The cost for OPT to find x is at least $f+1$ since there are at least f items before x in OPT's list. OPT may also perform some swaps. If we let p denote the number of paid swaps, then we have $C^i_{\mathrm{OPT}} \geq f+1+p$. We must also consider how swaps affect Φ. Free swaps can only decrease the number of inversions since we know that the requested item x is now at the front of MTF's list. Each paid swap may increase the potential by 1. Thus, Φ may increase by as much as p. Adding this to the cost from above, we see that the amortized cost of this request for MTF is now $C^i_{\mathrm{MTF}} \leq 2f+1+p$.

Putting these two parts together, we have $C^i_{\mathrm{MTF}} \leq 2f+1+p < 2(f+1+p) \leq 2\,C^i_{\mathrm{OPT}}$. The amortized cost of MTF over the entire sequence is $C_{\mathrm{MTF}} = \sum_{i=1}^{m} C^i_{\mathrm{MTF}} < \sum_{i=1}^{m} 2\,C^i_{\mathrm{OPT}} = 2\,C_{\mathrm{OPT}}$. The real cost of MTF is the amortized cost minus the total change in potential, $\Phi_m - \Phi_0$. As noted above, $\Phi_m \geq 0$ and $\Phi_0 = 0$, so the real cost must be less than the amortized cost. Thus, the real cost is less than $2\,C_{\mathrm{OPT}}$, which proves that MTF is 2-competitive. ■

As noted above, the Sleator-Tarjan model, while greatly simplifying the analysis of MTF, may not be an accurate description of the costs in a real implementation. Initially, one might question the notion of a "free swap". However, this is a fairly reasonable model of reality: if we have already spent $O(i)$ time finding the item at index i, the $O(i)$ cost of inserting it somewhere earlier only affects the constant factor in the running time. Making these operations free only changes the competitiveness of any algorithm by a constant factor, and it simplifies the analysis. The real problem with this model is that it only allows swaps. For example, in a real implementation, we could reverse the list up to index i in $O(i)$ time. However, in this model, it would cost $\Theta(i^2)$ since $\Theta(i^2)$ swaps are required. Restricting to swaps benefits MTF by preventing OPT from changing its list too quickly, but in reality, this is not an accurate model of the real costs.

An alternative model of Munro [1] allows the items to be rearranged arbitrarily at no cost once we have advanced past them in the list. In other words, we can rearrange the list through index i arbitrarily at a cost of i. This is perhaps more natural because many complicated rearrangements can be performed in $O(i)$ time, so they are free in the same sense as the "free swaps" of the Sleator-Tarjan model. Unfortunately, in this model, MTF is not competitive. Furthermore, no algorithm can be $o(n/\log n)$ competitive.

19.4 Paging

Paging is a familiar problem from operating system and computer system design. The idea is to model memory has having two parts: there is an unlimited amount of slow memory and a cache containing k pages of fast memory. The difference between "fast" and "slow" in these contexts is so large that we can consider the cost of accessing slow memory to be 1 and the cost of accessing fast memory to be 0. The problem consists of a sequence of m page accesses. If the requested page is not in the cache, then in addition to paying a cost of 1, we must move that page into the cache and evict some other page. As expected, the goal is to choose the pages to evict so as to reduce the number of cache misses over the sequence.

There are a large number of familiar algorithms for paging. Let's consider three simple examples. The FIFO (first-in first-out) algorithm manages the cache as a queue: on each cache miss, it evicts the page that was brought into the cache at the earliest time. The LRU (least recently used) algorithm evicts the page that was least recently accessed. The RAND algorithm evicts a page at random. In this lecture, we will focus on deterministic algorithms like FIFO and LRU. The next lecture will consider randomized algorithms like RAND.

For the offline problem, there is a well-known optimal algorithm. It evicts the page whose next access is farthest into the future. We will not prove the optimality of this algorithm here. (It is a standard analysis of a greedy algorithm.)

For the online problem, there is good news and bad news about deterministic algorithms.

Theorem 3 (Bad News) *No deterministic algorithm is c-competitive for any $c < k$.*

Proof: Consider any deterministic algorithm A. We will construct a sequence on which A does poorly. The sequence will consist of accesses to just $k + 1$ pages. Before each request, A has some k pages in its cache, so one page is not in the cache. We simply request that page. Thus, we can make A miss on every request for a total cost of m.

To complete the proof, we must show that OPT can do significantly better than this. As mentioned above, OPT will evict the page accessed furthest in the future. In the worst case, the k-th access from now would miss: since every other page must be accessed before that one, the worst case would be if the k pages in the cache were accessed all in a row. Thus, OPT has at most $\frac{1}{k}\, m$ misses, so the competative ratio of A is at least $m/\frac{1}{k}\, m = k$. ■

Before we get to the good news, we need a quick definition.

Definition 3 *A deterministic paging algorithm is* conservative *if it has at most k cache misses on any sequence of requests to just k pages.*

Both FIFO and LRU are conservative. It is not hard to show that any conservative algorithm is k-competitive. However, we can say even more than this.

Theorem 4 (Good News) *Any conservative paging algorithm A is $k/(k-h+1)$-competative when compared to an offline OPT with a cache of size h.*

In particular, if OPT's cache is half the size, $h = \frac{1}{2}k$, then the algorithm is 2-competitive. It is also important to note that restricting OPT to a smaller cache breaks the lower bound construction from Theorem 3.

Proof: We will analyze the sequence of requests in phases. Each phase is defined to be maximally long but such that only k distinct pages are requested. The conservatism of A implies that it has at most k cache misses per phase: each phase is a sequence of requests on only k pages. The offline OPT must miss on the first access of each new phase because we know its cache contains some subset of the k distinct pages accessed in the previous phase and the first access is not to one of those pages. (This fact alone proves that A is k-competitive.) After this first request, for some element x, OPT has x plus $h - 1$ unknown pages in its cache. Since there are $k - 1$ remaining pages to be accessed in this phase, at least $(k - 1) - (h - 1) = k - h$ of them must miss: at most $h - 1$ of these pages could be in OPT's cache. Thus, the total number of cache misses for OPT is at least these $k - h$ plus the miss for x, which is $k - h + 1$, so the competitive ratio is at most $k/(k - h + 1)$. ■

References

[1] J. Ian Munro. On the competitiveness of linear search. In *Proceedings of the 8th Annual European Symposium on Algorithms (ESA)*, September 2000.